|  |  |
| --- | --- |
| **Joint Collaborative Team on Video Coding (JCT-VC)**  **of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11**  9th Meeting: Geneva, CH, 27 April – 7 May 2012 | Document: JCTVC-I1003\_d4 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **High efficiency video coding (HEVC) text specification draft 7** | | |
| *Status:* | Output Document of JCT-VC | | |
| *Purpose:* | 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

Draft 7 of High efficiency video coding.

Ed. Notes (D7):

* RQT related issues ([#459](http://hevc.kw.bbc.co.uk/trac/ticket/459))
* Log2MaxTrafoSize constraint ([#348](http://hevc.kw.bbc.co.uk/trac/ticket/348))
* log2TrafoWidth1 and log2TrafoHeight1 calculation for NSQT ([#458](http://hevc.kw.bbc.co.uk/trac/ticket/458))
* non\_square\_quadtree\_enabled\_flag ([#403](http://hevc.kw.bbc.co.uk/trac/ticket/403))
* Wrong 'else if...' for 'if (skip\_flag)' in CU syntax ([#452](http://hevc.kw.bbc.co.uk/trac/ticket/452))
* use\_delta\_flag[ j ] is not decoded when used\_by\_curr\_pic\_flag[ j ] is 1 ([#451](http://hevc.kw.bbc.co.uk/trac/ticket/451))
* combined\_inter\_pred\_ref\_idx does not exist anymore ([#448](http://hevc.kw.bbc.co.uk/trac/ticket/448))
* Typos in reference picture list combination ([#446](http://hevc.kw.bbc.co.uk/trac/ticket/446))
* Text cleanup of QP prediction / derivation ([#492](http://hevc.kw.bbc.co.uk/trac/ticket/492))
* Equation (7-59) and (7-60) for RPS derivation do not match HM6.0 ([#445](http://hevc.kw.bbc.co.uk/trac/ticket/445))
* Chroma NSRQT fixes ([#505](http://hevc.kw.bbc.co.uk/trac/ticket/505),[#506](http://hevc.kw.bbc.co.uk/trac/ticket/506))
* Missing decoding process for AMP fixed ([#361](http://hevc.kw.bbc.co.uk/trac/ticket/361))
* Deblocking processing order fixed ([#412](http://hevc.kw.bbc.co.uk/trac/ticket/412))
* Derivation of coefficient group scan fixed ([#372](http://hevc.kw.bbc.co.uk/trac/ticket/372))
* Phrase "one of the following conditions is true" fixed ([#540](http://hevc.kw.bbc.co.uk/trac/ticket/540))
* Non-normative aspects in scaling process removed ([#544](http://hevc.kw.bbc.co.uk/trac/ticket/544))
* scaling\_list\_present\_flag issues fixed ([#407](http://hevc.kw.bbc.co.uk/trac/ticket/407))
* CTB,CB,PB,TB/CTU,CU,PU,TU defined
* beta\_offset\_div2 and tc\_offset\_div2 semantics fixed ([#353](http://hevc.kw.bbc.co.uk/trac/ticket/353))
* pixel replaced by sample ([#406](http://hevc.kw.bbc.co.uk/trac/ticket/406))
* references in HRD Annex C fixed ([#557](http://hevc.kw.bbc.co.uk/trac/ticket/557))
* sao\_type\_idx binarization fixed ([#367](http://hevc.kw.bbc.co.uk/trac/ticket/367))
* raster scan to tile scan order fixed ([#376](http://hevc.kw.bbc.co.uk/trac/ticket/376))
* error in tileId[] derivation loop fixed ([#558](http://hevc.kw.bbc.co.uk/trac/ticket/558))
* CtbAddrRS update in slice data syntax fixed ([#345](http://hevc.kw.bbc.co.uk/trac/ticket/345))
* SAO process issues fixed ([#504](http://hevc.kw.bbc.co.uk/trac/ticket/504))
* SAO clipping added ([#517](http://hevc.kw.bbc.co.uk/trac/ticket/517))
* fixed-length (FL) binarization process fixed ([#518](http://hevc.kw.bbc.co.uk/trac/ticket/518))
* intra transform skipping shift with high bit-depth fixed ([#560](http://hevc.kw.bbc.co.uk/trac/ticket/560))
* missing syntax element sao\_type\_idx added in sao\_param syntax table ([#576](http://hevc.kw.bbc.co.uk/trac/ticket/576))
* residual coding semantics typo fixed ([#333](http://hevc.kw.bbc.co.uk/trac/ticket/333))
* PCM alignment at PU syntax fixed ([#346](http://hevc.kw.bbc.co.uk/trac/ticket/346))
* non-break hyphen and minus fixed ([#347](http://hevc.kw.bbc.co.uk/trac/ticket/347))
* uneccessary shared\_pps\_info\_enabled\_flag semantics removed ([#351](http://hevc.kw.bbc.co.uk/trac/ticket/351))
* skip\_flag semantics fixed ([#355](http://hevc.kw.bbc.co.uk/trac/ticket/355))
* POC usage description fixed ([#359](http://hevc.kw.bbc.co.uk/trac/ticket/359))
* replaced coding tree depth variables cuDepth and cbDepth with ctDepth ([#385](http://hevc.kw.bbc.co.uk/trac/ticket/385))
* PicWidthInSamples fixed ([#443](http://hevc.kw.bbc.co.uk/trac/ticket/443))
* RefPicList naming fixed ([#478](http://hevc.kw.bbc.co.uk/trac/ticket/478))
* References to StCurr0/1 changed to StCurrBefore/After ([#482](http://hevc.kw.bbc.co.uk/trac/ticket/482))
* two references in 8.1 fixed ([#479](http://hevc.kw.bbc.co.uk/trac/ticket/479))
* one reference in 8.1 fixed ([#509](http://hevc.kw.bbc.co.uk/trac/ticket/509))
* replaced 'nal\_ref\_idc' by 'nal\_ref\_flag' in subclause 8.3.6 ([#510](http://hevc.kw.bbc.co.uk/trac/ticket/510))
* removed non-breaking space between log2\_min\_coding\_block\_size\_minus and 3 ([#386](http://hevc.kw.bbc.co.uk/trac/ticket/386))
* removed the "+1" in clause C in description of DPB size ([#522](http://hevc.kw.bbc.co.uk/trac/ticket/522))
* moved the restriction on the relationship between tiles and slices to subclause 6.3 ([#331](http://hevc.kw.bbc.co.uk/trac/ticket/331))
* Incorporated CBF coding without derivation process (JCTVC-I0152)
* Incorporated Unified CBFU and CBFV Coding in RQT (JCTVC-I0332)
* Incorporated BoG on I\_PCM / lossless deblocking unification (JCTVC-I0586)
* Incorporated transform and quantization bypass (JCTVC-I0529)
* Incorporated intra 4x4 transform skipping (JCTVC-I0408)
* Incorporated modifed deblocking threshold derivation table (JCTVC-I0258)
* Incorporated constrained motion data compression (JCTVC-I0182)
* Removed SAO parameters from APS (JCTVC-I0021)
* Incorporated SAO offset signaling with magnitude and sign (JCTVC-I0168)
* Incorporated SAO offset magnitude TU binarization (JCTVC-I0066)
* Incorporated no SAO merge at tile boundaries (JCTVC-I0172)
* Incorporated reordering of slice type values (JCTVC-I0500)
* Incorporated having tile syntax only in PPS and reordering of pic\_parameter\_set\_id in slice header to solve the slice header parsing issue (JCTVC-I0113)
* Incorporated moving list scaling syntax as well as deblocking filter parameters from APS to SPS and PPS (JCTVC-I0465)
* Incorporated a note on the presense of required parameter sets for random access (JCTVC-I0067)
* Incorporated an additional constraint on RPS for TLA pictures (JCTVC-I0236)
* Incorporated changing the derivation of the variable prevRefPic used in derivation of picture order count (JCTVC-I0345)
* Incorporated high-level syntax clean-ups on TMVP enabling as well as signalling of collocated picture, CU QP delta, entropy slice header, and slice header syntax (JCTVC-I0127, JCTVC-I0266, and JCTVC-I0420)
* Incorporated mandating nal\_ref\_flag to be 1 for CRA pictures (JCTVC-I0143)
* Incorporated broken link access (BLA) pictures, signaling of leading/TFD pictures, signalling of presence of leading/TFD pictures, and allocation of NAL unit types (JCTVC-I0275, JCTVC-I0278, JCTVC-I0404 and JCTVC-I0607)
* Incorporated removal of combined list (JCTVC-I0125)
* Incorporated entropy slice enabling in PPS, support of dependent slice, a constraint on prevRefPic, slice header byte alignment, entropy slice header, relationship between TLA and temporal\_id\_nesting\_flag, tile and WPP byte alginment, semantics of num\_reorder\_pics[ i ], and semantics of temporal\_id (JCTVC-I0138, JCTVC-I0229, JCTVC-I0330 and JCTVC-I0600)
* Incorporated WPP simplification and a restriction on coexistence of WPP and slices (JCTVC-I0360 and JCTVC-I0361)
* Incorporated removal of entry point markers, addition of VUI flag tiles\_fixed\_structure\_flag, mandating entry point signalling for each tile and WPP sub-stream, and entry point offsets being relative to end of slice header (JCTVC-I0159, JCTVC-I0233, JCTVC-I0237 and JCTVC-I0357)
* Incorporated a fix for an unhandled LTRP case, a fix to the POC MSB cycle coding, coding of LTRP POC LSB directly as u(v), and no MVP scaling for LTRPs (JCTVC-I0234, JCTVC-I0340, JCTVC-I0422)
* Incorporated HRD buffering for CRA/BLA pictures and sub-picture based CPB operation (JCTVC-I0277 and JCTVC-I0588)
* Incorporated introduction of video parameter set, and extension mechanisms for slice header and slice layer RBSP (JCTVC-I0230 and JCTVC-I0235)
* Incorporated increasing of POC range to 64 bits, a limit of POC difference between the current picture and a long-term reference picture, changes to scaling list syntax, and changed signalling for profile and level (JCTVC-I0045, JCTVC-I0059, and JCTVC-I0499)
* Incorporated a bug fix for the recovery point SEI message and a change to the decoded picture hash SEI message (JCTVC-I0044 and JCTVC-I0218)
* Incorporated a change to the field indication SEI message (JCTVC-I0393)

Ed. Notes (D6):

* Incorporated limiting dynamic range when qmatrix is used (JCTVC-H0541)
* Incorporated simplified intra horizontal and vertical modes (JCTVC-H0238)
* Incorporated DC mode as a default mode (JCTVC-H0242)
* Incorporated compatible QP prediction with RC and AQ (JCTVC-H0204)
* Incorporated burst transmission of I\_PCM (JCTVC-H0051)
* SAO syntax fix (Ticket #308)
* Unused semantics removal related to reference picture list modification (Ticket #293)
* hPos and vPos table fix in SAO EO (Ticket #300)
* Typo in weighted prediction fix (Ticket #309)
* Incorporated clipping operation in strong deblocking (JCTVC-H0275)
* Incorporated removing sign of SAO offset (JCTVC-H0434)
* Fix QP’Cb and QP’Cr to consider QpBdOffsetC (Ticket #313)
* Fix padding issue of LM mode
* Incorporated deblocking filter simplification (JCTVC-H0473)
* Fix ALF syntax mismatch
* Incorporated ALF with single filter type (JCTVC-H0068)
* Incorporated intra mode coding clean-up and simplification (JCTVC-H0712)
* Fix clipping in DST (Ticket #307)
* Considering pcm\_loop\_filter\_disable\_flag in SAO (Ticket #301)
* Fix wrong geometry of sub-pel interpolation filter (Ticket #318)
* Fix max value of num\_ref\_idx\_l0/l1\_default\_active\_minus1 (Ticket #281)
* Fix 16x16 and 32x32 quantization matrices (Ticket #320)
* Incorporated quantization matrix signalling (JCTVC-H0237)
* Incorporated deblocking parameter signalling (JCTVC-H0424/H0398)
* Incorporated chroma mode signalling (JCTVC-H0475/H0326)
* Incorporated 4x4 and 8x8 default quantization matrices (JCTVC-H0461)
* Incorporated lossless mode (JCTVC-H0530)
* Incorporated downsampling of q-matrix (JCTVC-H0230)
* Fix ALF chroma coefficients prediction (Ticket #321)
* Incorporated two stage design ALF with LCU-based syntax (JCTVC-H0274)
* Incorporated SAO with LCU-based syntax (JCTVC-H0273)
* Incorporated change to add a condition for presence of ref\_idx\_list\_curr (JCTVC-H0137 proposal #1)
* Incorporated unification of reference picture list modification processes (JCTVC-H0138)
* Incorporated change to NAL unit header and output flag (part of JCTVC-H0388)
* Incorporated coding treeblock and coding block scanning and address derivation.
* Incorporated changes relating to allowing the bitstream to start with a CRA picture (JCTVC-H0496)
* Incorporated multiple sign bits hiding (JCTVC-H0481)
* Incorporated 8x8 diagonal scan by 4x4 diagonal sub-scans (JCTVC-H0526/H0399)
* Incorporated abs\_greater1 and abs\_greater2 context reduction (JCTVC-H0130)
* Incorporated 8 bit codeword, change the Rice parameter to 4 (JCTVC-H0498)
* Incorporated high throughput binarization for CABAC (JCTVC-H0554)
* Incorporated sharing sig\_coeff\_flag cxtCnt=0 at high frequency area (JCTVC-H0095)
* Incorporated unified sig\_coeff\_flag context selection for 16x16 and 32x32 (JCTVC-H0290)
* Incorporated simplification on sig\_coeff\_group\_flag coding (JCTVC-H0131)
* Incorporated Profiles and Levels (JCTVC-H0738)
* Incorporated CABAC initialization process (JCTVC-H0535)
* Incorporated part\_mode context reduction (JCTVC-H0545)
* Incorporated merge index context reduction (JCTVC-H0251)
* Incorporated CABAC bit to bin expantion ratio limit (JCTVC-H0450)
* Incorporated CABAC\_init\_flag (JCTVC-H0540)
* Incorporated last\_sig\_coeff\_position\_prefix for luma context reduction (JCTVC-H0537)
* Incorporated last\_sig\_coeff\_position\_prefix for chroma context reduction (JCTVC-H0514)
* Incorporated unified transform and coefficient tree (JCTVC-H0123)
* Fix chroma cbf syntax mismatch (Ticket #295)
* Incorporated decisions relating to picture size and cropping parameters (response to JCTVC-H0485)
* Incorported adoptions on tiles, WPP and entropy slices documented in JCTVC-H0737 (response to JCTVC-H0439, H0463, H0513, H0517, and H0556 )
* Incorported decisions on long-term reference picture signalling (response to JCTVC-H0200 and JCTVC-H0531) and a resriction on POC values (response to JCTVC-H0449)
* Merge estimation region syntax (JCTVC-H0082)
* Setting the merge TMVP refidx to 0 for the non-first partition (JCTVC-H0278 / JCTVC-H0199)
* One merge candidate list for all partitions inside a 8x8 CU (JCTVC-H0240 Variant 2 conditioned on log2\_parallel\_merge\_level\_minus2 > 0 as described in H0082 section 6)
* Removing non-scaled bi-predictive merging candidates (JCTVC-H0250 /JCTVC-H0164)
* Removing the list empty check and the duplicate check in AMVP for zero motion vector (JCTVC-H0239)
* Removing redundant spatial candidates check in AMVP (JCTVC-H0316 first part)
* Clipping scaled MV to 16 bit and adjust it according to profile/level decisions (JCTVC-H0216 / JCTVC-H0555)
* Fix AMVP non-scaled/scaled candidate WD/HM mismatch by correcting the WD (JCTVC-H0462)
* Motion prediction at entropy slice boundary (JCTVC-H0362)
* Limiting collocated temporal reference to one per picture (JCTVC-H0442)
* Modification of bi-prediction syntax CE9 BP08 (JCTVC-H0111)
* CE9 SP (JCTVC-H0252)
* Incorported adoptions related to reference picture set, SPS syntax and HRD (JCTVC-H0568, H0566, H0567, H0423 and H0412).
* Rewrote SAO and ALF syntax and semantics and part of the processes.
* Incorporated harmonization of number of ALF classes between RA and BA modes (JCTVC-H0409)
* Incorporated harmonization of ALF luma and chroma center coefficient prediction (JCTVC-H0483)
* Incorporated JCTVC-H0174-B with modification of number of bands and coding of the offset band according to JCTVC-H0406 (5b FLC/bypass coding)..
* Imported VUI from AVC with HEVC modifications (based upon JCTVC-F289)
* Imported SEI messages from AVC with HEVC semantic restrictions (based upon JCTVC-E346)
* Incorporated display orientation SEI (VCEG-AR12\_r2)
* Incorporated temporal structure SEI (JCTVC-H0423)
* Incorporated decoded picture hash SEI (JCTVC-E490)
* Incorporated field indication SEI and VUI (JCTVC- H0720)
* Incorporated inheriting the QP prediction value at the left edge from the slice header in which the LCU belongs (JCTVC-H0226)
* Fixed missing definition of "cu\_qp\_delta\_enabled\_flag" (Ticket #310)
* Incorporated start-code based markers for signalling of tile entry points (JCTVC-F594)
* Made the following clean-up changes:
  + Addressing editing notes and cross references in subclauses 0.2, 0.3, 0.6, 0.7 and Clause 3
  + Various changes to the definitions, including
    - Correcting of the definition of "IDR picture", as marking all reference pictures as "unused for reference" is not anymore immediately after the decoding of the IDR picture due to the reference picture set based picture buffering mechanism
    - Resolving editing notes related to the definitions of "leading picture" and "output order"
    - Resolving editing notes related to the definitions of "reference picture list (X)" (X being 0 or 1)
    - Correcting for the indention of some bullet items in subclause 8.3.3 (Decoding process for generating unavailable reference pictures)
    - Improving definitions of "tile", "tree"
    - Added a definition of "z-scan"
  + Removing mentioning of redundant pictures and changing "primary (coded) picture" to "(coded) picture", auxiliary (coded) picture, and data partitioning
  + Moving of the sentence "The first coded picture in a bitstream shall be an IDR picture or a CRA picture." from the decoder conformance subclause to the bitstream conformance subclause
* Incorporated the ability, at the slice level, to disable loop filtering across slice boundaries (in response to JCTVC-H0391)

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 signalling 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 initialization
* 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. (GJS): Entropy slice was a better name, although it may not have been so great either. We should generally avoid naming things in ways that sound like advertising rather than technical descriptions (excessive use of "advanced", etc.).]
* 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 colour planes) be included from the start? Currently, it has been left in the text as it doesn't seem to affect much text.
* Use of bin string and bit string should be consistent.
* 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.
* 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 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.
* Absolute operation |·| is not defined. It could be replaced by Abs(·) or defined separately in arithmetic operators subclause. [Ed. (GJS): I think it's better not to use | | for absolute value. Are we actually using it?]

**CONTENTS**

*Page*

[Abstract i](#_Toc327299536)

[0 Introduction 1](#_Toc327299537)

[0.1 Prologue 1](#_Toc327299538)

[0.2 Purpose 1](#_Toc327299539)

[0.3 Applications 1](#_Toc327299540)

[0.4 Publication and versions of this Specification 1](#_Toc327299541)

[0.5 Profiles and levels 2](#_Toc327299542)

[0.6 Overview of the design characteristics 2](#_Toc327299543)

[0.7 How to read this Specification 2](#_Toc327299544)

[0.8 Luma inter transform block split direction derivation process 3](#_Toc327299545)

[0.9 Chroma inter transform block split direction derivation process 3](#_Toc327299546)

[1 Scope 4](#_Toc327299547)

[2 Normative references 4](#_Toc327299548)

[2.1 General 4](#_Toc327299549)

[2.2 Identical Recommendations | International Standards 4](#_Toc327299550)

[2.3 Paired Recommendations | International Standards equivalent in technical content 4](#_Toc327299551)

[2.4 Additional references 5](#_Toc327299552)

[3 Definitions 5](#_Toc327299553)

[4 Abbreviations 11](#_Toc327299554)

[5 Conventions 12](#_Toc327299555)

[5.1 Arithmetic operators 12](#_Toc327299556)

[5.2 Logical operators 12](#_Toc327299557)

[5.3 Relational operators 13](#_Toc327299558)

[5.4 Bit-wise operators 13](#_Toc327299559)

[5.5 Assignment operators 13](#_Toc327299560)

[5.6 Range notation 13](#_Toc327299561)

[5.7 Mathematical functions 13](#_Toc327299562)

[5.8 Order of operation precedence 14](#_Toc327299563)

[5.9 Variables, syntax elements, and tables 15](#_Toc327299564)

[5.10 Text description of logical operations 16](#_Toc327299565)

[5.11 Processes 17](#_Toc327299566)

[6 Source, coded, decoded and output data formats, scanning processes, and neighbouring relationships 17](#_Toc327299567)

[6.1 Bitstream formats 17](#_Toc327299568)

[6.2 Source, decoded, and output picture formats 17](#_Toc327299569)

[6.3 Spatial subdivision of pictures, slices and tiles 19](#_Toc327299570)

[6.4 Derivation process for the availability of a coding block with a minimum coding block address 21](#_Toc327299571)

[6.5 Scanning processes 21](#_Toc327299572)

[6.5.1 Coding tree block raster and tile scanning conversation process 21](#_Toc327299573)

[6.5.2 Z scanning array initialization process 22](#_Toc327299574)

[6.5.3 Zig-zag scanning array initialization process 23](#_Toc327299575)

[6.5.4 Up-right diagonal scanning array initialization process 23](#_Toc327299576)

[6.5.5 Horizontal scanning array initialization process 25](#_Toc327299577)

[6.5.6 Vertical scanning array initialization process 25](#_Toc327299578)

[6.6 Luma inter transform block split direction derivation process 26](#_Toc327299579)

[6.7 Chroma inter transform block split direction derivation process 26](#_Toc327299580)

[7 Syntax and semantics 27](#_Toc327299581)

[7.1 Method of specifying syntax in tabular form 27](#_Toc327299582)

[7.2 Specification of syntax functions and descriptors 28](#_Toc327299583)

[7.3 Syntax in tabular form 29](#_Toc327299584)

[7.3.1 NAL unit syntax 29](#_Toc327299585)

[7.3.2 Raw byte sequence payloads and RBSP trailing bits syntax 30](#_Toc327299586)

[7.3.2.1 Video parameter set RBSP syntax 30](#_Toc327299587)

[7.3.2.2 Sequence parameter set RBSP syntax 31](#_Toc327299588)

[7.3.2.3 Picture parameter set RBSP syntax 33](#_Toc327299589)

[7.3.2.4 Scaling list data syntax 34](#_Toc327299590)

[7.3.2.5 Scaling list syntax 34](#_Toc327299591)

[7.3.2.6 Adaptation parameter set RBSP syntax 35](#_Toc327299592)

[7.3.2.7 Supplemental enhancement information RBSP syntax 35](#_Toc327299593)

[7.3.2.8 Access unit delimiter RBSP syntax 36](#_Toc327299594)

[7.3.2.9 Filler data RBSP syntax 36](#_Toc327299595)

[7.3.2.10 Slice layer RBSP syntax 36](#_Toc327299596)

[7.3.2.11 RBSP slice trailing bits syntax 36](#_Toc327299597)

[7.3.2.12 RBSP trailing bits syntax 36](#_Toc327299598)

[7.3.2.13 Byte alignment syntax 37](#_Toc327299599)

[7.3.3 Slice header syntax 38](#_Toc327299600)

[7.3.3.1 Short-term reference picture set syntax 40](#_Toc327299601)

[7.3.3.2 Reference picture list modification syntax 41](#_Toc327299602)

[7.3.3.3 Adaptive loop filter parameter syntax 42](#_Toc327299603)

[7.3.3.4 Adaptive loop filter unit syntax 43](#_Toc327299604)

[7.3.3.5 Adaptive loop filter info syntax 44](#_Toc327299605)

[7.3.3.6 Adaptive loop filter coding unit control parameter syntax 44](#_Toc327299606)

[7.3.3.7 Prediction weight table syntax 45](#_Toc327299607)

[7.3.4 Slice data syntax 46](#_Toc327299608)

[7.3.4.1 Sample adaptive offset parameter syntax 47](#_Toc327299609)

[7.3.5 Coding tree syntax 48](#_Toc327299610)

[7.3.6 Coding unit syntax 49](#_Toc327299611)

[7.3.7 Prediction unit syntax 51](#_Toc327299612)

[7.3.7.1 Motion vector difference coding syntax 52](#_Toc327299613)

[7.3.7.2 PCM sample syntax 52](#_Toc327299614)

[7.3.8 Transform tree syntax 53](#_Toc327299615)

[7.3.9 Transform unit syntax 55](#_Toc327299616)

[7.3.10 Residual coding syntax 56](#_Toc327299617)

[7.4 Semantics 58](#_Toc327299618)

[7.4.1 NAL unit semantics 58](#_Toc327299619)

[7.4.1.1 Encapsulation of an SODB within an RBSP (informative) 61](#_Toc327299620)

[7.4.1.2 Order of NAL units and association to coded pictures, access units, and video sequences 62](#_Toc327299621)

[7.4.2 Raw byte sequence payloads and RBSP trailing bits semantics 65](#_Toc327299622)

[7.4.2.1 Video parameter set RBSP semantics 65](#_Toc327299623)

[7.4.2.2 Sequence parameter set RBSP semantics 66](#_Toc327299624)

[7.4.2.3 Picture parameter set RBSP semantics 71](#_Toc327299625)

[7.4.2.4 Scaling list data semantics 74](#_Toc327299626)

[7.4.2.5 Scaling list semantics 75](#_Toc327299627)

[7.4.2.6 Adaptation parameter set RBSP semantics 77](#_Toc327299628)

[7.4.2.7 Supplemental enhancement information RBSP semantics 77](#_Toc327299629)

[7.4.2.8 Access unit delimiter RBSP semantics 78](#_Toc327299630)

[7.4.2.9 Filler data RBSP semantics 78](#_Toc327299631)

[7.4.2.10 Slice layer RBSP semantics 78](#_Toc327299632)

[7.4.2.11 RBSP slice trailing bits semantics 78](#_Toc327299633)

[7.4.2.12 RBSP trailing bits semantics 79](#_Toc327299634)

[7.4.2.13 Byte alignment semantics 79](#_Toc327299635)

[7.4.3 Slice header semantics 79](#_Toc327299636)

[7.4.3.1 Short-term reference picture set semantics 83](#_Toc327299637)

[7.4.3.2 Reference picture list modification semantics 85](#_Toc327299638)

[7.4.3.3 Adaptive loop filter parameter semantics 85](#_Toc327299645)

[7.4.3.4 Adaptive loop filter unit semantics 86](#_Toc327299646)

[7.4.3.5 Adaptive loop filter info semantics 86](#_Toc327299647)

[7.4.3.6 Adaptive loop filter coding unit control parameter semantics 87](#_Toc327299648)

[7.4.3.7 Weighted prediction parameters semantics 87](#_Toc327299649)

[7.4.4 Slice data semantics 88](#_Toc327299650)

[7.4.4.1 Sample adaptive offset parameter semantics 88](#_Toc327299651)

[7.4.5 Coding tree semantics 90](#_Toc327299652)

[7.4.6 Coding unit semantics 90](#_Toc327299653)

[7.4.7 Prediction unit semantics 91](#_Toc327299654)

[7.4.7.1 Motion vector difference coding semantics 92](#_Toc327299655)

[7.4.8 Transform tree semantics 93](#_Toc327299656)

[7.4.9 Transform coefficient semantics 94](#_Toc327299657)

[7.4.10 Residual coding semantics 94](#_Toc327299658)

[8 Decoding process 96](#_Toc327299659)

[8.1 General [Ed. (GJS): Create similar subclauses for other places where there is "orphaned" text that is at the parent level of a section that contains subclauses.] 96](#_Toc327299660)

[8.2 NAL unit decoding process 97](#_Toc327299661)

[8.3 Slice decoding process 97](#_Toc327299662)

[8.3.1 Decoding process for picture order count 97](#_Toc327299663)

[8.3.2 Decoding process for reference picture set 98](#_Toc327299664)

[8.3.4 Decoding process for reference picture lists construction 102](#_Toc327299665)

[8.3.4.1 General 102](#_Toc327299666)

[8.3.4.2 Initialization process for reference picture lists 102](#_Toc327299667)

[8.3.5 Marking of the current picture after decoding 103](#_Toc327299698)

[8.4 Decoding process for coding units coded in intra prediction mode 103](#_Toc327299699)

[8.4.1 Derivation process for luma intra prediction mode 104](#_Toc327299700)

[8.4.2 Derivation process for chroma intra prediction mode 107](#_Toc327299701)

[8.4.3 Decoding process for intra blocks 107](#_Toc327299702)

[8.4.3.1 Intra sample prediction 108](#_Toc327299703)

[8.5 Decoding process for coding units coded in inter prediction mode 115](#_Toc327299704)

[8.5.1 Inter prediction process 116](#_Toc327299705)

[8.5.2 Decoding process for prediction units in inter prediction mode 118](#_Toc327299706)

[8.5.2.1 Derivation process for motion vector components and reference indices 119](#_Toc327299707)

[8.5.2.2 Decoding process for inter prediction samples 131](#_Toc327299708)

[8.5.3 Decoding process for the residual signal of coding units coded in inter prediction mode 139](#_Toc327299709)

[8.5.3.1 Decoding process for luma residual blocks 140](#_Toc327299710)

[8.5.3.2 Decoding process for chroma residual blocks 141](#_Toc327299711)

[8.6 Scaling, transformation and array construction process prior to deblocking filter process 143](#_Toc327299712)

[8.6.1 Derivation process for quantization parameters 143](#_Toc327299713)

[8.6.2 Scaling and transformation process 144](#_Toc327299714)

[8.6.3 Scaling process for transform coefficients 145](#_Toc327299715)

[8.6.4 Transformation process for scaled transform coefficients 145](#_Toc327299716)

[8.6.4.1 Transformation process 147](#_Toc327299717)

[8.6.5 Array construction process 148](#_Toc327299718)

[8.7 In-loop filter process 148](#_Toc327299719)

[8.7.1 Deblocking filter process 149](#_Toc327299720)

[8.7.1.1 Derivation process of transform block boundary 150](#_Toc327299721)

[8.7.1.2 Derivation process of prediction block boundary 151](#_Toc327299722)

[8.7.1.3 Derivation process of boundary filtering strength 152](#_Toc327299723)

[8.7.1.4 Edge filtering process 153](#_Toc327299724)

[8.7.2 Sample adaptive offset process 162](#_Toc327299725)

[8.7.2.1 Coding tree block modification process 162](#_Toc327299726)

[8.7.3 Adaptive loop filter process 164](#_Toc327299727)

[8.7.3.1 Boundary padding process 165](#_Toc327299728)

[8.7.3.2 Derivation process for filter coefficients 167](#_Toc327299729)

[8.7.3.3 Derivation process for filter index array for luma samples 169](#_Toc327299730)

[8.7.3.4 Filtering process for luma samples 170](#_Toc327299731)

[8.7.3.5 Filtering process for chroma samples 171](#_Toc327299732)

[9 Parsing process 172](#_Toc327299733)

[9.1 Parsing process for Exp-Golomb codes 172](#_Toc327299734)

[9.1.1 Mapping process for signed Exp-Golomb codes 174](#_Toc327299735)

[9.2 CABAC parsing process for slice data 175](#_Toc327299736)

[9.2.1 Initialization process 176](#_Toc327299737)

[9.2.1.1 Initialization process for context variables 176](#_Toc327299738)

[9.2.1.2 Memorization process for context variables 183](#_Toc327299739)

[9.2.1.3 Synchronization process for context variables 183](#_Toc327299740)

[9.2.1.4 Initialization process for the arithmetic decoding engine 183](#_Toc327299741)

[9.2.2 Binarization process 184](#_Toc327299742)

[9.2.2.1 Unary (U) binarization process 188](#_Toc327299743)

[9.2.2.2 Truncated unary (TU) binarization process 189](#_Toc327299744)

[9.2.2.3 Truncated Rice (TR) binarization process 189](#_Toc327299745)

[9.2.2.4 k-th order Exp-Golomb (EGk) binarization process 189](#_Toc327299746)

[9.2.2.5 Fixed-length (FL) binarization process 190](#_Toc327299747)

[9.2.2.6 Binarization process for cu\_qp\_delta 190](#_Toc327299748)

[9.2.2.7 Binarization process for part\_mode 190](#_Toc327299749)

[9.2.2.8 Binarization process for coeff\_abs\_level\_remaining 191](#_Toc327299750)

[9.2.2.9 Binarization process for intra\_chroma\_pred\_mode 192](#_Toc327299751)

[9.2.3 Decoding process flow 192](#_Toc327299752)

[9.2.3.1 Derivation process for ctxIdx 192](#_Toc327299753)

[9.2.3.2 Arithmetic decoding process 202](#_Toc327299754)

[9.2.4 Arithmetic encoding process (informative) 209](#_Toc327299755)

[9.2.4.1 Initialization process for the arithmetic encoding engine (informative) 209](#_Toc327299756)

[9.2.4.2 Encoding process for a binary decision (informative) 210](#_Toc327299757)

[9.2.4.3 Renormalization process in the arithmetic encoding engine (informative) 211](#_Toc327299758)

[9.2.4.4 Bypass encoding process for binary decisions (informative) 213](#_Toc327299759)

[9.2.4.5 Encoding process for a binary decision before termination (informative) 214](#_Toc327299760)

[9.2.4.6 Byte stuffing process (informative) 216](#_Toc327299761)

[10 Specification of bitstream subsets 216](#_Toc327299762)

[10.1 Sub-bitstream extraction process 216](#_Toc327299763)

[Annex A Profiles and levels 217](#_Toc327299764)

[A.1 Overview of profiles and levels 217](#_Toc327299765)

[A.2 Requirements on video decoder capability 217](#_Toc327299766)

[A.3 Profiles 217](#_Toc327299767)

[A.3.1 General 217](#_Toc327299768)

[A.3.2 Main profile 217](#_Toc327299769)

[A.4 Levels 218](#_Toc327299770)

[A.4.1 General 218](#_Toc327299771)

[A.4.2 Profile-specific level limits for the Main profile 219](#_Toc327299772)

[A.4.3 Effect of level limits on picture rate (informative) 220](#_Toc327299773)

[Annex B Byte stream format 223](#_Toc327299774)

[B.1 Byte stream NAL unit syntax and semantics 223](#_Toc327299775)

[B.1.1 Byte stream NAL unit syntax 223](#_Toc327299776)

[B.1.2 Byte stream NAL unit semantics 223](#_Toc327299777)

[B.2 Byte stream NAL unit decoding process 224](#_Toc327299778)

[B.3 Decoder byte-alignment recovery (informative) 224](#_Toc327299779)

[Annex C Hypothetical reference decoder 225](#_Toc327299780)

[C.1 General 225](#_Toc327299781)

[C.2 Operation of coded picture buffer (CPB) 227](#_Toc327299782)

[C.2.1 Timing of bitstream arrival 227](#_Toc327299783)

[C.2.2 Timing of decoding unit removal and decoding of decoding unit 229](#_Toc327299784)

[C.3 Operation of the decoded picture buffer (DPB) 229](#_Toc327299785)

[C.4 Bitstream conformance 231](#_Toc327299786)

[C.5 Decoder conformance 232](#_Toc327299787)

[C.5.1 Operation of the output order DPB 233](#_Toc327299788)

[C.5.2 Removal of pictures from the DPB 233](#_Toc327299789)

[C.5.2.1 "Bumping" process 234](#_Toc327299790)

[C.5.3 Picture decoding, marking and storage 234](#_Toc327299791)

[Annex D Supplemental enhancement information 235](#_Toc327299792)

[D.1 SEI payload syntax 236](#_Toc327299793)

[D.1.1 Buffering period SEI message syntax 237](#_Toc327299794)

[D.1.2 Picture timing SEI message syntax 237](#_Toc327299795)

[D.1.3 Pan-scan rectangle SEI message syntax 238](#_Toc327299796)

[D.1.4 Filler payload SEI message syntax 238](#_Toc327299797)

[D.1.5 User data registered by Rec. ITU-T T.35 SEI message syntax 238](#_Toc327299798)

[D.1.6 User data unregistered SEI message syntax 238](#_Toc327299799)

[D.1.7 Recovery point SEI message syntax 238](#_Toc327299800)

[D.1.8 Scene information SEI message syntax 238](#_Toc327299801)

[D.1.9 Full-frame snapshot SEI message syntax 238](#_Toc327299802)

[D.1.10 Progressive refinement segment start SEI message syntax 238](#_Toc327299803)

[D.1.11 Progressive refinement segment end SEI message syntax 238](#_Toc327299804)

[D.1.12 Film grain characteristics SEI message syntax 238](#_Toc327299805)

[D.1.13 Deblocking filter display preference SEI message syntax 238](#_Toc327299806)

[D.1.14 Post-filter hint SEI message syntax 238](#_Toc327299807)

[D.1.15 Tone mapping information SEI message syntax 238](#_Toc327299808)

[D.1.16 Frame packing arrangement SEI message syntax 238](#_Toc327299809)

[D.1.17 Display orientation SEI message syntax 239](#_Toc327299810)

[D.1.18 SOP description SEI message syntax 239](#_Toc327299811)

[D.1.19 Field indication SEI message syntax 239](#_Toc327299812)

[D.1.20 Decoded picture hash SEI message syntax 240](#_Toc327299813)

[D.1.21 Reserved SEI message syntax 240](#_Toc327299814)

[D.2 SEI payload semantics 240](#_Toc327299815)

[D.2.1 Buffering period SEI message semantics 240](#_Toc327299816)

[D.2.2 Picture timing SEI message semantics 241](#_Toc327299817)

[D.2.3 Pan-scan rectangle SEI message semantics 242](#_Toc327299821)

[D.2.4 Filler payload SEI message semantics 242](#_Toc327299822)

[D.2.5 User data registered by ITU-T Rec. T.35 SEI message semantics 242](#_Toc327299823)

[D.2.6 User data unregistered SEI message semantics 242](#_Toc327299824)

[D.2.7 Recovery point SEI message semantics 242](#_Toc327299825)

[D.2.8 Scene information SEI message semantics 243](#_Toc327299828)

[D.2.9 Full-frame snapshot SEI message semantics 243](#_Toc327299829)

[D.2.10 Progressive refinement segment start SEI message semantics 243](#_Toc327299830)

[D.2.11 Progressive refinement segment end SEI message semantics 243](#_Toc327299831)

[D.2.12 Film grain characteristics SEI message semantics 243](#_Toc327299832)

[D.2.13 Deblocking filter display preference SEI message semantics 243](#_Toc327299833)

[D.2.14 Post-filter hint SEI message semantics 243](#_Toc327299834)

[D.2.15 Tone mapping information SEI message semantics 243](#_Toc327299835)

[D.2.16 Frame packing arrangement SEI message semantics 243](#_Toc327299836)

[D.2.17 Display orientation SEI message semantics 244](#_Toc327299837)

[D.2.18 SOP description SEI message semantics 245](#_Toc327299838)

[D.2.19 Field indication SEI message semantics 245](#_Toc327299839)

[D.2.20 Decoded picture hash SEI message semantics 248](#_Toc327299840)

[D.2.21 Reserved SEI message semantics 249](#_Toc327299841)

[Annex E Video usability information 251](#_Toc327299842)

[E.1 VUI syntax 252](#_Toc327299843)

[E.1.1 VUI parameters syntax 252](#_Toc327299844)

[E.1.2 HRD parameters syntax 253](#_Toc327299845)

[E.2 VUI semantics 253](#_Toc327299846)

[E.2.1 VUI parameters semantics 253](#_Toc327299847)

[E.2.2 HRD parameters semantics 263](#_Toc327299848)

**List of figures**

[Figure 6‑1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture [Ed. Re-draw figure] 18](#_Toc327299849)

[Figure 6‑2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture [Ed. Re-draw figure] 19](#_Toc327299850)

[Figure 6‑3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture [Ed. Re-draw figure] 19](#_Toc327299851)

[Figure 6‑4 – A picture with 11 by 9 coding tree blocks that is partitioned into two slices (informative) [Ed. Re-draw figure] 20](#_Toc327299852)

[Figure 6‑5 – A picture with 13 by 8 coding tree blocks that is partitioned into three tiles (informative) 20](#_Toc327299853)

[Figure 7‑1 – Structure of an access unit not containing any NAL units with nal\_unit\_type equal to 0, , or 30, or in the ranges of 9..24, 25..28, or 32..63 65](#_Toc327299854)

[Figure 8‑1 – Intra prediction mode directions (informative) 105](#_Toc327299855)

[Figure 8‑2 – Intra prediction angle definition (informative) 112](#_Toc327299856)

[Figure 8‑3 – Spatial motion vector neighbours 127](#_Toc327299857)

[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 133](#_Toc327299858)

[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 135](#_Toc327299859)

[Figure 9‑1 – Spatial neighbour T that is used to invoke the coding tree block availability derivation process relative to the current coding tree block (informative) 175](#_Toc327299860)

[Figure 9‑2 – Overview of the arithmetic decoding process for a single bin (informative) 203](#_Toc327299861)

[Figure 9‑3 – Flowchart for decoding a decision 205](#_Toc327299862)

[Figure 9‑4 – Flowchart of renormalization 208](#_Toc327299863)

[Figure 9‑5 – Flowchart of bypass decoding process 208](#_Toc327299864)

[Figure 9‑6 – Flowchart of decoding a decision before termination 209](#_Toc327299865)

[Figure 9‑7 – Flowchart for encoding a decision 211](#_Toc327299866)

[Figure 9‑8 – Flowchart of renormalization in the encoder 212](#_Toc327299867)

[Figure 9‑9 – Flowchart of PutBit(B) 213](#_Toc327299868)

[Figure 9‑10 – Flowchart of encoding bypass 214](#_Toc327299869)

[Figure 9‑11 – Flowchart of encoding a decision before termination 215](#_Toc327299870)

[Figure 9‑12 – Flowchart of flushing at termination 216](#_Toc327299871)

[Figure C‑1 – Structure of byte streams and NAL unit streams for HRD conformance checks [Ed. (GJS): Fix figure numbering in annex(es)] 225](#_Toc327299872)

[Figure C‑2 – HRD buffer model 226](#_Toc327299873)

[Figure D‑1 – Nominal vertical and horizontal sampling locations of 4:2:0 samples in top and bottom fields 247](#_Toc327299874)

[Figure D‑2 – Nominal vertical and horizontal sampling locations of 4:2:2 samples in top and bottom fields 247](#_Toc327299875)

[Figure D‑3 – Nominal vertical and horizontal sampling locations of 4:4:4 samples in top and bottom fields 248](#_Toc327299876)

[Figure E‑1 – Location of chroma samples for top and bottom fields for chroma\_format\_idc equal to 1 (4:2:0 chroma format) as a function of chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field 260](#_Toc327299877)

**List of tables**

[Table 5‑1 – Operation precedence from highest (at top of table) to lowest (at bottom of table) 15](#_Toc327299878)

[Table 6‑1 – SubWidthC, and SubHeightC values derived from chroma\_format\_idc and separate\_colour\_plane\_flag 18](#_Toc327299879)

[Table 7‑1 – NAL unit type codes and NAL unit type classes 58](#_Toc327299880)

[Table 7‑2 – Specification of SizeID 75](#_Toc327299881)

[Table 7‑3 – Specification of MatrixID according to SizeID, prediction type and colour component 75](#_Toc327299882)

[Table 7‑4 – Specification of default values of ScalingList[ 0 ][ MatrixID ][ i ] with i=0..15 76](#_Toc327299883)

[Table 7‑5 – Specification of default values of ScalingList[ 1..3 ][ MatrixID ][ 0 ][ i ] with i=0..63 76](#_Toc327299884)

[Table 7‑6 – Meaning of pic\_type 78](#_Toc327299885)

[Table 7‑7 – Name association to slice\_type 79](#_Toc327299886)

[Table 7‑8 – used\_by\_curr\_pic\_flag[ j ] and use\_delta\_flag[ j ] operations for the reference picture set 83](#_Toc327299887)

[Table 7‑9 – Specification of the SAO type 89](#_Toc327299888)

[Table 7‑10 – Name association to prediction mode and partitioning type 91](#_Toc327299889)

[Table 7‑11 – Name association to inter prediction mode 92](#_Toc327299890)

[Table 7‑12 – Specification of ScanType[ log2TrafoSize − 2 ][ IntraPredMode ] 94](#_Toc327299891)

[Table 8‑1 – Specification of intra prediction mode and associated names 105](#_Toc327299892)

[Table 8‑2 – 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 1 107](#_Toc327299893)

[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 0 107](#_Toc327299894)

[Table 8‑4 – Specification of intraHorVerDistThres[ nS ] for various prediction unit sizes 110](#_Toc327299895)

[Table 8‑5 – Specification of intraPredAngle 113](#_Toc327299896)

[Table 8‑6 – Specification of invAngle 113](#_Toc327299897)

[Table 8‑7 – Specification of lmDiv 115](#_Toc327299898)

[Table 8‑8 – Specification of l0CandIdx and l1CandIdx 124](#_Toc327299899)

[Table 8‑9 – Assignment of the luma prediction sample predSampleLXL[ xL, yL ] 134](#_Toc327299900)

[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 136](#_Toc327299901)

[Table 8‑11 – Specification of QPC as a function of qPI 144](#_Toc327299902)

[Table 8‑12 – Specification of horizTrType and vertTrType 146](#_Toc327299903)

[Table 8‑13 – Derivation of threshold variables β′ and tC′ from input Q 161](#_Toc327299904)

[Table 8‑14 – Specification of hPos and vPos according to the sample adaptive offset type 164](#_Toc327299905)

[Table 8‑15 – Specification of the number of bits in alf fractional part, alfPrecisionBit 168](#_Toc327299906)

[Table 8‑16. avgVarTab values for different avgVar and directions. 170](#_Toc327299907)

[Table 8‑17 – Specification of horPos[ i ] according to alfFilterShape for adaptive loop filter process 171](#_Toc327299908)

[Table 8‑18 – Specification of verPos[ i ] according to alfFilterShape for adaptive loop filter process 171](#_Toc327299909)

[Table 9‑1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative) 173](#_Toc327299910)

[Table 9‑2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative) 173](#_Toc327299911)

[Table 9‑3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v) 174](#_Toc327299912)

[Table 9‑4 – Association of ctxIdx and syntax elements for each initialisationType in the initialization process 177](#_Toc327299913)

[Table 9‑5 – Values of variable initValue for sao\_merge\_left\_flag ctxIdx 178](#_Toc327299914)

[Table 9‑6 – Values of variable initValue for sao\_merge\_up\_flag ctxIdx 178](#_Toc327299915)

[Table 9‑7 – Values of variable initValue for sao\_type\_idx ctxIdx 178](#_Toc327299916)

[Table 9‑8 – Values of variable initValue for sao\_offset\_abs ctxIdx 178](#_Toc327299917)

[Table 9‑9 – Values of variable initValue for alf\_cu\_flag ctxIdx 178](#_Toc327299918)

[Table 9‑10 – Values of variable initValue for split\_coding\_unit\_flag ctxIdx 178](#_Toc327299919)

[Table 9‑11 – Values of variable initValue for cu\_transquant\_bypass\_flag ctxIdx 179](#_Toc327299920)

[Table 9‑12 – Values of variable initValue for skip\_flag ctxIdx 179](#_Toc327299921)

[Table 9‑13 – Values of variable initValue for cu\_qp\_delta ctxIdx 179](#_Toc327299922)

[Table 9‑14 – Values of variable initValue for pred\_mode and part\_mode 179](#_Toc327299923)

[Table 9‑15 – Values of variable initValue for prev\_intra\_luma\_pred\_flag ctxIdx 179](#_Toc327299924)

[Table 9‑16 – Values of variable initValue for intra\_chroma\_pred\_mode ctxIdx 179](#_Toc327299925)

[Table 9‑17 – Value of variable initValue for merge\_flag ctxIdx 180](#_Toc327299926)

[Table 9‑18 – Values of variable initValue for merge\_idx ctxIdx 180](#_Toc327299927)

[Table 9‑19 – Values of variable initValue for inter\_pred\_idc ctxIdx 180](#_Toc327299928)

[Table 9‑20 – Values of variable initValue for ref\_idx\_l0, ref\_idx\_l1 ctxIdx 180](#_Toc327299929)

[Table 9‑21 – Values of variable initValue for abs\_mvd\_greater0\_flag and abs\_mvd\_greater1\_flag ctxIdx 180](#_Toc327299930)

[Table 9‑22 – Values of variable initValue for mvp\_l0\_flag, mvp\_l1\_flag ctxIdx 180](#_Toc327299931)

[Table 9‑23 – Values of variable initValue for no\_residual\_data\_flag ctxIdx 181](#_Toc327299932)

[Table 9‑24 – Values of variable initValue for split\_transform\_flag ctxIdx 181](#_Toc327299933)

[Table 9‑25 – Values of variable initValue for cbf\_luma ctxIdx 181](#_Toc327299934)

[Table 9‑26 – Values of variable initValue for cbf\_cb and cbf\_cr ctxIdx 181](#_Toc327299935)

[Table 9‑27 – Values of variable initValue for transform\_skip\_flag ctxIdx 181](#_Toc327299936)

[Table 9‑28 – Values of variable initValue for last\_significant\_coeff\_x\_prefix ctxIdx 181](#_Toc327299937)

[Table 9‑29 – Values of variable initValue for last\_significant\_coeff\_y\_prefix ctxIdx 182](#_Toc327299938)

[Table 9‑30 – Values of variable initValue for significant\_coeff\_group\_flag ctxIdx 182](#_Toc327299939)

[Table 9‑31 – Values of variable initValue for significant\_coeff\_flag ctxIdx 182](#_Toc327299940)

[Table 9‑32 – Values of variable initValue for coeff\_abs\_level\_greater1\_flag ctxIdx 183](#_Toc327299941)

[Table 9‑33 – Values of variable initValue for coeff\_abs\_level\_greater2\_flag ctxIdx 183](#_Toc327299942)

[Table 9‑34 – Syntax elements and associated types of binarization, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset 184](#_Toc327299943)

[Table 9‑35 – Bin string of the unary binarization (informative) 189](#_Toc327299944)

[Table 9‑36 – Binarization for part\_mode 191](#_Toc327299945)

[Table 9‑37 – Specifcation of cTRMax depending on cRiceParam 191](#_Toc327299946)

[Table 9‑38 – Specifcation of intra\_chroma\_pred\_mode depending on chroma\_pred\_from\_luma\_enabled\_flag 192](#_Toc327299947)

[Table 9‑39 – Assignment of ctxIdxInc to binIdx for all ctxIdxTable and ctxIdxOffset values 194](#_Toc327299948)

[Table 9‑40 – Specifcation of ctxIdxInc using left and above syntax elements 198](#_Toc327299949)

[Table 9‑41 – Specifcation of lastCtx[ i ] 198](#_Toc327299950)

[Table 9‑42 – Specifcation of ctxIdxMap4x4[ i ] 200](#_Toc327299951)

[Table 9‑43 – Specifcation of ctxIdxMap8x8[ i ] 200](#_Toc327299952)

[Table 9‑44 – Specification of rangeTabLPS depending on pStateIdx and qCodIRangeIdx 206](#_Toc327299953)

[Table 9‑45 – State transition table 207](#_Toc327299954)

[Table A‑1 – Level limits 219](#_Toc327299955)

[Table A‑2 – Specification of cpbBrVclFactor and cpbBrNalFactor 220](#_Toc327299956)

[Table A‑3 – SliceRate limits for Main profile 220](#_Toc327299957)

[Table A‑4 – Maximum picture rates (pictures per second) at level 1 to 4.3 for some example picture sizes 221](#_Toc327299958)

[Table A‑5 – Maximum picture rates (pictures per second) at level 5 to 6.2 for some example picture sizes 222](#_Toc327299959)

[Table D‑1 – Indicated interpretation of field\_pic\_flag and progressive\_source\_flag 246](#_Toc327299960)

[Table D‑2 – Interpretation of hash\_type 249](#_Toc327299961)

[Table E‑1 – Meaning of sample aspect ratio indicator 254](#_Toc327299962)

[Table E‑2 – Colour primaries 255](#_Toc327299963)

[Table E‑3 – Transfer characteristics 256](#_Toc327299964)

[Table E‑4 – Matrix coefficients 259](#_Toc327299965)

**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 standardization. National Bodies that are members of ISO and IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. 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  through 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. This Recommendation | International Standard was developed in the JCT-VC.

## Purpose

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

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 as well as to enable the use of multi-core parallel encoding and decoding devices. 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.

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 (video streaming, etc.)

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 (MPEG). 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 .

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.

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 quantized, 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 quantized transform coefficient information and encoded using 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  () and moves on to clause  (). Clause  should be read for the geometrical relationship of the source, input, and output of the decoder. Clause  (Vertical scanning array initialization process

Inputs to this process are

– a block width blkWidth,

– a block height blkHeight.

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

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

## Luma inter transform block split direction derivation process

Inputs to this process are

– the current luma transform width log2TrafoWidth,

– the current luma transform height log2TrafoHeight,

– the current depth of the residual quadtree trafoDepth.

Output of this process is the variable interTbSplitDirectionL.

The variable log2TrafoSize is set equal to ( log2TrafoWidth + log2TrafoHeight ) >> 1.

When a luma transform block is split into four blocks with smaller horizontal and/or vertical size for the purpose of transform coding, the variable interTbSplitDirectionL defines where the four blocks are located inside the parent block. interTbSplitDirectionL equal to 0 results in the four smaller blocks being vertically aligned, i.e. one column of blocks. interTbSplitDirectionL equal to 1 results in the four smaller blocks being horizontally aligned, i.e. one row of blocks. interTbSplitDirectionL equal to 2 results in two rows with two blocks being horizontally aligned, i.e. two rows and two columns of blocks. interTbSplitDirectionL is derived as follows.

if( ( nsrqt\_enabled\_flag  
 && ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_2NxN  
 | | PartMode = = PART\_2NxnU | | PartMode = = PART\_2NxnD ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth < log2TrafoHeight ) ) )  
{  
 interTbSplitDirectionL = 0  
}  
else if( ( nsrqt\_enabled\_flag  
 && ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_Nx2N  
 | | PartMode = = PART\_nLx2N | | PartMode = = PART\_nRx2N ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth > log2TrafoHeight ) ) )  
{  
 interTbSplitDirectionL = 1  
}  
else   
{  
 interTbSplitDirectionL = 2  
}

## Chroma inter transform block split direction derivation process

Inputs to this process are

– the current luma transform width log2TrafoWidth,

– the current luma transform height log2TrafoHeight,

– the current depth of the residual quadtree trafoDepth.

Output of this process is the variable interTbSplitDirectionC.

When a chroma transform block is split into four blocks with smaller horizontal and/or vertical size for the purpose of transform coding, the variable interTbSplitDirectionC defines where the four blocks are located inside the parent block. interTbSplitDirectionC equal to 0 results in the four smaller blocks being vertically aligned, i.e. one column of blocks. interTbSplitDirectionC equal to 1 results in the four smaller blocks being horizontally aligned, i.e. one row of blocks. interTbSplitDirectionC equal to 2 results in two rows with two blocks being horizontally aligned, i.e. two rows and two columns of blocks. interTbSplitDirectionC is derived as follows.

– If ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoWidth is greater than log2TrafoHeight, interTbSplitDirectionC is set to 1.

– Otherwise, if ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoHeight is greater than log2TrafoWidth, interTbSplitDirectionC is set to 0.

– Otherwise, if ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoHeight is equal to log2TrafoWidth, interTbSplitDirectionC is set to 2.

– Otherwise, the luma inter transform block split direction derivation process as specified in 6.6 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionC.

Syntax and semantics) specifies the order to parse syntax elements from the bitstream. See subclauses - for syntactical order and see subclause  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  (). Finally, clause  specifies how the syntax elements are mapped into decoded samples. Throughout reading this Specification, the reader should refer to clauses  (), (), and () as needed. Annexes  through also form an integral part of this Recommendation | International Standard.

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

## General

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.]

## Identical Recommendations | International Standards

– None.

## Paired Recommendations | International Standards equivalent in technical content

– ITU-T H.264 (in force), *Advanced video coding for generic audiovisual services*.

ISO/IEC 14496-10: in force, *Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding*.

## Additional references

– 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) adapted 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 *coded picture*. In addition to the *coded slice NAL units* of the *coded picture*, the access unit may also contain other *NAL units* not containing *slices* of the *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. **broken link access (BLA) unit**: An *access unit* in which the *coded picture* is a *BLA picture*.
  15. **broken link access (BLA) picture**: A *RAP picture* for which each *slice* has nal\_unit\_type equal to 6 or 7.
  16. **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.
  17. **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.
  18. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex .
  19. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  20. **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 *coded picture* is a *CRA picture*.
  2. **clean random access (CRA) picture**: A *RAP picture* for which each *slice* has nal\_unit\_type equal to 4 or 5.
  3. **coded picture**: A *coded representation* of a *picture* containing all *coding tree units* of the *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 .
  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 a *CRA access unit* that is the first access unit in the bitstream, an *IDR access unit* or a *BLA access unit*, followed by zero or more non-IDR and non-BLA *access* *units* including all subsequent *access units* up to but not including any subsequent *IDR or BLA access unit*.
  8. **coding block**: A square NxN *block* of samples. The division of a *coding tree block* into *coding blocks* is a *partitioning*.
  9. **coding tree block**: A square NxN *block* of samples. The division of one of the arrays that make up a *picture* that has three sample arrays or of the array that make up a *picture* in monochrome format or a *picture* that is coded using three separate colour planes into *coding tree blocks* is a *partitioning*.
  10. **coding tree unit**: A *coding tree block* of *luma* samples, two corresponding *coding tree blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding tree block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples. The division of a *slice* into *coding tree units* is a *partitioning*.
  11. **coding unit**: A *coding block* of *luma* samples, two corresponding *coding blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples. The division of a *coding tree unit* into *coding units* is a *partitioning*.
  12. **column**: An integer number of *coding tree blocks*. *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*. The division of each *picture* into *columns* is a *partitioning*.
  13. **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.
  14. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
  15. **DC transform coefficient**: A *transform coefficient* for which the *frequency index* is zero in all dimensions.
  16. **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.
  17. **decoded picture buffer (DPB)**: A buffer holding *decoded pictures* for reference, output reordering, or output delay specified for the *hypothetical reference decoder* in Annex .
  18. **decoder**: An embodiment of a *decoding process*.
  19. **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*.
  20. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
  21. **decoding process**: The process specified in this Recommendation | International Standard that reads a *bitstream* and derives *decoded* *pictures* from it.
  22. **decoding unit**: An *access unit* or a subset of an *access unit*. If SubPicCpbFlag is equal to 0, a decoding unit is an *access unit*. Otherwise, a decoding unit consists of one or more *VCL NAL units* in an *access unit* and the associated non-VCL *NAL units*. For the first *VCL NAL unit* in an *access unit*, the associated non-VCL *NAL units* are and the filler data *NAL units*, if any, immediately following the first *VCL NAL unit* and all non-VCL *NAL units* in the *access unit* that precede the first *VCL NAL unit*. For a *VCL NAL unit* that is not the first *VCL NAL unit* in an *access unit*, the associated non-VCL *NAL units* are the filler data *NAL unit*, if any, immediately following the *VCL NAL unit*.
  23. **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.*
  24. **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*.
  25. **encoder**: An embodiment of an *encoding process*.
  26. **encoding process**: A process, not specified in this Recommendation | International Standard, that produces a *bitstream* conforming to this Recommendation | International Standard.
  27. **flag**: A variable that can take one of the two possible values 0 and 1.
  28. **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to an *inverse transform* part of the *decoding process.*
  29. **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.
  30. **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*.
  31. **I slice**: A *slice* that is decoded using *intra prediction* only.
  32. **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.
  33. **instantaneous decoding refresh (IDR) access unit**: An *access unit* in which the *coded picture* is an *IDR picture*.
  34. **instantaneous decoding refresh (IDR) picture**: A *RAP* *picture* for which each slice has nal\_unit\_type equal to 8.
  35. **inter coding**: Coding of a *block*, *slice*, or *picture* that uses *inter prediction*.
  36. **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*.
  37. **intra coding**: Coding of a *block, slice*, or *picture* that uses *intra prediction*.
  38. **intra prediction**: A *prediction* derived from only data elements (e.g. sample value) of the same decoded *slice*.
  39. **intra slice**: See *I slice*.
  40. **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*.
  41. **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 *coding tree* layers.
  42. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  43. **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*.
  44. **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into *reference picture list 0* (*list 1*).
  45. **list 0 (list 1) prediction**: *Inter prediction* of the content of a *slice* using a *reference index* pointing into *reference picture list 0* (*list 1*).
  46. **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* in case the *decoded pictures* are to be output from the *decoded picture buffer*. The output order of a picture is specified by the *picture order count* value, regardless of whether the picture is to be output.
  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 *quantization 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 one of the following *coded pictures*:

the previous *IDR picture* in *decoding order*, if any

the previous *BLA picture* in *decoding order*, if any

the previous *CRA picture* in *decoding order*, if any and if the previous *CRA picture* is the first *coded picture* in the *bitstream* in *decoding order*.

If more than one of the above *coded pictures* is present, the picture order count is relative to the last of such *coded pictures* in *decoding order*.

* 1. **prediction**: An embodiment of the *prediction process*.
  2. **prediction block**: A rectangular MxN *block* of samples on which the same *prediction* is applied. The division of a *coding block* into *prediction blocks* is a *partitioning*.
  3. **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.
  4. **prediction unit**: A *prediction block* of *luma* samples, two corresponding *prediction blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *prediction block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to predict the *prediction block* samples.
  5. **predictive slice**: See *P slice*.
  6. **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.
  7. **profile**: A specified subset of the syntax of this Recommendation | International Standard.
  8. **quadtree**: A *tree* in which a parent node can be split into four child nodes. A child node may become parent node for another split into four child nodes.
  9. **quantization parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
  10. **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
  11. **random access point (RAP) access unit**: An *access unit* in which the *coded picture* is a *RAP picture*.
  12. **random access point (RAP) picture**: A coded *picture* containing only *I slices* and for which each slice has nal\_unit\_type in the range of 4 to 8, inclusive; a RAP picture may be a *BLA picture*, a *CRA picture* or an *IDR picture*; all *coded pictures* that follow the RAP picture both in *decoding order* and *output order* do not use *inter prediction* from any *picture* that precedes the RAP picture either in *decoding order* or *output order*; and any *picture* that precedes the RAP picture in *decoding order* also precedes the RAP picture in *output order*.

NOTE – If all preceding access units in decoding order are not present, provided that each parameter set referred to by a RAP picture and all subsequent coded pictures in decoding order is present before its activation, the RAP picture and all subsequent coded pictures in decoding order that also follow the RAP picture in output order can be correctly decoded. [Ed. (GJS/YK): The wording of this note may be improved.]

* 1. **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.
  2. **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.
  3. **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.*
  4. **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*.
  5. **reference index**: An index into a *reference picture list*.
  6. **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*.
  7. **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)*.*
  8. **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 1*.
  9. **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*.
  10. **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*.
  11. **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.
  12. **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.
  13. **row**: An integer number of *coding tree blocks*. *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*. The division of each *picture* into *rows* is *a partitioning*.
  14. **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).
  15. **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.
  16. **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.*
  17. **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*.*
  18. **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.
  19. **slice**: An integer number of *coding tree blocks* ordered consecutively in the *raster scan*. The division of each *picture* into slices is a *partitioning*. The *coding tree block addresses* are derived from the first *coding tree block address* in a slice (as represented in the *slice header*)*.*
  20. **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *coding blocks* represented in the *slice*.
  21. **source**: Term used to describe the video material or some of its attributes before encoding.
  22. **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*.
  23. **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.
  24. **syntax element**: An element of data represented in the *bitstream*.
  25. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  26. **tagged for discard (TFD) access unit:** An *access unit* in which the *coded picture* is a *TFD picture.*
  27. **tagged for discard (TFD) picture:** A *coded picture* for which each *slice* has nal\_unit\_type equal to 2; a TFD picture is associated with the previous *CRA picture* or *BLA picture* in *decoding order* and precedes the associated *picture* in *output order*; when the associated *picture* is a *BLA picture*, or when the associated *picture* is a *CRA picture* that is the first *coded picture* in the *bitstream*, the TFD picture may not be correctly decodable and is not output.
  28. **temporal layer access (TLA) unit:** An *access unit* in which the *coded picture* is a *TLA picture*.
  29. **temporal layer access (TLA) picture**: A *coded picture* for which each *slice* has nal\_unit\_type equal to 3.
  30. **tile**: An integer number of *coding tree blocks* co-occurring in one *column* and one *row*, ordered consecutively in *coding tree block* *raster scan* of the *tile*. The division of each *picture* into *tiles* is a *partitioning*. *Tiles* in a *picture* are ordered consecutively in tile *raster scan* of the *picture*.
  31. **tile scan**: A specific seqential ordering of *coding tree blocks* *partitioning* a *picture*. The tile scan order traverses the *coding tree blocks* in *coding tree block* *raster scan* in a *tile*. Although a *slice* contains coding tree blocks that are consecutive in *coding tree block* *raster scan* of a *tile*, these *coding tree blocks* are not necessarily consecutive in *coding tree block* *raster scan* of the picture.
  32. **transform block**: A rectangular MxN *block* of samples on which the same *transform* is applied. The division of a *coding block* into *transform blocks* is a *partitioning*.
  33. **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*.
  34. **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.
  35. **transform unit**: A *transform block* of *luma* samples, two corresponding *transform blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *transform block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to transform the *transform block* samples.
  36. **tree**: A tree is a finite set of nodes with a unique root node. A terminating node is called a *leaf*.
  37. **universal unique identifier (UUID)**: An identifier that is unique with respect to the space of all universal unique identifiers.
  38. **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.
  39. **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.
  40. **VCL NAL unit**: A collective term for *coded slice NAL units*.
  41. **z-scan**: A specific seqential ordering of *blocks* *partitioning* a *picture*. When the *blocks* are of the same size as *coding tree blocks*, the z-scan order is identical to *coding tree block* *raster scan* of the *picture*. When the *blocks* are of a smaller size than *coding tree blocks*, i.e., *coding tree blocks* are further partitioned into smaller *coding blocks*, the z-scan order traverses from *coding tree block* to *coding tree block* in *coding tree block* *raster scan* of the *picture*, and inside each *coding tree block*, which may be divided into *quadtrees* hierachically to lower levels, the z-scan order traverses from *quadree* to *quadree* of a particular level in *quadtree-*of-the-particular-level *raster scan* of the *quadtree* of the immediately higher level.
  42. **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:

BLA Broken Link Access

CABAC Context-based Adaptive Binary Arithmetic Coding

CB Coding Block

CBR Constant Bit Rate

CRA Clean Random Access

CPB Coded Picture Buffer

CTB Coding Tree Block

CTU Coding Tree Unit

CU Coding Unit

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

PB Prediction Block

PU Prediction Unit

RAP Random Access Point

RBSP Raw Byte Sequence Payload

SEI Supplemental Enhancement Information

SODB String Of Data Bits

TB Transform Block

TFD Tagged For Discard

TU Transform Unit

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, the results of integer division and arithmetic shift operations are defined more precisely, and additional operations are defined, such as exponentiation and real-valued division. 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 non-negative 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 non-negative 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.

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  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 ) 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 one or more 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 coding block may be referred to by the variable name having a value equal to the address of the specific coding block.

# 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 .

## 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 .

The variables SubWidthC, and SubHeightC are specified in , depending on the chroma format sampling structure, which is specified through chroma\_format\_idc and separate\_colour\_plane\_flag. An entry marked as "-" in 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 . Alternative chroma sample relative locations may be indicated in video usability information (see Annex ).



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 .



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 .



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

## Spatial subdivision of pictures, slices and tiles

This subclause specifies how a picture is partitioned into slices, tiles and coding tree blocks. Pictures are divided into slices and tiles. A slice is a sequence of coding tree blocks. Likewise, a tile is a sequence of coding tree blocks. The samples are processed in units of coding tree blocks. The array size for each luma coding tree block in samples in both width and height is CtbSize. The width and height of the array for each chroma coding tree block are CtbWidthC and CtbHeightC, respectively.

For example, a picture may be divided into two slices as shown in . As another example, a picture may be divided into three tiles as shown in .

Unlike slices, tiles are always rectangular and always contain an integer number of coding tree blocks in coding tree block raster scan. A tile may consist of coding tree blocks contained in more than one slice. Similarly, a slice may consist of coding tree blocks contained in more than one tile.

One or both of the following conditions shall be fulfilled for each slice and tile:

– All coded blocks in a slice belong to the same tile.

– All coded blocks in a tile belong to the same slice.

NOTE 2 – Within the same picture, there may be both slices that contain multiple tiles and tiles that contain multiple slices.

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a slice contains only coding tree blocks 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 coding tree block address for the first coding tree block of each coded slice NAL unit.

NOTE – When separate\_colour\_plane\_flag is equal to 0, each coding tree block of a picture is contained in exactly one slice. When separate\_colour\_plane\_flag is equal to 1, each coding tree block of a colour component is contained in exactly one slice (i.e., information for each coding tree block 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 coding tree blocks that is partitioned into two slices (informative) [Ed. Re-draw figure]



Figure ‑ – A picture with 13 by 8 coding tree blocks that is partitioned into three tiles (informative)

Each coding tree block 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 coding tree block. The quadtree is split until a leaf is reached, which is referred to as the coding block. When the picture width is not an integer number of coding tree block sizes, the coding tree blocks at the right picture boundary are incomplete. When the picture height is not an integer number of coding tree block sizes, the coding tree blocks at the bottom picture boundary are incomplete.

The coding block 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 transform tree specifies the position and size of transform blocks. 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 blocks and associated syntax structures are encapsulated in a "unit" as follows.

– One prediction block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three prediction blocks (luma and chroma) and associated prediction syntax structures units are encapsulated in a prediction unit.

– One transform block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three transform blocks (luma and chroma) and associated transform syntax structures units are encapsulated in a transform unit.

– One coding block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three coding blocks (luma and chroma), the associated coding syntax structures and the associated prediction and transform units units are encapsulated in a coding unit.

– One coding tree block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three coding tree blocks (luma and chroma), the associated coding tree syntax structures and the associated coding units are encapsulated in a coding tree unit.

## Derivation process for the availability of a coding block with a minimum coding block address

Inputs to this process are

– a minimum coding block address minCbAddrZS in z-scan order

– the current minimum coding block address currMinCBAddrZS in z-scan order

Output of this process is the availability of the coding block with mininum coding block address cbAddrZS in z-scan order cbAvailable.

NOTE 1 – The meaning of availability is determined when this process is invoked.

NOTE 2 – Any coding block, regardless of its size, is associated with a mininum coding block address, which is the address of the coding block with the mininum coding block size in z-scan order.

– If one or more of the following conditions are true, cbAvailable is set to FALSE.

– minCbAddrZS is less than 0

– minCbAddrZS is greater than currMinCBAddrZS

– the coding block with mininum coding block address minCbAddrZS belongs to a different slice than the coding block with the current mininum coding block address currMinCBAddrZS and the dependent\_slice\_flag of the slice containing the coding block with the current mininum coding block address currMinCBAddrZS is equal to 0.

– the coding block with mininum coding block address minCbAddrZS is contained in a different tile than the coding block with the current mininum coding block address currMinCBAddrZS.

– Otherwise, cbAvailable is set to TRUE.

## Scanning processes

### Coding tree block raster and tile scanning conversation process

Inputs of this process are:

– variables picHeightInCtbs and picWidthInCtbs specifying the height and the width of a picture in units of CTBs.

Outputs of this process are:

– a vector colWidth[ i ] specifying the width of the i-th tile column in units of CTBs with the column i ranging from 0 to num\_tile\_columns\_minus1, inclusive,

– a vector ctbAddrRStoTS[ ctbAddrRS ] specifying the conversation from a CTB address in raster scan order to a CTB address in tile scan order with the index ctbAddrRS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive,

– a vector ctbAddrTStoRS[ ctbAddrTS ] specifying the conversation from a CTB address in tile scan order to a CTB address in raster scan order with the index ctbAddrTS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive,

– a vector tileId[ ctbAddrTS ] specifying the conversation from a CTB address in tile scan order to a tile id with ctbAddrTS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive.

The vector colWidth is derived as follows:

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

The values of rowHeight[ j ], specifying the height of the j-th tile row in units of CTBs, are derived as follows::

for( j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 if( uniform\_spacing\_flag )  
 rowHeight[ j ] = ( ( j + 1 ) \* picHeightInCtbs ) / ( num\_tile\_rows\_minus1 + 1 ) –   
 ( j \* picHeightInCtbs ) / ( num\_tile\_rows\_minus1 + 1)   
 else  
 rowHeight[ j ] = row\_height[ j ]

The values of colBd[ i ], specifying the location of the left column boundary of the i-th tile column in units of coding tree blocks, are derived as follows:

for( colBd[ 0 ] = 0, i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 colBd[ i + 1 ] = colBd[ i ] + colWidth[ i ]

The values of rowBd[ j ], specifying the location of the top row boundary of the j-th tile row in units of coding tree blocks, are derived as follows:

for( rowBd[ 0 ] = 0, j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 rowBd[ j + 1 ] = rowBd[ j ] + rowHeight[ j ]

The vector ctbAddrRStoTS is derived as follows:

for( ctbAddrRS = 0; ctbAddrRS < picHeightInCtbs \* picWidthInCtbs; ctbAddrRS++ ) {  
 tbX = ctbAddrRS % picWidthInCtbs  
 tbY = ctbAddrRS / picWidthInCtbs  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 if( tbX >= colBd[ i ] )  
 tileX = i  
 for( j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 if( tbY >= rowBd[ j ] )  
 tileY = j  
 ctbAddrRStoTS[ ctbAddrRS ] = 0  
 for( i = 0; i < tileX; i++ )  
 ctbAddrRStoTS[ ctbAddrRS ] += rowHeight[ tileY ] \* colWidth[ i ]  
 for( j = 0; j < tileY; j++ )  
 ctbAddrRStoTS[ ctbAddrRS ] += picWidthInCtbs \* rowHeight[ j ]  
 ctbAddrRStoTS[ ctbAddrRS ] += ( tbY − rowBd[ tileY ] ) \* colWidth[ tileX ] + tbX − colBd[ tileX ]  
 }

The vector ctbAddrTStoRS is derived as follows:

for( ctbAddrRS = 0; ctbAddrRS < picHeightInCtbs \* picWidthInCtbs; ctbAddrRS++ )  
 ctbAddrTStoRS[ ctbAddrRStoTS[ ctbAddrRS ] ] = ctbAddrRS

The vector tileId is derived as follows:

for( j = 0, tIdx = 0; j <= num\_tile\_rows\_minus1; j++ )  
 for( i = 0; i <= num\_tile\_columns\_minus1; i++, tIdx++ )  
 for( y = rowBd[ j ]; y < rowBd[ j + 1 ]; y++ )  
 for( x = colBd[ i ]; x < colBd[ i + 1 ]; x++ )  
 tileId[ ctbAddrRStoTS[ y\*picWidthInCtbs + x ] ] = tIdx

### Z scanning array initialization process

Inputs of this process are:

– a minimum CB size log2MinCbSize,

– a CTB size log2CtbSize,

– variables picHeightInCtbs and picWidthInCtbs specifying the height and the width of a picture in units of CTBs,

– a vector ctbAddrRStoTS[ ctbAddrRS ] specifying the conversation from a CTB address in raster scan order to a CTB address in tile scan order with the index ctbAddrRS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive.

Output of this process is an array minCbAddrZS[ x ][ y ] specifying the conversation from a location ( x, y ) in units of minimum CBs to a minimum CB address in z-scan order with x ranging from 0 to picWidthInMinCbs − 1, inclusive, and y ranging from 0 to picHeightInMinCbs − 1, inclusive.

The array minCbAddrZS is derived as follows.

for( y = 0; y < picHeightInMinCbs; y++ )  
 for( x = 0; x < picWidthtInMinCbs; x++) {  
 tbX = ( x << log2MinCbSize ) >> log2CtbSize  
 tbY = ( y << log2MinCbSize ) >> log2CtbSize  
 ctbAddrRS = picWidthInCtbs \* tbY + tbX  
 minCbAddrZS[ x ][ y ] = ctbAddrRStoTS[ ctbAddrRS ] << ( log2\_diff\_max\_min\_coding\_block\_size \* 2 )  
 for( i = 0, p = 0; i < log2\_diff\_max\_min\_coding\_block\_size; i++ ) {  
 m = 1 << i  
 p += ( m & x ? m\*m : 0 ) + ( m & y ? 2 \* m \* m : 0 )  
 }  
 minCbAddrZS[ x ][ y ] += p  
 }

### Zig-zag scanning array initialization process

Input to this process is a block size blkSize.

Output of this process is the array zigZag[ sPos ][ sComp ]. The array index sPos specify the scan position ranging from 0 to ( blkSize \* blkSize ) − 1. The array index sComp equal to 0 specifies the horizontal component and the array index sComp equal to 1 specifies the vertical component. The array zigZag is derived as follows.

zigZag[ 0 ][ 0 ] = 0  
zigZag[ 0 ][ 1 ] = 0  
i = 1  
x = 1  
y = 0  
stopLoop = ( blkSize = = 1 ) ? TRUE : FALSE  
while( !stopLoop ) {  
 while( x >= 0 ) {  
 if(x >= 0 && x < blkSize && y >= 0 && y < blkSize ) {  
 zigZag[ i ][ 0 ] = x  
 zigZag[ i ][ 1 ] = y  
 i++  
 }  
 x− −  
 y++  
 }  
 x = 0  
 while( y >= 0 ) {  
 if(x >= 0 && x < blkSize && y >= 0 && y < blkSize ) {  
 zigZag[ i ][ 0 ] = x  
 zigZag[ i ][ 1 ] = y  
 i++  
 }  
 x++  
 y− −  
 }  
 y = 0  
 if( i >= blkSize \* blkSize )   
 stopLoop = TRUE  
}

### Up-right diagonal scanning array initialization process

Inputs to this process are

– a block width blkWidth,

– a block height blkHeight.

Output of this process is the array diagScan[ sPos ][ sComp ]. The array index sPos specify the scan position ranging from 0 to ( blkWidth \* blkHeight ) − 1. The array index sComp equal to 0 specifies the horizontal component and the array index sComp 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 8 and blkHeight is less than 8, 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 4 or blkHeight is greater than 4), 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 ) )   
}

### Horizontal scanning array initialization process

Inputs to this process are

– a block width blkWidth,

– a block height blkHeight.

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

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

### Vertical scanning array initialization process

Inputs to this process are

– a block width blkWidth,

– a block height blkHeight.

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

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

## Luma inter transform block split direction derivation process

Inputs to this process are

– the current luma transform width log2TrafoWidth,

– the current luma transform height log2TrafoHeight,

– the current depth of the residual quadtree trafoDepth.

Output of this process is the variable interTbSplitDirectionL.

The variable log2TrafoSize is set equal to ( log2TrafoWidth + log2TrafoHeight ) >> 1.

When a luma transform block is split into four blocks with smaller horizontal and/or vertical size for the purpose of transform coding, the variable interTbSplitDirectionL defines where the four blocks are located inside the parent block. interTbSplitDirectionL equal to 0 results in the four smaller blocks being vertically aligned, i.e. one column of blocks. interTbSplitDirectionL equal to 1 results in the four smaller blocks being horizontally aligned, i.e. one row of blocks. interTbSplitDirectionL equal to 2 results in two rows with two blocks being horizontally aligned, i.e. two rows and two columns of blocks. interTbSplitDirectionL is derived as follows.

if( ( nsrqt\_enabled\_flag  
 && ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_2NxN  
 | | PartMode = = PART\_2NxnU | | PartMode = = PART\_2NxnD ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth < log2TrafoHeight ) ) )  
{  
 interTbSplitDirectionL = 0  
}  
else if( ( nsrqt\_enabled\_flag  
 && ( ( log2TrafoSize = = Log2MaxTrafoSize  
 | | ( log2TrafoSize < Log2MaxTrafoSize && trafoDepth = = 0 ) )   
 && log2TrafoSize > ( Log2MinTrafoSize + 1 )  
 && ( PartMode = = PART\_Nx2N  
 | | PartMode = = PART\_nLx2N | | PartMode = = PART\_nRx2N ) )  
 | | ( log2TrafoSize = = ( Log2MinTrafoSize + 1 )  
 && log2TrafoWidth > log2TrafoHeight ) ) )  
{  
 interTbSplitDirectionL = 1  
}  
else   
{  
 interTbSplitDirectionL = 2  
}

## Chroma inter transform block split direction derivation process

Inputs to this process are

– the current luma transform width log2TrafoWidth,

– the current luma transform height log2TrafoHeight,

– the current depth of the residual quadtree trafoDepth.

Output of this process is the variable interTbSplitDirectionC.

When a chroma transform block is split into four blocks with smaller horizontal and/or vertical size for the purpose of transform coding, the variable interTbSplitDirectionC defines where the four blocks are located inside the parent block. interTbSplitDirectionC equal to 0 results in the four smaller blocks being vertically aligned, i.e. one column of blocks. interTbSplitDirectionC equal to 1 results in the four smaller blocks being horizontally aligned, i.e. one row of blocks. interTbSplitDirectionC equal to 2 results in two rows with two blocks being horizontally aligned, i.e. two rows and two columns of blocks. interTbSplitDirectionC is derived as follows.

– If ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoWidth is greater than log2TrafoHeight, interTbSplitDirectionC is set to 1.

– Otherwise, if ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoHeight is greater than log2TrafoWidth, interTbSplitDirectionC is set to 0.

– Otherwise, if ( log2TrafoWidth + log2TrafoHeight ) >> 1 is equal to 4 and log2TrafoHeight is equal to log2TrafoWidth, interTbSplitDirectionC is set to 2.

– Otherwise, the luma inter transform block split direction derivation process as specified in 6.6 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionC.

# 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 , 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  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 , 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 .

– 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.

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

– tu(v): truncated unary coded syntax element with left bit first. The parsing process for this descriptor is specified in subclause .

– 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 .

## 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) |
| **temporal\_id** | u(3) |
| **reserved\_one\_5bits** | u(5) |
| NumBytesInRBSP = 0 |  |
| for( i = 2; 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

#### Video parameter set RBSP syntax

|  |  |
| --- | --- |
| video\_parameter\_set\_rbsp( ) { | Descriptor |
| **vps\_max\_temporal\_layers\_minus1** | u(3) |
| **vps\_max\_layers\_minus1** | u(5) |
| **video\_parameter\_set\_id** | ue(v) |
| **vps\_temporal\_id\_nesting\_flag** | u(1) |
| for( i = 0; i <= vps\_max\_temporal\_layers\_minus1; i++ ) { |  |
| **vps\_max\_dec\_pic\_buffering[** i **]** | ue(v) |
| **vps\_num\_reorder\_pics[** i **]** | ue(v) |
| **vps\_max\_latency\_increase[** i **]** | ue(v) |
| } |  |
| **vps\_extension\_flag** | u(1) |
| if( vps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **vps\_extension\_data\_flag** | u(1) |
| } |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Sequence parameter set RBSP syntax

|  |  |
| --- | --- |
| seq\_parameter\_set\_rbsp( ) { | Descriptor |
| **profile\_space** | u(3) |
| **profile\_idc** | u(5) |
| **constraint\_flags** | u(16) |
| **level\_idc** | u(8) |
| for( i = 0; i < 32; i++ ) |  |
| **profile\_compatability\_flag[** i **]** | u(1) |
| **seq\_parameter\_set\_id** | ue(v) |
| **video\_parameter\_set\_id** | ue(v) |
| **chroma\_format\_idc** | ue(v) |
| if( chroma\_format\_idc = = 3 ) |  |
| **separate\_colour\_plane\_flag** | u(1) |
| **sps\_max\_temporal\_layers\_minus1** | u(3) |
| **pic\_width\_in\_luma\_samples** | ue(v) |
| **pic\_height\_in\_luma\_samples** | ue(v) |
| **pic\_cropping\_flag** | u(1) |
| if( pic\_cropping\_flag ) { |  |
| **pic\_crop\_left\_offset** | ue(v) |
| **pic\_crop\_right\_offset** | ue(v) |
| **pic\_crop\_top\_offset** | ue(v) |
| **pic\_crop\_bottom\_offset** | 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\_sample\_bit\_depth\_luma\_minus1** | u(4) |
| **pcm\_sample\_bit\_depth\_chroma\_minus1** | u(4) |
| } |  |
| **log2\_max\_pic\_order\_cnt\_lsb\_minus4** | ue(v) |
| for( i = 0; i <= sps\_max\_temporal\_layers\_minus1; i++ ) { |  |
| **sps\_max\_dec\_pic\_buffering[** i **]** | ue(v) |
| **sps\_num\_reorder\_pics[** i **]** | ue(v) |
| **sps\_max\_latency\_increase[** i **]** | ue(v) |
| } |  |
| **restricted\_ref\_pic\_lists\_flag** | u(1) |
| if( restricted\_ref\_pic\_lists\_flag ) |  |
| **lists\_modification\_present\_flag** | u(1) |
| **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** | u(1) |
| if( scaling\_list\_enable\_flag ) { |  |
| **sps\_scaling\_list\_data\_present\_flag** | u(1) |
| if( sps\_scaling\_list\_data\_present\_flag ) |  |
| scaling\_list\_param( ) |  |
| } |  |
| **chroma\_pred\_from\_luma\_enabled\_flag** | u(1) |
| **transform\_skip\_enabled\_flag** | u(1) |
| **seq\_loop\_filter\_across\_slices\_enabled\_flag** | u(1) |
| **asymmetric\_motion\_partitions\_enabled\_flag** | u(1) |
| **nsrqt\_enabled\_flag** | u(1) |
| **sample\_adaptive\_offset\_enabled\_flag** | u(1) |
| **adaptive\_loop\_filter\_enabled\_flag** | u(1) |
| if( adaptive\_loop\_filter\_enabled\_flag ) |  |
| **alf\_coef\_in\_slice\_flag** | u(1) |
| if( pcm\_enabled\_flag ) |  |
| **pcm\_loop\_filter\_disable\_flag** | u(1) |
| **sps\_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\_short\_term\_ref\_pic\_sets** | ue(v) |
| for( i = 0; i < num\_short\_term\_ref\_pic\_sets; i++) |  |
| short\_term\_ref\_pic\_set( i ) |  |
| **long\_term\_ref\_pics\_present\_flag** | u(1) |
| **sps\_temporal\_mvp\_enable\_flag** | u(1) |
| **vui\_parameters\_present\_flag** | u(1) |
| if( vui\_parameters\_present\_flag ) |  |
| vui\_parameters( ) |  |
| **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) |
| **sign\_data\_hiding\_flag** | u(1) |
| if( sign\_data\_hiding \_flag ) |  |
| **sign\_hiding\_threshold** | u(4) |
| **cabac\_init\_present\_flag** | 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) |
| **slice\_granularity** | u(2) |
| **diff\_cu\_qp\_delta\_depth** | ue(v) |
| **cb\_qp\_offset** | se(v) |
| **cr\_qp\_offset** | se(v) |
| **weighted\_pred\_flag** | u(1) |
| **weighted\_bipred\_idc** | u(2) |
| **output\_flag\_present\_flag** | u(1) |
| **transquant\_bypass\_enable\_flag** | u(1) |
| **dependent\_slice\_enabled\_flag** | u(1) |
| **tiles\_or\_entropy\_coding\_sync\_idc** | u(2) |
| if( tiles\_or\_entropy\_coding\_sync\_idc = = 1 ) { |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| **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) |
| } |  |
| **loop\_filter\_across\_tiles\_enabled\_flag** | u(1) |
| } else if( tiles\_or\_entropy\_coding\_sync\_idc = = 3 ) |  |
| **cabac\_independent\_flag** | u(1) |
| **deblocking\_filter\_control\_present\_flag** | u(1) |
| if( deblocking\_filter\_control\_present\_flag ) { |  |
| **deblocking\_filter\_override\_enabled\_flag** | u(1) |
| **pps\_disable\_deblocking\_filter\_flag** | u(1) |
| if( !pps\_disable\_deblocking\_filter\_flag ) { |  |
| **beta\_offset\_div2** | se(v) |
| **tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| **pps\_scaling\_list\_data\_present\_flag** | u(1) |
| if( pps\_scaling\_list\_data\_present\_flag ) |  |
| scaling\_list\_param( ) |  |
| **log2\_parallel\_merge\_level\_minus2** | ue(v) |
| **slice\_header\_extension\_present\_flag** | u(1) |
| **slice\_extension\_present\_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** |
| 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 ], sizeID , matrixID ) |  |
| } |  |
| } |  |

#### Scaling list syntax

|  |  |
| --- | --- |
| scaling\_list( scalingList, sizeID , matrixID ) { | Descriptor |
| nextCoef = 8 | u(1) |
| coefNum = Min( 64, ( 1 << ( 4 + ( sizeID << 1) ) ) ) |  |
| if( sizeID > 1 ) { |  |
| **scaling\_list\_dc\_coef\_minus8[**sizeID − 2 ][ matrixID ] | se(v) |
| nextCoef = scaling\_list\_dc\_coef\_minus8[ sizeID – 2 ][ matrixID ] + 8 |  |
| } |  |
| 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

|  |  |  |  |
| --- | --- | --- | --- |
| aps\_rbsp( ) { | Descriptor | | |
| **aps\_id** | ue(v) | | |
| **aps\_adaptive\_loop\_filter\_flag** | u(1) | | |
| if( aps\_adaptive\_loop\_filter\_flag ) |  | | |
| alf\_param( ) |  | | |
| **aps\_sub\_stream\_entry\_present\_flag** | | u(1) | |
| if (aps\_sub\_pic\_entry\_present\_flag) { | |  | |
| **num\_entry\_point\_offsets** | | | ue(v) |
| if( num\_entry\_point\_offsets > 0 ) { | | |  |
| **offset\_len\_minus1** | | | ue(v) |
| for( i = 0; i < num\_entry\_point\_offsets; i++ ) | | |  |
| **entry\_point\_offset**[ i ] | | | u(v) |
| } | |  | |
| **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 |
| **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( ) |  |
| if( slice\_extension\_present\_flag ) { |  |
| **slice\_extention\_flag** | u(1) |
| if( slice\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **slice\_extension\_data\_flag** | u(1) |
| } |  |
| 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) |
| } |  |

#### Byte alignment syntax

|  |  |
| --- | --- |
| byte\_alignment( ) { | Descriptor |
| **bit\_equal\_to\_one** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **bit\_equal\_to\_zero** /\* equal to 0 \*/ | f(1) |
| } |  |

[Ed. Note (YK): The byte\_alignment() syntax structure and the rbsp\_trailing\_bits() syntax structure are essentially identical.]

### Slice header syntax

|  |  |
| --- | --- |
| slice\_header( ) { | Descriptor |
| **first\_slice\_in\_pic\_flag** | u(1) |
| **pic\_parameter\_set\_id** | ue(v) |
| if( !first\_slice\_in\_pic\_flag ) |  |
| **slice\_address** | u(v) |
| if( dependent\_slice\_enabled\_flag && !first\_slice\_in\_pic\_flag ) |  |
| **dependent\_slice\_flag** | u(1) |
| if( !dependent\_slice\_flag ) { |  |
| **slice\_type** | ue(v) |
| if( output\_flag\_present\_flag ) |  |
| **pic\_output\_flag** | u(1) |
| if( **separate\_colour\_plane\_flag** = = 1 ) |  |
| **colour\_plane\_id** | u(2) |
| if( RapPicFlag ) { |  |
| **rap\_pic\_id** | ue(v) |
| **no\_output\_of\_prior\_pics\_flag** | u(1) |
| } |  |
| if( !IdrPicFlag ) { |  |
| **pic\_order\_cnt\_lsb** | u(v) |
| **short\_term\_ref\_pic\_set\_sps\_flag** | u(1) |
| if( !short\_term\_ref\_pic\_set\_sps\_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++ ) { |  |
| **poc\_lsb\_lt**[ i ] | u(v) |
| **delta\_poc\_msb\_present\_flag**[ i ] | u(1) |
| if( delta\_poc\_msb\_present\_flag[ i ] ) |  |
| **delta\_poc\_msb\_cycle\_lt**[ i ] | ue(v) |
| **used\_by\_curr\_pic\_lt\_flag**[ i ] | u(1) |
| } |  |
| } |  |
| } |  |
| if( sample\_adaptive\_offset\_enabled\_flag ) { |  |
| **slice\_sample\_adaptive\_offset\_flag[** 0 **]** | u(1) |
| if( slice\_sample\_adaptive\_offset\_flag[ 0 ] ) { |  |
| **slice\_sample\_adaptive\_offset\_flag[** 1 **]** | u(1) |
| **slice\_sample\_adaptive\_offset\_flag[** 2 **]** | u(1) |
| } |  |
| } |  |
| if(adaptive\_loop\_filter\_enabled\_flag ) |  |
| **aps\_id** | ue(v) |
| if( slice\_type = = P | | slice\_type = = B ) { |  |
| if( sps\_temporal\_mvp\_enable\_flag ) |  |
| **pic\_temporal\_mvp\_enable\_flag** | u(1) |
| **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) |
| } |  |
| } |  |
| if( lists\_modification\_present\_flag ) |  |
| ref\_pic\_list\_modification( ) |  |
| if( slice\_type = = B ) |  |
| **mvd\_l1\_zero\_flag** | u(1) |
| if( cabac\_init\_present\_flag && slice\_type != I ) |  |
| **cabac\_init\_flag** | u(1) |
| **slice\_qp\_delta** | se(v) |
| if( deblocking\_filter\_control\_present\_flag ) { |  |
| if( deblocking\_filter\_override\_enabled\_flag ) |  |
| **deblocking\_filter\_override\_flag** | u(1) |
| if( deblocking\_filter\_override\_flag ) { |  |
| **slice\_header\_disable\_deblocking\_filter\_flag** | u(1) |
| if( !slice\_header\_disable\_deblocking\_filter\_flag ) { |  |
| **beta\_offset\_div2** | se(v) |
| **tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| } |  |
| if( pic\_temporal\_mvp\_enable\_flag ) { |  |
| if( slice\_type = = B ) |  |
| **collocated\_from\_l0\_flag** | u(1) |
| if( slice\_type != I &&   ((collocated\_from\_l0\_flag && num\_ref\_idx\_l0\_active\_minus1 > 0) | |  (!collocated\_from\_l0\_flag && num\_ref\_idx\_l1\_active\_minus1 > 0) ) |  |
| **collocated\_ref\_idx** | ue(v) |
| } |  |
| if( ( weighted\_pred\_flag && slice\_type = = P) | |  ( weighted\_bipred\_idc = = 1 && slice\_type = = B ) ) |  |
| pred\_weight\_table( ) |  |
| if( slice\_type = = P | | slice\_type = = B ) |  |
| **five\_minus\_max\_num\_merge\_cand** | ue(v) |
| if( adaptive\_loop\_filter\_enabled\_flag ) { |  |
| **slice\_adaptive\_loop\_filter\_flag** | u(1) |
| if( slice\_adaptive\_loop\_filter\_flag && alf\_coef\_in\_slice\_flag ) |  |
| alf\_param( ) |  |
| if( slice\_adaptive\_loop\_filter\_flag && !alf\_coef\_in\_slice\_flag ) |  |
| alf\_cu\_control\_param( ) |  |
| } |  |
| if( seq\_loop\_filter\_across\_slices\_enabled\_flag &&  ( slice\_adaptive\_loop\_filter\_flag | | slice\_sample\_adaptive\_offset\_flag | |  !disable\_deblocking\_filter\_flag ) ) |  |
| **slice\_loop\_filter\_across\_slices\_enabled\_flag** | u(1) |
| } |  |
| if( tiles\_or\_entropy\_coding\_sync\_idc = = 1 | |  tiles\_or\_entropy\_coding\_sync\_idc = = 2 ) { |  |
| **num\_entry\_point\_offsets** | ue(v) |
| if( num\_entry\_point\_offsets > 0 ) { |  |
| **offset\_len\_minus1** | ue(v) |
| for( i = 0; i < num\_entry\_point\_offsets; i++ ) |  |
| **entry\_point\_offset**[ i ] | u(v) |
| } |  |
| } |  |
| if( slice\_header\_extension\_present\_flag ) { |  |
| **slice\_header\_extension\_length** | ue(v) |
| for( i = 0; i < slice\_header\_extension\_length; i++) |  |
| **slice\_header\_extension\_data\_byte** | u(8) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### 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++ ) { |  |
| **used\_by\_curr\_pic\_flag**[ j ] | u(1) |
| if( !**used\_by\_curr\_pic\_flag**[ j ]) |  |
| **use\_delta\_flag**[ 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 && NumPocTotalCurr > 1 ) |  |
| for( i = 0; i <= num\_ref\_idx\_l0\_active\_minus1; i++ ) |  |
| **list\_entry\_l0**[ i ] | u(v) |
| } |  |
| if( slice\_type = = B ) { |  |
| **ref\_pic\_list\_modification\_flag\_l1** | u(1) |
| if( ref\_pic\_list\_modification\_flag\_l1 && NumPocTotalCurr > 1 ) |  |
| for( i = 0; i <= num\_ref\_idx\_l1\_active\_minus1; i++ ) |  |
| **list\_entry\_l1**[ i ] | u(v) |
| } |  |
| } |  |

#### Adaptive loop filter parameter syntax

|  |  |
| --- | --- |
| alf\_param() { | Descriptor |
| **alf\_cb\_enable\_flag** | u(1) |
| **alf\_cr\_enable\_flag** | u(1) |
| **alf\_one\_luma\_unit\_per\_slice\_flag** |  |
| if( alf\_cb\_enable\_flag ) |  |
| **alf\_one\_cb\_unit\_per\_slice\_flag** | u(1) |
| if( alf\_cr\_enable\_flag ) |  |
| **alf\_one\_cr\_unit\_per\_slice\_flag** |  |
| if( !alf\_coef\_in\_slice\_flag ) { |  |
| **alf\_num\_lcu\_in\_width\_minus1** | ue(v) |
| **alf\_num\_lcu\_in\_height\_minus1** | ue(v) |
| } else |  |
| **alf\_num\_lcu\_in\_slice\_minus1** | u(v) |
| endCtbrY = (numCtb − 1 + firstCtbAddr) / numCtbInWidth |  |
| endCtbrX = (numCtb − 1 + firstCtbAddr) % numCtbInWidth |  |
| for( i = 0; i < numCtb; i++ ) { |  |
| rx = ( i + firstCtbAddr) % numCtbInWidth |  |
| ry = ( i + firstCtbAddr) / numCtbInWidth |  |
| endrX = ( ry = = endCtbrY ) ? ( endCtbrX ) : ( numCtbInWidth − 1 ) |  |
| if( ( rx = = 0 ) && ( i − numCtbInWidth >= 0 ) && ( alf\_one\_luma\_unit\_per\_slice\_flag = =0 ) ) |  |
| **alf\_repeat\_row \_flag**[0] | u(1) |
| alf\_unit( rx, ry, 0, i, endrX, alf\_one\_luma\_unit\_per\_slice\_flag ) |  |
| if( alf\_cb\_enable\_flag ) { |  |
| if( ( rx = = 0 ) && ( i − numCtbInWidth >= 0 ) && ( alf\_one\_cb\_unit\_per\_slice\_flag = =0 ) ) |  |
| **alf\_repeat\_row \_flag**[1] | u(1) |
| alf\_unit(rx, ry, 1, i, endrX, alf\_one\_cb\_unit\_per\_slice\_flag ) |  |
| } |  |
| if( alf\_cr\_enable\_flag ) { |  |
| if( ( rx = = 0) && ( i − numCtbInWidth>= 0) && ( alf\_one\_cr\_unit\_per\_slice\_flag = =0 ) ) |  |
| **alf\_repeat\_row \_flag**[2] | u(1) |
| alf\_unit(rx, ry, 2, i, endrX, alf\_one\_cr\_unit\_per\_slice\_flag ) |  |
| } |  |
| } |  |
| } |  |

#### Adaptive loop filter unit syntax

|  |  |
| --- | --- |
| alf\_unit(rx, ry, cIdx, lcuIdx, endrX, oneUnitFlag ) { | Descriptor |
| if( oneUnitFlag ) { |  |
| if( lcuIdx = = 0 ) { |  |
| **alf\_lcu\_enable\_flag**[ cIdx ][ ry ][ rx ] | u(1) |
| if( alf\_lcu\_enable\_flag[ cIdx ][ ry ][ rx ] ) |  |
| alf\_info( rx, ry, cIdx ) |  |
| } |  |
| } else { |  |
| if( !alf\_repeat\_row \_flag[ cIdx ] ) { |  |
| if( rx = = 0 || alfRun[ cIdx ][ ry ][ rx ] < 0 ) |  |
| if( lcuIdx –numCtbInWidth < 0 ) |  |
| alfRun[ cIdx ][ ry ][ rx ] = 0 + **alf\_run\_diff** | u(v) |
| else |  |
| alfRun[ cIdx ][ ry ][ rx ] = alfRun[ cIdx ][ ry − 1 ][ rx ]+ **alf\_run\_diff** | s(v) |
| alfRun[ cIdx ][ ry ][ rx+ 1 ] = alfRun[ cIdx ][ ry ][ rx ] − 1 |  |
| if( rx == 0 || alfRun [ cIdx ] [ rx ][ ry ] < 0) { |  |
| if( ry > 0 && ( lcuIdx − numCtbInWidth > = 0 | | alfAcrossSlice) ) |  |
| **alf\_merge\_up\_flag** | u(1) |
| if( !alf\_merge\_up\_flag ) { |  |
| **alf\_lcu\_enable\_flag**[ cIdx ][ ry ][ rx ] | u(1) |
| if( alf\_lcu\_enable\_flag[ cIdx ][ ry ][ rx ] ) |  |
| alf\_info( rx, ry, cIdx ) |  |
| } |  |
| } |  |
| } else |  |
| alfRun[ cIdx ][ ry ][ rx ] = alfRun[ cIdx ][ ry − 1 ][ rx ] |  |
| } |  |
| } |  |

#### Adaptive loop filter info syntax

|  |  |
| --- | --- |
| alf\_info( rx, ry, cIdx ) { | **Descriptor** |
| if(NumALFFiltersInStoredBuffer[cIdx] > 0) |  |
| **alf\_new\_filter\_set\_flag** | u(1) |
| if( alf\_new\_filter\_set\_flag = = 0 && NumALFFiltersInStoredBuffer[cIdx] > 0) |  |
| **alf\_stored\_filter\_set\_idx**[ cIdx ] | u(v) |
| else { |  |
| if( cIdx = = 0 ) { |  |
| **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 < 15; i++ ) |  |
| **alf\_filter\_pattern\_flag[ cIdx ][ ry ][ rx ][ i ]** | u(1) |
| if( alf\_no\_filters\_minus1 > 0 ) |  |
| **alf\_pred\_flag[ cIdx ][ ry ][ rx ]** | u(1) |
| for( i = 0; i < AlfNumFilters; i++ ) |  |
| **alf\_nb\_pred\_luma\_flag[ cIdx ][ ry ][ rx ][ i ]** | u(1) |
| if( AlfNumFilters > 1 ) { |  |
| **alf\_min\_kstart\_minus1** | ue(v) |
| for( i = 1; i < 4; i++ ) |  |
| **alf\_golomb\_index\_flag[ i ]** | u(1) |
| } |  |
| for( i = 0; i < AlfNumFilters; i++ ) |  |
| for( j = 0; j < AlfCodedLength; j++ ) |  |
| **alf\_filt\_coeff[ cIdx ][ ry ][ rx ][ i ][ j ]** | ge(v) |
| } else |  |
| for( j = 0; j < AlfCodedLength; j++ ) |  |
| **alf\_filt\_coeff[ cIdx ][ ry ][ rx ][ 0 ][ j ]** | 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 ) |  |
| 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 ) |  |
| 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) |
| } |  |
| } |  |
| } |  |
| } |  |

### Slice data syntax

|  |  |
| --- | --- |
| slice\_data( ) { | Descriptor |
| CtbAddrRS = SliceCtbAddrRS |  |
| CtbAddrTS = CtbAddrRStoTS[ CtbAddrRS ] |  |
| moreDataFlag = 1 |  |
| if( adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag ) |  |
| AlfCuFlagIdx = −1 |  |
| do { |  |
| xCtb = InverseRasterScan( CtbAddrRS, CtbSize, CtbSize, pic\_width\_in\_luma\_samples, 0 ) |  |
| yCtb = InverseRasterScan( CtbAddrRS, CtbSize, CtbSize, pic\_width\_in\_luma\_samples, 1 ) |  |
| NumPCMBlock = 0 |  |
| CtbAddrInSlice = CtbAddrRS − ( slice\_address >> SliceGranularity ) |  |
| for( cIdx = 0; cIdx < 3; cIdx++ ) { |  |
| if( slice\_sample\_adaptive\_offset\_flag[ cIdx ] ) |  |
| sao\_param( xCtb, yCtb, cIdx ) |  |
| } |  |
| moreDataFlag = coding\_tree( xCtb, yCtb, Log2CtbSize, 0 ) |  |
| CtbAddrTS++ |  |
| CtbAddrRS = CtbAddrTStoRS[ CtbAddrTS ] |  |
| if( moreDataFlag && ( ( tiles\_or\_entropy\_coding\_sync\_idc = = 1 &&  TileId[ CtbAddrTS ] != TileId[ CtbAddrTS − 1 ] ) | |  ( tiles\_or\_entropy\_coding\_sync\_idc = = 2 && CtbAddrTS % PicWidthInCtbs = = 0 ) ) ) |  |
| rbsp\_alignment( ) |  |
| } while( moreDataFlag ) |  |
| } |  |

#### Sample adaptive offset parameter syntax

|  |  |
| --- | --- |
| sao\_param( rx, ry, cIdx ){ | Descriptor |
| if( rx > 0 ) { |  |
| leftCtbInSlice = CtbAddrInSlice > 0 |  |
| leftCtbInTile = TileId[ CtbAddrTS ] = = TileId[ CtbAddrRStoTS[ CtbAddrRS − 1 ] ] |  |
| if( leftCtbInSlice && leftCtbInTile ) |  |
| **sao\_merge\_left\_flag** | ae(v) |
| } |  |
| if( ry > 0 && !sao\_merge\_left\_flag ) { |  |
| upCtbInSlice = (CtbAddrTS – CtbAddrRStoTS[CtbAddrRS − PicWidthInCtbs]) <= CtbAddrInSlice |  |
| upCtbInTile = TileId[ CtbAddrTS ] = = TileId[ CtbAddrRStoTS[ CtbAddrRS − PicWidthInCtbs ] ] |  |
| if( upCtbInSlice && upCtbInTile ) |  |
| **sao\_merge\_up\_flag** | ae(v) |
| } |  |
| if( !sao\_merge\_up\_flag && !sao\_merge\_left\_flag ) { |  |
| **sao\_type\_idx**[ cIdx ][ rx ][ ry ] | ae(v) |
| if( sao\_type\_idx[ cIdx ][ rx ][ ry ] = =5 ) |  |
| **sao\_band\_position**[ cIdx ][ rx ][ ry ] | ae(v) |
| if( sao\_type\_idx[ cIdx ][ rx ][ ry ] != 0 ) |  |
| for( i = 0; i < 4; i++ ) |  |
| **sao\_offset\_abs**[ cIdx ][ rx][ ry ][ i ] | ae(v) |
| if( sao\_type\_idx[ cIdx ][ rx ][ ry ] = = 5 ) { |  |
| for( i = 0; i < 4; i++ ) { |  |
| if( sao\_offset\_abs[ cIdx ][ rx ][ ry ] != 0 ) |  |
| **sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

### Coding tree syntax

|  |  |
| --- | --- |
| coding\_tree( x0, y0, log2CbSize, ctDepth ) { | Descriptor |
| if( x0 + ( 1 << log2CbSize ) <= pic\_width\_in\_luma\_samples &&  y0 + ( 1 << log2CbSize ) <= pic\_height\_in\_luma\_samples &&  MinCbAddrZS[ x0 >> Log2MinCbSize ][ y0 >> Log2MinCbSize ] >=   SliceCbAddrZS &&  log2CbSize > Log2MinCbSize && NumPCMBlock = = 0 ) |  |
| **split\_coding\_unit\_flag[** x0 **][** y0 **]** | ae(v) |
| if( adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag &&  ctDepth <= alf\_cu\_control\_max\_depth &&  ( ctDepth = = alf\_cu\_control\_max\_depth | |  split\_coding\_unit\_flag[ x0 ][ y0 ] = = 0 ) ) { |  |
| AlfCuFlagIdx++ |  |
| if( ( diff\_cu\_qp\_delta\_depth > 0 ) && log2CbSize >= Log2MinCUDQPSize ) |  |
| IsCuQpDeltaCoded = 0 |  |
| if( split\_coding\_unit\_flag[ x0 ][ y0 ] ) { |  |
| x1 = x0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| y1 = y0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| if( MinCbAddrZS[ x1 >> Log2MinCbSize ][ y0 >> Log2MinCbSize ] >   SliceCbAddrZS ) |  |
| moreDataFlag = coding\_tree( x0, y0, log2CbSize − 1, ctDepth + 1 ) |  |
| if( MinCbAddrZS[ x0 >> Log2MinCbSize ][ y1 >> Log2MinCbSize ] >   SliceCbAddrZS && moreDataFlag &&  x1 < pic\_width\_in\_luma\_samples ) |  |
| moreDataFlag = coding\_tree( x1, y0, log2CbSize − 1, ctDepth + 1 ) |  |
| if( MinCbAddrZS[ x1 >> Log2MinCbSize ][ y1 >> Log2MinCbSize ] >   SliceCbAddrZS && moreDataFlag &&  y1 < pic\_height\_in\_luma\_samples ) |  |
| moreDataFlag = coding\_tree( x0, y1, log2CbSize − 1, ctDepth + 1 ) |  |
| if( moreDataFlag &&   x1 < pic\_width\_in\_luma\_samples &&   y1 < pic\_height\_in\_luma\_samples) |  |
| moreDataFlag = coding\_tree( x1, y1, log2CbSize − 1, ctDepth + 1 ) |  |
| } else { |  |
| if(adaptive\_loop\_filter\_flag && alf\_cu\_control\_flag ) |  |
| AlfCuFlag[ x0 ][ y0 ] = alf\_cu\_flag[ AlfCuFlagIdx ] |  |
| if( NumPCMBlock = = 0 ) |  |
| coding\_unit( x0, y0, log2CbSize ) |  |
| else |  |
| pcm\_sample( x0, y0, log2CbSize ) |  |
| if( !( ( x0 + 1 << log2CbSize ) % ( 1 << SliceGranularity ) ) &&  !( ( y0 + 1 << log2CbSize ) % ( 1 << SliceGranularity ) ) &&  NumPCMBlock = = 0 ) { |  |
| **end\_of\_slice\_flag** | ae(v) |
| moreDataFlag = !end\_of\_slice\_flag |  |
| } else |  |
| moreDataFlag = 1 |  |
| } |  |
| return moreDataFlag |  |
| } |  |

### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, log2CbSize ) { | Descriptor |
| CurrCbAddrTS = MinCbAddrZS[ x0 >> Log2MinCbSize ][ y0 >> Log2MinCbSize ] |  |
| if( transquant\_bypass\_enable\_flag ) { |  |
| **cu\_transquant\_bypass\_flag** | ae(v) |
| } |  |
| if( slice\_type != I ) |  |
| **skip\_flag[** x0 **][** y0 **]** | ae(v) |
| if( skip\_flag[ x0 ][ y0 ] ) |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| else { |  |
| if( slice\_type != I ) |  |
| **pred\_mode\_flag** | ae(v) |
| if( PredMode != MODE\_INTRA | | log2CbSize = = Log2MinCbSize ) |  |
| **part\_mode** | ae(v) |
| x1 = x0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| y1 = y0 + ( ( 1 << log2CbSize ) >> 1 ) |  |
| x2 = x1 − ( ( 1 << log2CbSize ) >> 2 ) |  |
| y2 = y1 − ( ( 1 << log2CbSize ) >> 2 ) |  |
| x3 = x1 + ( ( 1 << log2CbSize ) >> 2 ) |  |
| y3 = y1 + ( ( 1 << log2CbSize ) >> 2 ) |  |
| if( PartMode = = PART\_2Nx2N ) |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| else if( PartMode = = PART\_2NxN ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y1 , log2CbSize ) |  |
| } else if( PartMode = = PART\_Nx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x1, y0 , log2CbSize ) |  |
| } else if( PartMode = = PART\_2NxnU ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y2 , log2CbSize ) |  |
| } else if( PartMode = = PART\_2NxnD ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y3 , log2CbSize ) |  |
| } else if( PartMode = = PART\_nLx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x2, y0 , log2CbSize ) |  |
| } else if( PartMode = = PART\_nRx2N ) { |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x3, y0 , log2CbSize ) |  |
| } else { /\* PART\_NxN \*/ |  |
| prediction\_unit( x0, y0 , log2CbSize ) |  |
| prediction\_unit( x1, y0 , log2CbSize ) |  |
| prediction\_unit( x0, y1 , log2CbSize ) |  |
| prediction\_unit( x1, y1 , log2CbSize ) |  |
| } |  |
| if( !pcm\_flag ) { |  |
| if( PredMode != MODE\_INTRA &&   !(PartMode = = PART\_2Nx2N && merge\_flag[x0][y0]) ) |  |
| **no\_residual\_data\_flag** | ae(v) |
| if( !no\_residual\_data\_flag ) { |  |
| MaxTrafoDepth = ( PredMode = = MODE\_INTRA ?   max\_transform\_hierarchy\_depth\_intra + IntraSplitFlag :   max\_transform\_hierarchy\_depth\_inter ) |  |
| transform\_tree( x0, y0, x0, y0, x0, y0, log2CbSize, log2CbSize, log2CbSize, 0, 0 ) |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Prediction unit syntax

|  |  |
| --- | --- |
| prediction\_unit( x0, y0, log2CbSize ) { | 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 &&  log2CbSize >= Log2MinIPCMCUSize &&  log2CbSize <= Log2MaxIPCMCUSize ) |  |
| **pcm\_flag** | ae(v) |
| if( pcm\_flag ) { |  |
| **num\_subsequent\_pcm** | tu(3) |
| NumPCMBlock = num\_subsequent\_pcm + 1 |  |
| while( !byte\_aligned( ) ) |  |
| **pcm\_alignment\_zero\_bit** | f(1) |
| pcm\_sample( x0, y0, log2CbSize ) |  |
| } else { |  |
| **prev\_intra\_luma\_pred\_flag[** x0 **][** y0 **]** | ae(v) |
| if( prev\_intra\_luma\_pred\_flag[ x0 ][ y0 ] ) |  |
| **mpm\_idx[** x0 **][** y0 **]** | ae(v) |
| else |  |
| **rem\_intra\_luma\_pred\_mode[** x0 **][**y0 **]** | ae(v) |
| **intra\_chroma\_pred\_mode**[ x0 ][ y0 ] | ae(v) |
| SignalledAsChromaDC =   ( 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\_idc[** x0 **][** y0 **]** | ae(v) |
| if( inter\_pred\_idc[ x0 ][ y0 ] != Pred\_L1 ) { |  |
| 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\_idc[ x0 ][ y0 ] != Pred\_L0 ) { |  |
| if( num\_ref\_idx\_l1\_active\_minus1 > 0 ) |  |
| **ref\_idx\_l1[** x0 **][** y0 **]** | ae(v) |
| if( mvd\_l1\_zero\_flag &&  inter\_pred\_idc[ x0 ][ y0 ] = = Pred\_BI) { |  |
| mvd\_l1[ x0 ][ y0 ][ 0 ] = 0 |  |
| mvd\_l1[ x0 ][ y0 ][ 1 ] = 0 |  |
| } else |  |
| 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 ] ) |  |
| } |  |

#### PCM sample syntax

|  |  |
| --- | --- |
| pcm\_sample( x0, y0, log2CbSize ) { | **Descriptor** |
| for( i = 0; i < 1 << ( log2CbSize << 1 ); i++ ) |  |
| **pcm\_sample\_luma**[ i ] | u(v) |
| for( i = 0; i < ( 1 << ( log2CbSize << 1 ) ) >> 1; i++ ) |  |
| **pcm\_sample\_chroma**[ i ] | u(v) |
| NumPCMBlock− − |  |
| } |  |

### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0L, y0L,  x0C, y0C, xBase, yBase, log2CbSize, log2TrafoWidth, log2TrafoHeight,  trafoDepth, blkIdx ) { | Descriptor |
| log2TrafoSize = ( log2TrafoWidth + log2TrafoHeight ) >> 1 |  |
| if( log2TrafoSize <= Log2MaxTrafoSize &&   log2TrafoSize > Log2MinTrafoSize &&  trafoDepth < MaxTrafoDepth && !(IntraSplitFlag && trafoDepth = = 0) ) |  |
| **split\_transform\_flag**[ x0L ][ y0L ][ trafoDepth ] | ae(v) |
| if( trafoDepth = = 0 | | log2TrafoSize > 2 ) { |  |
| if( trafoDepth = = 0 | | cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1 ] ) |  |
| **cbf\_cb**[ x0C ][ y0C ][ trafoDepth ] | ae(v) |
| if( trafoDepth = = 0 | | cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1 ] ) |  |
| **cbf\_cr**[ x0C ][ y0C ][ trafoDepth ] | ae(v) |
| } |  |
| if( split\_transform\_flag[ x0L ][ y0L ][ trafoDepth ] ) { |  |
| if( interTbSplitDirectionL = = 2 ) { |  |
| x1L = x0L + ( ( 1 << log2TrafoWidth ) >> 1 ) |  |
| y1L = y0L |  |
| x2L = x0L |  |
| y2L = y0L + ( ( 1 << log2TrafoHeight ) >> 1 ) |  |
| x3L = x1L |  |
| y3L = y2L |  |
| } else { |  |
| x1L = x0L + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionL |  |
| y1L = y0L + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionL ) |  |
| x2L = x1L + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionL |  |
| y2L = y1L + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionL ) |  |
| x3L = x2L + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionL |  |
| y3L = y2L + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionL ) |  |
| } |  |
| if( interTbSplitDirectionC = = 2 && log2TrafoSize > 3 ) { |  |
| x1C = x0C + ( ( 1 << log2TrafoWidth ) >> 1 ) |  |
| y1C = y0C |  |
| x2C = x0C |  |
| y2C = y0C + ( ( 1 << log2TrafoHeight ) >> 1 ) |  |
| xC3 = xC1 |  |
| yC3 = yC2 |  |
| } else if ( log2TrafoSize > 3 ) { |  |
| x1C = x0C + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionC |  |
| y1C = y0C + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionC ) |  |
| x2C = x1C + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionC |  |
| y2C = y1C + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionC ) |  |
| x3C = x2C + ( ( 1 << log2TrafoWidth ) >> 2 ) \* interTbSplitDirectionC |  |
| y3C = y2C + ( ( 1 << log2TrafoHeight) >> 2 ) \* ( 1 − interTbSplitDirectionC ) |  |
| } else { |  |
| x1C = x0C |  |
| y1C = y0C |  |
| x2C = x0C |  |
| y2C = y0C |  |
| x3C = x0C |  |
| y3C = y0C |  |
| } |  |
| if( interTbSplitDirectionL ! = 2 ) { |  |
| log2TrafoWidth = log2TrafoWidth − 2 \* interTbSplitDirectionL + 1 |  |
| log2TrafoHeight = log2TrafoHeight + 2 \* interTbSplitDirectionL − 1 |  |
| } |  |
| transform\_tree( x0L, y0L, x0C, y0C, x0L, y0L, log2CbSize,   log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 0 ) |  |
| transform\_tree( x1L, y1L, x1C, y1C, x0L, y0L, log2CbSize,   log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 1 ) |  |
| transform\_tree( x2L, y2L, x2C, y2C, x0L, y0L, log2CbSize,   log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 2 ) |  |
| transform\_tree( x3L, y3L, x3C, y3C, x0L, y0L, log2CbSize,   log2TrafoWidth − 1, log2TrafoHeight − 1, trafoDepth + 1, 3 ) |  |
| } else { |  |
| if( PredMode = = MODE\_INTRA | | trafoDepth != 0 | |  cbf\_cb[ x0 ][ y0 ][ trafoDepth ] | | cbf\_cr[ x0 ][ y0 ][ trafoDepth ] ) |  |
| **cbf\_luma**[ x0L ][ y0L ][ trafoDepth ] | ae(v) |
| transform\_unit (x0L, y0L, x0C, y0C, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx) |  |
| } |  |
| } |  |

### Transform unit syntax

|  |  |
| --- | --- |
| transform\_unit( x0L, y0L, x0C, y0C, log2TrafoWidth, log2TrafoHeight, trafoDepth, blkIdx ) { | Descriptor |
| if( cbf\_luma[ x0L ][ y0L ][ trafoDepth ] | | cbf\_cb[ x0C ][ y0C ][ trafoDepth ] | |  cbf\_cr[ x0C ][ y0C ][ trafoDepth ] { |  |
| if( ( diff\_cu\_qp\_delta\_depth > 0 ) && !IsCuQpDeltaCoded ) { |  |
| **cu\_qp\_delta** | ae(v) |
| IsCuQpDeltaCoded = 1 |  |
| } |  |
| log2TrafoSize = ( ( log2TrafoWidth + log2TrafoHeight ) >> 1 ) |  |
| if( PredMode = = MODE\_INTRA ) { |  |
| scanIdx = ScanType[ log2TrafoSize − 2 ][ IntraPredMode ] |  |
| scanIdxC = ScanType[ log2TrafoSize − 2 ][ IntraPredModeC ] |  |
| } else { |  |
| scanIdx = 0 |  |
| scanIdxC = 0 |  |
| } |  |
| if( cbf\_luma[ x0L ][ y0L ][ trafoDepth ] ) |  |
| residual\_coding( x0L, y0L, log2TrafoWidth, log2TrafoHeight, scanIdx, 0 ) |  |
| if( log2TrafoSize > 2 ) { |  |
| if( cbf\_cb[ x0C ][ y0C ][ trafoDepth ] ) |  |
| residual\_coding( x0C, y0C, log2TrafoWidth − 1, log2TrafoHeight − 1, scanIdxC, 1 ) |  |
| if( cbf\_cr[ x0C ][ y0C ][ trafoDepth ] ) |  |
| residual\_coding( x0C, y0C, log2TrafoWidth − 1, log2TrafoHeight − 1, scanIdxC, 2 ) |  |
| } else if( blkIdx = = 3 ) { |  |
| if( cbf\_cb[ x0C ][ y0C ][ trafoDepth ] ) |  |
| residual\_coding( x0C, y0C, log2TrafoWidth, log2TrafoHeight, scanIdxC, 1 ) |  |
| if( cbf\_cr[ x0C ][ y0C ][ trafoDepth ] ) |  |
| residual\_coding( x0C, y0C, log2TrafoWidth, log2TrafoHeight, scanIdxC, 2 ) |  |
| } |  |
| } |  |
| } |  |

### Residual coding syntax

|  |  |
| --- | --- |
| residual\_coding( x0, y0, log2TrafoWidth, log2TrafoHeight, scanIdx, cIdx ) { | Descriptor |
| if( log2TrafoWidth = = 1 | | log2TrafoHeight = = 1 ) { |  |
| log2TrafoWidth = 2 |  |
| log2TrafoHeight = 2 |  |
| } |  |
| If( transform\_skip\_enabled\_flag && !cu\_ transquant\_bypass \_flag &&  (PredMode = = MODE\_INTRA) &&   ( log2TrafoWidth = = 2) && (log2TrafoHeight = = 2) ) |  |
| **transform\_skip\_flag**[ x0 ][ y0 ][ cIdx ] | ae(v) |
| **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( scanIdx = = 1 && log2TrafoWidth = = 3 && log2TrafoHeight = = 3 ) { |  |
| xCG = 0 |  |
| yCG = i |  |
| } else if( scanIdx = = 2 && log2TrafoWidth = = 3 && log2TrafoHeight = = 3 ) { |  |
| xCG = i |  |
| yCG = 0 |  |
| } else { |  |
| xCG = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ i << 4 ][ 0 ] >> 2 |  |
| yCG = ScanOrder[ log2TrafoWidth ][ log2TrafoHeight ][ scanIdx ][ i << 4 ][ 1 ] >> 2 |  |
| } |  |
| implicitNonZeroCoeff = 0 |  |
| if( (i < numLastSubset) && (i > 0) ) { |  |
| **significant\_coeff\_group\_flag**[ xCG ][ yCG ] | ae(v) |
| implicitNonZeroCoeff = 1 |  |
| } |  |
| 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 ] &&   ( n > 0 | | implicitNonZeroCoeff = = 0 ) ) { |  |
| **significant\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| if( significant\_coeff\_flag[ xC ][ yC ] = = 1 ) |  |
| implicitNonZeroCoeff = 0 |  |
| } |  |
| } |  |
| firstNZPosInCG = 16 |  |
| lastNZPosInCG = −1 |  |
| numSigCoeff = 0 |  |
| firstGreater1CoeffIdx = −1 |  |
| 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 ] ) { |  |
| if( numSigCoeff < 8 ) { |  |
| **coeff\_abs\_level\_greater1\_flag[** n **]** | ae(v) |
| numSigCoeff++ |  |
| if( coeff\_abs\_level\_greater1\_flag[ n ] && firstGreater1CoeffIdx = = −1 ) |  |
| firstGreater1CoeffIdx = n |  |
| } |  |
| if( lastNZPosInCG = = −1) |  |
| lastNZPosInCG = n |  |
| firstNZPosInCG = n |  |
| } |  |
| } |  |
| signHidden = ( lastNZPosInCG – firstNZPosInCG >= sign\_hiding\_threshold &&   !cu\_ transquant\_bypass \_flag) ? 1 : 0 |  |
| if( firstGreater1CoeffIdx != −1 ) |  |
| **coeff\_abs\_level\_greater2\_flag[** firstGreater1CoeffIdx**]** | 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 ] &&  (!sign\_data\_hiding\_flag | | !signHidden | | n != firstNZPosInCG) ) |  |
| **coeff\_sign\_flag[** n **]** | ae(v) |
| } |  |
| numSigCoeff = 0 |  |
| sumAbs = 0 |  |
| 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 ] ) { |  |
| baseLevel = 1 + coeff\_abs\_level\_greater1\_flag[ n ] + coeff\_abs\_level\_greater2\_flag[ n ] |  |
| if( baseLevel = = ( ( numSigCoeff < 8 ) ? ( (n = = firstGreater1CoeffIdx) ? 3 : 2 ) : 1 ) ) |  |
| **coeff\_abs\_level\_remaining[** n **]** | ae(v) |
| transCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   ( coeff\_abs\_level\_remaining[ n ] + baseLevel ) \* ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| if( sign\_data\_hiding\_flag && signHidden ) { |  |
| sumAbs += ( coeff\_abs\_level\_remaining[ n ] + baseLevel ) |  |
| if( n = = firstNZPosInCG && (sumAbs%2 = = 1) ) |  |
| transCoeffLevel[x0][y0][cIdx][xC][yC] = −  transCoeffLevel[x0][y0][cIdx][xC][yC] |  |
| } |  |
| numSigCoeff++ |  |
| } 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  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.

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

nal\_ref\_flag shall be equal to 1 for NAL units with nal\_unit\_type equal to 4, 5, 6, 7, or 8.

nal\_ref\_flag shall be equal to 0 for all NAL units having nal\_unit\_type equal to 29, 30, or 31.

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

NAL units that use nal\_unit\_type equal to 0 or in the range of 48..63, inclusive, shall not affect the decoding process specified in this Recommendation | International Standard.

NOTE 2 – NAL unit types 0 and 48..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 48..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 48 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 48 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-RAP, non-TFD and non-TLA picture slice\_layer\_rbsp( ) | VCL |
| 2 | Coded slice of a TFD picture  slice\_layer\_rbsp( ) | VCL |
| 3 | Coded slice of a non-TFD TLA picture  slice\_layer\_rbsp( ) | VCL |
| 4, 5 | Coded slice of a CRA picture  slice\_layer\_rbsp( ) | VCL |
| 6, 7 | Coded slice of a BLA picture  slice\_layer\_rbsp( ) | VCL |
| 8 | Coded slice of an IDR picture slice\_layer\_rbsp( ) | VCL |
| 9..24 | Reserved | n/a |
| 25 | Video parameter set video\_parameter\_set\_rbsp( ) | non-VCL |
| 26 | Sequence parameter set seq\_parameter\_set\_rbsp( ) | non-VCL |
| 27 | Picture parameter set pic\_parameter\_set\_rbsp( ) | non-VCL |
| 28 | Adaptation parameter set aps\_rbsp( ) | non-VCL |
| 29 | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-VCL |
| 30 | Filler data filler\_data\_rbsp( ) | non-VCL |
| 31 | Supplemental enhancement information (SEI) sei\_rbsp( ) | non-VCL |
| 32..47 | Reserved | n/a |
| 48..63 | Unspecified | non-VCL |

NOTE 4 - A CRA picture having nal\_unit\_type equal to 4 may have associated TFD pictures present in the bitstream. A CRA picture having nal\_unit\_type equal to 5 does not have associated TFD pictures present in the bitstream. A BLA picture having nal\_unit\_type equal to 6 may have associated TFD pictures present in the bitstream. A BLA picture having nal\_unit\_type equal to 7 does not have associated TFD pictures present in the bitstream.

Coded slice NAL unit collectively refers to a VCL NAL unit, which has nal\_unit\_type in the range of 1 to 8, inclusive. The variable IdrPicFlag is specified as

IdrPicFlag = ( ( nal\_unit\_type = = 8 ) ? 1 : 0 ) (7‑1)

The variable RapPicFlag is specified as

RapPicFlag = ( ( nal\_unit\_type >= 4 && nal\_unit\_type <= 8 ) ? 1 : 0 ) (7‑2)

When the value of nal\_unit\_type is equal to any particular value in the range of 1 to 8, inclusive, for a NAL unit of a particular picture, all VCL NAL units of that particular picture shall have nal\_unit\_type equal to that particular value.

When the value of nal\_unit\_type is equal to 4 or 5 for all VCL NAL units of a particular picture, the particular picture is referred to as a CRA picture.

When the value of nal\_unit\_type is equal to 6 or 7 for all VCL NAL units of a particular picture, the particular picture is referred to as a BLA picture.

All coded pictures that follow a CRA or BLA picture both in decoding order and output order shall not use inter prediction from any picture that precedes the CRA or BLA picture either in decoding order or output order, and any picture that precedes the CRA or BLA picture in decoding order shall also precede the CRA or BLA picture in output order.

[Ed. Note (GJS/YK): The constraint on inter prediction should be expressed as a constraint on the reference picture set or the final reference picture list, whichever is easier to express.]

It is a requirement of bitstream conformance that no TFD pictures shall be present in the bitstream that are associated with a CRA picture having nal\_unit\_type equal to 5 or a BLA picture having nal\_unit\_type equal to 7.

When the value of nal\_unit\_type is equal to 8 for all VCL NAL units of a particular picture, that particular picture is referred to as an IDR picture. All coded pictures that follow an IDR picture in decoding order shall not use inter prediction from any picture that precedes the IDR picture in decoding order, and any picture that precedes the IDR picture in decoding order shall also precede the IDR picture in output order.

[Ed. Note (GJS/YK): The constraint on inter prediction should be expressed as a constraint on the reference picture set or the final reference picture list, whichever is easier to express.]

RAP picture collectively refers to a coded picture that is a CRA picture, a BLA picture or an IDR picture, and RAP access unit collectively refers to an access unit that is a CRA access unit, a BLA access unit or an IDR access unit.

NOTE 5 - Any parameter set (video parameter set, sequence parameter set, picture parameter set, or adaptation parameter set) must be available before the activation of the parameter set. To be able to perform random access from any particular RAP picture by discarding all access units before the particular RAP access unit (and to correctly decode the particular RAP access unit and all the subsequent access units in both decoding and output order), the following condition must be satisfied: each parameter set that is activated during the decoding of the particular RAP access unit or during the decoding of any subsequent access unit in decoding order is either present or provided through external means at or subsequent to that particular RAP access unit and prior to any NAL unit activating that parameter set.

When the value of nal\_unit\_type is equal to 3 for all VCL NAL units of a particular picture, that particular picture is referred to as a TLA picture. A TLA picture and all coded pictures with temporal\_id greater than or equal to the temporal\_id of the TLA picture that follow the TLA picture in decoding order shall not use inter prediction from any picture with temporal\_id greater than or equal to the temporal\_id of the TLA picture that precedes the TLA picture in decoding order. A TLA picture shall not be TFD picture; hence TLA picture is also referred to as non-TFD TLA picture.

[Ed. Note (GJS/YK): The constraint on inter prediction should be expressed as a constraint on the reference picture set or the final reference picture list, whichever is easier to express.]

When temporal\_id\_nesting\_flag is equal to 1 and temporal\_id is greater than 0, nal\_unit\_type shall be equal to 3

**temporal\_id** specifies a temporal identifier for the NAL unit. The value of temporal\_id shall be the same for all VCL NAL units of an access unit. When an access unit is a RAP access unit, temporal\_id for all VCL NAL units of the access unit shall be equal to 0. When nal\_unit\_type is equal to 3, temporal\_id shall not be equal to 0.

The temporal\_id of an access unit is derived as equal to the temporal\_id value of the VCL NAL units in the access unit.

For a non-VCL NAL unit, the value of temporal\_id shall be equal to the minimum value of the temporal\_id values of all access units the non-VCL NAL unit applies to. When nal\_unit\_type is equal to 7 (sequence parameter set), temporal\_id shall be equal to 0. When nal\_unit\_type is equal to 9 (access unit delimiter) or 12 (filler data), temporal\_id shall be equal to the temporal\_id of the access unit containing the non-VCL NAL unit. When nal\_unit\_type is equal to 8 (picture parameter set) or 14 (adaptation parameter set), temporal\_id may be less than, equal to, or greater than the temporal\_id of the containing access unit. When nal\_unit\_type is equal to 6 (SEI), temporal\_id shall not be less than the temporal\_id of the containing access unit.

NOTE 6 – When nal\_unit\_type is equal to 7 (sequence parameter set), temporal\_id must be equal to 0, as a sequence parameter set applies at least to a RAP access unit. When nal\_unit\_type is equal to 9 (access unit delimiter) or 12 (filler data), temporal\_id must be equal to the temporal\_id of the access unit containing the non-VCL NAL unit, as access unit delimiter or filler data only applies to the containing access unit. When nal\_unit\_type is equal to 8 (picture parameter set) or 14 (adaptation parameter set), temporal\_id may be less than, equal to, or greater than the temporal\_id of the containing access unit, as a picture parameter set or adaptation parameter set may be repeated in access units not referring to the picture parameter set or adaptation parameter set for improved error resilience, and all picture parameter sets or adaptation parameter sets may be included in the beginning of a bitstream wherein the first coded picture has temporal\_id equal to 0. When nal\_unit\_type is equal to 6 (SEI), temporal\_id may be equal to or greater than the temporal\_id of the containing access unit, as an SEI NAL unit may contain a picture buffering SEI message or a picture timing SEI message that applies to a bitstream subset including access units for which the temporal\_id values are greater than the temporal\_id of the access unit containing the SEI NAL unit.

**reserved\_one\_5bits** shall be equal to '00001'. Other values of reserved\_one\_5bits may be specified in the future by ITU‑T | ISO/IEC. Decoders shall ignore (i.e. remove from the bitstream and discard) NAL units with values of reserved\_one\_5bits not equal to '00001'.

**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 contains the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain. [Ed. (GJS): Generally, there are way too many places where the word "shall" is used. If it is not a testable constraint on the content of a conforming bitstream and is not a testable constraint on the output behaviour of the decoder, it should not be expressed using a "shall".]

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 .

NOTE 7 – 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 8 – 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 described 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 syntax elements).

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 the bit pattern with the pattern:

'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.

– 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.

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 , , and specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

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

This subclause specifies the activation process of video, sequence, picture and adaptation parameter sets.

NOTE 1 – The video sequence, picture and adaptation parameter set mechanism decouples the transmission of infrequently changing information from the transmission of coded block data. Video, 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, included in at least one access unit with temporal\_id equal to or less than the temporal\_id of the adaptation parameter set NAL unit, unless the adaptation parameter set is provided through external means.

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, included in at least one access unit with temporal\_id equal to or less than the temporal\_id of the picture parameter set NAL unit, unless the picture parameter set is provided through external means.

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, included in at least one access unit with temporal\_id equal to 0, unless the sequence parameter set is provided through external means. An activated sequence parameter set RBSP shall remain active for the entire coded video sequence.

NOTE 2 – Because an IDR or BLA 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 or BLA 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.

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

When a video parameter set RBSP (with a particular value of video\_parameter\_set\_id) is not already active and it is referred to by activation of a sequence parameter set RBSP (using that value of video\_parameter\_set\_id), it is activated. This video parameter set RBSP is called the active video parameter set RBSP until it is deactivated by the activation of another video parameter set RBSP. A video parameter set RBSP, with that particular value of video\_parameter\_set\_id shall be available to the decoding process prior to its activation, included in at least one access unit with temporal\_id equal to 0, unless the video parameter set is provided through external means. An activated video parameter set RBSP shall remain active for the entire coded video sequence.

Any video parameter set NAL unit containing the value of video\_parameter\_set\_id for the active video parameter set RBSP for a coded video sequence shall have the same content as that of the active video 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 sequence parameter set NAL unit of another coded video sequence.

NOTE 3 – If video parameter set RBSP, sequence parameter set RBSP, picture 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 video parameter set RBSP, sequence parameter set RBSP, picture parameter set RBSP or adaptation parameter set RBSP respectively. Otherwise (video parameter set RBSP, sequence parameter set RBSP, picture 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 video parameter sets, sequence parameter sets, picture parameter sets and adaptation parameter sets and other syntax elements are expressions of constraints that apply only to the active video parameter set, the active sequence parameter set, the active picture parameter set and the active adaptation parameter set. If any video parameter set RBSP, sequence parameter set RBSP, picture parameter set RBSP or 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 ), the values of parameters of the active video parameter set, the active sequence parameter set, the active picture parameter set RBSP and the active adaptation parameter set RBSP shall be considered in effect. For interpretation of SEI messages, the values of the active video parameter set, the active sequence parameter set, the active picture parameter set RBSP and the active adaptation parameter set RBSP for the operation of the decoding process for the VCL NAL units of the 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. The order of NAL units and coded pictures and their association to access units is described in subclause .

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

##### 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  that are decoded using the decoding process specified in clauses -.

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

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 coded picture specifies the start of a new access unit:

– access unit delimiter NAL unit (when present),

– video parameter set NAL unit (when present),

– sequence parameter set NAL unit (when present),

– picture parameter set NAL unit (when present),

– adaptation parameter set NAL unit (when present),

– SEI NAL unit (when present),

– NAL units with nal\_unit\_type in the range of 33 to 43, inclusive (when present),

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

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

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 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.

– NAL units having nal\_unit\_type equal to 0, 30, or in the range of 44 to 63, inclusive, shall not precede the first VCL NAL unit of the coded picture.

NOTE 1 – Video parameter set NAL units, sequence parameter set NAL units, picture parameter set NAL units or adaptation parameter set NAL units may be present in an access unit, but cannot follow the last VCL NAL unit of the 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, or 30, or in the ranges of 9..24, 25..28, or 32..63, is shown in . [Ed. (TW/YK):adjust text and figure, replace "Primary coded picture" with "Coded picture", remove the "Redundant coded picture" box, remove the "Auxiliary coded picture" box, either remove "End of sequence" box and "End of stream" box or add the two NAL unit types into Table 7-1.]



Figure 7‑1 – Structure of an access unit not containing any NAL units with nal\_unit\_type equal to 0, , or 30, or in the ranges of 9..24, 25..28, or 32..63

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

Any coded slice NAL unit or coded slice data partition A NAL unit of the 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 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.

– RapPicFlag differs in value.

– RapPicFlag is equal to 1 for both and rap\_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

#### Video parameter set RBSP semantics

**vps\_max\_temporal\_layers\_minus1** plus 1 specifies the maximum number of temporal sub-layers that may be present in the bitstream.

[Ed.(YK): Add a definition of sub-layer in Clause 3.]

**vps\_max\_layers\_minus1** plus 1 specifies the maximum number of layers that may be present in the coded video sequence. The value of vps\_num\_layers\_minus1 shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. Other values for vps\_num\_layers\_minus1 are reserved for future use by ITU-T | ISO/IEC. Decoders should ignore vps\_num\_layers\_minus1 not equal to 0 in a video parameter set NAL unit.

[Ed.(YK): Add a definition of layer (probably it should "scalable layer", as there is already a definition of layer for something else) in Clause 3.]

**video\_parameter\_set\_id** identifies a video parameter set. The value of video\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

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

Dependent on vps\_temporal\_id\_nesting\_flag, the following applies.

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

– Otherwise (vps\_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 that is 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 that is less than tIdB, which follows the access unit auB and precedes the access unit auA in decoding order.

NOTE 1 – The syntax element vps\_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 that is greater than tIdN, is always possible.

**vps\_max\_dec\_pic\_buffering[** i **]** specifies the required size of the decoded picture buffer in units of picture storage buffers for the bitstream subset as specified in subclause  with i as input. The coded video sequence shall not require a decoded picture buffer with size of more than Max( 1, vps\_max\_dec\_pic\_buffering[ i ] ) picture storage 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 vps\_max\_dec\_pic\_buffering[ i ] shall be in the range of 0 to MaxDpbSize (as specified in subclause ), inclusive.

**vps\_num\_reorder\_pics[** i **]** indicates the maximum allowed number of pictures preceding any picture in decoding order and succeeding that picture in output order for the bitstream subset as specified in subclause  with i as input. The value of vps\_num\_reorder\_pics[ i ] shall be in the range of 0 to vps\_max\_dec\_pic\_buffering[ i ], inclusive.

**vps\_max\_latency\_increase[** i **]** not equal to 0 is used to compute the value of MaxLatencyPictures[ i ] as specified by setting MaxLatencyPictures[ i ] to vps\_num\_reorder\_pics[ i ] + vps\_max\_latency\_increase[ i ] for the bitstream subset as specified in subclause  with i as input. When vps\_max\_latency\_increase[ i ] is not equal to 0, the value of MaxLatencyPictures[ i ] specifies the maximum number of pictures that can precede any picture in the coded video sequence in output order and follow that picture in decoding order for the bitstream subset as specified in subclause  with i as input. When vps\_max\_latency\_increase[ i ] is equal to 0, no corresponding limit is expressed. The value of vps\_max\_latency\_increase[ i ] shall be in the range of 0 to 232 − 1, inclusive.

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

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

#### Sequence parameter set RBSP semantics

**profile\_space** specifies the context for the intepretation of profile\_idc and profile\_combatibility\_flag[ i ] for all possible values of i. The value of profile\_space shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. Other values for for profile\_space are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the coded video sequence if profile\_space is not equal to 0.

When profile\_space is equal to 0, **profile\_idc** indicates the profile to which the coded video sequence conforms.

[Ed. (DS): We might prefer not to use the profile\_idc value zero, or reserve it to mean "no profile signalled, bitstream is unconstrained"; this gives us one spare bit in the profile compatibility flags array.]

**constraint\_flags** shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. Other values for for constraint\_flags are reserved for future use by ITU-T | ISO/IEC. Decoders should ignore the value of constraint\_flags.

**level\_idc** indicates the level to which the coded video sequence conforms. A greater value of level\_idc indicates a higher level.

When profile\_space is equal to 0, **profile\_compatability\_flag[** i **]** equal to 1 indicates that the coded video sequence conforms to the profile indicated by profile\_idc equal to i. When profile\_space is equal to 0, profile\_compatability\_flag[ profile\_idc ] shall be equal to 1.

NOTE 1 – The level\_idc definitions need to be independent of profile\_idc, as a decoder may not recognize the the profile\_idc value but instead recognize, and be able to decode, one or more of the profiles indicated by the profile compatibility flags.

NOTE 2 – For bitstreams compatible with more than one profile, the value of profile\_idc should be set to the "best viewed as" profile.

NOTE 3 – The constraint\_flags can be used to indicate the respect of further constraints on the bitstream (e.g. that a selected tool, permitted by the profiles signalled, is nonetheless not used). These flags are ideally profile independent, but unlike level\_idc, it is allowed to be contextual on profile\_idc.

**video\_parameter\_set\_id** refers to the active video parameter set. The value of video\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

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

**separate\_colour\_plane\_flag** equal to 1 specifies that the three colour components of the 4:4:4 chroma format are coded separately. separate\_colour\_plane\_flag equal to 0 specifies that the colour components are not coded separately. When separate\_colour\_plane\_flag is not present, it is inferred to be equal to 0. When separate\_colour\_plane\_flag is equal to 1, the coded picture consists of three separate components, each of which consists of coded samples of one colour plane (Y, Cb or Cr) that each use the monochrome coding syntax. In this case, each colour plane is associated with a specific colour\_plane\_id value.

NOTE 4 – There is no dependency in decoding processes between the colour planes having different colour\_plane\_id values. For example, the decoding process of a monochrome picture with one value of colour\_plane\_id does not use any data from monochrome pictures having different values of colour\_plane\_id for inter prediction.

Depending on the value of separate\_colour\_plane\_flag, the value of the variable ChromaArrayType is assigned as follows:

– If separate\_colour\_plane\_flag is equal to 0, ChromaArrayType is set equal to chroma\_format\_idc.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), ChromaArrayType is set equal to 0.

**sps\_max\_temporal\_layers\_minus1** plus 1 specifies the maximum number of temporal layers that may be present in the sequence. The value of sps\_max\_temporal\_layers\_minus1 shall be in the range of 0 to 7, inclusive. The value of sps\_max\_temporal\_layers\_minus1 shall be equal to vps\_max\_temporal\_layers\_minus1.

**pic\_width\_in\_luma\_samples** specifies the width of each decoded picture in units of luma samples. pic\_width\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of ( 1 << Log2MinCbSize ).

**pic\_height\_in\_luma\_samples** specifies the height of each decoded picture in units of luma samples. pic\_height\_in\_luma\_samples shall not be equal to 0 and shall be an integer multiple of ( 1 << Log2MinCbSize ).

**pic\_cropping\_flag** equal to 1 indicates that the picture cropping offset parameters follow next in the sequence parameter set. pic\_cropping\_flag equal to 0 indicates that the picture cropping offset parameters are not present.

**pic\_crop\_left\_offset**, **pic\_crop\_right\_offset**, **pic\_crop\_top\_offset**, and **pic\_crop\_bottom\_offset** specify the samples of the pictures in the coded video sequence that are output from the decoding process, in terms of a rectangular region specified in picture coordinates for output.

The variables CropUnitX and CropUnitY are derived as follows:

– If ChromaArrayType is equal to 0, CropUnitX and CropUnitY are derived as:

CropUnitX = 1 (7-3)  
CropUnitY = 1 (7-4)

– Otherwise (ChromaArrayType is equal to 1, 2, or 3), CropUnitX and CropUnitY are derived as:

CropUnitX = SubWidthC (7-5)  
CropUnitY = SubHeightC (7-6)

The picture cropping rectangle contains luma samples with horizontal picture coordinates from CropUnitX \* pic\_crop\_left\_offset to pic\_width\_in\_luma\_samples − ( CropUnitX \* pic\_crop\_right\_offset + 1 ) and vertical picture coordinates from CropUnitY \* pic\_crop\_top\_offset to pic\_height\_in\_luma\_samples − ( CropUnitY \* pic\_crop\_bottom\_offset + 1 ), inclusive. The value of pic\_crop\_left\_offset shall be in the range of 0 to ( pic\_width\_in\_luma\_samples / CropUnitX ) − ( pic\_crop\_right\_offset + 1 ), inclusive; and the value of pic\_crop\_top\_offset shall be in the range of 0 to ( pic\_height\_in\_luma\_samples / CropUnitY ) − ( pic\_crop\_bottom\_offset + 1 ), inclusive.

When pic\_cropping\_flag is equal to 0, the values of pic\_crop\_left\_offset, pic\_crop\_right\_offset, pic\_crop\_top\_offset, and pic\_crop\_bottom\_offset is inferred to be equal to 0.

When ChromaArrayType is not equal to 0, the corresponding specified samples of the two chroma arrays are the samples having picture coordinates ( x / SubWidthC, y / SubHeightC ), where ( x, y ) are the picture coordinates of the specified luma samples.

NOTE 5 – The picture cropping parameters are only applied at the output. All internal decoding processes are applied to the uncropped picture size.

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

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

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 quantization parameter range offset QpBdOffsetC, as specified by

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (7‑9)  
QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (7‑10)

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 less than or equal to the value of BitDepthY.

PCMBitDepthY = 1 + pcm\_sample\_bit\_depth\_luma\_minus1 (7‑11)

**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 less than or equal to the value of BitDepthC.

PCMBitDepthC = 1 + pcm\_sample\_bit\_depth\_chroma\_minus1 (7‑12)

**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 ) (7‑13)

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

**sps\_max\_dec\_pic\_buffering[** i **]** specifies the required size of the decoded picture buffer in units of picture storage buffers for the bitstream subset as specified in subclause  with i as input. The coded video sequence shall not require a decoded picture buffer with size of more than Max( 1, sps\_max\_dec\_pic\_buffering[ i ] ) picture storage 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 sps\_max\_dec\_pic\_buffering[ i ] shall be in the range of 0 to MaxDpbSize (as specified in subclause ), inclusive. The value of sps\_max\_dec\_pic\_buffering[ i ] shall be equal to vps\_max\_dec\_pic\_buffering[ i ] for each possible value of i.

**sps\_num\_reorder\_pics[** i **]** indicates the maximum allowed number of pictures preceding any picture in decoding order and succeeding that picture in output order for the bitstream subset as specified in subclause  with i as input. The value of sps\_num\_reorder\_pics[ i ] shall be in the range of 0 to sps\_max\_dec\_pic\_buffering[ i ], inclusive. The value of sps\_num\_reorder\_pics[ i ] shall be equal to vps\_num\_reorder\_pics[ i ] for each possible value of i.

**sps\_max\_latency\_increase[** i **]** not equal to 0 is used to compute the value of MaxLatencyPictures[ i ] as specified by setting MaxLatencyPictures[ i ] equal to sps\_num\_reorder\_pics[ i ] + sps\_max\_latency\_increase[ i ] for the bitstream subset as specified in subclause  with i as input. When sps\_max\_latency\_increase[ i ] is not equal to 0, the value of MaxLatencyPictures[ i ] specifies the maximum number of pictures that can precede any picture in the coded video sequence in output order and follow that picture in decoding order for the bitstream subset as specified in subclause  with i as input. When sps\_max\_latency\_increase[ i ] is equal to 0, no corresponding limit is expressed. The value of sps\_max\_latency\_increase[ i ] shall be in the range of 0 to 232 − 1, inclusive. The value of sps\_max\_latency\_increase[ i ] shall be equal to vps\_max\_latency\_increase[ i ] for each possible value of i.

**restricted\_ref\_pic\_lists\_flag** equal to 1 indicates that all slices having the same value of slice\_type that belong to the same picture have identical reference picture lists. restricted\_ref\_pic\_lists\_flag equal to 0 indicates that there may be slices having the same value of slice\_type that belong to the same picture that have different reference picture lists. num\_ref\_idx\_active\_override\_flag, num\_ref\_idx\_l0\_active\_minus1, and num\_ref\_idx\_l1\_active\_minus1, when present , shall be identical for all slices having the same value of slice\_type that belong to the same picture when restricted\_ref\_pic\_lists\_flag is equal to 1.

**lists\_modification\_present\_flag** equal to 0 specifies that the syntax structure ref\_pic\_list\_modification( ) is not present in the slice header. lists\_modification\_present\_flag equal to 1 specifies that the syntax structure ref\_pic\_list\_modification( ) is present in the slice header. When not present, the value of lists\_modification\_present\_flag is inferred to be equal to 1.

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

The variable Log2MinCbSize 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 variables Log2CtbSize, PicWidthInCtbs, PicHeightInCtbs, PicWidthInMinCbs, and PicHeightInMinCbs are derived as follows.

Log2CtbSize = log2\_min\_coding\_block\_size\_minus3 + 3 + log2\_diff\_max\_min\_coding\_block\_size (7‑14)  
PicWidthInCtbs = Ceil( pic\_width\_in\_luma\_samples ÷ ( 1 << Log2CtbSize ) ) (7‑15)  
PicHeightInCtbs = Ceil( pic\_height\_in\_luma\_samples ÷ ( 1 << Log2CtbSize ) ) (7‑16)  
PicWidthInMinCbs = pic\_width\_in\_luma\_samples / ( 1 << Log2MinCbSize ) (7‑17)  
PicHeightInMinCbs = pic\_height\_in\_luma\_samples / ( 1 << Log2MinCbSize ) (7‑18)

The variable CtbSize, which specifies the width and the height, of the square array for each luma coding tree block, is derived as follows.

CtbSize = 1 << Log2CtbSize (7‑19)

The variables CtbWidthC and CtbHeightC, which specify the width and height, respectively, of the array for each chroma coding tree block, are derived as follows.

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

– Otherwise, CtbWidthC and CtbHeightC are derived as

CtbWidthC = CtbSize / SubWidthC (7‑20)  
CtbHeightC = CtbSize / SubHeightC (7‑21)

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

The variable Log2MinTrafoSize is set equal to log2\_min\_transform\_block\_size\_minus2 + 2. The bitstream shall not contain data that result in Log2MinTrafoSize greater than or equal to Log2MinCbSize.

**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 Min( Log2CtbSize, 5).

The array ScanOrder[ log2BlockWidth ][ log2BlockHeight ][ scanIdx ][ sPos ][ sComp ] specifies the mapping of the scan position sPos, ranging from 0 to ( ( 1 << log2BlockWidth ) \* ( 1 << log2BlockHeight ) ) − 1, inclusive, 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 sComp equal to 0 specifies the horizontal component and the array index sComp equal to 1 specifies the vertical component. The array ScanOrder is derived as follows.

For the variables log2BlockWidth and log2BlockHeight ranging from Max( 2, Log2MinTrafoSize − 1) to Log2MaxTrafoSize, inclusive, the scanning order array ScanOrder is derived as follows.

– The up-right diagonal scanning array initialization process as specified in 6.5.4 is invoked with 1 << log2BlockWidth and 1 << log2BlockHeight as input and the output is assigned to ScanOrder[ 0 ][ log2BlockWidth ][ log2BlockHeight ].

– The horizontal scanning array initialization process as specified in 6.5.5 is invoked with 1 << log2BlockWidth and 1 << log2BlockHeight as input and the output is assigned to ScanOrder[ 1 ][ log2BlockWidth ][ log2BlockHeight ].

– The vertical scanning array initialization process as specified in 6.5.6 is invoked with 1 << log2BlockWidth and 1 << log2BlockHeight as input and the output is assigned to ScanOrder[ 2 ][ log2BlockWidth ][ log2BlockHeight ].

**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( Log2CtbSize, 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( Log2CtbSize, 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 . scaling\_list\_enable\_flag equal to 0 specifies that scaling list is not used for scaling process for transform coefficients in .

**sps\_scaling\_list\_data\_present\_flag** equal to 1 specifies that scaling list parameters are present in the sequence parameter set. sps\_scaling\_list\_data\_present\_flag equal to 0 specifies that scaling list parameters are not present in the sequence parameter set. When not present, the value of sps\_scaling\_list\_data\_present\_flag is inferred to be 0.

**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.

**seq\_loop\_filter\_across\_slices\_enabled\_flag** equal to 1 specifies that the slice\_loop\_filter\_across\_slices\_enabled\_flag determines whether in-loop filtering operations are performed across slice boundaries; otherwise, the in-loop filtering operations are slice-independent and not applied across slice boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter.

**asymmetric\_motion\_partitions\_enabled\_flag** equal to 1 specifies that asymmetric motion partitions may be used in coding tree blocks; asymmetric\_motion\_partitions\_enabled\_flag equal to 0 specifies that asymmetric motion partitions cannot be used in coding tree blocks.

**nsrqt\_enabled\_flag** equal to 1 specifies that non-square residual quadtree partitions may be used in coding tree blocks; nsrqt\_enabled\_flag equal to 0 specifies that non-square quadtree partitions cannot be used in coding tree blocks.

**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.

**alf\_coef\_in\_slice\_flag** equal to 1 specifies that the ALF parameters alf\_param() are present in the slice header; equal to 0 specifies that the ALF parameters alf\_param() are not present in the APS.

**pcm\_loop\_filter\_disable\_flag** specifies whether the loop filter process is disabled on reconstructed samples of I\_PCM blocks. If the pcm\_loop\_filter\_disable\_flag value is equal to 1, deblocking filter, sample adaptive offset and adaptive loop filter processes on the reconstructed samples of I\_PCM blocks are disabled; otherwise if the pcm\_loop\_filter\_disable\_flag value is equal to 0, deblocking filter, sample adaptive offset and adaptive loop filter processes on the reconstructed samples of I\_PCM blocks are not disabled. When pcm\_loop\_filter\_disable\_flag is not present, it is inferred to be equal to 0.

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

**sps\_temporal\_id\_nesting\_flag** specifies whether inter prediction is additionally restricted for the coded video sequence. sps\_temporal\_id\_nesting\_flag shall be equal to vps\_temporal\_id\_nesting\_flag.

Dependent on sps\_temporal\_id\_nesting\_flag, the following applies.

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

– Otherwise (sps\_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 that is 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 that is less than tIdB, which follows the access unit auB and precedes the access unit auA in decoding order. [Ed. Note (GJS/YK): It might be better to express the constraint on inter prediction to be a constraint on the final reference picture list.]

NOTE 6 – The syntax element sps\_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 that is greater than 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\_short\_term\_ref\_pic\_sets** specifies the number of short-term reference picture sets that are specified in the sequence parameter set. The value of num\_short\_term\_ref\_pic\_sets shall be in the range of 0 to 64, inclusive.

NOTE 7 – 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 in the coded video sequence. 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 in the coded video sequence.

**sps\_temporal\_mvp\_enable\_flag** equal to 1 specifies that pic\_temporal\_mvp\_enable\_flag may be present in the slice headers and temporal motion vector predictors may be used. sps\_temporal\_mvp\_enable\_flag equal to 0 specifies that pic\_temporal\_mvp\_enable\_flag shall not be present in the slice headers and temporal motion vector predictors shall not be used.

**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. Its value does not affect decoder 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.

**sign\_data\_hiding\_flag** equal to 0 specifies that sign bit hiding is disabled. sign\_data\_hiding\_flag equal to 1 specifies that sign bit hiding is enabled

**sign\_hiding\_threshold** specifies a threshold to be compared with the number of coefficients that are between the last non-zero coefficient and the first non-zero coefficient along the scanning path within a coefficient group to determine whether the sign bit of the first nonzero coefficient is present in the bitstream or whether it will be inferred.

**cabac\_init\_present\_flag** equal to 1 specifies that cabac\_init\_flag is present in the slice header.

**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 15, inclusive. [Ed. (GJS): Constrain to the limit imposed by max DPB size instead of 15.]

**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 15, 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 coding blockscoded using Inter prediction modes for the prediction of coding blockscoded using Intra prediction modes. constrained\_intra\_pred\_flag equal to 1 specifies constrained intra prediction, in which case prediction of coding blockscoded using Intra prediction modes only uses residual data and decoded samples from coding blocks coded using Intra prediction mdoes.

**slice\_granularity** indicates the slice granularity within a picture. The value of slice\_granularity shall not be larger than Min( Log2CtbSize − 4, log2\_diff\_max\_min\_coding\_block\_size ). The variable SliceGranularity is set to the value of ( slice\_granularity << 1 ).

**diff\_cu\_qp\_delta\_depth** specifies the difference between the control granularity for QPY values for coding units within a picture and slice\_granularity plus 1. The value of diff\_cu\_qp\_delta\_depth shall be in the range of 0 to log2\_diff\_max\_min\_coding\_block\_size − slice\_granularity + 1, inclusive.

The variable Log2MinCUDQPSize specifying the minimum luma coding block size of coding units that convey cu\_qp\_delta, is derived as follows.

Log2MinCUDQPSize = Log2CtbSize − diff\_cu\_qp\_delta\_depth − slice\_granularity + 1 (7‑22)

**cb\_qp\_offset, 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 is applied to B slices. weighted\_bipred\_idc equal to 1 specifies that explicit weighted prediction is 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.

**output\_flag\_present\_flag** equal to 1 indicates that the pic\_output\_flag syntax element is present in the associated slice headers. output\_flag\_present\_flag equal to 0 indicates that the pic\_output\_flag syntax element is not present in the associated slice headers.

**transquant\_bypass\_enable\_flag** specifies whether a cu\_transquant\_bypass\_flag syntax element is coded or not. If transquant\_bypass\_enable\_flag is equal to 1, cu\_transquant\_bypass\_flag is coded, otherwise it is not coded.

**dependent\_slice\_enabled\_flag** equal to 1 specifies the presence of the syntax element dependent\_slice\_flag in the slice header for coded pictures referring to the picture parameter set. dependent\_slice\_enabled\_flag equal to 0 specifies the absence of the syntax element dependent\_slice\_flag in the slice header for coded pictures referring to the picture parameter set. When tiles\_or\_entropy\_coding\_sync\_idc is equal to 3, the value of dependent\_slice\_enabled\_flag shall be equal to 1.

**tiles\_or\_entropy\_coding\_sync\_idc** equal to 0 specifies that there shall be only one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of cabac\_independent\_flag and dependent\_slice\_flag for coded pictures referring to the picture parameter set shall not be both equal to 1.

[Ed.(YK): When cabac\_independent\_flag and depedent\_slice\_flag are both equal to 1 for a slice, the slice is an entropy slice.]

tiles\_or\_entropy\_coding\_sync\_idcequal to 1 specifies that there may be more than one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of cabac\_independent\_flag and dependent\_slice\_flag for coded pictures referring to the picture parameter set shall not be both equal to 1.

tiles\_or\_entropy\_coding\_sync\_idcequal to 2 specifies that there shall be only one tile in each picture referring to the picture parameter set, a specific synchronization process for context variables shall be invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set and a specific memorization process for context variables shall be invoked after decoding two coding tree blocks of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of cabac\_independent\_flag and dependent\_slice\_flag for coded pictures referring to the picture parameter set shall not be both equal to 1.

tiles\_or\_entropy\_coding\_sync\_idcequal to 3 specifies that there shall be only one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of cabac\_independent\_flag and dependent\_slice\_flag for coded pictures referring to the picture parameter set may both be equal to 1.

When dependent\_slice\_enabled\_flag shall be equal to 0, tiles\_or\_entropy\_coding\_sync\_idc shall not be equal to 3.

It's a requirement of bitstream conformance that the value of tiles\_or\_entropy\_coding\_sync\_idc shall be the same for all picture parameter sets that are activated within a coded video sequence.

For each slice referring to the picture parameter set, when tiles\_or\_entropy\_coding\_sync\_idc is equal to 2 and the first coding block in the slice is not the first coding block in the first coding tree block of a row of coding tree blocks, the last coding block in the slice shall belong to the same row of coding tree blocks as the first coding block in the slice slice.

**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.

When num\_tile\_columns\_minus1 is equal to 0, num\_tile\_rows\_minus1 shall not be equal to 0.

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

**column\_width[**i **]** specifies the width of the i-th tile column in units of coding tree blocks.

**row\_height[**i **]** specifies the height of the i-th tile row in units of coding tree blocks.

The vector colWidth[ i ] specifies the width of the i-th tile column in units of CTBs with the column i ranging from 0 to num\_tile\_columns\_minus1, inclusive.

The vector CtbAddrRStoTS[ ctbAddrRS ] specifies the conversation from a CTB address in raster scan order to a CTB address in tile scan order with the index ctbAddrRS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive.

The vector CtbAddrTStoRS[ ctbAddrTS ] specifies the conversation from a CTB address in tile scan order to a CTB address in raster scan order with the index ctbAddrTS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive.

The vector TileId[ ctbAddrTS ] specifies the conversation from a CTB address in tile scan order to a tile id with ctbAddrTS ranging from 0 to (picHeightInCtbs \* picWidthInCtbs) − 1, inclusive.

The values of colWidth, CtbAddrRStoTS, CtbAddrTStoRS and TileId are derived by invoking the CTB raster and tile scanning conversation process as specified in subclause 6.5.1 with PicHeightInCtbs and PicWidthInCtbs as inputs and the output is assigned to colWidth, CtbAddrRStoTS and TileId.

The values of ColumnWidthInLumaSamples[ i ], specifying the width of the i-th tile column in units of luma samples, are set equal to colWidth[ i ] << Log2CtbSize.

The array MinCbAddrZS[ x ][ y ], specifying the conversation from a location ( x, y ) in units of minimum CBs to a minimum CB address in z-scan order with x ranging from 0 to picWidthInMinCbs − 1, inclusive, and y ranging from 0 to picHeightInMinCbs − 1, inclusive, is dervied by invoking the Z scanning order array initialization process as specified in subclause 6.5.2 with Log2MinCbSize, Log2CtbSize, PicHeightInCtbs, PicWidthInCtbs, and the vector CtbAddrRStoTS as inputs and the output is assigned to MinCbAddrZS.

**loop\_filter\_across\_tiles\_enabled\_flag** equal to 1 specifies that in-loop filtering operations are performed across tile boundaries. loop\_filter\_across\_tiles\_enabled\_flag equal to 0 specifies that in-loop filtering operations are not performed across tile boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset, and adaptive loop filter operations. When not present, the value of loop\_filter\_across\_tiles\_enabled\_flag is inferred to be equal to 1.

**cabac\_independent\_flag** equal to 1 specifies that CABAC decoding of coding blocks in a slice is independent from any state of the previously decoded slice. cabac\_independent\_flag equal to 0 specifies that CABAC decoding of coding blocks in a slice is dependent from the states of the previously decoded slice. When not present, the value of cabac\_independent\_flag is inferred to be equal to 0.

**deblocking\_filter\_control\_present\_flag** equal to 1 specifies the presence of deblocking filter control syntax elements in the picture parameter set and in the slice header for pictures referring to the picture parameter set. deblocking\_filter\_control\_present\_flag equal to 0 specifies the absence of deblocking filter control syntax elements in the picture parameter set and in the slice header for pictures referring to the picture parameter set.

**deblocking\_filter\_override\_enabled\_flag** equal to 1 specifies the presence of deblocking\_filter\_overriding\_flag in the slice header for pictures referring to the picture parameter set. deblocking\_filter\_override\_enabled\_flag equal to 0 specifies the absence of deblocking\_filter\_overriding\_flag in the slice header for pictures referring to the picture parameter set. When not present, the value of deblocking\_filter\_override\_enabled\_flag is inferred to be equal to 0.

**pps\_disable\_deblocking\_filter\_flag** equal to 1 specifies that the operation of deblocking filter shall not be applied for pictures referring to the picture parameter set when deblocking\_filter\_override\_enabled\_flag is equal to 0. pps\_disable\_deblocking\_filter\_flag equal to 0 specifies that the operation of the deblocking filter shall be applied for pictures referring to the picture parameter set when deblocking\_filter\_override\_enabled\_flag is equal to 0. When not present, the value of pps\_disable\_deblocking\_filter\_flag is inferred to be equal to 0.

**beta\_offset\_div2** and **tc\_offset\_div2** specify the default deblocking parameter offsets for β and tC (divided by 2) that are applied for pictures referring to the picture parameter set unless the default deblocking parameter offsets are overriden by the deblocking parameter offsets present in the slice header for pictures referring to the picture parameter set.

**pps\_scaling\_list\_data\_present\_flag** equal to 1 specifies that parameters are present to modify the scaling lists specified in the sequence parameter set. pps\_scaling\_list\_data\_present\_flag equal to 0 specifies that the scaling lists used for the pictures referring to the picture parameter set shall be inferred to be equal to those specified by the sequence parameter set. When scaling\_list\_enable\_flag is equal to 0, the value of pps\_scaling\_list\_data\_present\_flag shall be equal to 0.

**log2\_parallel\_merge\_level\_minus2** specifies the parallel processing level of merge/skip mode. The value of log2\_parallel\_merge\_level\_minus2shall be in the range of 0 tolog2\_min\_coding\_block\_size\_minus3 + 1 + log2\_diff\_max\_min\_coding\_block\_size, inclusive.

**slice\_header\_extension\_present\_flag** equal to 0 specifies that no slice header extension syntax elements are present in the slice header for coded pictures referring to the picture parameter set. slice\_header\_extension\_present\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for slice\_header\_extension\_present\_flag is reserved for future use by ITU-T | ISO/IEC.

**slice\_extension\_present\_flag** equal to 0 specifies that no slice extension syntax elements are present in the slice layer RBSP for coded pictures referring to the picture parameter set. slice\_extension\_present\_flag shall be equal to 0 in bitstreams conforming to this Recommendation | International Standard. The value of 1 for slice\_extension\_present\_flag is reserved for future use by ITU-T | ISO/IEC.

**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. Its value does not affect decoder conformance to profiles specified in this Recommendation | International Standard.

#### Scaling list data semantics

**scaling\_list\_pred\_mode\_flag** 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 scaling\_list\_pred\_matrix\_id\_delta. scaling\_list\_pred\_mode\_flag equal to 1 speficies that the value of the scaling list is encoded by DPCM and exp-Golomb code.

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

RefMatrixID = MatrixID – scaling\_list\_pred\_matrix\_id\_delta (7‑23)

where MatrixID is specified in . When scaling\_list\_pred\_matrix\_id\_delta is equal to 0, i.e., RefMatrixID is equal to MatrixID, scalingList is inferred from the default scaling list ScalingList[ SizeID ][ MatrixID ] as specified in and .

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 colour component

|  |  |  |  |
| --- | --- | --- | --- |
| **SizeID** | **Prediction type** | **Colour 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\_dc\_coef\_minus8**[ sizeID − 2 ][ matrixID ] plus 8 specifies the DC value of the scaling list for 16x16 size when sizeID is equal to 2 and specifies the DC value of the scaling list for 32x32 size when sizeID is equal to 3.

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

Table 7‑4 – Specification of default values of ScalingList[ 0 ][ MatrixID ][ 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][i]** | 16 | 16 | 16 | 17 | 17 | 17 | 21 | 20 | 20 | 21 | 25 | 30 | 25 | 41 | 41 | 70 |
| **ScalingList[0][3..5][i]** | 16 | 16 | 16 | 17 | 17 | 17 | 21 | 21 | 21 | 21 | 24 | 24 | 24 | 36 | 36 | 57 |

Table 7‑5 – Specification of default values of ScalingList[ 1..3 ][ 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..2][0..2][i]**  **ScalingList[3][0][i]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 17 | 16 | 17 | 16 | 17 | 18 |
| **ScalingList[1..2][3..5][i]**  **ScalingList[3][1][i]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 17 | 17 | 17 | 17 | 17 | 18 |
| **i − 16** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1..2][0..2][i]**  **ScalingList[3][0][i]** | 17 | 18 | 18 | 17 | 18 | 21 | 19 | 20 | 21 | 20 | 19 | 21 | 24 | 22 | 22 | 24 |
| **ScalingList[1..2][3..5][i]**  **ScalingList[3][1][i]** | 18 | 18 | 18 | 18 | 18 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 24 | 24 | 24 | 24 |
| **i − 32** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1..2][0..2][i]**  **ScalingList[3][0][i]** | 24 | 22 | 22 | 24 | 25 | 25 | 27 | 30 | 27 | 25 | 25 | 29 | 31 | 35 | 35 | 31 |
| **ScalingList[1..2][3..5][i]**  **ScalingList[3][1][i]** | 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 28 | 28 | 28 | 28 | 28 |
| **i − 48** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[1..2][0..2][i]**  **ScalingList[3][0][i]** | 29 | 36 | 41 | 44 | 41 | 36 | 47 | 54 | 54 | 47 | 65 | 70 | 65 | 88 | 88 | 115 |
| **ScalingList[1..2][3..5][i]**  **ScalingList[3][1][i]** | 28 | 33 | 33 | 33 | 33 | 33 | 41 | 41 | 41 | 41 | 54 | 54 | 54 | 71 | 71 | 91 |

The four-dimensional array ScalingFactor[ SizeID ][ MatrixID ][ trafoType ][ i ], with i=0..( 1 << ( 4 + ( SizeID << 1 ) ) ) − 1 specifies the array of values of scaling factors according to the variables SizeID specified in , MatrixID specified in 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 4x4, ScalingFactor[ 0 ][ MatrixID ][ 0 ][ ] is derived as follows:

ScalingFactor[ 0 ][ MatrixID ][ 0 ][ y\*4+x ] = ScalingList[ 0 ][ MatrixID ][ i ] (7‑24)  
with i=0..15 and MatrixID=0..5

where zigZag is an array specified in subclause 6.5.3 with 4 as input, x=zigZag[ i ][ 0 ] and y=zigZag[ i ][ 1 ]

The elements of the quantization matrix of size 8x8, ScalingFactor[ 1 ][ MatrixID ][ 0 ][ ] is derived as follows:

ScalingFactor[ 1 ][ MatrixID ][ 0 ][ y\*8+x ] = ScalingList[ 1 ][ MatrixID ][ i ] (7‑25)  
with i=0..63 and MatrixID=0..5

where zigZag is an array specified in subclause 6.5.3 with 8 as input, x=zigZag[ i ][ 0 ] and y=zigZag[ i ][ 1 ]

The elements of the quantization matrix of size 16x16, ScalingFactor[ 2 ][ MatrixID ][ 0 ][ ] is derived as follows:

ScalingFactor[ 2 ][ MatrixID ][ 0 ][ (y\*2+j)\*16+(x\*2+k) ] = ScalingList[ 2 ][ MatrixID ][ i ] (7‑26)  
with i=0..63, j=0..1, k=0..1 and MatrixID=0..5

where zigZag is an array specified in subclause 6.5.3 with 8 as input, x=zigZag[ i ][ 0 ] and y=zigZag[ i ][ 1 ]

ScalingFactor[ 2 ][ MatrixID ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 0 ][ MatrixID ] + 8 (7‑27)  
with MatrixID=0..5

The elements of the quantization matrix of size 32x32, ScalingFactor[ 3 ][ MatrixID ][ 0 ][ ] is derived as follows:

ScalingFactor[ 3 ][ MatrixID ][ 0 ][ (y\*4+j)\*32+(x\*4+k) ] = ScalingList[ 3 ][ MatrixID ][ i ] (7‑28)  
with i=0..63, j=0..3, k=0..3 and MatrixID=0..1

where zigZag is an array specified in subclause 6.5.3 with 8 as input, x=zigZag[ i ][ 0 ] and y=zigZag[ i ][ 1 ]

ScalingFactor[ 3 ][ MatrixID ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 1 ][ MatrixID ] + 8 (7‑29)  
with MatrixID=0..1

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

ScalingFactor[ 1 ][ MatrixID ][ 1 ][ j\*16+i ] = ScalingFactor[ 2 ][ MatrixID ][ 0 ][ (j\*4)\*16+i ] (7‑30)  
with i=0..16, j=0..4 and MatrixID=0..5

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

ScalingFactor[ 1 ][ MatrixID ][ 2 ][ j\*4+i ] = ScalingFactor[ 2 ][ MatrixID ][ 0 ][ j\*16+i\*4 ] (7‑31)  
with i=0..4, j=0..16 and MatrixID=0..5

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

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

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

ScalingFactor[ 2 ][ MatrixID ][ 2 ][ j\*8+i ] = ScalingFactor[ 3 ][ MatrixID ][ 0 ][ j\*32+i\*4 ] (7‑33)  
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\_adaptive\_loop\_filter\_flag** equal to 1 specifies that the ALF is on for slices referring to the current APS; equal to 0 specifies that the ALF is off for slices referring to the current APS. When there is no active APS, the aps\_adaptive\_loop\_filter\_flag value is inferred to be 0.

**aps\_sub\_stream\_entry\_present\_flag** equal to 1 specifies that the sub-stream entry points are present in the APS; equal to 0 specifies that the sub-stream entry points are not present in the APS

**num\_entry\_point\_offsets** specifies the number of entry\_point\_offset[ i ] syntax elements in the APS.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 0, the value of num\_entry\_point\_offsets shall be in the range of 0 to number of slices in the picture referring to the APS minus -1, inclusive. When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1, the value of num\_entry\_point\_offsets shall be in the range of 0 to ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ) − 1, inclusive. When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2, the value of num\_entry\_point\_offsets shall be in the range of 0 to PicHeightInCtbs − 1, inclusive. When not present, the value of num\_entry\_point\_offsets is inferred to be equal to 0.

**offset\_len\_minus1** plus 1 specifies the length, in bits, of the entry\_point\_offset[ i ] syntax elements.

**entry\_point\_offset**[ i ] specifies the i-th entry point offset, in bytes and shall be represented by offset\_len\_minus1 plus 1 bits. The coded slice data after the APS consists of num\_entry\_point\_offsets + 1 subsets, with subset index values ranging from 0 to num\_entry\_point\_offsets, inclusive. Subset 0 consists of bytes 0 to entry\_point\_offset[ 0 ] − 1, inclusive, of the coded picture data, subset k, with k in the range of 1 to num\_entry\_point\_offsets - 1, inclusive, consists of bytes entry\_point\_offset[ i − 1 ] to entry\_point\_offset[ i ] + entry\_point\_offset[ i − 1 ] − 1, inclusive, of the coded picture data, and the last subset (with subset index equal to num\_entry\_point\_offsets) consists of the remaining bytes of the coded picture data.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 0 and num\_entry\_point\_offsets is greater than 0, each subset i with i in the range of 0 to num\_entry\_point\_offsets − 1, inclusive, shall contain all coded bits of an integer number of slices, the last subset (with subset index equal to num\_entry\_point\_offsets) shall contain all coded bits of the remaining coding blocks included in the picture.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1 and num\_entry\_point\_offsets is greater than 0, each subset shall contain all coded bits of an integer number of tiles, and the number of subsets (i.e, the value of num\_entry\_point\_offsets + 1) shall be equal to or less than the number of tiles in the picture; for level 5.2 and above, each subset shall contain all coded bits of a tile, and the number of subsets (i.e, the value of num\_entry\_point\_offsets + 1) shall be equal to the number of tiles in the picture.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2 and num\_entry\_point\_offsets is greater than 0, each subset i with i in the range of 0 to num\_entry\_point\_offsets − 1, inclusive, shall contain all coded bits of an integer number of rows of coding tree blocks, the last subset (with subset index equal to num\_entry\_point\_offsets) shall contain all coded bits of the remaining coding blocks included in the picture.

**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. Its value does not affect decoder 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 . 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 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.

**pic\_type** indicates that the slice\_type values for all slices of the coded picture are members of the set listed in for the given value of pic\_type.

Table 7‑6 – Meaning of pic\_type

|  |  |
| --- | --- |
| **pic\_type** | **slice\_type values that may be present in the coded picture** |
| 0 | I |
| 1 | P, I |
| 2 | B, P, I |

#### 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, slice data, and possibly slice extension data.

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

**slice\_extension\_data\_flag** may have any value. Its value does not affect decoder conformance to profiles specified in this Recommendation | International Standard.

#### 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 , is invoked to decode the contents of all VCL NAL units of a coded picture.

Let the variables RawMinCUBits and PicSizeInMinCbs be derived as

RawMinCUBits = (1 << Log2MinCbSize ) \* ( 1 << Log2MinCbSize ) \* BitDepthY + 2 \* (1 << Log2MinCbSize − 1) ) \* ( 1 << Log2MinCbSize − 1 ) \* BitDepthC, and

PicSizeInMinCbs = Ceil( pic\_width\_in\_luma\_samples ÷ ( 1 << Log2MinCbSize ) )\* Ceil( pic\_height\_in\_luma\_samples ÷ ( 1 << Log2MinCbSize ) )

BinCountsInNALunits shall not exceed ( 32 ÷ 3 ) \* NumBytesInVclNALunits + ( RawMinCUBits \* PicSizeInMinCbs ) ÷ 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.

#### Byte alignment semantics

**bit\_equal\_to\_one** shall be equal to 1.

**bit\_equal\_to\_zero** shall be equal to 0.

### Slice header semantics

When present, the value of the slice header syntax elements pic\_parameter\_set\_id, pic\_output\_flag, rap\_pic\_id, no\_output\_of\_prior\_pics\_flag, pic\_order\_cnt\_lsb, short\_term\_ref\_pic\_set\_sps\_flag, short\_term\_ref\_pic\_set\_idx, num\_long\_term\_pics and pic\_temporal\_mvp\_enable\_flag shall be the same in all slice headers of a coded picture. When present, the value of the slice header syntax elements poc\_lsb\_lt[ i ], delta\_poc\_msb\_present\_flag[ i ], delta\_poc\_msb\_cycle\_lt[ 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 SliceCbAddrZS and SliceCtbAddrRS are both set to 0 and the decoding starts with the first coding tree block in the picture.

**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.

**slice\_address** specifies the address in slice granularity resolution in which the slice starts. The length of the slice\_address syntax element is ( Ceil( Log2( PicWidthInCtbs \* PicHeightInCtbs ) ) + SliceGranularity ) bits.

The variable SliceCtbAddrRS, specifying the coding tree block in which the slice starts in coding tree block raster scan order, is derived as follows.

SliceCtbAddrRS = ( slice\_address >> SliceGranularity ) (7‑34)

The variable SliceCbAddrZS, specifying the address of first coding block in the slice in minimum coding block granularity in z-scan order, is derived as follows.

SliceCbAddrZS = slice\_address << ( ( log2\_diff\_max\_min\_coding\_block\_size − SliceGranularity ) <<1 ) (7‑35)

[Ed. (BB): Change it that SliceCbAddrZS is in z-scan order in a coding tree block while the CTBs are in tiles scan order]

The slice decoding starts with the largest coding unit possible at the slice starting coordinate. [Ed. (BB): More precise description is needed here.]

**dependent\_slice\_flag** equal to 1 specifies that the value of each slice header syntax element not present is inferred to be equal to the value of corresponding slice header syntax element in the preceding slice containing the coding tree block for which the coding tree block address is SliceCtbAddrRS − 1. When not present, the value of dependent\_slice\_flag is inferred to be equal to 0. The value of dependent\_slice\_flag shall be equal to 0 when SliceCtbAddrRS equal to 0.

**slice\_type** specifies the coding type of the slice according to .

Table 7‑7 – Name association to slice\_type

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

When nal\_unit\_type is equal to a value in the range of 4 to 8, inclusive (RAP picture), slice\_type shall be equal to 2.

When sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] is equal to 0, slice\_type shall be equal to 2.

**colour\_plane\_id** specifies the colour plane associated with the current slice RBSP when separate\_colour\_plane\_flag is equal to 1. The value of colour\_plane\_id shall be in the range of 0 to 2, inclusive. colour\_plane\_id equal to 0, 1, and 2 correspond to the Y, Cb, and Cr planes, respectively.

NOTE 1 – There is no dependency between the decoding processes of pictures having different values of colour\_plane\_id.

**pic\_output\_flag** affects the decoded picture output and removal processes as specified in Annex C. When pic\_output\_flag is not present, it is inferred to be equal to 1.

**rap\_pic\_id** identifies a RAP picture. The values of rap\_pic\_id in all the slices of a RAP picture shall remain unchanged. When two consecutive access units in decoding order are both RAP access units, the value of rap\_pic\_id in the slices of the first such RAP access unit shall differ from the rap\_pic\_id in the second such RAP access unit. The value of rap\_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 or a BLA picture. See Annex . When the IDR or BLA picture is the first IDR or BLA picture in the bitstream, the value of no\_output\_of\_prior\_pics\_flag has no effect on the decoding process. When the IDR or BLA picture is not the first IDR or BLA picture in the bitstream and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] 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 sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] 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 is inferred to be equal to 0.

**short\_term\_ref\_pic\_set\_sps\_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 sequence parameter set. short\_term\_ref\_pic\_set\_sps\_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 sequence 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 sequence parameter set.

The variable StRpsIdx is derived as follows.

if( short\_term\_ref\_pic\_set\_sps\_flag )  
 StRpsIdx = short\_term\_ref\_pic\_set\_idx (7‑36)  
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 sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] – NumNegativePics[ StRpsIdx ] – NumPositivePics[ StRpsIdx ] , inclusive. When not present, the value of num\_long\_term\_pics is inferred to be equal to 0.

**poc\_lsb\_lt**[ i ] specifies 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. The length of the poc\_lsb\_lt[ i ] syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits. For any values of j and k in the range of 0 to num\_long\_term\_pics – 1, inclusive, if j is less than k, poc\_lsb\_lt[ j ] shall not be less than poc\_lsb\_lt[ k ].

**delta\_poc\_msb\_present\_flag[ i ]** equal to 1 specifies that delta\_poc\_msb\_cycle\_lt\_minus1[ i ] is present. delta\_poc\_msb\_present\_flag[ i ]equal to 0 specifies that delta\_poc\_msb\_cycle\_lt\_minus1[ i ] is not present. delta\_poc\_msb\_present\_flag[ i ] shall be equal to 1 when there is more than one reference picture in the decoded picture buffer with the least significant bits of the picture order count value equal to poc\_lsb\_lt[ i ].

**delta\_poc\_msb\_cycle\_lt**[ i ]is used to determine the value of the most 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.

The variable DeltaPocMSBCycleLt[ i ] is derived as follows.

if( i = = 0 | | poc\_lsb\_lt[ i − 1 ] ! = poc\_lsb\_lt[ i ] )   
 DeltaPocMSBCycleLt[ i ] = delta\_poc\_msb\_cycle\_lt[ i ]   
 else (7‑37)  
 DeltaPocMSBCycleLt[ i ] = delta\_poc\_msb\_cycle\_lt[ i ] + DeltaPocMSBCycleLt[ i − 1 ]

The value of DeltaPocMSBCycleLt[ i ] \* MaxPicOrderCntLsb + pic\_order\_cnt\_lsb – poc\_lsb\_lt[ i ] shall be in the range of 1 to 224 – 1, 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.

**pic\_temporal\_mvp\_enable\_flag** specifies whether temporal motion vector predictors can be used for inter prediction. If pic\_temporal\_mvp\_enable\_flag is equal to 0, the temporal motion vector predictors shall not be used in decoding of the current picture. If pic\_temporal\_mvp\_enable\_flag is equal to 1, temporal motion vector predictors may be used in decoding of the current picture. When not present, the value of pic\_temporal\_mvp\_enable\_flag shall be inferred to be equal to 0.

When both pic\_temporal\_mvp\_enable\_flag and temporal\_id are equal to 0, all coded pictures that follow the current picture in decoding order shall not use temporal motion vector vectors from any picture that precedes the current picture in decoding order.

**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.

**num\_ref\_idx\_l0\_active\_minus1** specifies the maximum reference index for reference picture list 0 that may be used to decode the slice. num\_ref\_idx\_l0\_active\_minus1 shall be in the range of 0 to 15, inclusive. [Ed. (GJS): Constrain to the value imposed by the max DPB size limit instead of 15. (YK): It should be allowed to use the max number of active entries greater than the max DPB size limit as part of the level definition.]

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 is inferred to be equal to num\_ref\_idx\_l0\_default\_active\_minus1.

**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 is 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.

**mvd\_l1\_zero\_flag** equal to 1 indicates that difference between a list 1 vector component and its prediction, mvd\_l1[][][], is not parsed and it is set equal to 0. mvd\_l1\_zero\_flag equal to 0 indicates that mvd\_l1[][][] is parsed.

**cabac\_init\_flag** specifies the method for determining the initialisation table used in the initialisation process for context variables. The value of cabac\_init\_flag shall be in the range of 0 to 1, inclusive. When cabac\_init\_flag is not present, it is inferred to be 0.

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

SliceQPY = 26 + pic\_init\_qp\_minus26 + slice\_qp\_delta (7‑38)

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

**deblocking\_filter\_override\_flag** equal to 0 specifies that deblocking parameters from the active picture parameter set shall be used for deblocking the current slice. deblocking\_filter\_override\_flag equal to 0 specifies that deblocking parameters from the slice header shall be used for deblocking the current slice. When not present, the value of deblocking\_filter\_override\_flag is inferred to be equal to 1.

**slice\_header\_disable\_deblocking\_filter\_flag** equal to 1 specifies that the operation of the deblocking filter shall not be applied for the current slice. slice\_header\_disable\_deblocking\_filter\_flag equal to 0 specifies that the operation of the deblocking filter shall be applied for the current slice.

**beta\_offset\_div2** and **tc\_offset\_div2** specify the deblocking parameter offsets for β and tC (divided by 2) for the current slice.

**slice\_sample\_adaptive\_offset\_flag[** cIdx **]** equal to 1 specifies that SAO is enabled for the colour component cIdx in the current slice; equal to 0 specifies that SAO is disabled for the colour component cIdx in the current slice. When slice\_sample\_adaptive\_offset\_flag[ cIdx ] is not present, slice\_sample\_adaptive\_offset\_flag[ cIdx ] is inferred to be equal to 0.

**collocated\_from\_l0\_flag** equal to 1 specifies the picture that contains the collocated partition shall be derived from list 0, otherwise the picture shall be derived from list 1.When collocated\_from\_l0\_flag is not present, it is inferred to be equal to 1.

**collocated\_ref\_idx** specifies the reference index of the picture that contains the collocated partition. When the current slice is a P slice collocated\_ref\_idx refers to list 0. When the current slice is a B slice collocated\_ref\_idx refers to list 0 if collocated\_from\_l0 is 1, otherwise it refers to list 1. Collocated\_ref\_idx shall always refer to a valid list entry, and the resulting picture shall be the same for all slices of a coded picture. When collocated\_ref\_idx is not present, it is inferred to be equal to 0.

**five\_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 − five\_minus\_max\_num\_merge\_cand (7‑39)

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

**slice\_adaptive\_loop\_filter\_flag** equal to 1 specifies that the ALF is on for the current slice; equal to 0 specifies that the ALF is off for the current slice. It is a requirement of the bitstream that if there is an active APS, the value of slice\_adaptive\_loop\_filter\_flag shall be equal to aps\_adaptive\_loop\_filter\_flag.

**slice\_loop\_filter\_across\_slices\_enabled\_flag** equal to 1 specifies that in-loop filtering operations are performed across slice boundaries; otherwise, the in-loop operations are slice-independent and not applied across slice boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter. When slice\_loop\_filter\_across\_slices\_enabled\_flag is not present, it is inferred to be equal to seq\_loop\_filter\_across\_slices\_enabled\_flag.

**num\_entry\_point\_offsets** specifies the number of entry\_point\_offset[ i ] syntax elements in the slice header. When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1, the value of num\_entry\_point\_offsets shall be in the range of 0 to ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ) − 1, inclusive. When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2, the value of num\_entry\_point\_offsets shall be in the range of 0 to PicHeightInCtbs − 1, inclusive. When not present, the value of num\_entry\_point\_offsets is inferred to be equal to 0.

**offset\_len\_minus1** plus 1 specifies the length, in bits, of the entry\_point\_offset[ i ] syntax elements.

**entry\_point\_offset**[ i ] specifies the i-th entry point offset, in bytes and shall be represented by offset\_len\_minus1 plus 1 bits. The coded slice data after the slice header consists of num\_entry\_point\_offsets + 1 subsets, with subset index values ranging from 0 to num\_entry\_point\_offsets, inclusive. Subset 0 consists of bytes 0 to entry\_point\_offset[ 0 ] − 1, inclusive, of the coded slice data, subset k, with k in the range of 1 to num\_entry\_point\_offsets - 1, inclusive, consists of bytes entry\_point\_offset[ k − 1 ] to entry\_point\_offset[ k ] + entry\_point\_offset[ k − 1 ] − 1, inclusive, of the coded slice data, and the last subset (with subset index equal to num\_entry\_point\_offsets) consists of the remaining bytes of the coded slice data.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1 and num\_entry\_point\_offsets is greater than 0, each subset shall contain all coded bits of exactly one tile, and the number of subsets (i.e., the value of num\_entry\_point\_offsets + 1) shall be equal to or less than the number of tiles in the slice.

NOTE 2 – When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1, each slice must include either a subset of one tile (in which case signalling of entry points is unnecessary) or an integer number of complete tiles.

When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2 and num\_entry\_point\_offsets is greater than 0, each subset k with k in the range of 0 to num\_entry\_point\_offsets − 1, inclusive, shall contain all coded bits of exactly one row of coding tree blocks, the last subset (with subset index equal to num\_entry\_point\_offsets) shall contain all coded bits of the remaining coding blocks included in the slice, wherein the remaining coding blocks consist of either exactly one row of coding tree blocks or a subset of one row of coding tree blocks, and the number of subsets (i.e., the value of num\_entry\_point\_offsets + 1) shall be equal to the number of rows of coding tree blocks in the slice, wherein a subset of one row of coding tree blocks in the slice is also counted..

NOTE 3 – When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2, a slice may include a number of rows of coding tree blocks and a subset of a row of coding tree blocks. For example, if a slice include two and a half rows of coding tree blocks, the the number of subsets (i.e., the value of num\_entry\_point\_offsets + 1) shall be equal to 3.

**slice\_header\_extension\_length** specifies the length of the slice header extension data in bytes, not including the bits used for signalling slice\_header\_extension\_length itself. The value of slice\_header\_extension\_length shall be in the range of 0 to 256, inclusive.

**slice\_header\_extension\_data\_byte** may have any value. Decoders shall ignore the value of slice\_header\_extension\_data\_byte. Its value does not affect decoder conformance to profiles specified in this Recommendation | International Standard.

#### Short-term reference picture set semantics

A short-term reference picture set may be present in a sequence 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 sequence 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 sequence 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) (7‑40)

**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) (7‑41)

**used\_by\_curr\_pic\_flag**[ j ] indicates whether a picture is used for reference by the current picture.

**use\_delta\_flag[ j ]** specifies whether there is a corresponding i-th reference picture for determining the value of DeltaPoc[ Rdx ][ j ]. When use\_delta\_flag[ j ] is not present, it is inferred to be equal to 1.

Table ‑ – used\_by\_curr\_pic\_flag[ j ] and use\_delta\_flag[ j ] operations for the reference picture set

|  |  |  |
| --- | --- | --- |
| used\_by\_curr\_pic\_flag[ j ] | use\_delta\_flag[ j ] | **Properties of i-th reference picture** |
| 1 | inferred | 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 determining the value of 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( used\_by\_curr\_pic\_flag[ j ] | | use\_delta\_flag[ j ] ) {  
 DPoc = DeltaPoc[ RIdx ][ j ] + DeltaRPS  
 if( DPoc < 0 ) {  
 DeltaPocS0[ idx ][ i0 ] = DPoc (7‑42)  
 UsedByCurrPicS0[ idx ][ i0 ] = used\_by\_curr\_pic\_flag[ j ] (7‑43)  
 i0++  
 } else {  
 DeltaPocS1[ idx ][ i1 ] = DPoc (7‑44)  
 UsedByCurrPicS1[ idx ][ i1 ] = used\_by\_curr\_pic\_flag[ j ] (7‑45)  
 i1++  
 }  
 }  
NumNegativePics[ idx ] = i0 (7‑46)  
NumPositivePics[ idx ] = i1 (7‑47)

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

The elements of the lists DeltaPocS0[ idx ] and UsedByCurrPicS0[ idx ] are sorted together by the decoder, such that DeltaPocS0[ idx ][ i0 ] > DeltaPocS0[ idx ][ i0 + 1 ] for i0 in the range of 0 to NumNegativePics[ idx ] − 2.

The elements of the lists DeltaPocS1[ idx ] and UsedByCurrPicS1[ idx ] are sorted together by the decoder, such that DeltaPocS1[ idx ][ i1 ] < DeltaPocS1[ idx ][ i1 + 1 ] for i1 in the range of 0 to NumPositivePics[ idx ] − 2.

[Ed. (GJS): The reason for the language "sorted together" is that the list of flags needs to be reordered the same way that the list of delta pocs needs to be reordered. The reason for the language "by the decoder" is to make it clear that the sorting is an action, not just a property that will already be true at that point, and not an expression of a constraint on the encoder. This phrasing is not ideal, but it is an improvement.]

**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 sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ], inclusive.

The variable NumNegativePics[ idx ] is derived as follows.

NumNegativePics[ idx ] = num\_negative\_pics (7‑48)

**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 sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] − num\_negative\_pics, inclusive.

The variable NumPositivePics[ idx ] is derived as follows.

NumPositivePics[ idx ] = num\_positive\_pics (7‑49)

**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 ) (7‑50)  
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 ] (7‑51)

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 (7‑52)  
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 ] (7‑53)

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 ] (7‑54)  
for( i = 0; i < NumPositivePics[ idx ]; i++ )  
 DeltaPoc[ idx ][ i + NumNegativePics[ idx ] ] = DeltaPocS1[ idx ][ i ] (7‑55)

NumDeltaPocs[ idx ] = NumNegativePics[ idx ] + NumPositivePics[ idx ] (7‑56)

#### Reference picture list modification semantics

**ref\_pic\_list\_modification\_flag\_lX** (with X equal to 0 or 1) equal to 1 indicates that reference picture list X is specified explicitly as a list of list\_entry\_lX[ i ] values (with X being 0 or 1). ref\_pic\_list\_modification\_flag\_lX equal to 0 indicates that reference picture list X is determined implicitly. When ref\_pic\_list\_modification\_flag\_lX is not present, it is inferred to be equal to 0.

**list\_entry\_lX**[ i ](with X equal to 0 or 1) specifies the index of the reference picture in RefPicSetCurrTempListX to be placed at the current position of reference picture list LX (with X being 0 or 1). The length of the list\_entry\_lX[ i ] syntax element is Ceil( Log2( NumPocTotalCurr) ) bits. The value of list\_entry\_lX[ i ] shall be in the range of 0 to NumPocTotalCurr − 1, inclusive. If the syntax element list\_entry\_lX[ i ] is not present, it is inferred to be equal to 0. [Ed. (GJS): Unfortunately, NumPocTotalCurr is derived in the decoding process section. Properly, some restructuring of the text should be done to move its derivation to the semantics section, so that the parsing process does not need to reach into the decoding process in order to do its job.]

#### Adaptive loop filter parameter semantics

The variable NumALFFiltersInStoredBuffer[ cIdx ] is set equal to 0 for every colour component cIdx equal to 0, 1 and 2.

**alf\_cb\_enable\_flag** equal to 1 specifies that adaptive loop filter is applied on Cb component; equal to 0 specifies that adaptive loop filter is disabled on Cb component.

**alf\_cr\_enable\_flag** equal to 1 specifies that adaptive loop filter is applied on Cr component; equal to 0 specifies that adaptive loop filter is disabled on Cr component.

**alf\_one\_luma\_unit\_per\_slice\_flag** equal to 1 specifies that all of coding tree blocks in luma component of current slice are filtered by the same ALF parameter set; equal to 0 specifies that ALF parameter sets in luma component are adaptive on a coding tree block basis.

**alf\_one\_cb\_unit\_per\_slice\_flag** equal to 1 specifies that all of coding tree blocks in Cb component of current slice are filtered by the same ALF parameter set; equal to 0 specifies that ALF parameter sets in Cb component are adaptive on a coding tree block basis.

**alf\_one\_cr\_unit\_per\_slice\_flag** equal to 1 specifies that all of coding tree blocks in Cr component of current slice are be filtered by the same ALF parameter set; equal to 0 specifies that ALF parameter sets in Cr component are adaptive on a coding tree block basis.

**alf\_num\_lcu\_in\_width\_minus1** plus 1 specifies the number of coding tree blocks of picture width.

* If alf\_num\_lcu\_in\_width\_minus1 is present, the variable numCtbInWidth is set to (alf\_num\_lcu\_in\_width\_minus1 + 1).
* Otherwise (alf\_num\_lcu\_in\_width\_minus1 is not present), numCtbInWidth is set to PicWidthInCtbs.

**alf\_num\_lcu\_in\_height\_minus1** plus 1 specifies the number of coding tree blocks of picture height. The variable numCtbInHeight is set to (alf\_num\_lcu\_in\_height\_minus1+ 1).

* If alf\_num\_lcu\_in\_height\_minus1 is present, the variable numCtbInHeight is set to (alf\_num\_lcu\_in\_height\_minus1 + 1).
* Otherwise (alf\_num\_lcu\_in\_height\_minus1 is not present), numCtbInHeight is set to PicHeightInCtbs.

**alf\_num\_lcu\_in\_slice\_minus1** plus 1 specifies the number of coding tree blocks in the current slice. The length of the alf\_num\_lcu\_in\_slice\_minus1 syntax element is Ceil( Log2(PicWidthInCtbs\* PicHeightInCtbs) ) bits.

* If alf\_num\_lcu\_in\_slice\_minus1 is present, numCtb is set equal to (alf\_num\_lcu\_in\_slice\_minus1 + 1), alfAcrossSlice is set to slice\_loop\_filter\_across\_slices\_enabled\_flag, and firstCtbAddr is set to SliceCtbAddrRS.
* Otherwise (alf\_num\_lcu\_in\_slice\_minus1 is not present), numCtb is set equal to (numCtbInWidth\*numCtbInHeight), alfAcrossSlice is set to 1, and firstCtbAddr is set to 0.

**alf\_repeat\_row\_flag**[ cIdx ]equal to 1 specifies that the ALF parameter sets of coding tree blocks in the current coding tree block row shall be the same as those of the above coding tree blocks for the colour component cIdx; equal to 0 specifies that the current coding tree block row has new ALF parameter sets for the colour component cIdx. When alf\_repeat\_row\_flag[ cIdx ] is not present, it is inferred to be 0.

#### Adaptive loop filter unit semantics

The number of times the ALF parameters corresponding to an coding tree block are repeated for subsequent coding tree blocks in the same row is represented by alfRun[ cIdx ][ rx ][ ry ]. The array index cIdx specifies the colour component; cIdx is equal to 0 for luma, equal to 1 for Cb, and equal to 2 for Cr. The array indices rx and ry specify the location ( rx, ry ) of the considered coding tree block relative to the top-left coding tree block of the picture.

**alf\_lcu\_enable\_flag**[ cIdx ][ ry ][ rx ]equal to 1 specifies that the adaptive loop filter shall be applied on current coding tree block at position rx and ry in the colour component cIdx; equal to 0 specifies that adaptive loop filter shall be disabled for the current coding tree block at position rx and ry in the colour component cIdx.

**alf\_run\_diff** specifies the alfRun of current coding tree block if the current row is the first row, otherwise specifies the difference between the run of current coding tree block and the run of the above coding tree block. When alfRun is greater than or equal to 0, the syntax elements in alf\_info() are derived from the corresponding syntax elements of the left coding tree block. The length of the alf\_run\_diff syntax element is Ceil( Log2(alf\_num\_lcu\_in\_width\_minus1 − rx + 2) ) bits.

**alf\_merge\_up\_flag** equal to 1 specifies that the syntax elements in alf\_info() are derived from the corresponding syntax elements of the above coding tree block; equal to 0 specifies that the syntax elements in alf\_info() are not derived from the corresponding syntax elements of the above coding tree block. When alf\_merge\_up\_flag is not present, it is inferred to be equal to 0.

#### Adaptive loop filter info semantics

**alf\_new\_filter\_set\_flag** equal to 1 specifies that the current coding tree block uses a new filter set; equal to 0 specifies that the current coding tree block uses the stored filter set with the buffer index equal to alf\_stored\_filter\_set\_idx[ cIdx ] of the colour component cIdx.

When alf\_new\_filter\_set\_flag is not present, it is inferred to be 1.

When alf\_new\_filter\_set\_flag is equal to 1, NumALFFiltersInStoredBuffer[cIdx] is increased by 1.

**alf\_stored\_filter\_set\_idx**[ cIdx ]specifies the buffer index of the stored filters for the colour component cIdx. The length of the alf\_stored\_filter\_set\_idx[ cIdx ] syntax element is Floor( Log2(Min( 1 , NumALFFiltersInStoredBuffer[cIdx] − 1) ) ) + 1 bits.

**alf\_no\_filters\_minus1** is used to derive the number of filter sets for the current coding tree block. It is a requirement of bitstream conformance that the values of alf\_no\_filters\_minus1 are in the range of 0 to 2, inclusive.

**alf\_start\_second\_filter** specifies the block adaptive (BA) mode index of luma samples for which the second filter is applied.

**alf\_filter\_pattern\_flag**[ cIdx ][ ry ][ rx ][ i ] equal to 1 specifies that the filter index for the i-th BA mode index is increased by 1: equal to 0 specifies that the filter index for the i-th BA mode index is the same as i − 1 BA mode index.

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 (7‑57)

* Otherwise (alf\_no\_filters\_minus1 is equal to 2)

AlfNumFilters = Σi alf\_filter\_pattern\_flag[ i ] with i = 0..15 (7‑58)

The mapping between the i-th block adaptive (BA) mode index for luma samples and the filter index MapFiltFidx[ ry ][ rx ][ i ] specifying is derived as follows:

* If AlfNumFilters is equal to 1,

MapFiltFidx[ ry ][ rx ][ i ] = 0 with i = 0 .. 15 (7‑59)

* Otherwise, if AlfNumFilters is equal to 2,

MapFiltFidx[ ry ][ rx ][ i ] = 0 with i = 0 .. alf\_start\_second\_filter−1 (7‑60)  
MapFiltFidx[ ry ][ rx ] [ i ] = 1 with i = alf\_start\_second\_filter .. 15

* Otherwise (AlfNumFilters is greater than 2),

MapFiltFidx[ ry ][ rx ][ 0 ] = 0  
MapFiltFidx[ ry ][ rx ][ i+1 ] = MapFiltFidx[ ry ][ rx ][ i ] + alf\_filter\_pattern[ ry ][ rx ][ i − 1 ] (7‑61)  
 with i = 1..15

**alf\_pred\_flag**[ cIdx ][ ry ][ rx ] equal to 1 specifies that the filter coefficients for the current coding tree block with at location ( rx, ry ) for the colour component cIdx are coded in a predictive way; equal to 0 specifies that the filter coefficients are coded independently.

**alf\_min\_kstart\_minus1** plus 1 specifies the minimum order k of k-th order exponential golomb code for the luma filter coefficients for the adaptive loop filter.

**alf\_golomb\_index\_flag[ i ]** specifies the difference in the orders of k-th order exponential golomb codes between i-th group and (i+1)-th group of the luma filter coefficients. There are several groups of the luma filter coefficients where each group may have a different order k. The group value of the j-th luma filter coefficient alfCoeffGroup[ j ] is derived as follows.

alfCoeffGroup[ j ] = { 2, 3, 3, 4, 3, 1, 2, 3, 4, 1 } with j = 0..9 (7‑62)

The table of different orders k of the j-th coefficient alfkVal[ alfCoeffGroup[ j ] ] with j=0..9 is derived as follows.

alfkVal[ i ] = (alf\_min\_kstart\_minus1 +1)+ Σr alf\_golomb\_index\_flag[r] with i= 1..3 and r= 1.. i (7‑63)

The k-th order of j-th alf\_filter\_coeff alfkMap[j] is derived as follows.

alfkMap[j] = alfkVal[ alfCoeffGroup[ j ] ] with j = 0..9

[Ed. (WJ): this k is used for input to parse alf\_filter\_coeff [ ] coded as ge(v), but not considered yet]

When alf\_golomb\_index\_flag [i ] is not present, the alfkMap[ j ] = alfCoeffGroup[ j ]

**alf\_nb\_pred\_luma\_flag**[ cIdx ][ ry ][ rx ][ i ] equal to 1 specifies whether the filter coefficients of i-th filter for the current coding tree block at location ( rx, ry ) for the colour component cIdx are coded in a predictive way using spatially neighbouring filter coefficients; euqal to 0 specifies that their are not coded using spatially neighbouring filter coefficients.

**alf\_filt\_coeff**[ cIdx ][ ry ][ rx ][ i ][ j ] specifies the j-th filter coefficient of i-th filter used in the adaptive loop filtering process for the current coding tree block at location ( rx, ry ) for the colour component cIdx.

#### 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 0, the filtering process shall be applied to all luma samples in the current slice, otherwise, the filtering process shall be applied only when alf\_cu\_flag is equal to 1.

**alf\_cu\_control\_max\_depth** specifies the maximum split depth from tree block 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 syntax element, where the variable specifying the number of alf\_cu\_flag shall be computed as

NumAlfCuFlag = alf\_length\_cu\_control\_info + ( PicHeightInCtbs \* PicWidthInCtbs ) (7‑64)

**alf\_cu\_flag**[ i ] specifies the information whether adaptive loop filtering is applied to coding bock i, where i is between 0 and ( NumAlfCuFlag − 1 ).

#### 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 ] is 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 ] is 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 ] is 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 ) (7‑65)

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 ] is 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.

### Slice data semantics

**end\_of\_slice\_flag** equal to 0 specifies that another coding block is following in the slice. end\_of\_slice\_flag equal to 1 specifies the end of the slice and that no further coding block follows.

#### Sample adaptive offset parameter semantics

**sao\_merge\_left\_flag** equal to 1 specifies that the syntax elements sao\_type\_idx, sao\_band\_position, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the left coding tree block; equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the left coding tree block. When sao\_merge\_left\_flag is not present, it is inferred to be equal to 0.

**sao\_merge\_up\_flag** equal to 1 specifies that the syntax elements sao\_type\_idx, sao\_band\_position, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the above coding tree block; equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above coding tree block. When sao\_merge\_up\_flag is not present, it is inferred to be equal to 0.

**sao\_type\_idx**[ cIdx ][ rx ][ ry ] indicates the offset type as specified in Table 7‑9**Error! Reference source not found.** of current coding tree block at position rx and ry for the colour component cIdx.

When sao\_type\_idx[ cIdx ][ rx ][ ry ] is not present, it is inferred as follows.

* If sao\_merge\_left\_flag is equal to 1, sao\_type\_idx[ cIdx ][ rx ][ ry ] is set equal to sao\_type\_idx[ cIdx ][ rx − 1 ][ ry  ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_type\_idx[ cIdx ][ rx ][ ry ] is set equal to sao\_type\_idx[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, sao\_type\_idx[ cIdx ][ rx ][ ry ] is set equal to 0.

Table 7‑9 – Specification of the SAO type

|  |  |
| --- | --- |
| **sao\_type\_idx[ cIdx ][ rx ][ ry ]** | **SAO type (informative)** |
| 0 | Not applied |
| 1 | 1D 0-degree edge offset |
| 2 | 1D 90-degree edge offset |
| 3 | 1D 135-degree edge offset |
| 4 | 1D 45-degree edge offset |
| 5 | Band offset |

**sao\_band\_position**[ cIdx ][ rx ][ ry ] indicates the displacement of the band offset of the sample range when sao\_type\_idx[ cIdx ][ rx ][ ry ] is equal to 5.

When sao\_band\_position[ cIdx ][ rx ][ ry ] is not present it is inferred as follows.

* If sao\_merge\_left\_flag is equal to 1, sao\_band\_position[ cIdx ][ rx ][ ry ] is set equal to sao\_band\_position[ cIdx ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_band\_position[ cIdx ][ rx ][ ry ] is set equal to sao\_band\_position[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, sao\_band\_position[ cIdx ][ rx ][ ry ] is set equal to 0.

**sao\_offset\_abs**[ cIdx ][ rx ][ ry ][ i ] indicates the offset value of i-th category of current coding tree block at position rx and ry for the colour component cIdx.

When sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is not present, it is inferred as follows.

* If sao\_merge\_left\_flag is equal to 1, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is set equal to sao\_offset\_abs[ cIdx ][ rx − 1 ][ ry ][ i ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is set equal to sao\_offset\_abs[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is set equal to 0.

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.

**sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ]specifies the sign of sao\_offset[ cIdx ][ rx ][ ry ][ i ] when sao\_type\_index is equal to 5.

When sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is not present, it is inferred as follows.

* If sao\_merge\_left\_flag is equal to 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is set equal to sao\_offset\_sign[ cIdx ][ rx − 1 ][ ry ][ i ]
* Otherwise, if sao\_merge\_up\_flag is equal to 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is set equal to sao\_offset\_sign[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is set equal 0.

The variable offsetSign is derived as follows.

* If sao\_type\_idx[ cIdx ][ rx ][ ry ] is less than 5 and i is greater than 1,offsetSign is set equal to −1.
* Otherwise, if sao\_type\_idx[ cIdx ][ rx ][ ry ] is less than 5 and i is less than 2, offsetSign is set equal to 1.
* Otherwise, if sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is equal to 0, offsetSign is set to equal to 1.
* Otherwise, offsetSign is set to equal to −1.

The array SaoOffsetVal is derived as follows.

SaoOffsetVal[ cIdx ][ rx ][ ry ][ 0 ] = 0 (7‑66)

SaoOffsetVal[ cIdx ][ rx ][ ry ][ i + 1 ] =   
 offsetSign\*sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] << ( bitDepth – Min( bitDepth, 10 ) ) (7‑67)

### 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 log2CbSize is greater than Log2MinCbSize, the value of split\_coding\_unit\_flag[ x0 ][ y0 ] is inferred to be equal to 1.

– Otherwise (log2CbSize is equal to Log2MinCbSize), the value of split\_coding\_unit\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

### Coding unit semantics

**cu\_transquant\_bypass\_flag** equal to 1 specifies that the scaling and transform process as specified in subclause and the in-loop filter process as specified in subclause are bypassed. When cu\_transquant\_bypass\_flag is not present, it is inferred to be equal to 0.

**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 merging candidate index merge\_idx[ x0 ][ y0 ] 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 is 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 .

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 log2CbSize is greater than Log2MinCbSize, part\_mode shall be in the range of 0 to 2, inclusive and in the range of 4 to 7, inclusive.
* Otherwise, if log2CbSize 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 (log2CbSize 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.

**num\_subsequent\_pcm** specifies the number of subsequent I\_PCM coding units with the current log2CbSize that successively follow the current I\_PCM coding unit in the same depth of a TB. The values of pcm\_flags of the subsequent coding units are set equal to 1. It is a requirement of bitstream conformance that the immediate roots of the current and subsequent I\_PCM coding units are identical. The value num\_subsequent\_pcm shall be in the range of 0 to 3, inclusive.

**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 7‑10 – 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 |

**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 is inferred to be equal to 0.

### 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 is 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.

**prev\_intra\_luma\_pred\_flag**[ x0 ][ y0 ], **mpm\_idx[** 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 .

**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\_idc**[ x0 ][ y0 ] specifies whether list0, list1 or bi-prediction is used for the current prediction unit according to . 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 7‑11 – Name association to inter prediction mode

|  |  |  |
| --- | --- | --- |
| **slice\_type** | **inter\_pred\_idc** | **Name of inter\_pred\_idc** |
| P | inferred | Pred\_L0 |
| B | 0 | Pred\_L0 |
| 1 | Pred\_L1 |
|  | 2 | Pred\_BI |

When inter\_pred\_idc[ x0 ][ y0 ] is not present, it is 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 it is inferred to be equal to 0.

**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.

**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.

#### 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 luma inter transform block split direction derivation process as specified in 6.6 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionL.

The chroma inter transform block split direction derivation process as specified in 6.7 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionC.

**split\_transform\_flag**[ x0L ][ y0L ][ 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 x0L, y0L specify the location ( x0L, y0L ) 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.

The variable interSplitFlag is derived as follows.

* If max\_transform\_hierarchy\_depth\_inter is equal to 0 and PredMode is equal t oMODE\_INTER and PartMode is not equal to PART\_2Nx2N and trafoDepth is equal to 0, interSplitFlag is set to 1.
* Otherwise, interSplitFlag is set to 0.

When split\_transform\_flag[ x0L ][ y0L ][ trafoDepth ] is not present, it is inferred as follows:

* If one or more of the following conditions are true, the value of split\_transform\_flag[ x0L ][ y0L ][ trafoDepth ] is inferred to be equal to 1.
* log2TrafoSize is greater than Log2MaxTrafoSize
* IntraSplitFlag is equal to 1 and trafoDepth is equal to 0
* interSplitFlag is equal to 1
* Otherwise, the value of split\_transform\_flag[ x0L ][ y0L ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_luma**[ x0L ][ y0L ][ trafoDepth ] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0L, y0L specify the location ( x0L, y0L ) 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**[ x0C ][ y0C ][ trafoDepth ] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0C, y0C specify the location ( x0C, y0C ) 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[ x0C ][ y0C ][ trafoDepth ] is not present, the value of cbf\_cb[ x0C ][ y0C ][ trafoDepth ] is inferred as follows.

* If trafoDepth is greater than 0 and log2TrafoSize is equal to 2, cbf\_cb[ x0C ][ y0C ][ trafoDepth ] is inferred to be equal to cbf\_cb[ xBase ][ yBase ][ trafoDepth − 1 ]
* Otherwise, cbf\_cb[ x0C ][ y0C ][ trafoDepth ] is inferred to be equal to 0.

**cbf\_cr**[ x0C ][ y0C ][ trafoDepth ] equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices x0C, y0C specify the location ( x0C, y0C ) 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[ x0C ][ y0C ][ trafoDepth ] is not present, the value of cbf\_cr[ x0C ][ y0C ][ trafoDepth ] is inferred as follows.

* If trafoDepth is greater than 0 and log2TrafoSize is equal to 2, cbf\_cr[ x0C ][ y0C ][ trafoDepth ] is inferred to be equal to cbf\_cr[ xBase ][ yBase ][ trafoDepth − 1 ]
* Otherwise, cbf\_cr[ x0C ][ y0C ][ 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 7‑12 – Specification of ScanType[ log2TrafoSize − 2 ][ IntraPredMode ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IntraPredMode** | **log2TrafoSize − 2** | | | |
| **0** | **1** | **2** | **3** |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 – 5 | 0 | 0 | 0 | 0 |
| 6 – 14 | 2 | 2 | 0 | 0 |
| 15 – 21 | 0 | 0 | 0 | 0 |
| 22 – 30 | 1 | 1 | 0 | 0 |
| 31 – 35 | 0 | 0 | 0 | 0 |

**cu\_qp\_delta** specifies the difference between a luma quantization parameter to be used and its prediction for a coding unit quantization group where the coding unit quantization group is specified as follows:

* If a coding unit with the split\_coding\_unit\_flag[ x0 ][ y0 ] equal to 0 and the log2CbSize is greater than or equal to Log2MinCUDQPSize, the coding unit 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 log2CbSize is equal to Log2MinCUDQPSize, the coding unit 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 is inferred to be equal to 0 when it is not present for any coding unit quantization group.

The luma quantization parameter QP’Y and the chroma quantization parameters QP’Cb and QP’Cr are derived by invoking the process specified in subclause with ( x0 ,y0 ) as inputs.

### Residual coding semantics

**transform\_skip\_flag**[ x0 ][ y0 ][ cIdx ] specifies whether a transform is applied to the associated transform block or not: 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. transform\_skip\_flag[ x0 ][ y0 ][ cIdx ] equal to 1, specifies that no transform will be applied to the current transform block. When transform\_skip\_flag[ x0 ][ y0 ][ cIdx ] is not present, it is inferred to be equal to 0.

**last\_significant\_coeff\_x\_prefix** specifies 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** specifies 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** specifies 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 (7‑68)

* 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)) + (7‑69)  
 last\_significant\_coeff\_x\_suffix

**last\_significant\_coeff\_y\_suffix** specifies 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 (7‑70)

* 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)) + (7‑71)   
 last\_significant\_coeff\_y\_suffix

When scanIdx is equal to 2, the coordinates are swapped as follows.

temp = LastSignificantCoeffX  
LastSignificantCoeffX = LastSignificantCoeffY (7‑72)  
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 an array of 16 transform coefficient levels 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, 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 or more of the following conditions are 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 )
* Otherwise, 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 or both of the following conditions are true, significant\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 1
* ( xC, yC ) is equal to ( xCG << 2, yCG << 2 )
* implicitNonZeroCoeff is equal to 1
* Otherwise, 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\_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.

**coeff\_abs\_level\_remaining[** n **]** is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position n. The value of coeff\_abs\_level\_remaining is constrained by the limits in subclause .When coeff\_abs\_level\_remaining [ n ] is not present, it is inferred as 0.

# Decoding process

## General [Ed. (GJS): Create similar subclauses for other places where there is "orphaned" text that is at the parent level of a section that contains subclauses.]

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 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 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.

The decoding process operates as follows:

1. The decoding of NAL units is specified in subclause .
2. The processes in subclause  specify decoding processes using syntax elements in the slice layer and above:

– Variables and functions relating to picture order count are derived in subclause  (only needed to be invoked for one slice of a picture).

– The decoding process for reference picture set in subclause  is invoked, wherein reference pictures may be marked as "unused for reference" (only needed to be invoked for one slice of a picture).

– PicOutputFlag is set as follows:

– If the first coded picture in the bitstream is a CRA picture, and the current picture is a TFD picture associated with the CRA picture, or if the previous RAP picture preceding the current picture in decoding order is a BLA picture and the current picture is a TFD picture associated with the BLA picture, PicOutputFlag is set equal to 0 and the decoding process for generating unavailable reference pictures specified in subclause  is invoked (only needed to be invoked for one slice of a picture).

– Otherwise, PicOutputFlag is set equal to pic\_output\_flag.

– At the beginning of the decoding process for each P or B slice, the decoding process for reference picture lists construction specified in subclause 8.3.4 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", otherwise (it is a non-reference picture) the decoded picture is marked as "unused for reference", as specified in subclause 8.3.5.

1. The processes in subclauses , , , and specify decoding processes using syntax elements in the coding tree unit 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  describes the decoding process for VCL NAL units.

NAL units with nal\_unit\_type equal to 25, 26, 27 and 28 contain video parameter sets, sequence parameter sets, picture parameter sets and adaptation parameter sets, respectively. 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 or buffereing period SEI messages. 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. Adaptation parameter sets may be used in the decoding processes of other NAL units as determined by reference to an adaptation parameter set within the slice headers.

## 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 merge mode and motion vector prediction, to represent picture order differences between pictures for motion vector derivation, for implicit mode weighted prediction in B slices (see subclause ), and for decoder conformance checking (see subclause ).

Each coded picture is associated with one picture order count, denoted as PicOrderCntVal.

When none of the following conditions is true:

* The current picture is an IDR
* The current picture is a BLA picture
* The current picture is a CRA picture and is the first coded picture in the bitstream

the variables prevPicOrderCntLsb and prevPicOrderCntMsb are derived as follows. Let prevRefPic be the previous reference picture in decoding order that has temporal\_id equal to or less than the temporal\_id of the current picture. The variable prevPicOrderCntLsb is set equal to pic\_order\_cnt\_lsb of prevRefPic, and the variable prevPicOrderCntMsb is set equal to PicOrderCntMsb of prevRefPic.

The variable PicOrderCntMsb of the current picture is derived as follows.

* If the current picture is an IDR or a BLA picture, or if the first coded picture in the bitstream is a CRA picture and the current picture is the first coded picture in the bitstream, PicOrderCntMsb is set equal to 0.
* Otherwise, PicOrderCntMsb 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 (8‑1)  
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 (8‑2)

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 −263 to 263 − 1, inclusive. In one coded video sequence, the PicOrderCntVal values for any two coded pictures shall be different.

NOTE 2 – Retention of only the 32 least significant bits of PicOrderCntVal is sufficient for proper decoder operation, enabling unique identification of pictures.

The function PicOrderCnt( picX ) is specified as follows:

PicOrderCnt( picX ) = PicOrderCntVal of the picture picX (8‑3)

The function DiffPicOrderCnt( picA, picB ) is specified as follows:

DiffPicOrderCnt( picA, picB ) = PicOrderCnt( picA ) − PicOrderCnt( picB ) (8‑4)

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 3 – 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 4 – Many encoders assign PicOrderCntVal proportional to the sampling time of the corresponding picture relative to the sampling time of the previous IDR or BLA 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 . 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 signalling 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; PocStCurrBefore, PocStCurrAfter, PocStFoll, PocLtCurr, and PocLtFoll with NumPocStCurrBefore, NumPocStCurrAfter, NumPocStFoll, NumPocLtCurr, and NumPocLtFoll number of elements, respectively.

* If the current picture is an IDR or BAL picture, PocStCurrBefore, PocStCurrAfter, PocStFoll, PocLtCurr, and PocLtFoll are all set to empty, and NumPocStCurrBefore, NumPocStCurrAfter, 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 ] )  
 PocStCurrBefore[ j++ ] = PicOrderCntVal + DeltaPocS0[ StRpsIdx ][ i ]  
 else  
 PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS0[ StRpsIdx ][ i ]  
NumPocStCurrBefore = j  
  
for( i = 0, j = 0; i < NumPositivePics[ StRpsIdx ]; i++ )  
 if( UsedByCurrPicS1[ StRpsIdx ][ i ] )  
 PocStCurrAfter[ j++ ] = PicOrderCntVal + DeltaPocS1[ StRpsIdx ][ i ]  
 else  
 PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS1[ StRpsIdx ][ i ]  
NumPocStCurrAfter = j  
NumPocStFoll = k (8‑5)

for( i = 0, j = 0, k = 0; i < num\_long\_term\_pics; i++ )  
 if( delta\_poc\_msb\_present\_flag[ i ] )  
 if( used\_by\_curr\_pic\_lt\_flag[ i ] )  
 PocLtCurr[ j++ ] = PicOrderCntVal − DeltaPocMSBCycleLt[ i ] \* MaxPicOrderCntLsb −  
 pic\_order\_cnt\_lsb + poc\_lsb\_lt[ i ]  
 else  
 PocLtFoll[ k++ ] = PicOrderCntVal − DeltaPocMSBCycleLt[ i ] \* MaxPicOrderCntLsb −  
 pic\_order\_cnt\_lsb + poc\_lsb\_lt[ i ]  
 else  
 if( used\_by\_curr\_pic\_lt\_flag[ i ] )  
 PocLtCurr[ j++ ] = poc\_lsb\_lt[ i ]   
 else  
 PocLtFoll[ k++ ] = poc\_lsb\_lt[ i ]   
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 sequence 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 sequenceparameter 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; RefPicSetStCurrBefore, RefPicSetStCurrAfter, RefPicSetStFoll, RefPicSetLtCurr and RefPicSetLtFoll. The variable NumPocTotalCurr is set equal to NumPocStCurrBefore + NumPocStCurrAfter + NumPocLtCurr. When decoding a P or B slice, it is a requirement of bitstream conformance that the value of NumPocTotalCurr shall not be equal to 0.

NOTE 3 – RefPicSetStCurrBefore, RefPicSetStCurrAfter 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*.*

The derivation process for the reference picture set and picture marking are performed according to the following ordered steps, where DPB refers to the decoded picture buffer as described in Annex :

1. The following applies:

for( i = 0; i < NumPocLtCurr; i++ ) {  
 if( !delta\_poc\_msb\_present\_flag[ i ] ) {  
 if( there is a long-term reference picture picX in the DPB [Ed. (JB): Should be made more precise.]  
 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"  
 } else {   
 if( there is a long-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocLtCurr[ i ] )  
 RefPicSetLtCurr[ i ] = picX  
 else if( there is a short-term reference picture picY in the DPB  
 with PicOrderCntVal equal to PocLtCurr[ i ] )  
 RefPicSetLtCurr[ i ] = picY  
 else   
 RefPicSetLtCurr[ i ] = "no reference picture"  
 }  
} (8‑6)

for( i = 0; i < NumPocLtFoll; i++ ) {  
 if( !delta\_poc\_msb\_present\_flag[ 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"  
 } else {  
 if( there is a long-term reference picture picX in the DPB  
 with PicOrderCntVal to PocLtFoll[ i ] )  
 RefPicSetLtFoll[ i ] = picX  
 else if( there is a short-term reference picture picY in the DPB  
 with PicOrderCntVal 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 < NumPocStCurrBefore; i++ )  
 if( there is a short-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocStCurrBefore[ i ])  
 RefPicSetStCurrBefore[ i ] = picX  
 else  
 RefPicSetStCurrBefore[ i ] = "no reference picture"

for( i = 0; i < NumPocStCurrAfter; i++ )  
 if( there is a short-term reference picture picX in the DPB  
 with PicOrderCntVal equal to PocStCurrAfter[ i ])  
 RefPicSetStCurrAfter[ i ] = picX  
 else  
 RefPicSetStCurrAfter[ i ] = "no reference picture" (8‑7)

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 RefPicSetStCurrBefore, RefPicSetStCurrAfter 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, RefPicSetStCurrBefore, RefPicSetStCurrAfter 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 not present in the decoded picture buffer. Entries in RefPicSetStFoll or RefPicSetLtFoll that are equal to "no reference picture" should be ignored. Unless either of the following two conditions is true, an unintentional picture loss should be inferred for each entry in RefPicSetStCurrBefore, RefPicSetStCurrAfter and RefPicSetLtCurr that is equal to "no reference picture": a) the first coded picture in the bitstream is a CRA picture and the current coded picture is a TFD picture associated with the first coded picture in the bitstream; b) the previous RAP picture preceding the current coded picture in decoding order is a BLA picture and the current coded picture is a TFD picture associated with the BLA picture.

It is a requirement of bitstream conformance that 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 RefPicSetStCurrBefore, RefPicSetStCurrAfter and RefPicSetLtCurr.
* When the current picture is a TLA picture, there shall be no reference picture included in the reference picture set with temporal\_id greater than or equal to the temporal\_id of the current picture.
* There shall be no reference picture included in the reference picture set that precedes, in output order, any RAP picture that precedes the current picture both in decoding order and output order.
* Unless either of the following conditions is true, there shall be no entry in RefPicSetStCurrBefore, RefPicSetStCurrAfter or RefPicSetLtCurr that is equal to "no reference picture": a) the first coded picture in the bitstream is a CRA picture and the current coded picture is a TFD picture associated with the first coded picture in the bitstream; b) the previous RAP picture preceding the current coded picture in decoding order is a BLA picture and the current coded picture is a TFD picture associated with the BLA picture.

NOTE 5 – A reference picture cannot be included in more than one of the five reference picture set lists. [Ed. (YK): Why is this expressed as a note, not as a normative requirement?]

* + 1. **Decoding process for generating unavailable reference pictures**
       1. **General**

This process is invoked once per coded picture, after the invocation of the decoding process for reference picture set as specified in subclause , when the first coded picture in the bitstream is a CRA picture and the current coded picture is a TFD picture associated with the CRA picture, or when the previous RAP picture preceding the current coded picture in decoding order is a BLA picture and the current coded picture is a TFD picture associated with the BLA picture.

NOTE 1 – The entire specification herein of the decoding process for TFD pictures associated with a CRA picture at the beginning of the bitstream or for TFD pictures associated with a BLA picture is only included for purposes of specifying constraints on the allowed syntax content of such pictures. In actual decoders, any TFD pictures associated with a CRA picture at the beginning of the bitstream or any TFD pictures associated with a BLA picture may simply be ignored (removed from the bitstream and discarded), as they are not specified for output and have no effect on the decoding process of any other pictures that are specified for output.

When the first coded picture in the bitstream is a CRA picture and the current coded picture is a TFD picture associated with the CRA picture, or when the previous RAP picture preceding the current coded picture in decoding order is a BLA picture and the current coded picture is a TFD picture associated with the BLA picture, the following applies.

* For each RefPicSetStCurrBefore[ i ], with i in the range of 0 to NumPocStCurrBefore − 1, inclusive, that is equal to "no reference picture", a reference picture is generated as specified in subclause , and the following applies.
* The value of PicOrderCntVal for the generated reference picture is set to PocStCurrBefore[ i ].
* The value of PicOutputFlag for the generated reference picture is set to 0.
* The generated reference picture is marked as "used for short-term reference".
* RefPicSetStCurrBefore[ i ] is set to be the generated reference picture.
* For each RefPicSetStCurrAfter[ i ], with i in the range of 0 to NumPocStCurrAfter − 1, inclusive, that is equal to "no reference picture", a reference picture is generated as specified in subclause , and the following applies.
* The value of PicOrderCntVal for the generated reference picture is set to PocStCurrAfter[ i ].
* The value of PicOutputFlag for the generated reference picture is set to 0.
* The generated reference picture is marked as "used for short-term reference".
* RefPicSetStCurrAfter[ i ] is set to 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 , and the following applies.
* The value of pic\_order\_cnt\_lsb for the generated reference picture is set to PocLtCurr[ i ].
* The value of PicOutputFlag 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 the generated reference picture.
  + - 1. **Generation of one unavailable reference picture**

When this process is invoked, a unavailable reference picture is generated as follows:

* The value of each element in the sample array SL for the picture is set to 1 << ( BitDepthY − 1 ).
* The value of each element in the sample arrays SCb and SCr for the picture is set to 1 << ( BitDepthC − 1 ).
* The prediction mode PredMode for each minimum coded block of the picture is set to MODE\_INTRA.

### Decoding process for reference picture lists construction

#### General

This process is invoked at the beginning of the decoding process for each P or B slice.

Decoded reference pictures are marked as "used for short-term reference" or "used for long-term reference" as specified in subclause .

Reference pictures are addressed through reference indices as specified in subclause . A reference index is an index into a reference picture list. When decoding a P 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, the reference picture list RefPicList0, and for B slices RefPicList1, are derived as specified as specified in subclause .

#### Initialization process for reference picture lists

This process is invoked when decoding a P or B slice header.

The variable NumRpsCurrTempList0 is set equal to Max( num\_ref\_idx\_l0\_active\_minus1 + 1, NumPocTotalCurr ) and the list RefPicListTemp0 is constructed as follows:

rIdx = 0  
while( rIdx < NumRpsCurrTempList0 ) {  
 for( i = 0; i < NumPocStCurrBefore && rIdx < NumRpsCurrTempList0; rIdx++, i++ )  
 RefPicListTemp0[ rIdx ] = RefPicSetStCurrBefore[ i ]   
 for( i = 0; i < NumPocStCurrAfter && rIdx < NumRpsCurrTempList0; rIdx++, i++ ) (8‑8)  
 RefPicListTemp0[ rIdx ] = RefPicSetStCurrAfter[ i ]  
 for( i = 0; i < NumPocLtCurr && rIdx < NumRpsCurrTempList0; rIdx++, i++ )  
 RefPicListTemp0[ rIdx ] = RefPicSetLtCurr[ i ]  
}

The list RefPicList0 is constructed as follows:

for( rIdx = 0; rIdx ≤ num\_ref\_idx\_l0\_active\_minus1; rIdx++) (8‑9)  
 RefPicList0[ rIdx ] = ref\_pic\_list\_modification\_flag\_l0 ? RefPicListTemp0[ list\_entry\_l0[ rIdx ] ] :  
 RefPicListTemp0[ rIdx ]

When the slice is a B slice, the variable NumRpsCurrTempList1 is set equal to Max( num\_ref\_idx\_l1\_active\_minus1 + 1, NumPocTotalCurr ) and the list RefPicListTemp1 is constructed as follows:

rIdx = 0  
while( rIdx < NumRpsCurrTempList1 ) {  
 for( i = 0; i < NumPocStCurrAfter && rIdx < NumRpsCurrTempList1; rIdx++, i++ )  
 RefPicListTemp1[ rIdx ] = RefPicSetStCurrAfter[ i ]  
 for( i = 0; i < NumPocStCurrBefore && rIdx < NumRpsCurrTempList1; rIdx++, i++ ) (8‑10)  
 RefPicListTemp1[ rIdx ] = RefPicSetStCurrBefore[ i ]  
 for( i = 0; i < NumPocLtCurr && rIdx < NumRpsCurrTempList1; rIdx++, i++ )  
 RefPicListTemp1[ rIdx ] = RefPicSetLtCurr[ i ]  
}

When the slice is a B slice, the list RefPicList1 is constructed as follows:

for( rIdx = 0; rIdx ≤ num\_ref\_idx\_l1\_active\_minus1; rIdx++) (8‑11)  
 RefPicList0[ rIdx ] = ref\_pic\_list\_modification\_flag\_l1 ? RefPicListTemp1[ list\_entry\_l1[ rIdx ] ] :  
 RefPicListTemp1[ rIdx ]

### Marking of the current picture after decoding

This process is invoked after all slices of the current picture have been decoded.

– If nal\_ref\_flag of the current picture is equal to 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 log2CbSize 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 << log2CbSize ).

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 (8‑12)

– 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 is invoked with the luma location ( xB, yB ) and the variable log2PUSize set equal to log2CbSize 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 is invoked with the luma location ( xB, yB ), the variable log2TrafoSize set equal to log2CbSize, 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 is invoked with the luma location ( xBS, yBS ) and the variable log2PUSize set equal to log2CbSize − 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 is invoked with the luma location ( xBS, yBS ), the variable log2TrafoSize set equal to log2CbSize − 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 (8‑14)

– 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 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 is invoked with the chroma location ( xB/2, yB/2 ), the variable log2TrafoSize set equal to log2CbSize-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 is invoked with the chroma location ( xB/2, yB/2 ), the variable log2TrafoSize set equal to log2CbSize-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 ].

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\_DC |
| Otherwise (2..34) | Intra\_Angular |
| 35 | Intra\_FromLuma (used only for chroma) |

IntraPredMode[ xB ][ yB ] labelled 0, 1, 2, .., 35 represents directions of predictions as illustrated in .



Figure ‑ – 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 neighbouring locations ( xBA, yBA ) and ( xBB, yBB ) are set equal to ( xB−1, yB ) and ( xB, yB−1 ), respectively.
2. The minimum coding block addresses cbAddrA and cbAddrB of the coding tree blocks covering the locations ( xBA,  yBA ) and ( xBB, yBB ) respectively where are derived as follows.

cbAddrA =MinCbAddrZS[ xBA << Log2MinCbSize ][ yBA << Log2MinCbSize ] (8‑15)

cbAddrB =MinCbAddrZS[ xBB << Log2MinCbSize ][ yBB << Log2MinCbSize ] (8‑16)

1. The availability process for a minimum coding block address specified in subclause  is invoked once with the minmum coding block address cbAddrA as the input and the output assigned to availableA and once with the minmum coding block address cbAddrB as the input and the output assigned to availableB.
2. For N being either replaced A or B, the variables intraPredModeN are derived as follows.

* If availableN is equal to FALSE, intraPredModeN is set equal to Intra\_DC.
* Otherwise, if the coding unit covering ( xBN,  yBN ) is not coded as intra mode, intraPredModeN is set equal to Intra\_DC,
* Otherwise, if yB−1 is less than (( yB >> Log2CtbSize ) << Log2CtbSize), intraPredModeA is set equal to IntraPredMode[ xBA ][ yBA ] and intraPredModeB is set equal to Intra\_DC.
* 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. The candModeList[ x ] with x=0..2 is derived as follows:

* If candIntraPredModeB is equal to candIntraPredModeA, the following applies:
  + If candIntraPredModeA is less than 2 (either Intra\_Planar or Intra\_DC), candModeList[ x ] with x=0..2 is derived as:

candModeList[0] = Intra\_Planar (8‑17)  
candModeList[1] = Intra\_DC (8‑18)  
candModeList[2] = Intra\_Angular (26) (8‑19)

[Ed. (WJ): Intra\_Angular (26) means vertical direction]

* + Otherwise, candModeList[ x ] with x=0..2 is derived as:

candModeList[0] = candIntraPredModeA (8‑20)  
candModeList[1] = 2 + ( ( candIntraPredModeA − 2 − 1 ) % 32 (8‑21)  
candModeList[2] = 2 + ( ( candIntraPredModeA − 2 + 1 ) % 32 (8‑22)

* Otherwise (candIntraPredModeB is not equal to candIntraPredModeA), the following applies:
  + candModeList[0] and candModeList[1] are derived as follows:

candModeList[0] = candIntraPredModeA (8‑23)  
candModeList[1] = candIntraPredModeB (8‑24)

* + If none of candModeList[0] and candModeList[1] is equal to Intra\_Planar, candModeList[2] is set equal to Intra\_Planar,
  + Otherwise, if none of candModeList[0] and candModeList[1] is equal to Intra\_DC, candModeList[2] is set equal to Intra\_DC,
  + Otherwise, candModeList[2] is set equal to Intra\_Angular (26). [Ed. (WJ): Intra\_Angular (26) means vertical direction]

1. IntraPredMode[ xB ][ yB ] is derived by applying the following procedure:

* If prev\_intra\_pred\_flag[ xB ][ yB ] is equal to TRUE, the IntraPredMode[ xB ][ yB ] is set equal to candModeList[ mpm\_idx ],
* Otherwise IntraPredMode[ xB ][ yB ] is derived by applying the following ordered steps:

1. The array candModeList[x], x=0..2 is modified as the following ordered steps:
   1. If candModeList[0] is greater than candModeList[1], swap two values.
   2. If candModeList[0] is greater than candModeList[2], swap two values.
   3. If candModeList[1] is greater than candModeList[2], swap two values.

[Ed. (WJ): better expression rather than swap?]

1. IntraPredMode[xB][yB] is derived as the following ordered steps:
   1. IntraPredMode[ xB ][ yB ] = rem\_intra\_luma\_pred\_mode
   2. When IntraPredMode[ xB ][ yB ] is greater than or equal to candModeList[ 0 ], the value of IntraPredMode[ xB ][ yB ] is increased by one
   3. When IntraPredMode[ xB ][ yB ] is greater than or equal to candModeList[ 1 ], the value of IntraPredMode[ xB ][ yB ] is increased by one
   4. When IntraPredMode[ xB ][ yB ] is greater than or equal to candModeList[ 2 ], 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 or with intra\_chroma\_pred\_mode, IntraPredMode[ xB ][ yB ] and chroma\_pred\_from\_luma\_enabled\_flag as inputs.

Table 8‑2 – 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** | **26** | **10** | **1** | **X ( 0 <= X < 35 )** |
| 0 | 34 | 0 | 0 | 0 | 0 |
| 1 | 26 | 34 | 26 | 26 | 26 |
| 2 | 10 | 10 | 34 | 10 | 10 |
| 3 | 1 | 1 | 1 | 34 | 1 |
| 4 | LM | LM | LM | LM | LM |
| 5 | 0 | 26 | 10 | 1 | X |

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 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **intra\_chroma\_pred\_mode** | **IntraPredMode[ xB ][ yB ]** | | | | |
| **0** | **26** | **10** | **1** | **X ( 0 <= X < 35 )** |
| 0 | 34 | 0 | 0 | 0 | 0 |
| 1 | 26 | 34 | 26 | 26 | 26 |
| 2 | 10 | 10 | 34 | 10 | 10 |
| 3 | 1 | 1 | 1 | 34 | 1 |
| 4 | 0 | 26 | 10 | 1 | 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 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 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  (8‑25)

yBN = yB +y  (8‑26)

– 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"
  + the coding unit covering ( xBN, yBN ) is not available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ]
  + 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 ) is assigned to p[ x, y ].

When chroma\_pred\_from\_luma\_enabled\_flag is equal to 1, cIdx is equal to 1 or 2, and intraPredMode is equal to Intra\_FromLuma, the nS\*8+1 neighbouring samples PLM[ x, y ] that are constructed luma samples for Intra\_FromLuma prediction mode, with x = -1, y = -1..nS\*4-1 and x = 0..nS\*4-1, y=-1, are derived as following ordered steps:

1. For x = -1, y= 0..nS\*2-1 , if the chroma sample p[ x, y ] is marked as "not available for intra prediction" PLM[ x, 2y ] and PLM[ x, 2y+1 ] are marked as "not available for intra prediction", otherwise, PLM[ x, 2y ] and PLM[ x, 2y+1 ] are marked as "available for intra prediction" and the luma samples at the locations ( xB + x − 1, yB + 2y ) and ( xB + x − 1, yB + 2y + 1 ) are assigned to PLM[ x, 2y ] and PLM[ x, 2y+1 ], respectively.
2. For x = 0..nS\*2-1, y=-1 , if the chroma sample p[ x, y ] is marked as "not available for intra prediction" PLM[ 2x, y ] and PLM[ 2x+1, y ] are marked as "not available for intra prediction", otherwise, PLM[ 2x, y ] and PLM[ 2x+1, y ] are marked as "available for intra prediction" and the luma samples at the locations ( xB + 2x , yB + y ) and ( xB + 2x + 1, yB + y ) are assigned to PLM[ 2x, y ] and PLM[ 2x+1, y ].
3. For x = -1, y = -1, if the chroma sample p[ x, y ] is marked as "not available for intra prediction" PLM[ x, y ] is marked as "not available for intra prediction", otherwise, PLM[ x, y ] is marked as "available for intra prediction" and the luma sample at the location ( xB + x , yB + y ) 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 is invoked with the samples p[ x, y ] with x = −1, y = −1..nS\*2−1 and x = 0..nS\*2−1, y = −1 and nS 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 1 or 2, intraPredMode is equal to Intra\_FromLuma, and at least one luma sample PLM[ x, y ] with x = −1, y = −1..nS\*4−1 and x = 0..nS\*4−1, y = −1 is marked as "not available for intra prediction", the reference sample substitution process for intra sample prediction in subclause is invoked with the samples PLM[ x, y ] with x = −1, y = −1..nS\*4−1 and x = 0..nS\*4−1, y = −1 and nS\*2 as input and the modified samples PLM[ x, y ] with x = −1, y = −1..nS\*4−1 and x = 0..nS\*4−1, 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 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:
   * One of the intra prediction modes specified in subclause to 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 and the prediction size nS.

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. [Ed. (GJS): Global check/replace for incorrect use of ordinary hyphens ("-") versus non-breaking hyphens ("‑") versus minus signs ("−") versus en-dashes ("–").]

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.

The value of intraFilterType[ nS ][ IntraPredMode ] is derived as the following ordered steps:

1. The variable minDistVerHor is derived as:

minDistVerHor = Min( Abs( intraPredMode − 26 ), Abs( intraPredMode − 10 ) ) (8‑27)

1. If intraPredMode is equal to Intra\_DC, intraFilterType[ nS ][ IntraPredMode ] is set equal to 0,
2. Otherwise, if minDistVerHor is larger than intraHorVerDistThresh[ nS ], intraFilterType[ nS ][ IntraPredMode ] is set to 1, otherwise it is set to 0. intraHorVerDistThres[ nS ] is specified in .

Table 8‑4 – Specification of intraHorVerDistThres[ nS ] for various prediction unit sizes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **nS = 4** | **nS = 8** | **nS = 16** | **nS = 32** | **nS = 64** |
| **intraHorVerDistThresh[ nS ]** | 10 | 7 | 1 | 0 | 10 |

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 ] (8‑28)

pF[ nS\*2−1, −1 ] = p[ nS\*2−1, −1 ] (8‑29)

pF[ −1, y ] = ( p[ −1, y+1 ] + 2\*p[ −1, y ] + p[ −1, y−1 ] + 2 ) >> 2 for y = nS\*2−2..0 (8‑30)

pF[ −1, −1] = ( p[ −1, 0 ] + 2\*p[ −1, −1] + p[ 0, −1 ] + 2) >> 2 (8‑31)

pF[ x, −1 ] = ( p[ x−1, −1 ] + 2\*p[ x, −1 ] + p[ x+1, −1 ] + 2 ) >> 2 for x = 0..nS\*2−2 (8‑32)

##### Specification of Intra\_Angular (26) 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 26.

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 (8‑33)

predSamples[ x, y ] = Clip1Y( p[ x, −1 ] + ( ( p[ −1, y ] – p[ −1, −1 ] ) >> 1 ) ), with x = 0, y = 0..nS−1

– Otherwise,

predSamples[ x, y ] = p[ x, −1 ], with x, y = 0..nS−1 (8‑34)

##### Specification of Intra\_Angular (10) 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 10.

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 (8‑35)

predSamples[ x, y ] = Clip1Y( p[ −1, y ] + ( ( p[ x, −1 ] – p[ −1, −1 ] ) >> 1 ), with x = 0..nS−1, y = 0

– Otherwise,

predSamples[ x, y ] = p[ −1, y ], with x, y = 0..nS−1 (8‑36)

##### 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 1.

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 (8‑37)  
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 (8‑38)  
predSamples[ x, 0 ] = ( 1\*p[ x, −1 ] + 3\*DCVal + 2 ) >> 2, with x = 1..nS−1 (8‑39)  
predSamples[ 0, y ] = ( 1\*p[ −1, y ] + 3\*DCVal + 2 ) >> 2, with y = 1..nS−1 (8‑40)  
predSamples[ x, y ] = DCVal, with x, y = 1..nS−1 (8‑41)

* Otherwise, the prediction samples predSamples[ x, y ] are derived as

predSamples[ x, y ] = DCVal, with x, y = 0..nS−1 (8‑42)

##### Specification of Intra\_Angular (2..9, 11..25, 27..34) 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 2..9, 11..25 and 27..34.

[Ed. (WJ): do we really need to represent Intra\_Horizontal and Intra\_Vertical as one of Intra\_Angular although they use different functions? E.g. angular function can’t generate horizontal and vertical functions.]

illustrates the total 34 intra angles and specifies the mapping table between intraPredMode and the angle parameter intraPredAngle.



Figure ‑ – Intra prediction angle definition (informative)

Table 8‑5 – Specification of intraPredAngle

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intraPredMode** | 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 |
| **intraPredMode** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** |
| **intraPredAngle** | −26 | −32 | −26 | −21 | −17 | −13 | −9 | −5 | −2 | - | 2 | 5 | 9 | 13 | 17 | 21 | 26 | 32 |

further specifies the mapping table between intraPredMode and the inverse angle parameter invAngle.

Table 8‑6 – Specification of invAngle

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **intraPredMode** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **invAngle** | −256 | −315 | −390 | −482 | −630 | −910 | −1638 | −4096 |
| **intraPredMode** | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| **invAngle** | −315 | −390 | −482 | −630 | −910 | −1638 | −4096 | - |

The reference sample array refMain[ x ], with x= −nS..2\*nS is specified as follows.

– If intraPredMode is equal or greater than 18,

refMain[ x ] = p[ −1+x, −1 ], with x=0..nS (8‑43)

* If intraPredAngle is less than 0,

refMain[ x ] = p[ −1, −1+( ( x\*invAngle+128 )>>8 ) ], with x=( nS\*intraPredAngle ) >>5..−1 (8‑44)

* Otherwise,

refMain[ x ] = p[ −1+x, −1 ], with x=nS+1..2\*nS (8‑45)

Otherwise,

refMain[ x ] = p[ −1, −1+x ], with x=0..nS (8‑46)

* If intraPredAngle is less than 0,

refMain[ x ] = p[ −1+( ( x\*invAngle+128 )>>8 ), −1 ], with x=( nS\*intraPredAngle ) >>5..−1 (8‑47)

* Otherwise,

refMain[ x ] = p[ −1, −1+x ], with x=nS+1..2\*nS (8‑48)

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 (8‑49)

iFact = ( ( y + 1 )\*intraPredAngle ) && 31 (8‑50)

– 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 (8‑51)

* Otherwise, the value of the prediction samples predSamples[ x, y ] is derived by

predSamples[ x, y ] = refMain[ x+iIdx+1 ] (8‑52)

If intraPredMode is less than 18, the value of prediction samples predSamples[ x, y ] is swapped by that of predSamples[ y, x ] for y=0..nS−2, x=y+1..nS−1.

##### 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 ) (8‑53)  
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 ) (8‑54)

pY’[ x, −1 ] = ( PLM[ 2x−1, −1 ] + 2\*PLM[ 2x, −1 ] + PLM[ 2x+1, −1 ] + 2 ) >> 2, with x = 0..nS−1 (8‑55)

pY’[ −1, y ] = ( PLM[ −1, 2y ] + PLM[ −1, 2y+1 ] ) >> 1, with y = 0..nS−1 (8‑56)

pY’[ x, y ] = ( recSamplesL[ 2x, 2y ] + recSamplesL[ 2x, 2y+1 ] ) >> 1, with x, y = 0..nS−1 (8‑57)

1. Variables L, C, LL, LC and k2 are derived as follows:

L =  (8‑58)

C =  (8‑59)

LL =  (8‑60)

LC =  (8‑61)

k2 = Log2( (2\*nS) >> k3 ) (8‑62)

1. Variables a, b and k are derived as:

a1 = ( LC << k2 ) – L\*C (8‑63)  
a2 = ( LL << k2 ) – L\*L (8‑64)  
k1 = Max( 0, Log2( abs( a2 ) ) − 5 ) – Max( 0, Log2( abs( a1 ) ) − 14 ) + 2 (8‑65)  
a1s = a1 >> Max(0, Log2( abs( a1 ) ) − 14 ) (8‑66)  
a2s = abs( a2 >> Max(0, Log2( abs( a2 ) ) − 5 ) ) (8‑67)  
a3 = a2s < 1 ? 0 : Clip3( −215, 215−1, a1s\*lmDiv + ( 1 << ( k1 − 1 ) ) >> k1 ) (8‑68)

a = a3 >> Max( 0, Log2( abs( a3 ) ) − 6 ) (8‑69)  
k = 13 – Max( 0, Log2( abs( a ) ) − 6 ) (8‑70)

b = ( L – ( ( a\*C ) >> k1 ) + ( 1 << ( k2 − 1 ) ) ) >> k2 (8‑71)

where lmDiv is specified in 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 (8‑72)

Table 8‑7 – 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 log2CbSize 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 << log2CbSize and the variable nCSC is set equal to ( 1 << log2CbSize ) >> 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 is invoked with the luma location ( xC, yC ), coding unit size log2CbSize, 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 is invoked with the luma location ( xC, yC ), the size of the current coding unit log2CbSize 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 colour 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 colour 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 log2CbSize 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 << log2CbSize and the variable nCSC is set equal to ( 1 << log2CbSize ) >> 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 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 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 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 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 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\_2NxnU, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 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 >> 1 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 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, nCS1L >> 1 ), 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 + (nCS1L >> 1) 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\_2NxnD, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 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 + (nCS1L >> 1) 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 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to ( 0, nCS1L + (nCS1L >> 1) ), 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 >> 1 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\_nLx2N, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 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 >> 1 , 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 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to (nCS1L >> 1 , 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L + (nCS1L >> 1) , 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\_nRx2N, the following ordered steps apply:

1. The decoding process for prediction units in inter prediction mode as specified in subclause 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 + (nCS1L >> 1) , 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 is invoked with the luma location ( xC, yC ), the luma location ( xB, yB ) set equal to (nCS1L + (nCS1L >> 1) , 0 ), the size of the coding unit nCSL, the width of the luma prediction samples nPSW set equal to nCS1L >> 1 , 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 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 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 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 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 .

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,

* + 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 .

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 (8‑73)  
MvL1[ xB, yB ] = mvL1 (8‑74)

RefIdxL0[ xB, yB ] = refIdxL0 (8‑75)  
RefIdxL1[ xB, yB ] = refIdxL1 (8‑76)

PredFlagL0[ xB, yB ] = predFlagL0 (8‑77)  
PredFlagL1[ xB, yB ] = predFlagL1 (8‑78)

#### 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,
* 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,
* 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 is invoked with the luma location ( xC, yC ), the luma location ( xP, yP ), variables nCS, 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 is invoked with the the luma location ( xC, yC ), luma location ( xP, yP ), variables nCS, 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 refIdxLX and predFlagLX are derived as follows.

* If inter\_pred\_idc[ xP ][ yP ] is equal to Pred\_LX or Pred\_BI,

refIdxLX = ref\_idx\_lX[ xP ][ yP ] (8‑79)  
 predFlagLX = 1 (8‑80)

* Otherwise, the variables refIdxLX and predFlagLX are specified by

refIdxLX = −1 (8‑81)  
 predFlagLX = 0 (8‑82)

1. The variable mvdLX is derived as follows.

mvdLX[ 0 ] = mvd\_lX[ xP ][ yP ][ 0 ] (8‑83)  
 mvdLX[ 1 ] = mvd\_lX[ xP ][ yP ][ 1 ] (8‑84)

1. When predFlagLX is equal to 1, the variable mvpLX is derived as follows.

* The derivation process for luma motion vector prediction in subclause 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 ] (8‑85)  
 mvLX[ 1 ] = mvpLX[ 1 ] + mvdLX[ 1 ] (8‑86)

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 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 ( 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 ( xP, yP ) of the top-left luma sample of the current prediction unit relative to the top-left luma sample of the current picture,
* 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,
* 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 variables singleMCLFlag is derived as follows.

* If log2\_parallel\_merge\_level\_minus2 is greater than 0 and nCS is equal to 8, singleMCLFlag is set to 1.
* Otherwise, singleMCLFlag is set to 0.

When singleMCLFlag is equal to 1, xP is set equal to xC, yP is set equal to yC, and both nPSW and nPSH are set equal to nCS.

NOTE – When singleMCLFlag is equal to 1, all the prediction units of the current coding unit share a single merge candidate list, which is identical to the merge candidate list of the 2Nx2N prediction unit.

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 is invoked with luma location ( xP, yP ), the variable singleMCLFlag, 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 following conditions are true, refIdxLX is set equal to refIdxLX[ xP − 1, yP + nPSH − 1 ].
* the prediction unit covering luma location ( xP − 1, yP + nPSH − 1 ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ]
* the PartIdx is equal to 0
* PredMode is not MODE\_INTRA
* (xP >> (log2\_parallel\_merge\_level\_minus2 + 2)) !=   
  ((xP − 1) >> (log2\_parallel\_merge\_level\_minus2 + 2)) | |  
  (yP >> (log2\_parallel\_merge\_level\_minus2 + 2)) !=   
  ((yP + nPSH − 1) >> (log2\_parallel\_merge\_level\_minus2 + 2)
* Otherwise, refIdxLX is set equal to 0.

1. The derivation process for temporal luma motion vector prediction in subclause 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 (8‑87)

predFlagLXCol = availableFlagLXCol (8‑88)

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. The variable numMergeCand and numOrigMergeCand are set to the number of merging candidates in the mergeCandList.
3. When slice\_type is equal to B, the derivation process for combined bi-predictive merging candidates specified in subclause 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.
4. The derivation process for zero motion vector merging candidates specified in subclause 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.
5. 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 ] (8‑89)

mvLX[ 1 ] = mvLXN[ 1 ] (8‑90)

refIdxLX = refIdxLXN (8‑91)

predFlagLX = predFlagLXN (8‑92)

##### 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,
* a variable singleMCLFlag,
* 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.

– When yP−1 is less than (( yC >> Log2CtbSize ) << Log2CtbSize), the following applies.

xB0 = (xB0>>3)<<3) + ((xB0>>3)&1)\*7 (8‑93)  
xB1 = (xB1>>3)<<3) + ((xB1>>3)&1)\*7 (8‑94)  
xB2 = (xB2>>3)<<3) + ((xB2>>3)&1)\*7 (8‑95)

– If one or more of the following conditions are 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.

* (xP >> (log2\_parallel\_merge\_level\_minus2 + 2)) is equal to (xN >> (log2\_parallel\_merge\_level\_minus2 + 2)) and (yP >> (log2\_parallel\_merge\_level\_minus2 + 2)) is equal to (yN >> (log2\_parallel\_merge\_level\_minus2 + 2)).
* 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 [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ] or PredMode is MODE\_INTRA.
* singleMCLFlag is equal to 0 and 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
* singleMCLFlag is equal to 0 and 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
* N is equal to B1 and the prediction units covering luma location ( xP − 1, yP + nPSH − 1 ) ( N = A1 ) and luma location ( xN, yN ) (Cand. N) have the same motion vectors and the same reference indices
* N is equal to B0 and the prediction units covering luma location ( xP + nPSW − 1, yP − 1 ) ( N = B1 ) and luma location ( xN, yN ) (Cand. N) have the same motion vectors and the same reference indices
* N is equal to A0 and the prediction units covering luma location ( xP − 1, yP + nPSH − 1 ) ( N = A1 ) and luma location ( xN, yN ) (Cand. N) have the same motion vectors and the same reference indices
* N is equal to B2 and the prediction units covering luma location ( xP + nPSW − 1, yP − 1 ) ( N = B1 ) and luma location ( xN, yN ) (Cand. N) have the same motion vectors and the same reference indices
* N is equal to B2 and the prediction units covering luma location ( xP − 1, yP + nPSH − 1 ) ( N = A1 ) and luma location ( xN, yN ) (Cand. N) have the same motion vectors and the same reference indices

– 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 .
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,
   * + predFlagL0l0Cand = = 1
     + predFlagL1l1Cand = = 1
     + PicOrderCnt( RefPicList0[ refIdxL0l0Cand ] ) != PicOrderCnt( RefPicList1[ refIdxL1l1Cand ] ) | | mvL0l0Cand != mvL1l1Cand

the following applies.

* + - 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 (8‑96)

refIdxL1combCandk = refIdxL1l1Cand (8‑97)

predFlagL0combCandk = 1 (8‑98)

predFlagL1combCandk = 1 (8‑99)

mvL0combCandk[ 0 ] = mvL0l0Cand[ 0 ] (8‑100)

mvL0combCandk[ 1 ] = mvL0l0Cand[ 1 ] (8‑101)

mvL1combCandk[ 0 ] = mvL1l1Cand[ 0 ] (8‑102)

mvL1combCandk[ 1 ] = mvL1l1Cand[ 1 ] (8‑103)

numMergeCand = numMergeCand + 1 (8‑104)

* + - 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 8‑8 – 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 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.
   * + 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 (8‑105)

refIdxL1zeroCandm = −1 (8‑106)

predFlagL0zeroCandm = 1 (8‑107)

predFlagL1zeroCandm = 0 (8‑108)

mvL0zeroCandm[ 0 ] = 0 (8‑109)

mvL0zeroCandm[ 1 ] = 0 (8‑110)

mvL1zeroCandm[ 0 ] = 0 (8‑111)

mvL1zeroCandm[ 1 ] = 0 (8‑112)

numMergeCand = numMergeCand + 1 (8‑113)

* + - 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 (8‑114)

refIdxL1zeroCandm = zeroIdx (8‑115)

predFlagL0zeroCandm = 1 (8‑116)

predFlagL1zeroCandm = 1 (8‑117)

mvL0zeroCandm[ 0 ] = 0 (8‑118)

mvL0zeroCandm[ 1 ] = 0 (8‑119)

mvL1zeroCandm[ 0 ] = 0 (8‑120)

mvL1zeroCandm[ 1 ] = 0 (8‑121)

numMergeCand = numMergeCand + 1 (8‑122)

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 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 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 mvLXA and mvLXB have the same value, mvLXB is removed from the list. The variable numMVPCandLX is set to the number of elements within the mvpListLX and maxNumMVPCand is set to 2.
8. The motion vector predictor list is modifed as follows.
   * + If numMVPCandLX is less than 2, the following applies.

mvpListLX[ numMVPCandLX ][ 0 ] = 0 (8‑123)

mvpListLX[ numMVPCandLX ][ 1 ] = 0 (8‑124)

numMVPCandLX = numMVPCandLX + 1 (8‑125)

* + - Otherwise (numMVPCandLX is equal to or greater than 2), all motion vector predictor candidates mvpListLX[ idx ] with idx greater than 1 are removed from the list.

1. 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 ‑ – 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. When one or more of the following conditions are true, the variable isScaledFlagLX is set equal to 1.

* the prediction unit covering luma location ( xA0, yA0 ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ] and PredMode is not MODE\_INTRA.
* the prediction unit covering luma location ( xA1, yA1 ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ] 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 [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListX.
* Otherwise, if the prediction unit covering luma location ( xAk, yAk ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], PredMode is not MODE\_INTRA, predFlagLY[ xAk ][ yAk ] (with Y = !X) is equal to 1 and PicOrderCnt( RefPicListY[ refIdxLY[ xAk ][ yAk ] ] ) is equal to PicOrderCnt( RefPicListX[ 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 ListY 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 [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListX.
* Otherwise, if the prediction unit covering luma location ( xAk, yAk ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListY.
* When availableFlagLXA is equal to 1, and both RefPicListA[ refIdxA ] and RefPicListX[ refIdxLX ] are short-term reference pictures, mvLXA is derived as specified below.

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑126)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑127)

mvLXA = Clip3( −8192, 8191.75, Sign( DistScaleFactor \* mvLXA ) \*    
 ( (Abs( DistScaleFactor \* mvLXA ) + 127 ) >> 8 ) ) (8‑128)

where td and tb are derived as

td = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListA[ refIdxA ] ) ) (8‑129)

tb = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListX[ refIdxLX ] ) ) (8‑130)

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 its extended line. [Ed. (BB): Define MinPuSize in the SPS but the derivation should depend on the use of an AMP flag ]
2. When yP−1 is less than (( yC >> Log2CtbSize ) << Log2CtbSize), the following applies.

xB0 = (xB0>>3)<<3) + ((xB0>>3)&1)\*7 (8‑131)  
xB1 = (xB1>>3)<<3) + ((xB1>>3)&1)\*7 (8‑132)  
xB2 = (xB2>>3)<<3) + ((xB2>>3)&1)\*7 (8‑133)

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 [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListX.
* Otherwise, if the prediction unit covering luma location ( xBk, yBk ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], PredMode is not MODE\_INTRA, predFlagLY[ xBk ][ yBk ] (with Y = !X) is equal to 1 and PicOrderCnt( RefPicListY[ refIdxLY[ xBk ][ yBk ] ] ) is equal to PicOrderCnt( RefPicListX[ 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 ListY.

1. When isScaledFlagLX is equal to 0 and availableFlagLXB is equal to 1,  mvLXA is set equal to mvLXB and refIdxA is set equal to refIdxB and availableFlagLXA is set equal to 1.
2. 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 [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListX.
* Otherwise, if the prediction unit covering luma location ( xBk, yBk ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 ListY.
* When availableFlagLXB is equal to 1 and PicOrderCnt( RefPicListB[ refIdxB ] ) is not equal to PicOrderCnt( RefPicListX[ refIdxLX ] ) and both RefPicListB[ refIdxB ] and RefPicListX[ refIdxLX ] are short-term reference pictures, mvLXB is derived as specified below.

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑134)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑135)

mvLXB =Clip3( −8192, 8191.75, Sign( DistScaleFactor \* mvLXA ) \*   
 ( (Abs( DistScaleFactor \* mvLXA ) + 127 ) >> 8 ) ) (8‑136)

[Ed. (GJS): I believe the thing that is being clipped is an integer, so does that make sense?]

where td and tb are derived as

td = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListB[ refIdxB ] ) ) (8‑137)

tb = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListX[ refIdxLX ] ) ) (8‑138)

##### 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(RefPicListX[ refIdx ] of the picture picX) (8 141)

Depending on the values of slice\_type, collocated\_from\_l0\_flag, and collocated\_ref\_idx, the variable colPic, specifying the picture that contains the collocated 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 collocated partition as specified by RefPicList1[ collocated\_ref\_idx ].
* 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 collocated partition as specified by RefPicList0[ collocated\_ref\_idx ].

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 (8‑139)

* + If ( yP >> Log2CtbSize ) is equal to ( yPRb >> Log2CtbSize ), 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 >> Log2CtbSize ) is not equal to ( yPRb >> Log2CtbSize ) ), colPu is marked as "unavailable".

1. When colPu is coded in an intra prediction mode or colPu is marked as "unavailable", the following applies.
   * Central luma position of the current prediction unit is defined by

xPCtr = ( xP + ( nPSW >> 1 ) (8‑141)

yPCtr = ( yP + ( nPSH >> 1 ) (8‑142)

* + 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 function LongTermRefPic( picX, refIdx, LX ) is defined as follows. If the reference picture with index refIdx from reference picture list LX of the picture picX was marked as "used for long term reference" at the time when picX was the current picture, LongTermRefPic( picX, refIdx, LX ) returns 1; otherwise LongTermRefPic( picX, refIdx, LX ) returns 0.

The variables mvLXCol and availableFlagLXCol are derived as follows.

* If one or more of the following conditions are 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 marked as "unavailable".
  + pic\_temporal\_mvp\_enable\_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 RefPicListX[ refIdxLX ] is a long-term reference picture, or LongTermRefPic( colPic, refIdxCol, listCol ) is equal to 1, or PicOrderCnt( colPic ) – RefPicOrderCnt( colPic, refIdxCol, listCol ) is equal to PicOrderCntVal – PicOrderCnt( RefPicListX[ refIdxLX ] ),

mvLXCol = mvCol (8‑143)

* + Otherwise, mvLXCol is derived as scaled version of the motion vector mvCol as specified below

tx = ( 16384 + ( Abs( td ) >>1 ) ) / td (8‑144)

DistScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑145)

mvLXCol =  Clip3( −8192, 8191.75, Sign( DistScaleFactor \* mvCol ) \*    
 ( (Abs( DistScaleFactor \* mvCol ) + 127 ) >> 8 ) ) (8‑146)

where td and tb are derived as

td = Clip3( −128, 127, PicOrderCnt( colPic ) – RefPicOrderCnt( colPic, refIdxCol, listCol ) ) (8‑147)

tb = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicListX [ refIdxLX ] ) ) (8‑148)

##### 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 ] (8‑149)

mvCLX[ 1 ] = mvLX[ 1 ] (8‑150)

#### 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 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 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 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 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 (8‑151)  
yP = yC + yB (8‑152)

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 (8‑153)  
yIntL = yP + ( mvLX[ 1 ] >> 2 ) + yL (8‑154)

xFracL = mvLX[ 0 ] & 3 (8‑155)  
yFracL = mvLX[ 1 ] & 3 (8‑156)

– The prediction luma sample value predSampleLXL[ xL, yL ] is derived by invoking the process specified in subclause 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 (8‑157)  
yIntC = ( yP / 2 ) + ( mvCLX[ 1 ] >> 3 ) + yC (8‑158)

xFracC = mvLX[ 0 ] & 7 (8‑159)  
yFracC = mvLX[ 1 ] & 7 (8‑160)

– The prediction sample value predSampleLXCb[ xC, yC ] is derived by invoking the process specified in subclause 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 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 ‑ – 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 , 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, pic\_width\_in\_luma\_samples − 1, xIntL +i ) (8‑161)  
yAi, j = Clip3( 0, pic\_height\_in\_luma\_samples − 1, yIntL +j ) (8‑162)

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 (8‑163)

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 (8‑164)

c0,0 = ( A−2,0 − 5\*A−1,0 + 17\*A0,0 +  58\*A1,0 − 10\*A2,0 + 4\*A3,0 − A4,0 ) >> shift1 (8‑165)

d0,0 = ( −A0,−3 + 4\*A0,−2 − 10\*A0,−1 + 58\*A0,0 +  17\*A0,1 − 5\*A0,2 + A0,3 ) >> shift1 (8‑166)

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 (8‑167)

n0,0 = ( A0,−2 − 5\*A0,−1 + 17\*A0,0 +  58\*A0,1 − 10\*A0,2 + 4\*A0,3 − A0,4 ) >> shift1 (8‑168)

– The samples labelled e0,0, i0,0, p0,0, f0,0, j0,0, q0,0, g0,0, k0,0 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 (8‑169)

i0,0 = ( −a0,−3 + 4\*a0,−2 − 11\*a0,−1 + 40\*a0,0 +  40\*a0,1 − 11\*a0,2 + 4\*a0,3 − a0,4 ) >> shift2 (8‑170)

p0,0 = ( a0,−2 − 5\*a0,−1 + 17\*a0,0 +  58\*a0,1 − 10\*a0,2 + 4\*a0,3 − a0,4 ) >> shift2 (8‑171)

f0,0 = ( −b0,−3 + 4\*b0,−2 − 10\*b0,−1 + 58\*b0,0 +  17\*b0,1 − 5\*b0,2 + b0,3 ) >> shift2 (8‑172)

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 (8‑173)

q0,0 = ( b0,−2 − 5\*b0,−1 + 17\*b0,0 +  58\*b0,1 − 10\*b0,2 + 4\*b0,3 − b0,4 ) >> shift2 (8‑174)

g0,0 = ( −c0,−3 + 4\*c0,−2 − 10\*c0,−1 + 58\*c0,0 +  17\*c0,1 − 5\*c0,2 + c0,3 ) >> shift2 (8‑175)

k0,0 = ( −c0,−3 + 4\*c0,−2 − 11\*c0,−1 + 40\*c0,0 +  40\*c0,1 − 11\*c0,2 + 4\*c0,3 − c0,4 ) >> shift2 (8‑176)

r0,0 = ( c0,−2 − 5\*c0,−1 + 17\*c0,0 +  58\*c0,1 − 10\*c0,2 + 4\*c0,3 − c0,4 ) >> shift2 (8‑177)

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 . The value of predSampleLXL[ xL, yL ] shall be the output.

Table 8‑9 – 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 , 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, (pic\_width\_in\_luma\_samples / SubWidthC) − 1, xIntC +i ) (8‑178)  
yBi, j = Clip3( 0, (pic\_height\_in\_luma\_samples / SubWidthC) − 1, yIntC +j ) (8‑179)

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 (8‑180)

ac0,0 = ( −4\*B−1,0 + 54\*B0,0 + 16\*B1,0 − 2\*B2,0 ) >> shift1 (8‑181)

ad0,0 = ( −6\*B−1,0 + 46\*B0,0 + 28\*B1,0 − 4\*B2,0 ) >> shift1 (8‑182)

ae0,0 = ( −4\*B−1,0 + 36\*B0,0 + 36\*B1,0 − 4\*B2,0 ) >> shift1 (8‑183)

af0,0 = ( −4\*B−1,0 + 28\*B0,0 + 46\*B1,0 − 6\*B2,0 ) >> shift1 (8‑184)

ag0,0 = ( −2\*B−1,0 + 16\*B0,0 + 54\*B1,0 − 4\*B2,0 ) >> shift1 (8‑185)

ah0,0 = ( −2\*B−1,0 + 10\*B0,0 + 58\*B1,0 − 2\*B2,0 ) >> shift1 (8‑186)

– 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 (8‑187)

ca0,0 = ( −4\*B0,−1 + 54\*B0,0 + 16\*B0,1 − 2\*B0,2 ) >> shift1 (8‑188)

da0,0 = ( −6\*B0,−1 + 46\*B0,0 + 28\*B0,1 − 4\*B0,2 ) >> shift1 (8‑189)

ea0,0 = ( −4\*B0,−1 + 36\*B0,0 + 36\*B0,1 − 4\*B0,2 ) >> shift1 (8‑190)

fa0,0 = ( −4\*B0,−1 + 28\*B0,0 + 46\*B0,1 − 6\*B0,2 ) >> shift1 (8‑191)

ga0,0 = ( −2\*B0,−1 + 16\*B0,0 + 54\*B0,1 − 4\*B0,2 ) >> shift1 (8‑192)

ha0,0 = ( −2\*B0,−1 + 10\*B0,0 + 58\*B0,1 − 2\*B0,2 ) >> shift1 (8‑193)

– 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 (8‑194)

cX0,0 = ( −4\*aX0,−1 + 54\*aX0,0 + 16\*aX0,1 − 2\*aX0,2 ) >> shift2 (8‑195)

dX0,0 = ( −6\*aX0,−1 + 46\*aX0,0 + 28\*aX0,1 − 4\*aX0,2 ) >> shift2 (8‑196)

eX0,0 = ( −4\*aX0,−1 + 36\*aX0,0 + 36\*aX0,1 − 4\*aX0,2 ) >> shift2 (8‑197)

fX0,0 = ( −4\*aX0,−1 + 28\*aX0,0 + 46\*aX0,1 − 6\*aX0,2 ) >> shift2 (8‑198)

gX0,0 = ( −2\*aX0,−1 + 16\*aX0,0 + 54\*aX0,1 − 4\*aX0,2 ) >> shift2 (8‑199)

hX0,0 = ( −2\*aX0,−1 + 10\*aX0,0 + 58\*aX0,1 − 2\*aX0,2 ) >> shift2 (8‑200)

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 . The value of predSampleLXC[ xC, yC ] shall be the output.

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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 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 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 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 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 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 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 .

Outputs of this process are:

– the same as specified in subclause .

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 ) (8‑201)

– 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 ) (8‑202)

– 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 ) (8‑203)

– Otherwise,

predSamples[ x, y ] = Clip3( 0, ( 1 << bitDepth ) − 1 ,   
 ( predSamplesL0[ x, y ] + predSamplesL1[ x, y ] + offset2 ) >> shift2 ) (8‑204)

###### Weighted sample prediction process

Inputs to this process are:

– the same as specified in subclause .

– 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 .

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 ) (8‑205)  
else  
 predSamples[ x, y ] = Clip1H( predSamplesL0 [ x, y ] \* w0C + o0C ) (8‑206)

– 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 ) (8‑207)  
else  
 predSamples[ x, y ] = Clip1H( predSamplesL1 [ x, y ] \* w1C + o1C ) (8‑208)

– 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 ) ) (8‑209)

– 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 ) ) (8‑210)

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 (8‑211)

o0C = 0 (8‑212)

o1C = 0 (8‑213)

The variable WeightScaleFactor is derived from the values currPoc, refIdxL0 and refIdxL1 as follows:

tb = Clip3( −128, 127, PicOrderCntVal – PicOrderCnt( RefPicList0[ refIdxL0 ] ) ) (8‑214)

td = Clip3( −128, 127, PicOrderCnt( RefPicList1[ refIdxL1 ] )   
  – PicOrderCnt( RefPicList0[ refIdxL0 ] ) ) (8‑215)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑216)

WeightScaleFactor = Clip3( −1024, 1023, ( tb \* tx + 32 ) >> 6 ) (8‑217)

The variables w0C and w1C are derived as follows.

– If PicOrderCnt( RefPicList0[ refIdxL0 ] ) is equal to PicOrderCnt( RefPicList1[ refIdxL1 ] ) or ( WeightScaleFactor >> 2 ) < −64 or ( WeightScaleFactor >> 2 ) > 128, the following applies.

w0C=32 (8‑218)

w1C=32 (8‑219)

– Otherwise;

w0C = 64 – (WeightScaleFactor >> 2) (8‑220)

w1C = WeightScaleFactor >> 2 (8‑221)

– Otherwise (weighted\_pred\_flag is equal to 1 in P slice or weighted\_bipred\_idc is equal to 1 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 (8‑222)

w0C = LumaWeightL0[refIdxL0] (8‑223)

w1C = LumaWeightL1[refIdxL1] (8‑224)

o0C = luma\_offset\_l0[refIdxL0] \* ( 1 << ( BitDepthY − 8 ) ) (8‑225)

o1C = luma\_offset\_l1[refIdxL1] \* ( 1 << ( BitDepthY − 8 ) ) (8‑226)

– Otherwise (C is equal to Cb or Cr for chroma samples, with iCbCr = 0 for Cb, iCbCr = 1 for Cr),

logWDc = ChromaLog2WeightDenom + shift1 (8‑227)

w0C = ChromaWeightL0[refIdxL0][ iCbCr ] (8‑228)

w1C = ChromaWeightL1[refIdxL1][ iCbCr ] (8‑229)

o0C = ChromaOffsetL0[refIdxL0][ iCbCr ] \* ( 1 << ( BitDepthC − 8 ) ) (8‑230)

o1C = ChromaOffsetL1[refIdxL1][ iCbCr ] \* ( 1 << ( BitDepthC − 8 ) ) (8‑231)

### 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 log2CbSize 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 << log2CbSize and the variable nCSC is set equal to ( 1 << log2CbSize ) >> 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  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 log2CbSize, the variable log2TrafoHeight set equal to log2CbSize, 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  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 log2CbSize, the variable log2TrafoHeight set equal to log2CbSize, 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  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 log2CbSize, the variable log2TrafoHeight set equal to log2CbSize, 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 luma inter transform block split direction derivation process as specified in 6.6 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionL.
2. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If interTbSplitDirectionL 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 (interTbSplitDirectionL is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 2 \* interTbSplitDirectionL.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 2 \* (1 − interTbSplitDirectionL).
3. 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.
4. 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.
5. 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.
6. 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 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 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 log2TrafoSize is set equal to ( log2TrafoWidth + log2TrafoHeight ) >> 1.

The variable splitChromaFlag is derived as follows:

– If split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 1 and log2TrafoSize is greater than 3, splitChromaFlag is set equal to 1.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0 or log2TrafoSize is equal to 3), splitChromaFlag is set equal to 0.

Depending splitChromaFlag, the following applies:

– If splitChromaFlag is equal to 1, the following ordered steps apply:

1. The chroma inter transform block split direction derivation process as specified in 6.7 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionC.
2. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If interTbSplitDirectionC 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 (interTbSplitDirectionC is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionC.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionC).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionC.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionC).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionC.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionC).
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 2 \* interTbSplitDirectionC.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 2 \* (1 − interTbSplitDirectionC).
3. 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.
4. 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.
5. 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.
6. 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 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 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

### Derivation process for quantization parameters

Inputs of this process are:

– luma location ( xB, yB ) specifying the top-left luma sample of the current coding unit quantization group relative to the top‑left luma sample of the current picture,

Outputs of this process are:

– luma quantization parameter QP’Y,

– chroma quantization parameters QP’Cb and QP’Cr

Let a variable QPY\_A specifying the luma quantization parameter of the coding unit quantization group covering ( xB−1,  yB ), a variable ctbAddrA specifying the coding tree block address of coding tree block containing the coding unit quantization group covering ( xB−1,  yB ), a variable QPY\_B specifying the luma quantization parameter of the coding unit quantization group covering ( xB,  yB−1 ), a variable , tbAddrB specifying the coding tree block address of coding tree block containing the coding unit quantization covering ( xB,  yB−1 ), and a variable QPY\_PREV specifying the luma quantization parameter of the previous coding unit quantization group in decoding order, respectively.

When one or more of the following conditions are true, QPY\_PREV is initially set equal to SliceQPY at the start of each slice for the first coding unit quantization group in the slice.

– A quantization group of coding units is the first quantization group of coding units in a slice.

– A quantization group of coding units is the first quantization group of coding units in a tile.

– A quantization group of coding units is the first quantization group of coding units in a coding tree block row and tiles\_or\_entry\_coding\_sync\_idc is equal to 2.

The prediction of luma quantization parameter QPY\_PRED is derived as the following ordered steps:

1. If the coding unit quantization group covering ( xB−1,  yB ) in the current slice is not available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ] or tbAddrA is not equal to CurrTbAddr, then QPY\_A is set equal to QPY\_PREV.
2. If the coding unit quantization group covering ( xB,  yB−1 ) in the current slice is not available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ] or tbAddrB is not equal to CurrTbAddr, then QPY\_B is set equal to QPY\_PREV.
3. The prediction of luma quantization paramaeter QPY\_PRED is derived as:

QPY\_PRED =  (QPY\_A + QPY\_B + 1) >> 1 (8‑232)

The value of QPY is derived as

QPY = ( ( ( QPY\_PRED + cu\_qp\_delta +52+ 2\*QpBdOffsetY )%( 52 + QpBdOffsetY ) ) − QpBdOffsetY (8‑233)

The value of luma quantization parameter QP’Y is derived as

QP’Y = QPY + QpBdOffsetY (8‑234)

The values of QPCb and QPCr are equal to the value of QPC as specified in based on the index qPI equal to qPICb and qPICr derived as:

qPICb = Clip3( −QpBdOffsetC, 51, QPY + cb\_qp\_offset) (8‑235)

qPICr = Clip3( −QpBdOffsetC, 51, QPY + cr\_qp\_offset) (8‑236)

The values of chroma quantization parameters for Cb and Cr components, QP’Cb and QP’Cr are derived as:

QP’Cb = QPCb + QpBdOffsetC (8‑237)

QP’Cr = QPCr + QpBdOffsetC (8‑238)

Table ‑ – Specification of QPC as a function of qPI

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| qPI | <30 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 |
| QPC | = qPI | 29 | 30 | 31 | 32 | 32 | 33 | 34 | 34 | 35 | 35 | 36 | 36 | 37 | 37 | 37 | 38 | 38 | 38 | 39 | 39 | 39 | 39 |

### 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 (8‑239)

– Otherwise, if cIdx is equal to 1,

qP = QP’C b (8‑240)

– Otherwise (cIdx is equal to 2),

qP = QP’C r (8‑241)

The (nW)x(nH) array of residual samples r are derived as specified as follows:

* If cu\_transquant\_bypass\_flag is equal to 1, (nW)x(nH) array r is set equal to (nW)x(nH) array of transform coefficients transCoeffLevel[ xT ][ yT ][ cIdx ].
* Otherwise, the following ordered steps apply:

1. The scaling process for transform coefficients as specified in subclause 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 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, the transform skip flag transform\_skip\_flag[ xT ][ yT ][ cIdx ] 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 (8‑242)

The variable shift is derived as follows:

– If cIdx is equal to 0,

shift = BitDepthY + log2TrSize − 5 (8‑243)

– Otherwise,

shift = BitDepthC + log2TrSize − 5 (8‑244)

The scaling array levelScale[·] is specified as levelScale[k] = { 40, 45, 51, 57, 64, 72 } with k=0..5.

For the derivation of the scaled transform coefficients dij with i = 0..nW − 1, j = 0..nH − 1 the following applies.

– The scaling factor mij is derived as follows.

– If scaling\_list\_enable\_flag is equal to 0,

mij = 16 (8‑245)

– Otherwise (scaling\_list\_enable\_flag is equal to 1),

mij = ScalingFactor[ SizeID ][ RefMatrixID ][ trafoType ][ i\*nW+j ] (8‑246)

Where the variables SizeID and RefMatrixID are specified in and Equation , respectively, and trafoType is derived by

trafoType = ( ( nW = = nH ) ? 0 : ( ( nW > nH ) ? 1 : 2 ) ) (8‑247)

– The scaled transform coefficient dij is derived as follows.

dij = Clip3( −32768, 32767, ( (cij \* mij \* levelScale[ qP%6 ] << (qP/6)) + (1 << (shift − 1 )) ) >> shift ) (8‑248)

### 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 transSkipFlag specifying whether transform is applied to the current block,

– 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 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 8‑12 – Specification of horizTrType and vertTrType

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **IntraPredMode** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| **vertTrType** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **horizTrType** | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **IntraPredMode** | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| **vertTrType** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **horizTrType** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The constructed residual samples are derived as specified in the following ordered steps.

* If transSkipFlag is equal to 1, the following applies.

1. The variable shift is derived as follows:

– If cIdx is equal to 0,

shift = 13 – BitDepthY (8‑249)

– Otherwise,

shift = 13 – BitDepthC (8‑250)

1. The residual sample value rij with i=0..(nW)−1, j=0..(nH)−1 is derived as follows.

– If shift is greater than 0,

rij = ( dij + (1 << ( shift − 1) ) ) >> shift (8‑251)

– Otherwise,

rij = dij << ( −shift ) (8‑252)

* Otherwise (transSkipFlag is equal to 0), the following applies.

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 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 ) (8‑253)

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 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 (8‑254)

– Otherwise,

shift = 20 – BitDepthC (8‑255)

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 (8‑256)

#### 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 (8‑257)  
c1 = x2 + x3 (8‑258)  
c2 = x0 – x3 (8‑259)  
c3 = 74 \* x1 (8‑260)

1. The output values yi with i = 0..3 are then specified as follows:

y0 = 29\*c0 + 55\*c1 + c3 (8‑261)  
y1 = 55\*c2 − 29\*c1 + c3 (8‑262)  
y2 = 74\*( x0 - x2 + x3 ) (8‑263)  
y3 = 55\*c0 + 29 \* c2 – c3 (8‑264)

– 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 (8‑265)

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}

}

(8‑266)

[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 (8‑267)

## In-loop filter process

[Ed. (BB): TODO:

1. Define and derive in 8.6 reconstructed picture buffers prior to deblocking as in AVC: S′L, S′Cb, S′Cr
2. Invoke deblocking filter process depending on disable\_deblocking\_filter\_flag.
   1. Input: S′L, S′Cb, S′Cr
   2. Output: modified S′L, S′Cb, S′Cr after DF.
3. Invoke SAO filter process depending on sample\_adaptive\_offset\_enabled\_flag.
   1. Input: S′L, S′Cb, S′Cr
   2. Output: modified S′L, S′Cb, S′Cr after SAO.
4. Invoke ALF filter process depending on adaptive\_loop\_filter\_enabled\_flag and slice\_adaptive\_loop\_filter\_flag.
   1. Input: S′L, S′Cb, S′Cr
   2. Output: modified S′L, S′Cb, S′Cr after ALF.
5. Assign as in AVC: "the arrays S′L, S′Cb, S′Cr are assigned to the arrays SL, SCb, SCr (which represent the decoded picture), respectively"]

### Deblocking filter process

Inputs of this process are the reconstructed picture buffers prior to deblocking recPictureL, recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture buffers after deblocking recPictureL, recPictureCb and recPictureCr.

The vertical edges in a picture are filtered first. Then the horizontal edges in a picture are filtered with samples modified by deblocking filtering of vertical edges as the input. The vertical and horizontal edges in the coding tree blocks of each coding tree unit are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE – Although the filtering process is specified on a picture basis in this specification, the filtering process can be implemented on a coding unit basis with an equivalent result.

The deblocking filter process shall be applied to all prediction block edges and transform block 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 coincide with tile boundaries when loop\_filter\_across\_tiles\_enabled\_flag is equal to 0, and any edges coincide with slice boundaries of a particular slice when slice\_loop\_filter\_across\_slices\_enabled\_flag is equal to 0. For the transform units and prediction units with luma block edges less 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 follows.

For each coding unit with luma coding block size log2CbSize and location of top left sample of the luma coding block ( xC, yC ), the vertical edges are filtered by the following ordered steps.

1. The luma coding block size nS is set equal to 1 << log2CbSize.
2. The variable filterLeftCuEdgeFlag is derived as follows.

* If the left boundary of current luma coding block is the left boundary of the picture, or if the left boundary of current luma coding block is the left boundary of the tile and loop\_filter\_across\_tiles\_enabled\_flag is equal to 0, or if the left boundary of current luma coding block is the left boundary of the slice and slice\_loop\_filter\_across\_slices\_enabled\_flag is equal to 0, the variable filterLeftCuEdgeFlag is set equal to 0.
* Otherwise, the variable filterLeftCuEdgeFlag is set equal to 1.

1. All elements of the two-dimensional (nS)x(nS) array verEdgeFlags are initialized to zero.
2. The derivation process of transform block boundary specified in subclause 8.7.1.1 are invoked with the luma location ( xB, yB ) set equal to ( 0, 0 ), the transform block width log2TrafoWidth set equal to log2CbSize, the transform block height log2TrafoHeight set equal to log2CbSize, the variable trafoDepth set equal to 0, the variable filterLeftCbEdgeFlag, and the variable edgeType set equal to EDGE\_VER as the inputs and the modified array verEdgeFlags as output.
3. The derivation process of prediction block boundary specified in subclause  are invoked with the luma coding block size log2CbSize, the prediction partition mode PartMode, and the variable edgeType set equal to EDGE\_VER as inputs, and the modified array verEdgeFlags as output.
4. The derivation process of the boundary filtering strength specified in subclause  is invoked with the reconstructed luma picture buffer prior to deblocking recPictureL, the luma location ( xC, yC ), the luma coding block size log2CbSize, the variable edgeType set equal to EDGE\_VER, and verEdgeFlags as inputs and an (nS)x(nS) array verBS as output.
5. The vertical edge filtering process for a coding unit as specified in subclause 8.7.1.4.1 is invoked with the reconstructed picture buffers prior to deblocking recPictureL, recPictureCb and recPictureCr, the luma location ( xC, yC ), the luma coding block size log2CbSize and the array verBS as inputs and the modified reconstructued picture buffers recPictureL, recPictureCb and recPictureCr as output.

For each coding unit with luma coding block size log2CbSize and location of top left sample of the luma coding block ( xC, yC ), the horizontal edges are filtered by the following ordered steps.

1. The luma coding block size nS is set equal to 1 << log2CbSize.
2. The variable filterTopCbEdgeFlag is derived as follows.

* If the top boundary of current luma coding block is the top boundary of the picture, or if the top boundary of current luma coding block is the top boundary of the tile and loop\_filter\_across\_tiles\_enabled\_flag is equal to 0, or if the top boundary of current luma coding block is the top boundary of the slice and slice\_loop\_filter\_across\_slices\_enabled\_flag is equal to 0, the variable filterTopCbEdgeFlag is set equal to 0.
* Otherwise, the variable filterTopCbEdgeFlag is set equal to 1.

1. All elements of the two-dimensional (nS)x(nS) array horEdgeFlags are initialized to zero.
2. The derivation process of transform block boundary specified in subclause 8.7.1.1 are invoked with the luma location ( xB, yB ) set equal to ( 0, 0 ), the transform block width log2TrafoWidth set equal to log2CbSize, the transform block height log2TrafoHeight set equal to log2CbSize, the variable trafoDepth set equal to 0, the variable filterTopCbEdgeFlag, and the variable edgeType set equal to EDGE\_HOR as the inputs and the modified array horEdgeFlags as output.
3. The derivation process of prediction block boundary specified in subclause  are invoked with the luma coding block size log2CbSize, the prediction partition mode PartMode, and the variable edgeType set equal to EDGE\_HOR as inputs, and the modified array horEdgeFlags as output.
4. The derivation process of the boundary filtering strength specified in subclause  is invoked with the reconstructed luma picture buffer prior to deblocking recPictureL, the luma location ( xC, yC ), the luma coding block size log2CbSize, the variable edgeType set equal to EDGE\_HOR, and horEdgeFlags as inputs and an (nS)x(nS) array horBS as output.
5. The horizontal edge filtering process for a coding unit as specified in subclause 8.7.1.4.2 is invoked with the modified reconstructed picture buffers recPictureL, recPictureCb and recPictureCr, the luma location ( xC, yC ), the luma coding block size log2CbSize and the array horBS as inputs and the modified reconstructued picture buffers recPictureL, recPictureCb and recPictureCr as output.

#### Derivation process of transform block boundary

Inputs of this process are:

– a luma location ( xB, yB ) specifying the top-left sample of the current luma block relative to the top‑left sample of the current luma coding block,

– a variable log2TrafoWidth specifying the width of the current block,

– a variable log2TrafoHeight specifying the height of the current block,

– a variable trafoDepth,

– a variable filterEdgeFlag,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Output of this process is:

– a two-dimensional (nS)x(nS) array edgeFlags.

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 luma inter transform block split direction derivation process as specified in subclause 6.6 is invoked with log2TrafoWidth, log2TrafoHeight and trafoDepth as inputs and the output is assigned to interTbSplitDirectionL.
2. The variables xB1, yB1, xB2, yB2, xB3 and yB3 are derived as follows.
   * If interTbSplitDirectionL 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 (interTbSplitDirectionL is equal to 0 or 1), the following applies.
   * The variable xB1 is set equal to xB + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB1 is set equal to yB + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable xB2 is set equal to xB1 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB2 is set equal to yB1 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable xB3 is set equal to xB2 + ((1 << (log2TrafoWidth)) >> 2) \* interTbSplitDirectionL.
   * The variable yB3 is set equal to yB2 + ((1 << (log2TrafoHeight)) >> 2) \* (1 − interTbSplitDirectionL).
   * The variable log2TrafoWidth1 is set equal to log2TrafoWidth − 2 \* interTbSplitDirectionL.
   * The variable log2TrafoHeight1 is set equal to log2TrafoHeight − 2 \* (1 − interTbSplitDirectionL).
3. The deriviation process of transform block 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, the variable trafoDepth1 set equal to trafoDepth + 1, the variable filterEdgeFlag and the variable edgeType as inputs and the output is the modified version of array edgeFlags.
4. The deriviation process of transform block 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, the variable trafoDepth1 set equal to trafoDepth + 1, the variable filterEdgeFlag and the variable edgeType as inputs and the output is the modified version of array edgeFlags.
5. The deriviation process of transform block 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, the variable trafoDepth1 set equal to trafoDepth + 1, the variable filterEdgeFlag and the variable edgeType as inputs and the output is the modified version of array edgeFlags.
6. The deriviation process of transform block 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, the variable trafoDepth1 set equal to trafoDepth + 1, the variable filterEdgeFlag and the variable edgeType as inputs and the output is the modified version of array edgeFlags.

– Otherwise (split\_transform\_flag[ xB ][ yB ][ trafoDepth ] is equal to 0), the following applies:

* If edgeType is equal to EDGE\_HOR, the value of edgeFlags[ xB + k ][ yB ] for k = 0..( 1 << log2TrafoWidth ) − 1 is derived as follows.
* If yB is equal to 0, edgeFlags[ xB + k ][ yB ] is set equal to filterEdgeFlag.
* Otherwise edgeFlags[ xB + k ][ yB ] is set equal to 1.
* Otherwise (edgeType is equal to EDGE\_VER), the value of edgeFlags[ xB ][ yB + k ] for k = 0..( 1 << log2TrafoHeight ) − 1 is derived as follows.
* If xB is equal to 0, edgeFlags[ xB ][ yB + k ] is set equal to filterEdgeFlag.
* Otherwise edgeFlags[ xB ][ yB + k ] is set equal to 1.

#### Derivation process of prediction block boundary

Inputs of this process are:

– a variable log2CbSize specifying the luma coding block size,

– a prediction partition mode PartMode,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Output of this process is:

– a two-dimensional (nS)x(nS) array edgeFlags.

Depending on edgeType and PartMode, the following applies for k = 0.. ( 1 << log2CbSize ) − 1:

– If edgeType is equal to EDGE\_HOR,

– When PartMode is equal to PART\_2NxN or PART\_NxN,   
edgeFlags[ k ][ 1 << ( log2CbSize − 1 ) ] is set equal to 1.

– When PartMode is equal to PART\_2NxnU,   
edgeFlags[ k ][ (1 << ( log2CbSize − 1 )) – (1 << ( log2CbSize − 2 )) ] is set equal to 1.

– When PartMode is equal to PART\_2NxnD,   
edgeFlags[ k ][ (1 << ( log2CbSize − 1 )) + (1 << ( log2CbSize − 2 )) ] is set equal to 1.

– Otherwise (edgeType is equal to EDGE\_VER),

– When PartMode is equal to PART\_Nx2N or PART\_NxN,   
edgeFlags[ 1 << ( log2CbSize − 1 ) ][ k ] is set equal to 1.

– When PartMode is equal to PART\_nLx2N,   
edgeFlags[ (1 << ( log2CbSize − 1 )) – (1 << ( log2CbSize − 2 )) ][ k ] is set equal to 1.

– When PartMode is equal to PART\_nRx2N,   
edgeFlags[ (1 << ( log2CbSize − 1 )) + (1 << ( log2CbSize − 2 )) ][ k ] is set equal to 1.

#### Derivation process of boundary filtering strength

Inputs of this process are:

– a luma picture buffer recPictureL,

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable log2CbSize specifying the size of the current luma coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– a two-dimensional array of size (nS)x(nS), edgeFlags.

Output of this process is:

– a two-dimensional array of size (nS)x(nS), bS specifying the boundary filtering strength.

The boundary filtering strength array bS for the current coding unit is derived as follows.

The variables xDi, yDj, xN and yN are derived as follows.

* If edgeType is equal to EDGE\_VER, xDi is set equal to ( i << 3 ), yDj is set equal to ( j << 2 ), xN is set equal to (1 << ( log2CbSize − 3 )) − 1 and yN is set equal to (1 << ( log2CbSize − 2 )) − 1.
* Otherwise (edgeType is equal to EDGE\_HOR), xDi is set equal to ( i << 2 ), yDj is set equal to ( j << 3 ), xN is set equal to (1 << ( log2CbSize − 2 )) − 1 and yN is set equal to (1 << ( log2CbSize − 3 )) − 1.

For xDi with i=0..xN, the following applies.

For yDj with j=0..yN, the following applies.

* If edgeFlags[ xDi ][ yDj ] is equal to 1, the sample values are derived as follows.
* If edgeType is equal to EDGE\_VER, sample p0 = recPictureL[ xC + xDi − 1 ][ yC + yDj ] and q0 = recPictureL[ xC + xDi ][ yC + yDj ].
* Otherwise (edgeType is equal to EDGE\_HOR), sample p0 = recPictureL[ xC + xDi ][ yC + yDj − 1 ] and q0 = recPictureL[ xC + xDi ][ yC + yDj ].

Depending on p0 and q0, the variable bS[ xDi ][ yDj ] is derived as follows.

* If the sample p0 or q0 is in the luma coding block of a coding unit coded with intra prediction mode, the variable bS[ xDi ][ yDj ] is set equal to 2.
* Otherwise, if the block edge is also a transform block edge and the sample p0 or q0 is in a luma transform block which contains non-zero transform coefficient level.
* Otherwise, the following applies.
  + - When edgeType is equal to EDGE\_HOR and yC + yDj − 1 is less than (( yC >> Log2CtbSize ) << Log2CtbSize), sample p0 = recPictureL[ xL ][ yC + yDj – 1 ] where xL is equal to ((( xC + xDi ) >> 3) << 3) + ((( xC + xDi ) >> 3) & 1) \* 7.
    - If one or more of the following conditions are true, the variable bS[ xDi ][ yDj ] is set equal to 1.
    - For the prediction of the luma prediction block containing the sample p0 different reference pictures or a different number of motion vectors are used than for the prediction of the luma prediction block containing the sample q0.

NOTE 1 – The determination of whether the reference pictures used for the two luma prediction blocks are the same or different is based only on which pictures are referenced, without regard to whether a prediction is formed using an index into reference picture list 0 or an index into reference picture list 1, and also without regard to whether the index position within a reference picture list is different.

NOTE 2 – The number of motion vectors that are used for the prediction of a luma prediction block with lop left luma sample covering ( xB, yB ), is equal to PredFlagL0[ xB, yB ] + PredFlagL1[ xB, yB ].

* + - One motion vector is used to predict the luma prediction block containing the sample p0 and one motion vector is used to predict the luma prediction block containing the sample q0 and the absolute difference between the horizontal or vertical component of the motion vectors used is greater than or equal to 4 in units of quarter luma frame samples.
    - Two motion vectors and two different reference pictures are used to predict the luma prediction block containing the sample p0 and two motion vectors for the same two reference pictures are used to predict the luma prediction block containing the sample q0 and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two luma prediction blocks for the same reference picture is greater than or equal to 4 in units of quarter luma frame samples,
    - Two motion vectors for the same reference picture are used to predict the luma prediction block containing the sample p0 and two motion vectors for the same reference picture are used to predict the luma prediction block containing the sample q0 and all of the following conditions are true:
    - The absolute difference between the horizontal or vertical component of list 0 motion vectors used in the prediction of the two luma prediction bocks is greater than or equal to 4 in quarter luma frame samples or the absolute difference between the horizontal or vertical component of the list 1 motion vectors used in the prediction of the two luma prediction blocks is greater than or equal to 4 in units of quarter luma frame samples,
    - The absolute difference between the horizontal or vertical component of list 0 motion vector used in the prediction of the luma prediction block containing the sample p0 and the list 1 motion vector used in the prediction of the luma prediction block containing the sample q0 is greater than or equal to 4 in units of quarter luma frame samples or the absolute difference between the horizontal or vertical component of the list 1 motion vector used in the prediction of the luma prediction block containing the sample p0 and list 0 motion vector used in the prediction of the luma prediction block containing the sample q0 is greater than or equal to 4 in units of quarter luma frame samples.
    - Otherwise (none of the conditions above is true), the variable bS[ xDi ][ yDj ] is set equal to 0.
* Otherwise (edgeFlags[ xDi ][ yDj ] is equal to 0), the variable bS[ xDi ][ yDj ] is set equal to 0.

#### Edge filtering process

##### Vertical edge filtering process

Inputs of this process are:

– picture buffers recPictureL, recPictureCb and recPictureCr.

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable log2CbSize specifying the size of the current luma coding block,

– an array bS specifying the boundary filtering strength.

Outputs of this process are:

– the modified picture buffers recPictureL, recPictureCb and recPictureCr.

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( log2CbSize − 3 ).
2. For xDk set equal to xC+( k << 3 ), k=0..nD − 1, the following applies.

For yDm set equal to yC+( m << 2 ), m=0..nD\*2 − 1, the following applies.

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply.

1. The decision process for luma block edge as specified in subclause is invoked with the luma picture buffer recPictureL, the location of the luma coding block ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, and the boundary filtering strength bS[ xDk ][ yDm ] as inputs, the decisions dE, dEp, dEq, and the variables β, tC as outputs.
2. The filtering process for luma block edge as specified in subclause is invoked with the luma picture buffer recPictureL, the location of the luma coding block ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, the boundary filtering strength bS[ xDk ][ yDm ], the decisions dE, dEp, dEq, and the variables β, tC as inputs and the modified luma picture buffer recPictureL as output.

The filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( Max( log2CbSize, 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 applies.

* When bS[ xDk\*2 ][ yDm\*2 ] is greater than 1, the following ordered steps apply.

1. The filtering process for chroma block edge as specified in subclause is invoked with the chroma picture buffer recPictureCb, the location of the chroma coding block ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, and the boundary filtering strength bS[ xDk\*2 ][ yDm\*2 ] as inputs and the modified chroma picture buffer recPictureCb as output.
2. The filtering process for chroma block edge as specified in subclause is invoked with the chroma picture buffer recPictureCr, the location of the chroma coding block ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_VER, and the boundary filtering strength bS[ xDk\*2 ][ yDm\*2 ] as inputs and the modified chroma picture buffer recPictureCr as output.

##### Horizontal edge filtering process

Inputs of this process are:

– picture buffers recPictureL, recPictureCb and recPictureCr.

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a variable log2CbSize specifying the size of the current luma coding block,

– an array bS specifying the boundary filtering strength.

Outputs of this process are:

– the modified picture buffers recPictureL, recPictureCb and recPictureCr.

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( log2CbSize − 3 ).
2. For yDm set equal to yC+( m << 3 ), m=0..nD − 1, the following applies.

For xDk set equal to xC + ( k << 2 ), k = 0..nD\*2 − 1, the following applies.

* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply.

1. The decision process for luma block edge as specified in subclause 8.7.1.4.3 is invoked with the luma picture buffer recPictureL, the location of the luma coding block ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, and the boundary filtering strength bS[ xDk ][ yDm ] as inputs, the decisions dE, dEp, dEq, and the variables β, tC as outputs.
2. The filtering process for luma block edge as specified in subclause is invoked with the luma picture buffer recPictureL, the location of the luma coding block ( xC, yC ), the luma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, the boundary filtering strength bS[ xDk ][ yDm ], the decisions dEp, dEp, dEq, and the variables β, tC as inputs and the modified luma picture buffer recPictureL as output.

The filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable nD is set equal to 1 << ( Max( log2CbSize, 4 ) − 4 ).
2. 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 applies.

* When bS[ xDk\*2 ][ yDm\*2 ] is greater than 1, the following ordered steps apply.

1. The filtering process for chroma block edge as specified in subclause is invoked with the chroma picture buffer recPictureCb, the location of the chroma coding block ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, and the boundary filtering strength bS[ xDk\*2 ][ yDm\*2 ] as inputs and the modified chroma picture buffer recPictureCb as output.
2. The filtering process for chroma block edge as specified in subclause is invoked with the chroma picture buffer recPictureCr, the location of the chroma coding block ( xC/2, yC/2 ), the chroma location of the block ( xDk, yDm ), a variable edgeType set equal to EDGE\_HOR, and the boundary filtering strength bS[ xDk\*2 ][ yDm\*2 ] as inputs and the modified chroma picture buffer recPictureCr as output.

##### Decision process for luma block edge

Inputs of this process are:

– a luma picture buffer recPictureL,

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left sample of the current luma block relative to the top left sample of the current luma coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– a variable bS specifying the boundary filtering strength.

Outputs of this process are:

– variables dE, dEp, dEq containing decisions,

– variables β, tC.

If edgeType is equal to EDGE\_VER, the sample values pi,k and qi,k with i = 0..3 and k = 0, 3 are derived as follows:

qi,k = recPictureL[ xC + xB +i, yC + yB + k ] (8‑268)

pi,k = recPictureL[ xC + xB – i − 1, yC + yB + k ] (8‑269)

Otherwise (edgeType is equal to EDGE\_HOR), the sample values pi,k and qi,k with i = 0..3 and k = 0, 3 are derived as follows:

qi,k = recPictureL[ xC + xB +k, yC + yB + i ] (8‑270)

pi,k = recPictureL[ xC + xB +k, yC + yB – i − 1 ] (8‑271)

The variables QPQ and QPP are set equal to the QPY values of the coding units which include the coding blocks containing the sample q0,0 and p0,0, respectively.

A variable qPL is derived as follows:

qPL = ( ( QPQ + QPP + 1 ) >> 1 ) (‑)

The value of the variable β′ is determined as specified in based on the luma quantization parameter Q derived as:

Q = Clip3( 0, 51, qPL + ( beta\_offset\_div2 << 1 ) ) (‑)

where beta\_offset\_div2 is the value of the variable beta\_offset\_div2 for the slice that contains sample q0,0.

The variable β is derived as:

β = β′\*(1<<(BitDepthY − 8)) (‑)

The value of the variable tC′ is determined as specified as based on the luma quantization parameter Q derived as:

Q = Clip3( 0, 53, qPL + 2\*( bS − 1) + ( tc\_offset\_div2 << 1) ) (‑)

where tc\_offset\_div2 is the value of the variable tc\_offset\_div2 for the slice that contains sample q0,0.

The variable tC is derived as:

tC = tC′\*(1<<(BitDepthY − 8)) (‑)

Depending on edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq, and d are derived as follows:

dp0 = abs( p2,0 − 2\*p1,0 + p0,0 )

dp3 = abs( p2,3 − 2\*p1,3 + p0,3 ) (8‑277)

dq0 = abs( q2,0 − 2\*q1,0 + q0,0 ) (8‑278)

dq3 = abs( q2,3 − 2\*q1,3 + q0,3 ) (8‑279)

dpq0 = dp0 + dq0 (8‑280)

dpq3 = dp3 + dq3 (8‑281)

dp = dp0 + dp3 (8‑282)

dq = dq0 + dq3 (8‑283)

d = dpq0 + dpq3 (8‑284)

1. The variables dE, dEp and dEq are set equal to 0.
2. When d is less than β, the following ordered steps apply:
3. The variable dqp is set equal to 2\*dpq0.
4. For the sample location ( xC + xB, yC + yB ), the decision process for a luma sample as specified in subclause is invoked with sample values pi,0, qi,0 with i = 0..3, the variables dpq, β and tC as inputs and the output is assigned to the decision dSam0.
5. The variable dqp is set equal to 2\*dpq3.
6. For the sample location ( xC + xB, yC + yB + 3), the decision process for a luma sample as specified in subclause is invoked with sample values pi,3, qi,3 with i = 0..3, the variables dpq, β and tC as inputs and the output is assigned to the decision dSam3.
7. The variable dE is set equal to 1.
8. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
9. When dp is less than ( β + ( β >> 1 ) ) >> 3, the variable dEp is set equal to 1.
10. When dq is less than ( β + ( β >> 1 ) ) >> 3, the variable dEq is set equal to 1.

– Otherwise (edgeType is equal to EDGE\_HOR), the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq, and d are derived as follows:

dp0 = abs( p2,0 − 2\*p1,0 + p0,0 ) (8‑285)

dp3 = abs( p2,3 − 2\*p1,3 + p0,3 ) (8‑286)

dq0 = abs( q2,0 − 2\*q1,0 + q0,0 ) (8‑287)

dq3 = abs( q2,3 − 2\*q1,3 + q0,3 ) (8‑288)

dpq0 = dp0 + dq0 (8‑289)

dpq3 = dp3 + dq3 (8‑290)

dp = dp0 + dp3 (8‑291)

dq = dq0 + dq3 (8‑292)

d = dpq0 + dpq3 (8‑293)

1. The variables dE, dEp and dEq are set equal to 0.
2. When d is less than β, the following ordered steps apply:
3. The variable dqp is set equal to 2\*dpq0.
4. For the sample location ( xC + xB, yC + yB ), the decision process for a luma sample as specified in subclause is invoked with sample values pi,0, qi,0 with i = 0..3, the variables dpq, β and tC as inputs and the output is assigned to the decision dSam0.
5. The variable dqp is set equal to 2\*dpq3.
6. For the sample location ( xC + xB + 3, yC + yB ), the decision process for a luma sample as specified in subclause is invoked with sample values pi,3, qi,3 with i = 0..3, the variables dpq, β and tC as inputs and the output is assigned to the decision dSam3.
7. The variable dE is set equal to 1.
8. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
9. When dp is less than ( β + ( β >> 1 ) ) >> 3, the variable dEp is set equal to 1.
10. When dq is less than ( β + ( β >> 1 ) ) >> 3, the variable dEq is set equal to 1.

##### Filtering process for luma block edge

Inputs of this process are:

– a luma picture buffer recPictureL,

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left sample of the current luma block relative to the top left sample of the current luma coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– a variable bS specifying the boundary filtering strength,

– variables dE, dEp, dEq containing decisions,

– variables β, tC.

Output of this process is:

– the modified luma picture buffer recPictureL.

This process is invoked only when bS is not equal to 0.

Depending on edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0..3 are derived as follows:

qi,k = recPictureL[ xC + xB +i, yC + yB + k ] (‑)

pi,k = recPictureL[ xC + xB – i − 1, yC + yB + k ] (‑)

1. When dE is not equal to 0, for each sample location ( xC + xB, yC + yB + k ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample as specified in subclause is invoked with the sample values pi,k, qi,k with i = 0..3, the decision dE, variables dEp and dEq, 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. When nDp is greater than 0, the filtered sample values pi’ with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPictureL as follows:

recPictureL[ xC + xB + k, yC + yB – i − 1 ] = pi’ (8‑296)

1. When nDq is greater than 0, the filtered sample values qj’ with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPictureL as follows:

recPictureL[ xC + xB + k, yC + yB + j ] = qj’ (8‑297)

– Otherwise (edgeType is equal to EDGE\_HOR), the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..3 and k = 0..3 are derived as follows:

qi,k = recPictureL[ xC + xB +k, yC + yB + i ] (‑)

pi,k = recPictureL[ xC + xB +k, yC + yB – i − 1 ] (‑)

1. When dE is not equal to 0, for each sample location ( xC + xB + k, yC + yB ), k = 0..3 , the following ordered steps apply:
2. The filtering process for a luma sample as specified in subclause is invoked with the sample values pi,k, qi,k with i = 0..3, decision dE, variables dEp and dEq, 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. When nDp is greater than 0, the filtered sample values pi’ with i = 0..nDp − 1 replace the corresponding samples inside the sample array recPictureL as follows:

recPictureL[ xC + xB + k, yC + yB – i − 1 ] = pi’ (8‑300)

1. When nDq is greater than 0, the filtered sample values qj’ with j = 0..nDq − 1 replace the corresponding samples inside the sample array recPictureL as follows:

recPictureL[ xC + xB + k, yC + yB + j ] = qj’ (8‑301)

##### Filtering process for chroma block edge

Inputs of this process are:

– a chroma picture buffer s’,

– a luma location ( xC, yC ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,

– a luma location ( xB, yB ) specifying the top-left sample of the current luma block relative to the top left sample of the current luma coding block,

– a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,

– a variable bS specifying the boundary filtering strength.

Output of this process is:

– the modified chroma picture buffer s’.

If edgeType is equal to EDGE\_VER, the values pi and qi with i = 0..1 and k = 0..3 are derived as follows:

qi,k = s’[ xC + xB +i, yC + yB + k ] (8‑302)

pi,k = s’[ xC + xB – i − 1, yC + yB + k ] (8‑303)

Otherwise (edgeType is equal to EDGE\_HOR), the sample values pi and qi with i = 0..1 and k = 0..3 are derived as follows:

qi,k = s’[ xC + xB +k, yC + yB + i ] (8‑304)

pi,k = s’[ xC + xB +k, yC + yB – i − 1 ] (8‑305)

The variables QPQ and QPP are set equal to the QPY values of the coding units which include the coding blocks containing the sample q0,0 and p0,0, respectively.

The variable QPC is determined as specified in based on the index qPI derived as:

qPI = ( ( QPQ + QPP + 1 ) >> 1 ) (‑)

The value of the variable tC′ is determined as specified as based on the chroma quantization parameter Q derived as:

Q = Clip3( 0, 53, QPC + 2\*(bS − 1) + (tc\_offset\_div2 << 1 ) ) (8‑307)

The variable tC is derived as:

tC = tC′\*(1<<(BitDepthY − 8)) (‑)

where tc\_offset\_div2 is the value of the variable tc\_offset\_div2 for the slice that contains sample q0,0.

Depending on edgeType, the following applies:

– If edgeType is equal to EDGE\_VER, for each sample location ( xC + xB, yC + yB + k ), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in subclause is invoked with the sample values pi,k, qi,k, with i = 0..1, and the variable tC as inputs and the filtered sample values p0’ and q0’ as outputs.
2. 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’ (‑)

s’[ xC + xB − 1, yC + yB + k ] = p0’ (‑)

– Otherwise (edgeType is equal to EDGE\_HOR), for each sample location ( xC + xB + k, yC + yB ), k = 0..3, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in subclause is invoked with the sample values pi,k, qi,k, with i = 0..1, and the variable tC as inputs and the filtered sample values p0’ and q0’ as outputs.
2. 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’ (‑)

s’[ xC + xB +k, yC + yB − 1 ] = p0’ (‑)

##### Decision process for a luma sample

Inputs of this process are:

– sample values, pi and qi with i = 0..3,

– variables dpq, β and tC.

Output of this process is:

– a variable dSam containing a decision

The variable dSam is specified as follows:

– If dpq is less than ( β >> 2 ), abs( p3 – p0 ) + abs( q0 – q3 ) is less than ( β >> 3 ) and abs( 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,

– a variable dE,

– variables dEp1 and dEq1 containing decisions to filter samples 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:

– If the variable dE is equal to 2, the following strong filtering applies while nDp and nDq are set equal to 3:

p0’ = Clip3( p0−2\*tc, p0+2\*tc, ( p2 + 2\*p1 + 2\*p0 + 2\*q0 + q1 + 4 ) >> 3 ) (8‑313)

p1’ = Clip3( p1−2\*tc, p1+2\*tc, ( p2 + p1 + p0 + q0 + 2 ) >> 2 ) (8‑314)

p2’ = Clip3( p2−2\*tc, p2+2\*tc, ( 2\*p3 + 3\*p2 + p1 + p0 + q0 + 4 ) >> 3 ) (8‑315)

q0’ = Clip3( q0−2\*tc, q0+2\*tc, ( p1 + 2\*p0 + 2\*q0 + 2\*q1 + q2 + 4 ) >> 3 ) (8‑316)

q1’ = Clip3( q1−2\*tc, q1+2\*tc, ( p0 + q0 + q1 + q2 + 2 ) >> 2 ) (8‑317)

q2’= Clip3( q2−2\*tc, q2+2\*tc, ( p0 + q0 + q1 + 3\*q2 + 2\*q3 + 4 ) >> 3 ) (8‑318)

– Otherwise, nDp and nDq are set equal to 0 and the following weak filtering applies:

Δ = ( 9 \* ( q0 –  p0 ) − 3 \* ( q1 – p1 ) + 8 ) >> 4 (8‑319)

* + When abs(Δ) is less than tc\*10, the following ordered steps apply:
    - The filtered sample values p0’ and q0’ are specified as follows:

Δ = Clip3( −tc, tc, Δ ) (8‑320)

p0’ = Clip1Y( p0 + Δ ) (8‑321)

q0’ = Clip1Y( q0 − Δ ) (8‑322)

* + - When dEp1 is equal to 1, the filtered sample value p1’ is specified as follows:

Δp = Clip3( −(tc >> 1), tc >> 1, ( ( ( p2 + p0 + 1 ) >> 1 ) – p1 + Δ ) >>1 ) (8‑323)

p1’ = Clip1Y( p1 + Δp ) (8‑324)

* + - When dEq1 is equal to 1, the filtered sample value q1’ is specified as follows:

Δq = Clip3( −(tc >> 1), tc >> 1, ( ( ( q2 + q0 + 1 ) >> 1 ) – q1 – Δ ) >>1 ) (8‑325)

q1’ = Clip1Y( q1 + Δq ) (8‑326)

* + - nDp is set equal to dEp1+1 and nDq is set equal to dEq1+1.

When nDp is greater than 0 and one or more of the following conditions are true, each of the filtered sample values, pi’ with i = 0..nDp−1, is substituted by the corresponding input sample value pi.

– pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag of the coding unit which includes the coding block containing the sample pi is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit which includes the coding block containing the sample pi is equal to 1.

When nDq is greater than 0 and one or more of the following conditions are true, each of the filtered sample values, qj’ with j = 0..nDq−1, is substituted by the corresponding input sample value qj.

– pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag of the coding unit which includes the coding block containing the sample qj is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit which includes the coding block containing the sample qj is equal to 1.

Table 8‑13 – 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 |  |  |  |
| **β**′ | 38 | 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | - | - |  |  |  |
| **tC**′ | 5 | 5 | 6 | 6 | 7 | 8 | 9 | 10 | 11 | 13 | 14 | 16 | 18 | 20 | 22 | 24 |  |  |  |

##### Filtering process for a chroma sample

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 ) ) (8‑327)

p0’ = Clip1C( p0 + Δ ) (8‑328)

q0’ = Clip1C( q0 − Δ ) (8‑329)

When one or more of the following conditions are true, the filtered sample value, p0’ is substituted by the corresponding input sample value p0.

– pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag of the coding unit which includes the coding block containing the sample pi is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit which includes the coding block containing the sample pi is equal to 1.

When one or more of the following conditions are true, the filtered sample value, q0’ is substituted by the corresponding input sample value q0.

– pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag of the coding unit which includes the coding block containing the sample qj is equal to 1.

– cu\_transquant\_bypass\_flag of the coding unit which includes the coding block containing the sample qj is equal to 1.

### Sample adaptive offset process

Inputs of this process are the reconstructed picture buffers prior to sample adaptive offset recPictureL, recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture buffers after sample adaptive offset saoPictureL, saoPictureCb and saoPictureCr.

This process is performed on a coding tree block basis after the completion of the deblocking filter process for the decoded picture. This process is invoked when sample\_adaptive\_offset\_enabled\_flag is equal to 1.

The sample values in the modified reconstructed picture buffers after sample adaptive offset saoPictureL, saoPictureCb and saoPictureCr are initially set equal to the sample values in the reconstructed picture buffers prior to sample adaptive offset recPictureL, recPictureCb and recPictureCr.

For every coding tree unit with coding tree block location ( rx, ry ), where rx = 0..PicWidthInCtbs − 1 and ry = 0..PicHeightInCtbs − 1, the following applies:

– When slice\_sample\_adaptive\_offset\_flag[ 0 ] of the current slice is equal to 1, the coding tree block modification process as specified in subclause is invoked with recPicture set equal to recPictureL, cIdx set equal to 0, ( rx, ry ) and nS set equal to ( 1 << Log2CtbSize ) as inputs and the modified luma picture buffer saoPictureL as output.

– When slice\_sample\_adaptive\_offset\_flag[ 1 ] of the current slice is equal to 1, the coding tree block modification process as specified in subclause is invoked with recPicture set equal to recPictureCb, cIdx set equal to 1, ( rx, ry ) and nS set equal to ( 1 << (Log2CtbSize − 1) ) as inputs and the modified chroma picture buffer saoPictureCb as output.

– When slice\_sample\_adaptive\_offset\_flag[ 2 ] of the current slice is equal to 1, the coding tree block modification process as specified in subclause is invoked with recPicture set equal to recPictureCr, cIdx set equal to 2, ( rx, ry ) and nS set equal to ( 1 << (Log2CtbSize − 1) ) as inputs and the modified chroma picture buffer saoPictureCr as output.

#### Coding tree block modification process

Inputs to this process are:

– picture buffer recPicture for the colour component cIdx,

– a variable cIdx specifying colour component index,

– a pair of variables ( rx, ry ) specifying the coding tree block location,

– a coding tree block size nS.

Output of this process is a modified picture buffer saoPicture for the colour component cIdx.

The variable saoTypeIdx is set equal to sao\_type\_idx[ cIdx ][ rx ][ ry ].

The variable bitDepth is derived as follows.

– If cIdx is equal to 0, bitDepth is set equal to BitDepthY.

– Otherwise, bitDepth is set equal to BitDepthC.

The variables xC and yC are set equal to rx\*nS and ry\*nS, respectively.

For i = 0..nS−1 and j = 0..nS−1, depending on the value of saoTypeIdx, pcm\_loop\_filter\_disable\_flag, pcm\_flag, and cu\_transquant\_bypass\_flag of the coding unit which includes the coding block covering recPicture[ xC + i, yC + j ], the following applies:

– If one or more of the following conditions are true, saoPicture[ xC + i, yC + j ] is not modified.

* pcm\_loop\_filter\_disable\_flag and pcm\_flag are both equal to 1.
* cu\_transquant\_bypass\_flag is equal to 1.
* saoTypeIdx is equal to 0.

– Otherwise, if saoTypeIdx is equal to one of the values of 1, 2, 3 or 4, the following ordered steps apply:

1. The values of hPos[ k ] and vPos[ k ] for k=0..1 are specified in Table 8‑14 based on sao\_type\_idx[ cIdx ][ rx ][ ry ].
2. The variable edgeIdx is derived as follows.

* If one or more of the following conditions for k=0..1 are true, edgeIdx is set equal to 0.
* The sample at location (xC + i + hPos[ k ], yC + j + vPos[ k ]) is outside picture boundary
* slice\_loop\_filter\_across\_slices\_enabled\_flag is equal to 0 and the sample at location (xC + i + hPos[ k ], yC + j + vPos[ k ]) belongs to a different slice.
* loop\_filter\_across\_tiles\_enabled\_flag is equal to 0 and the sample at location (xC + i + hPos[ k ], yC + j + vPos[ k ]) belongs to a different tile.
* Otherwise, edgeIdx is dervied as follows.

edgeIdx = 2 + ∑k( Sign( recPicture[ xC + i, yC + j ] –   
 recPicture[ xC + i + hPos[ k ], yC + j + vPos[ k ] ] ) ) with k = 0..1 (8‑330)

When edgeIdx is equal to 0, 1, or 2, it is modified as follows.

edgeIdx = (edgeIdx = = 2) ? 0 : (edgeIdx + 1) (8‑331)

1. The modified picture buffer saoPicture[ xC + i, yC + j ] is derived as follows.

saoPicture[ xC + i, yC + j ] = Clip3( 0, (1 << bitDepth) – 1, recPicture[ xC + i, yC + j ] +   
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ edgeIdx ] ) (8‑332)

* Otherwise (saoTypeIdx is equal to 5), the following ordered steps apply:

1. The variable bandShift is set equal to bitDepth − 5.
2. The variable saoLeftClass is set equal to sao\_band\_position[ cIdx ][ rx ][ ry ].
3. The vector bandTable is defined with 32 elements and all elements are initially set to 0. Then, four of its elements (indicating the starting position of bands for explicit offsets) are modified as follows.

for( k = 0; k < 4; k++ )  
 bandTable[ (k + saoLeftClass) & 31 ] = k + 1 (8‑333)

1. The variable bandIdx is set equal to bandTable[ recPicture[ xC + i, yC + j ] >> bandShift ].
2. The modified picture buffer saoPicture[ xC + i, yC + j ] is derived as follows.

saoPicture[ xC + i, yC + j ] = Clip3( 0, (1 << bitDepth) – 1, recPicture[ xC + i, yC + j ] +   
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ bandIdx ] ) (8‑334)

Table 8‑14 – Specification of hPos and vPos according to the sample adaptive offset type

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sao\_type\_idx[ cIdx ][ 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 |

### Adaptive loop filter process

An adaptive loop filtering process shall be conditionally performed on a coding tree block basis after the completion of the sample adaptive offset process for the entire decoded picture, with all coding tree blocks in a picture processed in order of increasing coding tree block addresses.

Each coding tree block 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 slice\_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.

The current coding unit with the luma location ( xC, yC ) specifying the top-left luma sample is belonged to the current coding tree block at location ( rx, ry ), and the ALF parameter set of this current coding tree block, including alf\_lcu\_enable\_flag[cIdx][ry][rx], alf\_filter\_pattern\_flag[cIdx][ry][rx], alf\_filt\_coeff[cIdx][ry][rx] shall be used in the following process for three colour components cIdx, respectively. When alf\_lcu\_enable\_flag[cIdx][ry][rx] is equal to 1, ALF filtering process shall be applied, including filter coefficient derivation specified in subclause , filter index derivation for luma specified in subclause , filtering process for luma specified in subclause , and filtering process for chroma specified in subclause .

When interpreting the luma samples of the coding unit as to be filtered, depending on pcm\_flag, pcm\_loop\_filter\_disable\_flag, alf\_lcu\_enable\_flag[ 0 ][ ry ][ rx ] 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 cu\_transquant\_bypass\_flag is equal to 1, luma samples are not filtered,
* Otherwise, if alf\_lcu\_enable\_flag[ 0 ][ ry ][ rx ] is equal to 0, 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 Cb chroma samples of the coding unit as to be filtered, depending on pcm\_flag, pcm\_loop\_filter\_disable\_flag alf\_lcu\_enable\_flag[1][ry][rx], and alf\_cb\_enable\_flag, the following applies.

* If pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag is equal to 1, the samples of the colour component 1 are not filtered.
* Otherwise, if cu\_transquant\_bypass\_flag is equal to 1, the samples of the colour component 1 are not filtered.
* Otherwise, if alf\_cb\_enable\_flag is equal to 0, the samples in the colour component 1 are not filtered.
* Otherwise, if alf\_lcu\_enable\_flag[1][ry][rx] is equal to 1, the samples of the colour component 1 are filtered.
* Otherwise (alf\_lcu\_enable\_flag[1][ry][rx] is equal to 0), the samples of the colour component 1 are not filtered.

When interpreting the Cr samples of the coding unit as to be filtered, depending on pcm\_flag, pcm\_loop\_filter\_disable\_flag, alf\_lcu\_enable\_flag[2][ry][rx], and alf\_cr\_enable\_flag, the following applies.

* If pcm\_loop\_filter\_disable\_flag is equal to 1 and pcm\_flag is equal to 1, the samples of the colour component 2 are not filtered.
* Otherwise, if cu\_transquant\_bypass\_flag is equal to 1, the samples of the colour component 2 are not filtered.
* Otherwise, if alf\_cr\_enable\_flag is equal to 0, the samples in the colour component 2 are not filtered.
* Otherwise, if alf\_lcu\_enable\_flag[2][ry][rx] is equal to 1, the samples of the colour component 2 are filtered.
* Otherwise (alf\_lcu\_enable\_flag[2][ry][rx] is equal to 0), the samples of the colour component 2 are not filtered.

For the luma samples of the coding unit interpreted as to be filtered, the filter coefficient derivation specified in subclause is invoked with the colour component index cIdx equal to 0, and the filtering process for luma samples specified in subclause 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 log2CbSize, 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 filter coefficient derivation specified in subclause is invoked with the colour component index cIdx equal to 1, and the filtering process for chroma samples specified in subclause 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 log2CbSize−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 filter coefficient derivation specified in subclause is invoked with the colour component index cIdx equal to 2, and the filtering process for chroma samples specified in subclause 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 log2CbSize−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 ]

#### 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 log2CbSize 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 nExtSamples 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 << log2CbSize ), and a variable nB is set equal to Min( ( 1<< log2CbSize ), ( (1 << Log2CtbSize ) >> 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 << ( log2CbSize − 1 ) ), and a variable nB is set equal to Min( ( 1 << ( log2CbSize − 1 ) ), ( ( 1 << ( Log2CtbSize − 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 << ( log2CbSize − 1 ) ), and a variable nB is set equal to Min( ( 1 << ( log2CbSize − 1 ) ), ( ( 1 << ( Log2CtbSize − 1 ) ) >> slice\_granularity ) ).

The boundary padding process for each granularity block specified in subclause 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 nExtSamples 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 nExtSamples specifying the padding size,

Output of this process is the padded sample array s’’.

Depending on slice\_loop\_filter\_across\_slices\_enabled\_flag and loop\_filter\_across\_tiles\_enabled\_flag, the padded sample array s’’ is derived as follows.

* If both slice\_loop\_filter\_across\_slices\_enabled\_flag and loop\_filter\_across\_tiles\_enabled\_flag are equal to 1, s’’[ x ][ y ] is set equal to s’[ x ][ y ] for x = (xC-nExtSamples)..(xC+nB+nExtSamples-1) and y = (yC-nExtSamples)..(yC+nB+nExtSamples-1).
* Otherwise, there are 8 neighbouring blocks that may be referenced during the filtering process of the current block. The neighbouring blocks include left, right, above, bottom, above-left, above-right, below-left, and below-right blocks. The availability of each neighbouring block is derived as follows:
  + If all of the following conditions are true, the neighbouring block is marked as "available for ALF".
    - the neighbouring block and the current block belong to the same slice or slice\_loop\_filter\_across\_slices\_enabled\_flag is equal to 1.
    - the neighbouring block and the current block belong to the same tile or loop\_filter\_across\_tiles\_enabled\_flag is equal to 1.
  + Otherwise, the neighbouring block is marked as "not available for ALF"

The padded sample array s’’[ x ][ y ] is derived by the following ordered steps:

1. s’’[ xC+x ][ yC+y ] is set equal to s’[ xC+x ][ yC+y ] for x = (xC-nExtSamples)..(xC+nB+nExtSamples-1) and y = (yC-nExtSamples)..(yC+nB+nExtSamples-1). [Ed. (GJS): Clean up enumeration formatting (e.g. hanging indent and indentation of sub-bullets).]

2. When the left block is marked as "not available for ALF",

* + If the below-left block is marked as "available for ALF" and any of the following condition is true,
    - the left boundary of the current block does not coincide with an LCU boundary.
    - both the left and bottom boundaries of the current block coincide with LCU boundaries.

s’’[ xC+x ][ yC+y ] = s’[ xC ][ yC+y ] for x = −1..-nExtSamples and y = 0..(nB- nExtSamples−1)

s’’[ xC+x ][ yC+y ] = s’[ xC+x ][ yC+nB ] for x = −1..- nExtSamples and y= (nB-nExtSamples)..(nB−1)

* + Otherwise, the following applies:

s’’[ xC+x ][ yC+y ] = s’[ xC ][ yC+y ] for x = −1..- nExtSamples and y = 0..(nB−1)

3. When the right block is marked as "not available for ALF",

* + If the above-right block is marked as "available for ALF" and one of the following condition is true,
    - the right boundary of the current block coincides with an LCU boundary.
    - the top boundary of the current block does not coincides with an LCU boundary.

s’’[ xC+nB+x ][ yC+y ] = s’[ xC+nB+x ][ yC−1 ] for x= 0..(nExtSamples−1) and y= 0..(nExtSamples−1)

s’’[ xC+nB+x ][ yC+y ] = s’[ xC+nB−1 ][ yC+y ] for x= 0..(nExtSamples−1) and y= nExtSamples..(nB−1)

* + Otherwise, the following applies:

s’’[ xC+nB+x ][ yC+y ] = s’[ xC+nB−1 ][ yC+y ] for x= 0..(nExtSamples−1) and y= 0..(nB−1)

4. When the above block is marked as "not available for ALF",

* + If the above-right block is marked as "available for ALF" and one of the following condition is true,
    - the right boundary of the current block coincides with an LCU boundary.
    - the top boundary of the current block does not coincide with an LCU boundary.

s’’[ xC+x ][ yC+y ] = s’[ xC+x ][ yC ] for x= 0..(nB-nExtSamples−1) and y= −1..-nExtSamples

s’’[ xC+x ][ yC+y ] = s’[ xC+nB ][ yC+y ] for x= (nB-nExtSamples)..(nB−1) and y= −1..-nExtSamples

* + Otherwise, the following applies

s’’[ xC+x ][ yC+y ] = s’[ xC+x ][ yC ] for x= 0..(nB−1) and y= −1..-nExtSamples

5. When the below block is marked as "not available for ALF",

* + If the below-left block is marked as "available for ALF" and any of the following condition is true,
    - the left boundary of the current block does not coincide with an LCU boundary.
    - both the left and bottom boundaries of the current block coincide with LCU boundaries.

s’’[ xC+x ][ yC+nB+y ] = s’[ xC−1 ][ yC+nB+y ] for x= 0..( nExtSamples−1) and y= 0..(nExtSamples−1)

s’’[ xC+x ][ yC+nB+y ] = s’[ xC+x ][ yC+nB−1 ] for x= nExtSamples..(nB−1) and y= 0..(nExtSamples−1)

* + Otherwise, the following applies:

s’’[ xC+x ][ yC+nB+y ] = s’[ xC+x ][ yC+nB−1 ] for x= 0..(nB−1) and y= 0..(nExtSamples−1)

6. When the above-left block is marked as "not available for ALF",

* + If the above block is marked as "available for ALF" and any of the following condition is true,
    - the left block is marked as "not available for ALF".
    - the left block is marked as "available for ALF", and the left boundary of the current block coincides with an LCU boundary, and the top boundary of the current block does not coincide with an LCU boundary.

s’’[ xC+x ][ yC+y ] = s’[ xC ][ yC+y ] for x= −1..-nExtSamples and y= −1..-nExtSamples

* + Otherwise, the following applies:

s’’[ xC+x ][ yC+y ] = s’[ xC+x ][ yC ] for x= −1..-nExtSamples and y= −1..-nExtSamples

7. When the above-right block is marked as "not available for ALF",

* + If the above block is marked as "available for ALF" and the right block is marked as "not available for ALF",

s’’[ xC+nB+x ][ yC+y ] = s’[ xC+nB−1 ][ yC+y ] for x= 0..(nExtSamples−1) and y= −1..-nExtSamples

* + Otherwise (none of the conditions above is true),

s’’[ xC+nB+x ][ yC+y ] = s’[ xC+nB+x ][ yC ] for x= 0..(nExtSamples−1) and y= −1..-nExtSamples

8. When the below-left block is marked as "not available for ALF", the following applies:

* + If the below block is marked as "available for ALF" and the left block is marked as "not available for ALF",

s’’[ xC+ x ][ yC+nB+y ] = s’[ xC ][ yC+nB+y ] for x= −1..-nExtSamples and y= 0..(nExtSamples−1)

* + Otherwise, the following applies:

s’’[ xC+ x ][ yC+nB+y ] = s’[ xC+x ][ yC+nB−1 ] for x= −1..-nExtSamples and y= 0..(nExtSamples−1)

9. When the below-right block is marked as "not available for ALF",

* + If the below block is marked as "available for ALF" and one or more of the following conditions are true,
    - the right block is marked as "not available for ALF".
    - the right block is marked as "available for ALF", and the right boundary of the current block does not coincide with an LCU boundary or the bottom boundary of the current block coincides with an LCU boundary.

s’’[ xC+nB+x ][ yC+nB+y ] = s’[ xC+nB−1 ][ yC+nB+y ]   
for x= 0..(nExtSamples−1) and y= 0..(nExtSamples−1)

* + Otherwise, the following applies:

s’’[ xC+nB+x ][ yC+nB+y ] = s’[ xC+nB+x ][ yC+nB−1 ] for x= 0..(nExtSamples−1) and y= 0..(nExtSamples−1)

#### Derivation process for filter coefficients

Input to this process is a variable cIdx specifying colour component index,

Outputs of this process are filter coefficients c for the samples for the colour component cIdx.

The ALF coefficient precision alfPrecisionBit is derived from SliceQPY as specified in .

If cIdx is equal to 0, the derivation process for luma filter coefficients specified in subclause  is invoked with alfPrecisionBit as input and the ouput assigned to c.

Otherwise (cIdx is equal to 1 or 2), the derivation process for chroma coefficients specified in subclause  is invoked with cIdx and alfPrecisionBit as inputs and the ouput assigned to c.

Table ‑ – Specification of the number of bits in alf fractional part, alfPrecisionBit

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SliceQPY | 0 - 27 | 28 - 33 | 34 - 38 | 39 - 51 |
| alfPrecisionBit | 8 | 7 | 6 | 5 |

##### Derivation process for luma filter coefficients

Inputs to this process are

* + a variable alfPrecisionBit specifying the ALF coefficient precision

Outputs of this process are the luma filter coefficients cL.

The luma filter coefficients cL with elements cL[ i ][ j ], with i = 0..AlfNumFilters − 1 and j = 0..9 are derived as follows.

* If alf\_nb\_pred\_luma\_flag[ i ] is equal to 0, the following ordered steps apply:

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters−1, j = 0..8 are derived as follows:

* If alf\_pred\_flag is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_filt\_coeff [ 0 ][ ry ][ rx ][ i ][ j ] (8‑335)

* Otherwise (alf\_pred\_flag is equal to 1 and the value of i is greater than 1),

cL[ i ][ j ] = alf\_filt\_coeff [ 0 ][ ry ][ rx ][ i ][ j ] + cL[ i − 1 ][ j ] (8‑336)

1. The luma filter coefficients cL with elements cL[ i ][ j ] with i = 0..AlfNumFilters − 1 and j = 9 are derived as follows.

* If alf\_pred\_flag is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ j ] + ( 1 << alfPrecisionBit ) −  
 Σk(2\* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ])  (8‑337)  
 with k=0..j−2

* Otherwise (alf\_pred\_flag is equal to 1 and the value of i is greater than 1),

cL[ i ][ j ] = alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ j ] + cL[ i − 1 ][ j ] −  
 Σk(2\* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ])  (8‑338)  
 with k=0..j−2

* Otherwise (alf\_nb\_pred\_luma\_flag[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..7 are derived as follows:
   * + If alf\_pred\_flag is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_filt\_coeff [ 0 ][ ry ][ rx ][ i ][ j ] (8‑339)

* + - Otherwise (alf\_pred\_flag is equal to 1 and the value of i is greater than 1),

cL[ i ][ j ] = alf\_filt\_coeff [ 0 ][ ry ][ rx ][ i ][ j ] + cL[ i − 1 ][ j ] (8‑340)

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters−1, j = 8 are derived as follows:
   * + If alf\_pred\_flag is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_filt\_coeff [ 0 ][ ry ][ rx ][ i ][ j ] + ( 1 << alfPrecisionBit ) –  
 ( Σk2 \* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ] ) >> 2 (8‑341)  
 with k=0..j−1

* + - Otherwise,

alf\_coeff\_luma\_nb[ i ] = alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ j ] –  
 ( Σk2\* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ] ) >> 2 (8‑342)

cL[ i ][ j ] = alf\_coeff\_luma\_nb[ i ] + cL[ i − 1 ][ j ]  (8‑343)  
 with k=0..j−1

1. The luma filter coefficients cL with elements cL[ i ][ j ], i = 0..AlfNumFilters−1, j = 9 are derived as follows:
   * + If alf\_pred\_flag is equal to 0 or the value of i is equal to 0,

cL[ i ][ j ] = alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ j ] + ( 1 << alfPrecisionBit ) –  
 Σk(2\* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ]) − 2\*cL[ i ][ j − 1 ] (8‑344)  
 with k=0..j−2

* + - Otherwise,

cL[ i ][ j ] = alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ j ] + cL[ i − 1 ][ j ] –  
 Σk(2\* alf\_filt\_coeff[ 0 ][ ry ][ rx ][ i ][ k ]) − 2\*alf\_coeff\_luma\_nb[ i ] (8‑345)  
 with 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 ][ 18 ] = cL[ i ][ 9 ] (8‑346)

cL[ i ][ 17 – j ] = cL[ i ][ j ] (8‑347)  
 with j = 0..8

##### Derivation process for chroma filter coefficients

Inputs to this process are

* + a variable cIdx specifying colour component index,
  + a variable alfPrecisionBit specifying the ALF coefficient precision

Outputs of this process are filter coefficients cc for the samples for the colour component cIdx.

The chroma filter coefficients cC with elements cC[ i ], i = 0..9 is derived as follows:

* If i is equal to 9, the coefficient cC[i] is derived as

sum = Σj( alf\_filt\_coeff [ cIdx ][ ry ][ rx ][ 0 ] [ j ] << 1 ) (8‑348)  
 with j = 0..8

cC[ i ] = ( 1 << alfPrecisionBit ) − ( sum – alf\_filt\_coeff [ cIdx ][ ry ][ rx ][ 0 ][ i ] ) (8‑349)

* Otherwise (i is less than 9), considering the symmetry of the filter, the chroma filter coefficients cC with elements cC[ i ], i = 0..18 are derived as follows:

cC[ i ] = alf\_filt\_coeff [ cIdx ][ ry ][ rx ][ 0 ] [ i ] (8‑350)

cC[ i ][ 18 ] = cC[ i ][9 ] (8‑351)

cC[ i ][ 17 – j ] = cC[ i ][ j ] (8‑352)  
 with j = 0..8

#### 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 log2CbSize 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 << log2CbSize ).

The boundary padding process specified in subclause is invoked with the luma location ( xC, yC ), the size of coding unit log2CbSize 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:

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 ] | (8‑353)  
 varTempV[ x ][ y ] = | ( s’’[ xC+x, yC+y ] << 1 ) – s’’[ xC+x, yC+y−1 ] – s’’[ xC+x, yC+y+1 ] | (8‑354)

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 (8‑355)  
 varTempV1[ x ][ y ] = ΣiΣj varTempV[ (x << 2 ) + i ][ (y << 2) + j ] with i, j = 1, 2 (8‑356)  
 varTemp3[ x ][ y ] = ( varTempH1[ x ][ y ] + varTempV1[ x ][ y ] ) >> 2 (8‑357)

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, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5} (8‑358)  
 avgVar = varTab[Clip3( 0, 15, (varTemp3[ x >> 2 ][ y >> 2 ] \* 1024 ) >> (3 + BitDepthY) )] (8‑359)

1. The filter index fIdx[ x, y ] with x, y = 0..(nS)−1 is derived as

fIdx[ x ][ y ] = avgVarTab[direction][avgVar] (8‑360)

where avgVarTab[direction][avgVar] is defined in .

Table ‑. avgVarTab values for different avgVar and directions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **avgVar** | **0** | **1** | **2** | **3** | **4** | **5** |
| **Direction=0** | 0 | 1 | 2 | 3 | 4 | 5 |
| **Direction=1** | 0 | 6 | 7 | 8 | 9 | 10 |
| **Direction=2** | 0 | 11 | 12 | 13 | 14 | 15 |

#### 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 log2CbSize 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 is invoked with the luma location ( xC, yC ), the size of coding unit log2CbSize 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 << log2CbSize ).

A variable lcuHeight is set equal to ( 1 << Log2CtbSize ) 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 ) (8‑361)

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 greater than or equal to pic\_height\_in\_luma\_samples, dist2VB is set equal to 5.

1. The variable alfPrecisionBit is derived as specified in ., horPos[ i ] and verPos[ i ] are specified in and , respectively.
2. The following applies.

filtIdx = MapFiltFidx[ yC >> Log2MaxCUSize ][ xC >> Log2MaxCUSize ][ fIdx [ x ][ y ] ]

recFiltPictureL[ xC + x ][ yC + y ] = Σi(s’’[ xC + x + horPos[ i ], yC + y + verPos[ i ] ] \* cL[ filtIdx ][ i ])   
 with i=0..18

recFiltPictureL[ xC + x][ yC + y ] = ( recFiltPictureL[xC + x][yC + y] +   
 ( 1 << ( alfPrecisionBit − 1) ) ) >> alfPrecisionBit

Table 8‑17 – 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 | 17 | 18 |
| hosPos[i] | 0 | 0 | −1 | 0 | −1 | −4 | −3 | −2 | −1 | 0 | 1 | 2 | 3 | 4 | −1 | 0 | 1 | 0 | 0 |

Table 8‑18 – 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 | 17 | 18 |
| dist2VB < −3 or dist2VB > 2 | −3 | −2 | −1 | −1 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3 |
| dist2VB = = −3 or dist2VB = = 2 | 0 | −2 | −1 | −1 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 0 |
| dist2VB = = −2 or dist2VB = = 1 | 0 | 0 | −1 | −1 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| dist2VB = = −1 or dist2VB = = 0 | 0 | 0 | −1 | −1 | −1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | −1 | −1 | −1 | 0 | 0 |

#### 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 log2CbSize 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 is invoked with the chroma location ( xC, yC ), the size of coding unit log2CbSize 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 << log2CbSize ).

A variable lcuHeight is set equal to ( 1 << ( Log2CtbSize − 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 ) (8‑362)

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 greater than or equal to pic\_height\_in\_luma\_samples >> 1, dist2VB is set equal to 5.

1. The variable alfPrecisionBit is derived as specified in , horPos[ i ] and verPos[ i ] are specified in and , respectively.
2. The following applies.

filtIdx = MapFiltFidx[ yC >> Log2MaxCUSize ][ xC >> Log2MaxCUSize ][ fIdx [ x ][ y ] ]

recFiltPictureC[ xC + x ][ yC + y ] = Σi(s’’[ xC + x + horPos[ i ], yC + y + verPos[ i ] ] \* cC[ filtIdx ][ i ])   
 with i=0..18

recFiltPictureC[ xC + x][ yC + y ] = ( recFiltPictureC[xC + x][yC + y] +   
 ( 1 << ( alfPrecisionBit − 1) ) ) >> alfPrecisionBit

# 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  is equal to ue(v), se(v), tu(v) (see subclause ), or ae(v) (see subclause ).

## Parsing process for Exp-Golomb codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause  is equal to ue(v), se(v), ge(v) or tu(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v), or se(v) are Exp-Golomb-coded. Syntax elements coded as tu(v) are truncated unary 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++ ) (‑)  
 b = read\_bits( 1 )

The variable codeNum is then assigned as follows:

codeNum = 2leadingZeroBits − 1 + read\_bits( leadingZeroBits ) (‑)

where the value returned from read\_bits( leadingZeroBits ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

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 . The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as xi in , 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 |
| … | … |

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  with codeNum as the input.

– Otherwise, if the syntax element is coded as ge(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 codeNum which is equal to the value of the syntax element as follows:

q = −1

codeNum = 0

bit = 1

while( bit ){

bit = read\_bits( 1 )

q++

}

for(a = 0; a < k; ++a){

bit = read\_bits( 1 )

if( bit )

codeNum += (1 << a)

}

codeNum += q << k

if( codeNum != 0 ) {

bit = read\_bits( 1 )

codeNum = ( bit ) ? codeNum : ( −codeNum )

}

– Otherwise (the syntax element is coded as tu(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 rang is used in the derivation of the value of the syntax element value. codeNum which is equal to the value of the syntax element is given by a process equivalent to:

codeNum = 0 (‑)  
keepGoing = 1  
for(i = 0; i < x && keepGoing; i++){  
 keepGoing = read\_bits( 1 )   
 if( keepGoing )  
 codeNum ++  
}

### Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in subclause .

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. provides the assignment rule.

Table 9‑3 – 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  and .

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 , the initialization process of the CABAC parsing process is invoked as specified in subclause .

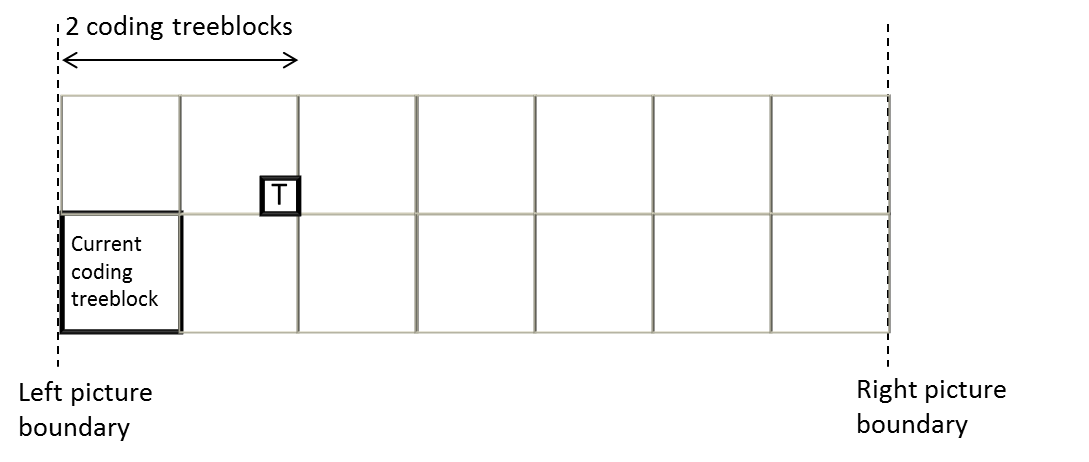


Figure ‑ – Spatial neighbour T that is used to invoke the coding tree block availability derivation process relative to the current coding tree block (informative)

The mininum coding block address of the coding tree block containing the spatial neighbor block T (), ctbMinCbAddrT, is derived using the location ( x0, y0 ) of the top-left luma sample of the current coding tree block as follows.

x = x0 + 2 << Log2CtbSize − 1  
y = y0 − 1  
ctbMinCbAddrT = MinCbAddrZS[ x >> Log2MinCbSize ][ y >> Log2MinCbSize ] (9‑4)

The variable availableFlagT is obtained by invoking the coding block availability derivation process as specified in subclause  with ctbMinCbAddrT as input.

When starting the parsing of a coding tree as specified in subclause , , the following ordered steps apply.

1. The arithmetic decoding engine is initialised as follows.
   * If CtbAddrRS is equal to slice\_address, dependent\_slice\_flag is equal to 1 and entropy\_coding\_reset\_flag is equal to 0, the following applies.
   * The synchronization process of the CABAC parsing process is invoked as specified in subclause  with TableStateIdxDS and TableMPSValDS as input.
   * The decoding process for binary decisions before termination as specified in subclause  is invoked, followed by the initialisation process for the arithmetic decoding engine as specified in subclause .
   * Otherwise if tiles\_or\_entropy\_coding\_sync\_idc is equal to 2, and CtbAddrRS % PicWidthInCtbs 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  with TableStateIdxWPP and TableMPSValWPP as input.
   * The decoding process for binary decisions before termination as specified in subclause  is invoked, followed by the initialisation process for the arithmetic decoding engine as specified in subclause .
2. When cabac\_independent\_flag is equal to 0 and dependent\_slice\_flag is equal to 1, or when tiles\_or\_entropy\_coding\_sync\_idc is equal to 2, the memorization process is applied as follows.
   * When tiles\_or\_entropy\_coding\_sync\_idc is equal to 2 and CtbAddrRS % PicWidthInCtbs is equal to 2, the memorization process of the CABAC parsing process is invoked as specified in subclause  with TableStateIdxWPP and TableMPSValWPP as output.
   * When cabac\_independent\_flag is equal to 0, dependent\_slice\_flag is equal to 1, and end\_of\_slice\_flag is equal to 1, the memorization process of the CABAC parsing process is invoked as specified in subclause  with TableStateIdxDS and TableMPSValDS as output.

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as described in subclause .

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause .

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 .

For each ctxIdx the arithmetic decoding process is invoked as specified in subclause .

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 pcm-flag and the decoded value of pcm\_flag is equal to 1, the decoding engine is initialised after the decoding of any pcm\_alignment\_zero\_bit, num\_subsequent\_pcm, and all pcm\_sample\_luma and pcm\_sample\_chroma data as specified in subclause .

### Initialization process

Outputs of this process are initialised CABAC internal variables.

The processes in subclauses  and are invoked when starting the parsing of the slice data of a slice in subclause  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.

#### Initialization process for context variables

Outputs of this process are the initialised CABAC context variables indexed by ctxIdxTable and ctxIdx.

to contain the values of the 8 bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in subclauses  to 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 .

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and intersecIdx are derived according to the following pseudo-code process:

slopeIdx = initValue >> 4  
intersecIdx = initValue & 15

Slope m and Intersec n are derived from the indices as follows:

m = slopeIdx\*5 − 45  
n = ( intersecIdx << 3 ) − 16

The two values assigned to pStateIdx and valMPS for the initialization are derived from SliceQPY, which is derived in Equation . Given the variable m and n, the initialization is specified by the following pseudo-code process:

preCtxState = Clip3( 1, 126, ( ( m \* Clip3( 0, 51, SliceQPY ) ) >> 4 ) + n )  
valMPS = ( preCtxState <= 63) ? 0 : 1  
pStateIdx = valMPS ? (preCtxState − 64) : (63 – preCtxState) (‑)

The variable initType in is derived as follows:

if( slice\_type = = I )  
 initialisatonType = 0  
else if(slice\_type = = P )  
 initialisatonType = cabac\_init\_flag ? 2 : 1  
else  
 initialisatonType = cabac\_init\_flag ? 1 : 2

Table 9‑4 – Association of ctxIdx and syntax elements for each initialisationType in the initialization process

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Syntax element** | **ctxIdxTable** | **initType** | | |
| **0** | **1** | **2** |
| sao\_param() | sao\_merge\_left\_flag |  | 0..2 | 3..5 | 6..8 |
| sao\_merge\_up\_flag |  | 0 | 1 | 2 |
| sao\_type\_idx |  | 0..1 | 2..3 | 4..5 |
| sao\_offset\_abs |  | 0..1 | 2..3 | 4..5 |
| alf\_cu\_control\_param() | alf\_cu\_flag |  | 0 | 1 | 2 |
| coding\_tree() | split\_coding\_unit\_flag |  | 0..2 | 3..5 | 6..8 |
| coding\_unit() | cu\_transquant\_bypass\_flag |  | 0 | 1 | 2 |
| skip\_flag |  |  | 0..2 | 3..5 |
| cu\_qp\_delta |  | 0..3 | 4..7 | 8..11 |
| pred\_mode\_flag |  |  | 0 | 1 |
| part\_mode |  | 2 | 3..5 | 6..8 |
| prediction\_unit() | prev\_intra\_luma\_pred\_flag |  | 0 | 1 | 2 |
| intra\_chroma\_pred\_mode |  | 0..1 | 2..3 | 4..5 |
| merge\_flag |  |  | 0 | 1 |
| merge\_idx |  |  | 0 | 1 |
| inter\_pred\_idc |  |  |  | 0..4 |
| ref\_idx\_l0, ref\_idx\_l1 |  |  | 0..2 | 3..5 |
| abs\_mvd\_greater0\_flag |  |  | 0 | 1 |
| abs\_mvd\_greater1\_flag |  |  | 2 | 3 |
| mvp\_l0\_flag, mvp\_l1\_flag |  |  | 0 | 1 |
| transform\_tree() | no\_residual\_data\_flag |  |  | 0 | 1 |
| split\_transform\_flag |  | 0..3 | 4..7 | 8..11 |
| cbf\_luma |  | 0..1 | 2..3 | 4..5 |
| cbf\_cb, cbf\_cr |  | 0..2 | 3..5 | 6..8 |
| residual\_coding() | transform\_skip\_flag[ ][ ][ 0 ] |  | 0 | 1 | 2 |
| transform\_skip\_flag[ ][ ][ 1 ] transform\_skip\_flag[ ][ ][ 2 ] |  | 3 | 4 | 5 |
| last\_significant\_coeff\_x\_prefix |  | 0..17 | 18..35 | 36..54 |
| last\_significant\_coeff\_y\_prefix |  | 0..17 | 18..35 | 36..54 |
| significant\_coeff\_group\_flag |  | 0..3 | 4..7 | 8..11 |
| significant\_coeff\_flag |  | 0..47 | 48..95 | 96..143 |
| coeff\_abs\_level\_greater1\_flag |  | 0..23 | 24..47 | 48..71 |
| coeff\_abs\_level\_greater2\_flag |  | 0..5 | 6..11 | 12..17 |

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  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 . 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 ‑ – Values of variable initValue for sao\_merge\_left\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **sao\_merge\_left\_flag ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 118 | 118 | 118 | 102 | 102 | 102 | 102 | 102 | 102 |

Table ‑ – Values of variable initValue for sao\_merge\_up\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization variable** | **sao\_merge\_up\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 109 | 102 | 102 |

Table ‑ – Values of variable initValue for sao\_type\_idx ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **sao\_type\_idx ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 61 | 104 | 168 | 120 | 184 | 120 |

Table ‑ – Values of variable initValue for sao\_offset\_abs ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **sao\_offset\_abs ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 171 | 119 | 199 | 169 | 119 | 151 |

Table ‑ – Values of variable initValue for alf\_cu\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization variable** | **alf\_cu\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 109 | 102 | 102 |

Table 9‑10 – Values of variable initValue for split\_coding\_unit\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization 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 ‑ – Values of variable initValue for cu\_transquant\_bypass\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization variable** | **cu\_transquant\_bypass\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 109 | 102 | 102 |

Table 9‑12 – Values of variable initValue for skip\_flag ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **skip\_flag ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 165 | 168 | 154 | 165 | 168 | 154 |

Table 9‑13 – Values of variable initValue for cu\_qp\_delta ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization 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 9‑14 – Values of variable initValue for pred\_mode and part\_mode

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **pred\_type ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 114 | 98 | 167 | 119 | 87 | 119 | 119 | 87 | 119 |

Table 9‑15 – Values of variable initValue for prev\_intra\_luma\_pred\_flag ctxIdx

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization variable** | **prev\_intra\_luma\_pred\_flag ctxIdx** | | |
| **0** | **1** | **2** |
| **initValue** | 167 | 119 | 150 |

Table 9‑16 – Values of variable initValue for intra\_chroma\_pred\_mode ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **intra\_chroma\_pred\_mode ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 53 | 103 | 85 | 87 | 101 | 87 |

Table 9‑17 – Value of variable initValue for merge\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialization variable** | **merge\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 72 | 119 |

Table 9‑18 – Values of variable initValue for merge\_idx ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialisation variable** | **merge\_idx ctxIdx** | |
| **0** | **1** |
| **initValue** | 100 | 116 |

Table 9‑19 – Values of variable initValue for inter\_pred\_idc ctxIdx

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialisation variable** | **inter\_pred\_idc ctxIdx** | | | | | | | |
| **0** | **1** | **2** | **3** | **4** |  |  |  |
| **initValue** | 95 | 79 | 63 | 31 | 31 |  |  |  |

Table 9‑20 – Values of variable initValue for ref\_idx\_l0, ref\_idx\_l1 ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **ref\_idx\_l0, ref\_idx\_l1 ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 102 | 118 | 103 | 118 | 118 | 134 |

Table 9‑21 – Values of variable initValue for abs\_mvd\_greater0\_flag and abs\_mvd\_greater1\_flag ctxIdx

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **abs\_mvd\_greater0\_flag, abs\_mvd\_greater1\_flag ctxIdx** | | | | |
| **0** | **1** | **2** | **3** |
| **initValue** | 120 | 166 | 135 | 166 |

Table 9‑22 – Values of variable initValue for mvp\_l0\_flag, mvp\_l1\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialization variable** | **mvp\_l0\_flag, mvp\_l1\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 134 | 134 |

Table 9‑23 – Values of variable initValue for no\_residual\_data\_flag ctxIdx

|  |  |  |
| --- | --- | --- |
| **Initialization variable** | **no\_residual\_data\_flag ctxIdx** | |
| **0** | **1** |
| **initValue** | 39 | 39 |

Table 9‑24 – Values of variable initValue for split\_transform\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization 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 9‑25 – Values of variable initValue for cbf\_luma ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **cbf\_luma ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 73 | 74 | 102 | 89 | 102 | 89 |

Table 9‑26 – Values of variable initValue for cbf\_cb and cbf\_cr ctxIdx

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **cbf\_cb ctxIdx** | | | | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **initValue** | 55 | 86 | 133 | 114 | 84 | 117 | 114 | 68 | 117 |

Table ‑ – Values of variable initValue for transform\_skip\_flag ctxIdx

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Initialization variable** | **transform\_skip\_flag ctxIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **5** |
| **initValue** | 109 | 102 | 102 | 102 | 102 | 102 |

Table 9‑28 – 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** | **15** | **16** | **17** |
| **initValue** | 72 | 72 | 71 | 72 | 104 | 89 | 88 | 89 | 59 | 73 | 89 | 106 | 60 | 59 | 43 | 54 | 70 | 53 |
|  | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** |
| **initValue** | 57 | 72 | 55 | 72 | 57 | 72 | 88 | 73 | 73 | 72 | 103 | 73 | 89 | 73 | 57 | 54 | 70 | 54 |
|  | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** |
| **initValue** | 88 | 72 | 71 | 72 | 57 | 72 | 88 | 73 | 73 | 72 | 103 | 73 | 89 | 73 | 57 | 54 | 70 | 69 |

Table 9‑29 – Values of variable initValue for last\_significant\_coeff\_y\_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** | **15** | **16** | **17** |
| **initValue** | 72 | 72 | 71 | 72 | 104 | 89 | 88 | 89 | 59 | 73 | 89 | 106 | 60 | 59 | 43 | 54 | 70 | 53 |
|  | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** |
| **initValue** | 57 | 72 | 55 | 72 | 57 | 72 | 88 | 73 | 73 | 72 | 103 | 73 | 89 | 73 | 57 | 54 | 70 | 54 |
|  | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** |
| **initValue** | 88 | 72 | 71 | 72 | 57 | 72 | 88 | 73 | 73 | 72 | 103 | 73 | 89 | 73 | 57 | 54 | 70 | 69 |

Table ‑ – Values of variable initValue for significant\_coeff\_group\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization 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 9‑31 – Values of variable initValue for significant\_coeff\_flag ctxIdx

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialization 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 9‑32 – 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 | 102 | 66 | 114 | 68 | 87 | 84 | 100 | 101 |
|  | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **initValue** | 104130 | 130 | 147 | 149 | 104 | 196 | 100 | 165 | 119 | 179 | 179 | 164 | 119 | 85 | 117 | 149 |
|  | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initValue** | 133 | 98 | 114 | 115 | 118 | 99 | 115 | 116 | 135 | 146 | 147 | 164 | 119 | 148 | 116 | 133 |
|  | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **initValue** | 119 | 179 | 148 | 164 | 119 | 85 | 117 | 149 | 133 | 98 | 114 | 115 | 118 | 99 | 115 | 100 |
|  | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** |  |  |  |  |  |  |  |  |
| **initValue** | 135 | 177 | 147 | 164 | 119 | 132 | 148 | 149 |  |  |  |  |  |  |  |  |

Table 9‑33 – 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 | 102 | 115 | 117 | 101 | 101 | 84 | 117 | 83 | 100 | 84 | 117 |
|  | **12** | **13** | **14** | **15** | **16** | **17** |  |  |  |  |  |  |
| **initValue** | 84 | 117 | 83 | 84 | 84 | 117 |  |  |  |  |  |  |

#### 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 pStateIdx and valMPS used in the initialization process of context variables that are assigned to all syntax elements in subclauses and except for the end\_of\_slice\_flag.

For each context variable, the corresponding entries pStateIdx and valMPS 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 pStateIdx and valMPS used in the memorization process of context variables that are assigned to all syntax elements in subclauses and except for the end\_of\_slice\_flag.

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 pStateIdx and valMPS of tables tableStateSync and tableMPSSync.

#### Initialization process for the arithmetic decoding engine

This process is invoked before decoding the first coding tree block 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 initialization 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 . The arithmetic decoding process for a binary decision (DecodeDecision) as specified in subclause  and the decoding process for a binary decision before termination (DecodeTerminate) as specified in subclause  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  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.

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  to , respectively. Other binarizations are specified in subclauses  to .

The binarizations for the syntax element coeff\_abs\_level\_remaining as specified in subclause  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 . 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 . When two values for each of these variables are specified for one syntax element in , 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 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 . 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 , a specific value of the variable ctxIdx is further specified in subclause . In such a case, 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 . For parsing the value of the corresponding bin from the bitstream, the arithmetic decoding process for decisions before termination (DecodeTerminate) as specified in subclause  is applied.

| Table 9‑34 – Syntax elements and associated types of binarization, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset | | | | | |
| --- | --- | --- | --- | --- | --- |
| **Syntax element** | **initType** | **Type of binarization** | **maxBinIdxCtx** | **ctxIdxTable** | **ctxIdxOffset** |
| sao\_merge\_left\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 3 |
| 2 | 0 |  | 6 |
| sao\_merge\_up\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 1 |
| 2 | 0 |  | 2 |
| sao\_type\_idx | 0 | U | 1 |  | 0 |
| 1 | 1 |  | 2 |
| 2 | 1 |  | 4 |
| sao\_band\_position | 0 | FL, cMax = 5 | na | na | na, (uses Decode Bypass) |
| 1 | na | na | na, (uses Decode Bypass) |
| 2 | na | na | na, (uses Decode Bypass) |
| sao\_offset\_abs | 0 | TU,  cMax = ( 1<< (Min(bitDepth, 10) − 5)) − 1 | 1 |  | 0 |
| 1 | 1 |  | 2 |
| 2 | 1 |  | 4 |
| sao\_offset\_sign | 0 | FL, cMax = 1 | na | na | na, (uses Decode Bypass) |
| 1 | na | na | na, (uses Decode Bypass) |
| 2 | na | na | na, (uses Decode Bypass) |
| alf\_cu\_flag | 0 | Fl, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 1 |
| 2 | 0 |  | 2 |
| end\_of\_slice\_flag | all | FL, cMax = 1 | 0 | 0 | 0 |
| split\_coding\_unit\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 3 |
| 2 | 0 |  | 6 |
| cu\_transquant\_bypass\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 1 |
| 2 | 0 |  | 2 |
| skip\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 3 |
| cu\_qp\_delta | 0 | as specified in subclause | 2 |  | 0 |
| 1 | 2 |  | 3 |
| 2 | 2 |  | 6 |
| pred\_mode\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| part\_mode | 0 | as specified in subclause | 0 |  | 2 |
| 1 | 3 |  | 3 |
| 2 | 3 |  | 6 |
| pcm\_flag | all | FL, cMax = 1 | 0 | 0 | 0 |
| prev\_intra\_luma\_pred\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 1 |
| 2 | 0 |  | 2 |
| mpm\_idx | all | TU, cMax = 2 | na | na | na, (uses Decode Bypass) |
| rem\_intra\_luma\_pred\_mode | all | FL, cMax = 31 | na | na | na, (uses Decode Bypass) |
| intra\_chroma\_pred\_mode | 0 | as specified in subclause | 1 |  | 0 |
| 1 | 1 |  | 2 |
| 2 | 1 |  | 4 |
| merge\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| merge\_idx | 1 | TU, cMax = MaxNumMergeCand − 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| inter\_pred\_idc |  | TU, cMax = 2 |  |  |  |
| 2 | 0 |  | 0 |
|  |  |  |  |  |  |
|  |  |  |  |
| ref\_idx\_l0 | 1 | TU, cMax = num\_ref\_idx\_l0\_active\_minus1 | 2 |  | 0 |
| 2 | 2 |  | 3 |
| ref\_idx\_l1 | 1 | TU, cMax = num\_ref\_idx\_l1\_active\_minus1 | 2 |  | 0 |
| 2 | 2 |  | 3 |
| abs\_mvd\_greater0\_flag[ ] | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| abs\_mvd\_greater1\_flag[ ] | 1 | FL, cMax = 1 | 0 |  | 2 |
| 2 | 0 |  | 3 |
| abs\_mvd\_minus2[ ] | 1/2 | EG1 | na | na | na, (uses Decode Bypass) |
| mvd\_sign\_flag[ ] | 1/2 | FL, cMax = 1 | na | na | na, (uses Decode Bypass) |
|  |  |  |  |  |  |
|  |  |  |  |
| mvp\_l0\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| mvp\_l1\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| no\_residual\_data\_flag | 1 | FL, cMax = 1 | 0 |  | 0 |
| 2 | 0 |  | 1 |
| split\_transform\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 4 |
| 2 | 0 |  | 8 |
| cbf\_luma | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 2 |
| 2 | 0 |  | 4 |
| cbf\_cb, cbf\_cr | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 3 |
| 2 | 0 |  | 6 |
| transform\_skip\_flag[ ][ ][ 0 ] | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 1 |
| 2 | 0 |  | 2 |
| transform\_skip\_flag[ ][ ][ 1 ] transform\_skip\_flag[ ][ ][ 2 ] | 0 | FL, cMax = 1 | 0 |  | 3 |
| 1 | 0 |  | 4 |
| 2 | 0 |  | 5 |
| last\_significant\_coeff\_x\_prefix | 0 | TU,  cMax = ( log2TrafoWidth << 1 ) − 1 | 8 |  | 0 |
| 1 | 8 |  | 18 |
| 2 | 8 |  | 36 |
| last\_significant\_coeff\_y\_prefix | 0 | TU,  cMax = ( log2TrafoHeight << 1 ) − 1 | 8 |  | 0 |
| 1 | 8 |  | 18 |
| 2 | 8 |  | 36 |
| 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 | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 4 |
| 2 | 0 |  | 8 |
| significant\_coeff\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 48 |
| 2 | 0 |  | 96 |
| coeff\_abs\_level\_greater1\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 24 |
| 2 | 0 |  | 48 |
| coeff\_abs\_level\_greater2\_flag | 0 | FL, cMax = 1 | 0 |  | 0 |
| 1 | 0 |  | 6 |
| 2 | 0 |  | 12 |
| coeff\_abs\_level\_remaining | all | prefix and suffix as specified in subclause | 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.

illustrates the bin strings of the unary binarization for a syntax element.

Table 9‑35 – 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 9.2.2.1 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 ) (9‑6)  
 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 .

#### 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 most significant bit with increasing values of binIdx towards the least 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 cLog2CbSize specifying the current CU size.

Output of this process is the binarization of the syntax element.

The binarization for part\_mode is given by depending on PredMode, cLog2CbSize and inter\_4x4\_enabled\_flag.

Table 9‑36 – Binarization for part\_mode

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PredMode** | **Value of part\_mode** | **PartMode** | **Bin string** | | |
| cLog2CbSize >  Log2MinCbSize | cLog2CbSize = = Log2MinCbSize | |
| cLog2CbSize = = 3 &&  !inter\_4x4\_enabled\_flag | cLog2CbSize > 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 coeff\_abs\_level\_remaining

Input to this process is a request for a binarization for the syntax element coeff\_abs\_level\_remaining[ 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\_remaining[ 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\_remaining[ 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\_remaining[ n − 1 ] of the same transform block.

The variable cRiceParam is derived from cLastSE and cLastRiceParam equivalent to the following pseudo-code

cRiceParam = cLastRiceParam

+ (cLastSE >= 3 && cLastRiceParam = = 0)

+ (cLastSE >= 6 && cLastRiceParam <= 1)

+ (cLastSE >= 12 && cLastRiceParam <= 2)

+ (cLastSE >= 23 && cLastRiceParam <= 3)

The variable cTRMax is derived from cRiceParam as given in .

Table 9‑37 – Specifcation of cTRMax depending on cRiceParam

|  |  |
| --- | --- |
| **cRiceParam** | **cTRMax** |
| 0 | 7 |
| 1 | 14 |
| 2 | 26 |
| 3 | 46 |
| 4 | 78 |

The binarization of coeff\_abs\_level\_remaining 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  for the syntax element coeff\_abs\_level\_remaining[ 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 .

#### Binarization process for intra\_chroma\_pred\_mode

Input to this process is a request for a binarization for the syntax element intra\_chroma\_pred\_mode and the chroma\_pred\_from\_luma\_enabled\_flag.

Output of this process is the binarization of the syntax element.

Table 9‑38 – Specifcation of intra\_chroma\_pred\_mode depending on chroma\_pred\_from\_luma\_enabled\_flag

|  |  |  |
| --- | --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **chroma\_pred\_from\_luma\_enabled\_flag = 1** | **chroma\_pred\_from\_luma\_enabled\_flag = 0** |
| 5 | 0 | n/a |
| 4 | 10 | 0 |
| 0 | 1100 | 100 |
| 1 | 1101 | 101 |
| 2 | 1110 | 110 |
| 3 | 1111 | 111 |

### Decoding process flow

Input to this process is a binarization of the requested syntax element, maxBinIdxCtx, bypassFlag, ctxIdxTable and ctxIdxOffset as specified in subclause .

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 and is invoked depending on the resulting prefix bit string as specified in subclauses and .

Depending on the variable bypassFlag, the following applies.

– If bypassFlag is equal to 1, the bypass decoding process as specified in subclause  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 .

2. Given ctxIdxTable and ctxIdx, the value of the bin from the bitstream as specified in subclause  is decoded.

#### Derivation process for ctxIdx

Inputs to this process are binIdx, maxBinIdxCtx, ctxIdxTable, and ctxIdxOffset.

Output of this process is ctxIdx.

Table 9‑39 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 , the ctxIdx for a binIdx is the sum of ctxIdxOffset and ctxIdxInc, which is found in . When more than one value is listed in 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 labelled with "na" correspond to values of binIdx that do not occur for the corresponding ctxIdxOffset.

| Table 9‑39 – Assignment of ctxIdxInc to binIdx for all ctxIdxTable and ctxIdxOffset values | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Syntax element** | **ctxIdxTable,  ctxIdxOffset** | | **binIdx** | | | | |
| **0** | **1** | **2** | **3** | **>=4** |
| sao\_merge\_left\_flag |  | 0 | cIdx | na | na | na | na |
| 3 | cIdx | na | na | na | na |
| 6 | cIdx | na | na | na | na |
| sao\_merge\_up\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| sao\_type\_idx |  | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 1 |
| sao\_offset\_abs |  | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 1 |
| alf\_cu\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| split\_coding\_unit\_flag |  | 0 | 0,1,2 (subclause ) | na | na | na | na |
| 3 | 0,1,2 (subclause ) | na | na | na | na |
| 6 | 0,1,2 (subclause ) | na | na | na | na |
| cu\_transquant\_bypass\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| skip\_flag |  | 0 | 0,1,2 (subclause ) | na | na | na | na |
| 3 | 0,1,2 (subclause ) | na | na | na | na |
| cu\_qp\_delta |  | 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 |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| part\_mode |  | 2 | 0 | na | na | na | na |
| 3 | 0 | 1 | 2 | na (uses Decode Bypass) | na |
| 6 | 0 | 1 | 2 | na (uses Decode Bypass) | na |
| part\_mode (cLog2CbSize = = Log2MinCbSize &&  cLog2CbSize > 3 | | inter\_4x4\_enabled\_flag) |  | 3 | 0 | 1 | 2 | na | na |
| 6 | 0 | 1 | 2 | na | na |
| prev\_intra\_luma\_pred\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| intra\_chroma\_pred\_mode (chroma\_pred\_from\_luma\_enabled\_flag == true) |  | 0 | 0 | 1 | na | na | na |
| 2 | 0 | 1 | na | na | na |
| 4 | 0 | 1 | na | na | na |
| intra\_chroma\_pred\_mode (chroma\_pred\_from\_luma\_enabled\_flag == false) |  | 0 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| 4 | 0 | na | na | na | na |
| merge\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| merge\_idx |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| inter\_pred\_idc |  | 0 | ctDepth | 4 | na | na | na |
| ref\_idx\_l0 |  | 0 | 0 | 1 | 2 | 2 | 2 |
| ref\_idx\_l0, ref\_idx\_l1 |  | 3 | 0 | 1 | 2 | 2 | 2 |
| abs\_mvd\_greater0\_flag[ ] |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| abs\_mvd\_greater1\_flag[ ] |  | 2 | 0 | na | na | na | na |
| 3 | 0 | na | na | na | na |
| mvp\_l0\_flag |  | 0 | 0 | na | na | na | na |
| mvp\_l0\_flag, mvp\_l1\_flag |  | 1 | 0 | na | na | na | na |
| no\_residual\_data\_flag |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| split\_transform\_flag |  | 0 | ctDepth + trafoDepth | na | na | na | na |
| 4 | ctDepth + trafoDepth | na | na | na | na |
| 8 | ctDepth + trafoDepth | na | na | na | na |
| cbf\_luma |  | 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 |  | 0 | trafoDepth | na | na | na | na |
| 3 | trafoDepth | na | na | na | na |
| 6 | trafoDepth | na | na | na | na |
| transform\_skip\_flag[ ][ ][ 0 ] |  | 0 | 0 | na | na | na | na |
| 1 | 0 | na | na | na | na |
| 2 | 0 | na | na | na | na |
| transform\_skip\_flag[ ][ ][ 1 ] transform\_skip\_flag[ ][ ][ 2 ] |  | 3 | 0 | na | na | na | na |
| 4 | 0 | na | na | na | na |
| 5 | 0 | na | na | na | na |
| last\_significant\_coeff\_x\_prefix |  | 0 | 0..17 (subclause ) | | | | |
| 18 | 0..17 (subclause ) | | | | |
| 36 | 0..17 (subclause ) | | | | |
| last\_significant\_coeff\_y\_prefix |  | 0 | 0..17 (subclause ) | | | | |
| 18 | 0..17 (subclause ) | | | | |
| 36 | 0..17 (subclause ) | | | | |
| significant\_coeff\_group\_flag |  | 0 | 0..3 (subclause ) | na | na | na | na |
| 4 | 0..3 (subclause ) | na | na | na | na |
| 8 | 0..3 (subclause ) | na | na | na | na |
| significant\_coeff\_flag |  | 0 | 0..47 (subclause 9.2.3.1.4) | na | na | na | na |
| 48 | 48..95 (subclause 9.2.3.1.4) | na | na | na | na |
| 96 | 96..143 (subclause 9.2.3.1.4) | na | na | na | na |
| coeff\_abs\_level\_greater1\_flag |  | 0 | 0..23 (subclause ) | na | na | na | na |
| 24 | 0..47 (subclause ) | na | na | na | na |
| 48 | 0..71 (subclause ) | na | na | na | na |
| coeff\_abs\_level\_greater2\_flag |  | 0 | 0..5 (subclause ) | na | na | na | na |
| 6 | 0..11 (subclause ) | na | na | na | na |
| 12 | 0..17 (subclause ) | na | na | na | na |

##### Assignment process of ctxIdxInc using neighbouring syntax elements

Subclause  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): Define MinPuSize in the SPS but the derivation should depend on the use of an AMP flag ]

* If the prediction unit covering location ( xL, yL ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 where MinPuSize = 1 << ( Log2MinCbSize − 1). The variable availableA is derived as follows.

* If the prediction unit covering location ( xA, yA ) is available [Ed. (BB): Rewrite it using MinCbAddrZS[ ][ ] and the availibility process for minimum coding blocks ], 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 split\_coding\_unit\_flag and skip\_flag is specified in .

Table 9‑40 – Specifcation of ctxIdxInc using left and above syntax elements

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax element** | **condL** | **condA** | **ctxIdxInc** |
| split\_coding\_unit\_flag | ctDepth[ xL ][ yL ]  >  ctDepth[ xP ][ yP ] | ctDepth[ xA ][ yA ]  >  ctDepth[ 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 elements last\_significant\_coeff\_x\_prefix and last\_significant\_coeff\_y\_prefix

Inputs to this process are the binIdx, the colour component index cIdx, the transform block width log2TrafoWidth and the transform block height log2TrafoHeight.

Output of this process is ctxIdxInc.

Table 9‑41 – 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 | 2 | 2 | 6 | 7 | 8 | 8 |
| **i** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** |
| **lastCtx[ i ]** | 9 | 9 | 2 | 2 | 10 | 11 | 12 | 12 | 13 | 13 | 14 | 14 | 2 | 2 |

The vector lastCtx[ i ] is specified in and ctxIdxInc is derived as follows.

When cIdx is equal to 0,

– For last\_significant\_coeff\_x\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = lastCtx[( ( log2TrafoWidth − 2 )\*( log2TrafoWidth + 1 ) ) + binIdx ] (9‑7)

– For last\_significant\_coeff\_y\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = lastCtx[ ( ( log2TrafoHeight − 2 )\*( log2TrafoHeight + 1 ) ) + binIdx ] (9‑8)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

– For last\_significant\_coeff\_x\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = (binIdx >>( log2TrafoWidth − 2 ) )+ 15 (9‑9)

– For last\_significant\_coeff\_y\_prefix, ctxIdxInc is derived as follows.

ctxIdxInc = (binIdx >>( log2TrafoWidth − 2 ) )+ 15 (9‑10)

[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 colour 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 (9‑11)

* When xCG is less than ( 1 << ( log2TrafoWidth − 2 ) ) − 1, ctxCG is modified as follows.

ctxCG += significant\_coeff\_group\_flag[ xCG + 1 ][ yCG ] (9‑12)

* When yCG is less than ( 1 << ( log2TrafoHeight − 2 ) ) − 1, ctxCG is modified as follows.

ctxCG += significant\_coeff\_group\_flag[ xCG ][ yCG + 1 ] (9‑13)

The context index increment ctxIdxInc is derived using the colour component index cIdx and ctxCG as follows.

* If cIdx is equal to 0, ctxIdxInc is derived as follows.

ctxIdxInc = min(ctxCG, 1) (9‑14)

* Otherwise (cIdx is greater than 0), ctxIdxInc is derived as follows.

ctxIdxInc = 2 + min(ctxCG, 1) (9‑15)

##### Derivation process of ctxIdxInc for the syntax element significant\_coeff\_flag

Inputs to this process are the colour 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 colour 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 as follows..

sigCtx = ctxIdxMap4x4[ ((cIdx > 0) ? 15 : 0) + (yC << 2) + xC ] (9‑16)

* Otherwise, if log2TrafoWidth is equal to log2TrafoHeight and log2TrafoWidth is equal to 3, sigCtx is derived using ctxIdxMap8x8[ ] specified in as follows.

sigCtx = ((xC + yC) = = 0) ? 10 : ctxIdxMap8x8[ ((yC >> 1 ) << 2) + (xC >> 1) ] (9‑17)

sigCtx += ( cIdx > 0) ? 6: 9 (9‑18)

* Otherwise, if xC + yC is equal to 0, sigCtx is derived as follows.

sigCtx = ( cIdx > 0) ? 17: 20 (9‑19)

* Otherwise, if (xC>>2) + (yC>>2) is less than (3 << (max(log2TrafoWidth, log2TrafoHeight) − 4)), sigCtx is derived using previously decoded bins of the syntax element significant\_coeff\_flag as follows.  
  [Ed. (IK): Check whether this is correct (original H0095 suggested check for greater or equal, but this would possibly conflict with H0290.]
* The variable sigCtx is initialized as follows.

sigCtx = 0 (9‑20)

* When xC is less than ( 1 << log2TrafoWidth ) − 1, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC + 1 ][ yC ] (9‑21)

* 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 ] (9‑22)

* When xC is less than ( 1 << log2Width ) − 2, the following applies.

sigCtx = sigCtx + significant\_coeff\_flag[ xC + 2 ][ yC ] (9‑23)

* 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 ] (9‑24)

* 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 ] (9‑25)

* The variable sigCtx is modified as follows.
  + If cIdx is equal to 0 and (xC>>2) + (yC>>2) are greater than 0, the following applies.

sigCtx = ( (sigCtx + 1) >> 1 ) + 24 (9‑26)

* + Otherwise, the following applies.

sigCtx = ( (sigCtx + 1) >> 1 ) + ( (cIdx > 0) ? 18 : 21 ) (9‑27)

* Otherwise( (xC>>2) + (yC>>2) is equal to or greater than (3 << (max(log2TrafoWidth, log2TrafoHeight) − 4)) ), sigCtx is derived as follows.

sigCtx = ( cIdx > 0) ? 18: 24

The context index increment ctxIdxInc is derived using the colour component index cIdx and sigCtx as follows.

* If cIdx is equal to 0, ctxIdxInc is derived as follows.

ctxIdxInc = sigCtx (9‑28)

* Otherwise (cIdx is greater than 0), ctxIdxInc is derived as follows.

ctxIdxInc = 27 + sigCtx (9‑29)

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 colour 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 (9‑30)

* Otherwise (i is greater than 0 and cIdx is equal to 0), the following applies.

ctxSet = 2 (9‑31)

* 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 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 (9‑32)

* 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 ) (9‑33)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

ctxIdxInc = ctxIdxInc + 16 (9‑34)

##### Derivation process of ctxIdxInc for the syntax element coeff\_abs\_level\_greater2\_flag

Inputs to this process are the colour 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 (9‑35)

* Otherwise (i is greater than 0 and cIdx is equal to 0), the following applies.

ctxSet = 2 (9‑36)

* The variable numGreater1 is set equal to 0.

ctxIdxInc = ctxSet (9‑37)

When cIdx is greater than 0, ctxIdxInc is modified as follows.

ctxIdxInc = ctxIdxInc + 4 (9‑38)

#### Arithmetic decoding process

Inputs to this process are the bypassFlag, ctxIdxTable, and ctxIdx as derived in subclause , and the state variables codIRange and codIOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

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  is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxIdxTable is equal  0 and ctxIdx is equal to 0, DecodeTerminate( ) as specified in subclause  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  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 quantized to a small set {Q1,…,Q4} of pre-set quantization 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.

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 (9‑39)

– Given qCodIRangeIdx and pStateIdx associated with ctxIdxTable and ctxIdx, the value of the variable rangeTabLPS as specified in is assigned to codIRangeLPS:

codIRangeLPS = rangeTabLPS[ pStateIdx ][ qCodIRangeIdx ] (9‑40)

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 . Depending on the current value of codIRange, renormalization is performed as specified in subclause .

###### 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 { (9‑41)  
 if( pStateIdx = = 0 )  
 valMPS = 1 − valMPS  
 pStateIdx = transIdxLPS( pStateIdx )  
}

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

[Ed. (BB): add ctxIdxTable to the figure]

Table 9‑44 – 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 9‑45 – 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 . 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. 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. 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 .

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 . 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.

#### Initialization 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 coding block 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 initialization 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 , , and .

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 , , and is 10 bits and 9 bits, respectively. The encoding process for a binary decision (EncodeDecision) as specified in subclause  and the encoding process for a binary decision before termination (EncodeTerminate) as specified in subclause  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  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.

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 quantized value of codIRange by using Equation . 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 , 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 . Depending on the current value of codIRange, renormalization is performed as specified in subclause . 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 9‑8 – Flowchart of renormalization in the encoder

The PutBit( ) procedure described in 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 9‑10 – 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 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 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 coding block 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 minimum CUs PicSizeInMinCbs 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 , BinCountsInNALunits is incremented as specified in subclauses , , and .

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 − RawMinCUBits \* PicSizeInMinCbs ) ÷ 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.

# Specification of bitstream subsets

Subclause  specifies the sub-bitstream extraction process.

## Sub-bitstream extraction process

It is requirement of bitstream conformance that any sub-bitstream that is the output of the process specified in this subclause with tIdTarget equal to any value in the range of 0 to 7, inclusive, shall be conforming to this Recommendation | International Standard.

NOTE – A conforming bitstream contains one or more coded slice NAL units with temporal\_id equal to 0.

Input to this process is a variable tIdTarget.

Output of this process is a sub-bitstream.

The sub-bitstream is derived by removing from the bitstream all NAL units for which temporal\_id is greater than tIdTarget.

1. Annex A  
     
   Profiles and levels

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

* 1. Overview of profiles and levels

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  are also referred to as the profiles specified in Annex .

* 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
     1. General

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 are constraints for sequence parameter sets that are activated in the bitstream.

* + 1. Main profile

Bitstreams conforming to the Main profile shall obey the following constraints:

* Sequence parameter sets shall have chroma\_format\_idc equal to 1 only.
* Sequence parameter sets shall have bit\_depth\_luma\_minus8 equal to 0 only.
* Sequence parameter sets shall have bit\_depth\_chroma\_minus8 equal to 0 only.
* Sequence parameter sets shall have adaptive\_loop\_filter\_enabled\_flag equal to 0 only.
* Sequence parameter sets shall have chroma\_pred\_from\_luma\_enabled\_flag equal to 0 only.
* Sequence parameter sets shall have inter\_4x4\_enabled\_flag equal to 0 only.
* Sequence parameter sets shall have asymmetric\_motion\_partitions\_enabled\_flag equal to 0 only.

[Ed. (KM): Confirm that this flag has been defined. (GJS): It is now in the SPS, but not yet connected to the decoding process.]

* Sequence parameter sets shall have nsrqt\_enabled\_flag equal to 0 only.
* Sequence parameter sets shall have seq\_parameter\_set\_id in the range of 0 to 15, inclusive.
* Log2CtbSize shall be in the range from 4 to 6, inclusive.
* SliceGranularity shall be equal to 0.
* Picture parameter sets shall have tiles\_or\_entropy\_coding\_sync\_idc in the range of 0 to 1, inclusive.
* Picture parameter sets shall have dependent\_slice\_enabled\_flag equal to 0.
* When tiles\_or\_entropy\_coding\_sync\_idc is equal to 1, ColumnWidthInLumaSamples[ i ] shall be greater than or equal to 384 for any i in the range of 0 to num\_tile\_columns\_minus1, inclusive.

[Ed. (KM): Confirm that this is the best formulation]

* Picture parameter sets shall have pic\_parameter\_set\_id in the range of 0 to 63, inclusive.
* Adaptation parameter sets shall have aps\_id in the range of 0 to 63, inclusive.
* The level constraints specified for the Main profile in subclause shall be fulfilled.

Conformance of a bitstream to the Main profile is indicated by profile\_idc being equal to 1.

Decoders conforming to the Main profile at a specific level (identified by a specific value of level\_idc) shall be capable of decoding all bitstreams in which the profile\_idc is equal to 1 and level\_idc represents a level lower than or equal to the specified level. For purposes of comparison of level capabilities, a particular level shall be considered to be a lower level than some other level if the level\_idc of the particular level is less than the level\_idc of the other level.

* 1. Levels
     1. General

The following is specified for expressing the constraints in this annex.

* Let access unit n be the n-th access unit in decoding order, with the first access unit being access unit 0.
* Let picture n be the coded picture or the corresponding decoded picture of access unit n.
* Let the variable fR be set to 1 ÷ 300.

Bitstreams conforming to a profile at a specified level shall obey the following constraints:

1. The nominal removal time of access unit n (with n > 0) from the CPB as specified in subclause , satisfies the constraint that tr,n( n ) − tr( n − 1 ) is greater than or equal to Max( PicSizeLuma ÷ MaxLumaPR, fR ), where MaxLumaPR is the value specified in that applies to picture n − 1, and PicSizeLuma is the number of pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples in picture n − 1.
2. The difference between consecutive output times of pictures from the DPB as specified in subclause , satisfies the constraint that Δto,dpb( n ) >= Max( PicSizeLuma ÷ MaxLumaPR, fR ), where MaxLumaPR is the value specified in for picture n, and PicSizeLuma is the number of pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples of picture n, provided that picture n is a picture that is output and is not the last picture of the bitstream that is output.
3. pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples <= MaxLumaFS, where MaxLumaFS is specified in
4. pic\_width\_in\_luma\_samples <= Sqrt( MaxLumaFS \* 8 )
5. pic\_height\_in\_luma\_samples <= Sqrt( MaxLumaFS \* 8 )
6. sps\_max\_dec\_pic\_buffering[ sps\_max\_temporal\_layers\_minus1 ] <= MaxDpbSize, where MaxDpbSize is specified in .

specifies the limits for each level. The use of the MinCR parameter column of is specified in subclause .

A level to which the bitstream conforms shall be indicated by the syntax element level\_idc as follows.

– level\_idc shall be set equal to a value of 30 times the level number specified in .

[Ed. (YK): Putting level\_idc 30 times instead of 10 times the level number can solve the level 1b problem occurred in AVC.]

Table A‑1 – Level limits

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Level** | **Max luma sample rate MaxLumaPR**  **(samples/sec)** | **Max luma picture size MaxLumaFS (samples)** | **Max bit rate MaxBR**  **(1000 bits/s)** | **Min Compression Ratio MinCR** | **MaxDpbSize (picture storage buffers)** | **Max CPB size**  **(1000 bits)** | num\_tile\_columns\_minus1+1 | num\_tile\_rows\_minus1 + 1 | **uniform\_spacing\_flag** |
| **1** | 552,960 | 36,864 | 128 | 2 | 6 | 350 | - | - | - |
| **2** | 3,686,400 | 122,880 | 1,000 | 2 | 6 | 1,000 | - | - | - |
| **3** | 13,762,560 | 458,752 | 5,000 | 2 | 6 | 5,000 | - | - | - |
| **3.1** | 33,177,600 | 983,040 | 9,000 | 2 | 6 | 9,000 | - | - | - |
| **4** | 62,668,800 | 2,088,960 | 15,000 | 4 | 6 | 15,000 | - | - | - |
| **4.1** | 62,668,800 | 2,088,960 | 30,000 | 4 | 6 | 30,000 | - | - | - |
| **4.2** | 133,693,440 | 2,228,224 | 30,000 | 4 | 6 | 30,000 | - | - | - |
| **4.3** | 133,693,440 | 2,228,224 | 50,000 | 4 | 6 | 50,000 | - | - | - |
| **5** | 267,386,880 | 8,912,896 | 50,000 | 6 | 6 | 50,000 | - | - | - |
| **5.1** | 267,386,880 | 8,912,896 | 100,000 | 8 | 6 | 100,000 | - | - | - |
| **5.2** | 534,773,760 | 8,912,896 | 150,000 | 8 | 6 | 150,000 | 1 | 2 | 1 |
| **6** | 1,002,700,800 | 33,423,360 | 300,000 | 8 | 6 | 300,000 | 2 | 2 | 1 |
| **6.1** | 2,005,401,600 | 33,423,360 | 500,000 | 8 | 6 | 500,000 | 2 | 4 | 1 |
| **6.2** | 4,010,803,200 | 33,423,360 | 800,000 | 6 | 6 | 800,000 | 2 | 8 | 1 |

Informative subclause  shows the effect of these limits on picture rates for several example picture formats.

* + 1. Profile-specific level limits for the Main profile

1. In bitstreams conforming to the Main profile, the removal time of access unit 0 shall satisfy the constraint that the number of slices in picture 0 is less than or equal to ( Max( PicSizeLuma, fR \* MaxLumaPR ) + MaxLumaPR \* ( tr( 0 ) − tr,n( 0 ) ) ) ÷ SliceRate, where MaxLumaPR and SliceRate are the values specified in and , respectively, that apply to picture 0 and PicSizeLuma is the number of pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples in picture 0.
2. In bitstreams conforming to the Main profile, the difference between consecutive removal time of access units n and n − 1 with n > 0 shall satisfy the constraint that the number of slices in picture n is less than or equal to MaxLumaPR \* ( tr( n ) − tr( n − 1 ) ) ÷ SliceRate, where MaxLumaPR and SliceRate are the values specified in and , respectively, that apply to picture n.
3. In bitstreams conforming to the Main profile, for the VCL HRD parameters, BitRate[ SchedSelIdx ] <= cpbBrVclFactor \* MaxBR and CpbSize[ SchedSelIdx ] <= cpbBrVclFactor \* MaxCPB for at least one value of SchedSelIdx, where cpbBrVclFactor is specified in and BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are given as follows.

* If vcl\_hrd\_parameters\_present\_flag is equal to 1, BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are given by Equations E‑37 and E‑38, respectively, using the syntax elements of the hrd\_parameters( ) syntax structure that immediately follows vcl\_hrd\_parameters\_present\_flag.
* Otherwise (vcl\_hrd\_parameters\_present\_flag is equal to 0), BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are inferred as specified in subclause E.2.2 for VCL HRD parameters.

MaxBR and MaxCPB are specified in in units of cpbBrVclFactor bits/s and cpbBrVclFactor bits, respectively. The bitstream shall satisfy these conditions for at least one value of SchedSelIdx in the range 0 to cpb\_cnt\_minus1, inclusive.

1. In bitstreams conforming to the Main profile, for the NAL HRD parameters, BitRate[ SchedSelIdx ] <= cpbBrNalFactor \* MaxBR and CpbSize[ SchedSelIdx ] <= cpbBrNalFactor \* MaxCPB for at least one value of SchedSelIdx, where cpbBrNalFactor is specified in and BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are given as follows.

* If nal\_hrd\_parameters\_present\_flag is equal to 1, BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are given by Equations E‑37 and E‑38, respectively, using the syntax elements of the hrd\_parameters( ) syntax structure that immediately follows nal\_hrd\_parameters\_present\_flag.
* Otherwise (nal\_hrd\_parameters\_present\_flag is equal to 0), BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ] are inferred as specified in subclause E.2.2 for NAL HRD parameters.

MaxBR and MaxCPB are specified in in units of cpbBrNalFactor bits/s and cpbBrNalFactor bits, respectively. The bitstream shall satisfy these conditions for at least one value of SchedSelIdx in the range 0 to cpb\_cnt\_minus1, inclusive.

1. In bitstreams conforming to the Main profile, the sum of the NumBytesInNALunit variables for access unit 0 is less than or equal to 1.5 \*  ( Max( PicSizeLuma, fR \* MaxLumaPR ) + MaxLumaPR \*  ( tr( 0 ) − tr,n( 0 ) )) ÷ MinCR, where MaxLumaPR and MinCR are the values specified in that apply to picture 0 and PicSizeLuma is the number of pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples in picture 0.
2. In bitstreams conforming to the Main profile, the sum of the NumBytesInNALunit variables for access unit n with n > 0 is less than or equal to 1.5 \* MaxLumaPR \* ( tr( n ) − tr( n − 1 ) ) ÷ MinCR, where MaxLumaPR and MinCR are the values specified in that apply to picture n.

[Ed. (KM): check against Annex E]

Table A‑2 – Specification of cpbBrVclFactor and cpbBrNalFactor

|  |  |  |
| --- | --- | --- |
| **Profile** | **cpbBrVclFactor** | **cpbBrNalFactor** |
| **Main** | 1 000 | 1 100 |

specifies limits for each level that are specific to bitstreams conforming to the Main profile. The mark "-" in denotes the absence of a corresponding limit.

NOTE – The mark "-" in does not indicate that the corresponding level is undefined for the Main profile.

Table A‑ – SliceRate limits for Main profile

|  |  |
| --- | --- |
| **Level number** | **SliceRate** |
| **1** | - |
| **2** | - |
| **3** | 22 |
| **3.1** | 60 |
| **4** | 60 |
| **4.1** | 24 |
| **4.2** | 24 |
| **4.3** | 24 |
| **5** | 24 |
| **5.1** | 24 |
| **5.2** | 24 |
| **6** | 24 |
| **6.1** | 24 |
| **6.2** | 24 |

* + 1. Effect of level limits on picture rate (informative)

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

Table A‑ – Maximum picture rates (pictures per second) at level 1 to 4.3 for some example picture sizes

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Level:** |  |  |  | **1** | **2** | **3** | **3.1** | **4, 4.1** | **4.2, 4.3** |
| **Max picture size (luma samples):** |  |  |  | 36,864 | 122,880 | 458,752 | 983,040 | 2,088,960 | 2,228,224 |
| **Max luma sample rate (samples/sec)** |  |  |  | 552,960 | 3,686,400 | 13,762,560 | 33,177,600 | 62,668,800 | 133,693,440 |
| **Format** | **Luma Width** | **Luma Height** | **Picture Size** |  |  |  |  |  |  |
| **SQCIF** | **128** | **96** | 16,384 | 33.7 | 225.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **QCIF** | **176** | **144** | 36,864 | 15.0 | 100.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **QVGA** | **320** | **240** | 81,920 | - | 45.0 | 168.0 | 300.0 | 300.0 | 300.0 |
| **525 SIF** | **352** | **240** | 98,304 | - | 37.5 | 140.0 | 300.0 | 300.0 | 300.0 |
| **CIF** | **352** | **288** | 122,880 | - | 30.0 | 112.0 | 270.0 | 300.0 | 300.0 |
| **525 HHR** | **352** | **480** | 196,608 | - | - | 70.0 | 168.7 | 300.0 | 300.0 |
| **625 HHR** | **352** | **576** | 221,184 | - | - | 62.2 | 150.0 | 283.3 | 300.0 |
| **VGA** | **640** | **480** | 327,680 | - | - | 42.0 | 101.2 | 191.2 | 300.0 |
| **525 4SIF** | **704** | **480** | 360,448 | - | - | 38.1 | 92.0 | 173.8 | 300.0 |
| **525 SD** | **720** | **480** | 393,216 | - | - | 35.0 | 84.3 | 159.3 | 300.0 |
| **4CIF** | **704** | **576** | 405,504 | - | - | 33.9 | 81.8 | 154.5 | 300.0 |
| **625 SD** | **720** | **576** | 442,368 | - | - | 31.1 | 75.0 | 141.6 | 300.0 |
| **480p (16:9)** | **854** | **480** | 458,752 | - | - | 30.0 | 72.3 | 136.6 | 291.4 |
| **SVGA** | **800** | **600** | 532,480 | - | - | - | 62.3 | 117.6 | 251.0 |
| **QHD** | **960** | **544** | 552,960 | - | - | - | 60.0 | 113.3 | 241.7 |
| **XGA** | **1024** | **768** | 786,432 | - | - | - | 42.1 | 79.6 | 170.0 |
| **720p HD** | **1280** | **720** | 983,040 | - | - | - | 33.7 | 63.7 | 136.0 |
| **4VGA** | **1280** | **960** | 1,228,800 | - | - | - | - | 51.0 | 108.8 |
| **SXGA** | **1280** | **1024** | 1,310,720 | - | - | - | - | 47.8 | 102.0 |
| **525 16SIF** | **1408** | **960** | 1,351,680 | - | - | - | - | 46.3 | 98.9 |
| **16CIF** | **1408** | **1152** | 1,622,016 | - | - | - | - | 38.6 | 82.4 |
| **4SVGA** | **1600** | **1200** | 1,945,600 | - | - | - | - | 32.2 | 68.7 |
| **1080 HD** | **1920** | **1088** | 2,088,960 | - | - | - | - | 30.0 | 64.0 |
| **2Kx1K** | **2048** | **1024** | 2,097,152 | - | - | - | - | - | 63.7 |
| **2Kx1080** | **2048** | **1088** | 2,228,224 | - | - | - | - | - | 60.0 |
| **4XGA** | **2048** | **1536** | 3,145,728 | - | - | - | - | - | - |
| **16VGA** | **2560** | **1920** | 4,915,200 | - | - | - | - | - | - |
| **3616x1536 (2.35:1)** | **3616** | **1536** | 5,603,328 | - | - | - | - | - | - |
| **3672x1536 (2.39:1)** | **3680** | **1536** | 5,701,632 | - | - | - | - | - | - |
| **4Kx2K** | **4096** | **2048** | 8,388,608 | - | - | - | - | - | - |
| **4096x2160** | **4096** | **2160** | 8,912,896 | - | - | - | - | - | - |
| **4096x2304 (16:9)** | **4096** | **2304** | 9,437,184 | - | - | - | - | - | - |
| **7680x4320** | **7680** | **4320** | 33,423,360 | - | - | - | - | - | - |

Table A‑ – Maximum picture rates (pictures per second) at level 5 to 6.2 for some example picture sizes

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Level:** |  |  |  | **5, 5.1** | **5.2** | **6** | **6.1** | **6.2** |
| **Max picture size (samples):** |  |  |  | 8,912,896 | 8,912,896 | 33,423,360 | 33,423,360 | 33,423,360 |
| **Max luma sample rate (samples/sec)** |  |  |  | 267,386,880 | 534,773,760 | 1,002,700,800 | 2,005,401,600 | 4,010,803,200 |
| **Format** | **Luma Width** | **Luma Height** | **Picture Size** |  |  |  |  |  |
| **SQCIF** | **128** | **96** | 16,384 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **QCIF** | **176** | **144** | 36,864 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **QVGA** | **320** | **240** | 81,920 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **525 SIF** | **352** | **240** | 98,304 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **CIF** | **352** | **288** | 122,880 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **525 HHR** | **352** | **480** | 196,608 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **625 HHR** | **352** | **576** | 221,184 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **VGA** | **640** | **480** | 327,680 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **525 4SIF** | **704** | **480** | 360,448 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **525 SD** | **720** | **480** | 393,216 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **4CIF** | **704** | **576** | 405,504 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **625 SD** | **720** | **576** | 442,368 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **480p (16:9)** | **854** | **480** | 458,752 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **SVGA** | **800** | **600** | 532,480 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **QHD** | **960** | **544** | 552,960 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **XGA** | **1024** | **768** | 786,432 | 300.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **720p HD** | **1280** | **720** | 983,040 | 272.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **4VGA** | **1280** | **960** | 1,228,800 | 217.6 | 300.0 | 300.0 | 300.0 | 300.0 |
| **SXGA** | **1280** | **1024** | 1,310,720 | 204.0 | 300.0 | 300.0 | 300.0 | 300.0 |
| **525 16SIF** | **1408** | **960** | 1,351,680 | 197.8 | 300.0 | 300.0 | 300.0 | 300.0 |
| **16CIF** | **1408** | **1152** | 1,622,016 | 164.8 | 300.0 | 300.0 | 300.0 | 300.0 |
| **4SVGA** | **1600** | **1200** | 1,945,600 | 137.4 | 274.8 | 300.0 | 300.0 | 300.0 |
| **1080 HD** | **1920** | **1088** | 2,088,960 | 128.0 | 256.0 | 300.0 | 300.0 | 300.0 |
| **2Kx1K** | **2048** | **1024** | 2,097,152 | 127.5 | 255.0 | 300.0 | 300.0 | 300.0 |
| **2Kx1080** | **2048** | **1088** | 2,228,224 | 120.0 | 240.0 | 300.0 | 300.0 | 300.0 |
| **4XGA** | **2048** | **1536** | 3,145,728 | 85.0 | 170.0 | 300.0 | 300.0 | 300.0 |
| **16VGA** | **2560** | **1920** | 4,915,200 | 54.4 | 108.8 | 204.0 | 300.0 | 300.0 |
| **3616x1536 (2.35:1)** | **3616** | **1536** | 5,603,328 | 47.7 | 95.4 | 178.9 | 300.0 | 300.0 |
| **3672x1536 (2.39:1)** | **3680** | **1536** | 5,701,632 | 46.8 | 93.7 | 175.8 | 300.0 | 300.0 |
| **4Kx2K** | **4096** | **2048** | 8,388,608 | 31.8 | 63.7 | 119.5 | 239.0 | 300.0 |
| **4096x2160** | **4096** | **2160** | 8,912,896 | 30.0 | 60.0 | 112.5 | 225.0 | 300.0 |
| **4096x2304 (16:9)** | **4096** | **2304** | 9,437,184 | - | - | 106.2 | 212.5 | 300.0 |
| **7680x4320** | **7680** | **4320** | 33,423,360 | - | - | 30.0 | 60.0 | 120.0 |

The following should be noted:

– This Recommendation | International Standard is a variable-picture-size specification. The specific picture sizes in and are illustrative examples only.

– As used in and , "525" refers to typical use for environments using 525 analogue scan lines (of which approximately 480 lines contain the visible picture region), and "625" refers to environments using 625 analogue scan lines (of which approximately 576 lines contain the visible picture region).

– XGA is also known as (aka) XVGA, 4SVGA aka UXGA, 16XGA aka 4Kx3K, CIF aka 625 SIF, 625 HHR aka 2CIF aka half 625 D-1, aka half 625 ITU-R BT.601, 525 SD aka 525 D-1 aka 525 ITU-R BT.601, 625 SD aka 625 D-1 aka 625 ITU-R BT.601.

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 Rec. ITU-T H.222.0 | ISO/IEC 13818-1 systems or Rec. ITU-T 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 ). 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 ).

**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 ) 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 one or more of the following conditions are true, the zero\_byte syntax element shall be present:

– the nal\_unit\_type within the nal\_unit( ) is equal to 25 (video parameter set), 26 (sequence parameter set), 27 (picture parameter set) or 28 (adaptation 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 .

**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 one or more 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 .

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 ), 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)

* 1. General

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 ).

shows the types of bitstream conformance points checked by the HRD.



Figure C‑1 – Structure of byte streams and NAL unit streams for HRD conformance checks [Ed. (GJS): Fix figure numbering in annex(es)]

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 .

Two types of HRD parameter sets (NAL HRD parameters and VCL HRD parameters) are used. The HRD parameter sets are signalled through video usability information as specified in subclauses  and , which is part of the sequence parameter set syntax structure.

All sequence parameter sets and picture parameter sets referred to in the VCL NAL units, and corresponding buffering period and picture timing SEI messages shall be conveyed to the HRD, in a timely manner, either in the bitstream (by non-VCL NAL units), or by other means not specified in this Recommendation | International Standard.

In Annexes , , and , the specification for "presence" of non-VCL NAL units is also satisfied when those NAL units (or just some of them) are conveyed to decoders (or to the HRD) by other means not specified by this Recommendation | International Standard. For the purpose of counting bits, only the appropriate bits that are actually present in the bitstream are counted.

NOTE 3 – As an example, synchronization of a non-VCL NAL unit, conveyed by means other than presence in the bitstream, with the NAL units that are present in the bitstream, can be achieved by indicating two points in the bitstream, between which the non‑VCL NAL unit would have been present in the bitstream, had the encoder decided to convey it in the bitstream.

When the content of a non-VCL NAL unit is conveyed for the application by some means other than presence within the bitstream, the representation of the content of the non-VCL NAL unit is not required to use the same syntax specified in this annex.

NOTE 4 – When HRD information is contained within the bitstream, it is possible to verify the conformance of a bitstream to the requirements of this subclause based solely on information contained in the bitstream. When the HRD information is not present in the bitstream, as is the case for all "stand-alone" Type I bitstreams, conformance can only be verified when the HRD data is supplied by some other means not specified in this Recommendation | International Standard.

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‑2 – HRD buffer model

[Ed. (JS): Remove "fields" from figure. (DF) Rename "frames" to "pictures"? (GJS): Yes, and I believe our current model now always stores the current picture, so the bypass around the DPB should be removed. (YK): And, "Access units" -> "decoding units"; "Reference fields or frames" -> "reference pictures"; "Fields of frames" (first instance) -> "decoded decoding units"; "Fields of frames" (second instance) -> "decoded pictures"; "Output cropped fields of frames" -> "Output cropped pictures"]

The CPB size (number of bits) is CpbSize[ SchedSelIdx ]. The DPB size (number of picture storage buffers) for temporal layer X is sps\_max\_dec\_pic\_buffering[ X ] for each X in the range of 0 to sps\_max\_temporal\_layers\_minus1, inclusive. [Ed. (GJS): Think about using MaxDpbSize versus max\_dec\_pic\_buffering (various places in document).]

The variable SubPicCpbPreferredFlag is either specified by external means, or when not specified by external means, set to 0.

The variable SubPicCpbFlag is derived as follows:

SubPicCpbFlag = SubPicCpbPreferredFlag && sub\_pic\_cpb\_params\_present\_flag (C‑1)

If SubPicCpbFlag is equal to 0, the CPB operates at access unit level and each decoding unit is an access unit. Otherwise the CPB operates at sub-picture level and each decoding unit is a subset of an access unit.

The HRD operates as follows. Data associated with decoding units that flow into the CPB according to a specified arrival schedule are delivered by the HSS. The data associated with each decoding unit are removed and decoded instantaneously by the instantaneous decoding process at CPB removal times. Each decoded picture is placed in the DPB. A decoded picture is removed from the DPB at the later of the DPB output time or the time that it becomes no longer needed for inter-prediction reference.

The operation of the CPB is specified in subclause C.2. The instantaneous decoder operation is specified in clauses 2-9. The operation of the DPB is specified in subclause C.3. The output cropping is specified in subclause C.3.2.

HSS and HRD information concerning the number of enumerated delivery schedules and their associated bit rates and buffer sizes is specified in subclauses , , , and . The HRD is initialised as specified by the buffering period SEI message as specified in subclauses D.1.1 and D.2.1. The removal timing of decoding units from the CPB and output timing of decoded pictures from the DPB are specified in the picture timing SEI message as specified in subclauses D.1.2 and D.2.1. All timing information relating to a specific decoding unit shall arrive prior to the CPB removal time of the decoding unit.

The HRD is used to check conformance of bitstreams and decoders as specified in subclauses C.4 and C.5, respectively.

NOTE 7 – While conformance is guaranteed under the assumption that all frame-rates and clocks used to generate the bitstream match exactly the values signalled in the bitstream, in a real system each of these may vary from the signalled or specified value.

All the arithmetic in this annex is done with real values, so that no rounding errors can propagate. For example, the number of bits in a CPB just prior to or after removal of a decoding unitis not necessarily an integer.

The variable tc is derived as follows and is called a clock tick:

tc = num\_units\_in\_tick  time\_scale (C‑1)

The variable tc\_sub is derived as follows and is called a sub-picture clock tick:

tc\_sub = num\_units\_in\_sub\_tick  time\_scale (C‑1)

The following is specified for expressing the constraints in this annex:

– Let access unit n be the n-th access unit in decoding order with the first access unit being access unit 0.

– Let picture n be the coded picture or the decoded picture of access unit n.

– Let decoding unit m be the m-th decoding unit in decoding order with the first decoding unit being decoding unit 0.

* 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 .

* + 1. Timing of bitstream arrival

The HRD may be initialised at any one of the buffering period SEI messages. Prior to initialisation, the CPB is empty.

NOTE – After initialisation, the HRD is not initialised again by subsequent buffering period SEI messages.

Each access unit is referred to as access unit n, where the number n identifies the particular access unit. The access unit that is associated with the buffering period SEI message that initialises the CPB is referred to as access unit 0. The value of n is incremented by 1 for each subsequent access unit in decoding order.

Each decoding unit is referred to as decoding unit m, where the number m identifies the particular decoding unit. The first decoding unit in decoding order in access unit 0 is referred to as decoding unit 0. The value of m is incremented by 1 for each subsequent decoding unit in decoding order.

The variables InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are set as follows.

– If either of the following conditions is true:, InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are set to the values of the corresponding initial\_alt\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_alt\_cpb\_removal\_delay\_offset[ SchedSelIdx ], respectively, of the associated buffering period SEI message:

– Access unit 0 is a CRA or BLA access unit for which the coded picture has nal\_unit\_type equal to 5 or 7, and the value of rap\_cpb\_params\_present\_flag of the associated buffering period SEI message is equal to 1;

– SubPicCpbFlag is equal to 1.

– Otherwise, InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are set to the values of the corresponding initial\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_cpb\_removal\_delay\_offset[ SchedSelIdx ], respectively, of the associated buffering period SEI message.

The time at which the first bit of decoding unit n begins to enter the CPB is referred to as the initial arrival time tai( m ).

The initial arrival time of decoding units is derived as follows.

– If the decoding unit is decoding unit 0, tai( 0 ) = 0,

– Otherwise (the decoding unit is decoding unit m with m > 0), the following applies.

– If cbr\_flag[ SchedSelIdx ] is equal to 1, the initial arrival time for decoding unit m, is equal to the final arrival time (which is derived below) of decoding unit m − 1, i.e.,

tai( m ) = taf( m − 1 ) (C‑2)

– Otherwise (cbr\_flag[ SchedSelIdx ] is equal to 0), the initial arrival time for decoding unit m is derived by

tai( m ) = Max( taf( m − 1 ), tai,earliest( m ) ) (C‑3)

where tai,earliest( m ) is derived as follows.

– If decoding unit m is not the first decoding unit of a subsequent buffering period, tai,earliest( m ) is derived as

tai,earliest( m ) = tr,n( m ) − ( InitCpbRemovalDelay[ SchedSelIdx ] +  
 InitCpbRemovalDelayOffset[ SchedSelIdx ] )  90000 (C‑4)

with tr,n( m ) being the nominal removal time of decoding unit m from the CPB as specified in subclause 0.

– Otherwise (decoding unit m is the first decoding unit of a subsequent buffering period), tai,earliest( m ) is derived as

tai,earliest( m ) = tr,n( m ) − ( InitCpbRemovalDelay[ SchedSelIdx ]  90000 ) (C‑5)

The final arrival time for decoding unit m is derived by

taf( m ) = tai( m ) + b( m )  BitRate[ SchedSelIdx ] (C‑6)

where b( m ) is the size in bits of decoding unit m, counting the bits of the VCL NAL units and the filler data NAL units for the Type I conformance point or all bits of the Type II bitstream for the Type II conformance point, where the Type I and Type II conformance points are as shown in Figure C‑1.

The values of SchedSelIdx, BitRate[ SchedSelIdx ], and CpbSize[ SchedSelIdx ] are constrained as follows.

– If the content of the active sequence parameter sets for the access unit containing decoding unit m and the previous access unit differ, the HSS selects a value SchedSelIdx1 of SchedSelIdx from among the values of SchedSelIdx provided in the active sequence parameter set for the access unit containing decoding unit m that results in a BitRate[ SchedSelIdx1 ] or CpbSize[ SchedSelIdx1 ] for the access unit containing decoding unit m. The value of BitRate[ SchedSelIdx1 ] or CpbSize[ SchedSelIdx1 ] may differ from the value of BitRate[ SchedSelIdx0 ] or CpbSize[ SchedSelIdx0 ] for the value SchedSelIdx0 of SchedSelIdx that was in use for the previous access unit.

– Otherwise, the HSS continues to operate with the previous values of SchedSelIdx, BitRate[ SchedSelIdx ] and CpbSize[ SchedSelIdx ].

When the HSS selects values of BitRate[ SchedSelIdx ] or CpbSize[ SchedSelIdx ] that differ from those of the previous access unit, the following applies.

– the variable BitRate[ SchedSelIdx ] comes into effect at time tai( m )

– the variable CpbSize[ SchedSelIdx ] comes into effect as follows.

– If the new value of CpbSize[ SchedSelIdx ] exceeds the old CPB size, it comes into effect at time tai( m ),

– Otherwise, the new value of CpbSize[ SchedSelIdx ] comes into effect at the CPB removal time of the last decoding unit of the access unit containing decoding unit m.

When SubPicCpbFlag is equal to 1, the initial CPB arrival time of access unit n tai( n ) is set to the initial CPB arrival time of the first decoding unit in access unit n, and the final CPB arrival time of access unit n taf( n ) is set to the final CPB arrival time of the last decoding unit in access unit n.

NOTE – When SubPicCpbFlag is equal to 0, each decoding unit is an access unit, hence the initial and final CPB arrival times of access unit n are the initial and final CPB arrival times of decoding unit n.

* + 1. Timing of decoding unit removal and decoding of decoding unit

If SubPicCpbFlag is equal to 0, the variable CpbRemovalDelay( m ) is set to the value of cpb\_removal\_delay in the picture timing SEI message associated with the access unit that is decoding unit m, and the variable Tc is set to tc. Otherwise the variable CpbRemovalDelay( m ) is set to the value of du\_cpb\_removal\_delay[ i ] for decoding unit m in the picture timing SEI message associated with the access unit that contains decoding unit m, and the variable Tc is set to tc\_sub.

When a decoding unit m is the decoding unit with n equal to 0 (the first decoding unit of the access unit that initialises the HRD), the nominal removal time of the decoding unit from the CPB is specified by

tr,n( 0 ) = InitCpbRemovalDelay[ SchedSelIdx ]  90000 (C‑7)

When a decoding unit m is the first decoding unit of the first access unit of a buffering period that does not initialise the HRD, the nominal removal time of the decoding unit from the CPB is specified by

tr,n( m ) = tr,n( mb ) + Tc \* CpbRemovalDelay( m ) (C‑8)

where tr,n( mb ) is the nominal removal time of the first decoding unit of the previous buffering period.

When a decoding unit m is the first decoding unit of a buffering period, mb is set equal to m at the removal time tr,n( m ) of the decoding unit m.

The nominal removal time tr,n( m ) of a decoding unit m that is not the first decoding unit of a buffering period is given by

tr,n( m ) = tr,n( mb ) + Tc \* CpbRemovalDelay( m ) (C‑9)

where tr,n( mb ) is the nominal removal time of the first decoding unit of the current buffering period.

The removal time of decoding unit m is specified as follows.

– If low\_delay\_hrd\_flag is equal to 0 or tr,n( m ) >= taf( m ), the removal time of decoding unit m is specified by

tr( m ) = tr,n( m ) (C‑10)

– Otherwise (low\_delay\_hrd\_flag is equal to 1 and tr,n( m ) < taf( m )), the removal time of decoding unit m is specified by

tr( m ) = tr,n( m ) + Tc \* Ceil( ( taf( m ) − tr,n( m ) ) c ) (C‑11)

NOTE 1 – The latter case indicates that the size of decoding unit m, b( m ), is so large that it prevents removal at the nominal removal time.

When SubPicCpbFlag is equal to 1, the nominal CPB removal time of access unit n tr,n( n ) is set to the nominal CPB removal time of the last decoding unit in access unit n, the CPB removal time of access unit n tr( n ) is set to the CPB removal time of the last decoding unit in access unit n.

NOTE 2 – When SubPicCpbFlag is equal to 0, each decoding unit is an access unit, hence the nominal CPB removal time and the CPB revmoal time of access unit n are the nominal CPB removal time and the CPB removal time of decoding unit n.

At CPB removal time of decoding unit m, the decoding unit is instantaneously decoded.

* 1. Operation of the decoded picture buffer (DPB)

The decoded picture buffer contains picture storage buffers. Each of the picture storage buffers may contain a decoded picture that is marked as "used for reference" or is held for future output. Prior to initialization, the DPB is empty (the DPB fullness is set to zero). The following steps of the subclauses of this subclause happen in the sequence as listed below.

* + 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) happens instantaneously at the CPB removal time of the first decoding unit of access unit n (containing the current picture) and proceeds as follows.

The decoding process for reference picture set as specified in subclause  is invoked.

If the current picture is an IDR or a BLA picture, the following applies:

1. When the IDR or BLA picture is not the first IDR or BLA picture decoded and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or sps\_max\_dec\_pic\_buffering[ i ] for any possible value of i 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 sps\_max\_dec\_pic\_buffering[ i ] 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 1 – Decoder implementations should try to handle picture or DPB size changes more gracefully than the HRD in regard to changes in pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples, or sps\_max\_dec\_pic\_buffering[ i ].

1. When no\_output\_of\_prior\_pics\_flag is equal to 1 or is inferred to be equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain, and DPB fullness is set to 0.

All pictures k in the DPB, for which all of the following conditions are true, are removed from the DPB:

– picture k is marked as "unused for reference",

– picture k has PicOutputFlag equal to 0 or its DPB output time is less than or equal to the CPB removal time of the first decoding unit (denoted as decoding unit m) of the current picture n; i.e., to,dpb( k ) <= tr( m )

When a picture is removed from the DPB, the DPB fullness is decremented by one.

* + 1. **Picture output**

The following happens instantaneously at the CPB removal time of access unit n, tr( n ).

Picture n is considered as decoded after the last decoding unit of the picture is decoded.

When picture n has PicOutputFlag equal to 1, its DPB output time to,dpb( n ) is derived by

to,dpb( n ) = tr( n ) + tc \* dpb\_output\_delay( n ) (C‑12)

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 PicOutputFlag is equal to 1 and to,dpb( n ) = tr( n ), the current picture is output.

– Otherwise, if PicOutputFlag is equal to 0, the current picture is not output, but will be stored in the DPB as specified in subclause .

– Otherwise (PicOutputFlag 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 ) 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‑13)

where nn indicates the picture that follows after picture n in output order and has PicOutputFlag equal to 1.

* + 1. **Current decoded picture marking and storage**

The following happens instantaneously at the CPB removal time of access unit n, tr( n ).

The current decoded picture is stored in the DPB in an empty picture storage 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

A bitstream of coded data conforming to this Recommendation | International Standard shall fulfil the following requirements.

It is a requirement of bitstream conformance that the bitstream shall be constructed according to the syntax, semantics, and constraints specified in this Recommendation | International Standard outside of this annex.

It is a requirement of bitstream conformance that the first coded picture in a bitstream shall be a RAP picture, i.e., an IDR picture or a CRA picture or a BLA picture.

It is a requirement of bitstream conformance that any TFD picture associated with a particular RAP picture shall follow in output order any another RAP picture preceding the particular RAP picture in decoding order.

NOTE 1 – This constraint guarantees decodability of a TFD picture if its associated RAP picture is a CRA picture and if that CRA picture is not the first coded picture in the bitstream.

It is a requirement of bitstream conformance that any coded picture that follows a RAP picture in decoding order and uses inter prediction either from any picture that precedes the RAP picture in decoding order or from another TFD picture associated with the RAP picture shall be a TFD picture associated with the RAP picture.

NOTE 2 – Any coded picture that follows a RAP picture in decoding order and uses inter prediction either from any picture that precedes the RAP picture in decoding order or from another TFD picture associated with the RAP picture must precede the RAP in output order.

For any current picture being decoded, let the variables maxPicOrderCnt and minPicOrderCnt be set equal to the maximum and the mininum, respectively, of the PicOrderCntVal values for the current picture and all pictures in the DPB that are currently marked as "used for short-term reference" or "needed for output", and let the variable prevRefPic be the previous reference picture in decoding order that has temporal\_id equal to or less than the temporal\_id of the current picture. It is a requirement of bitstream conformance that the value of maxPicOrderCnt – minPicOrderCnt shall be less than MaxPicOrderCntLsb / 2, and that the absolute difference between the PicOrderCntVal values for the current picture and for prevRefPic shall be less than MaxPicOrderCntLsb / 2.

[Ed. (YK): Rewording of "needed for output" needed, as a picture may only be marked as "needed for output" when the output order DPB operation is applied.]

The bitstream is tested by the HRD for conformance as specified below:

For Type I bitstreams, the number of tests carried out is equal to cpb\_cnt\_minus1 + 1 where cpb\_cnt\_minus1 is either the syntax element of hrd\_parameters( ) following the vcl\_hrd\_parameters\_present\_flag or is determined by the application by other means not specified in this Recommendation | International Standard. One test is carried out for each bit rate and CPB size combination specified by hrd\_parameters( ) following the vcl\_hrd\_parameters\_present\_flag. Each of these tests is conducted at the Type I conformance point shown in Figure C‑1.

For Type II bitstreams there are two sets of tests. The number of tests of the first set is equal to cpb\_cnt\_minus1 + 1 where cpb\_cnt\_minus1 is either the syntax element of hrd\_parameters( ) following the vcl\_hrd\_parameters\_present\_flag or is determined by the application by other means not specified in this Recommendation | International Standard. One test is carried out for each bit rate and CPB size combination. Each of these tests is conducted at the Type I conformance point shown in Figure C‑1. For these tests, only VCL and filler data NAL units are counted for the input bit rate and CPB storage.

The number of tests of the second set, for Type II bitstreams, is equal to cpb\_cnt\_minus1 + 1 where cpb\_cnt\_minus1 is either the syntax element of hrd\_parameters( ) following the nal\_hrd\_parameters\_present\_flag or is determined by the application by other means not specified in this Recommendation | International Standard. One test is carried out for each bit rate and CPB size combination specified by hrd\_parameters( ) following the nal\_hrd\_parameters\_present\_flag. Each of these tests is conducted at the Type II conformance point shown in Figure C‑1. For these tests, all NAL units (of a Type II NAL unit stream) or all bytes (of a byte stream) are counted for the input bit rate and CPB storage.

NOTE 3 – NAL HRD parameters established by a value of SchedSelIdx for the Type II conformance point shown in Figure C‑1 are sufficient to also establish VCL HRD conformance for the Type I conformance point shown in Figure C‑1 for the same values of InitCpbRemovalDelay[ SchedSelIdx ], BitRate[ SchedSelIdx ], and CpbSize[ SchedSelIdx ] for the VBR case (cbr\_flag[ SchedSelIdx ] equal to 0). This is because the data flow into the Type I conformance point is a subset of the data flow into the Type II conformance point and because, for the VBR case, the CPB is allowed to become empty and stay empty until the time a next picture is scheduled to begin to arrive. For example, when decoding a coded video sequence conforming to one or more of the profiles specified in Annex A using the decoding process specified in clauses 2-9, when NAL HRD parameters are provided for the Type II conformance point that not only fall within the bounds set for NAL HRD parameters for profile conformance in item d) of subclause A.4.2 but also fall within the bounds set for VCL HRD parameters for profile conformance in item c) of subclause A.4.2, conformance of the VCL HRD for the Type I conformance point is also assured to fall within the bounds of item c) of subclause A.4.2.

It is a requirement of bitstream conformance that all of the following conditions shall be fulfilled for each of the tests:

1. For each access unit n, with n>0, associated with a buffering period SEI message, with Δtg,90( n ) specified by

Δtg,90( n ) = 90000 \* ( tr,n( n ) − taf( n − 1 ) ) (C‑14)

the value of InitCpbRemovalDelay[ SchedSelIdx ] shall be constrained as follows.

– If cbr\_flag[ SchedSelIdx ] is equal to 0,

InitCpbRemovalDelay[ SchedSelIdx ] <= Ceil( Δtg,90( n ) ) (C‑15)

– Otherwise (cbr\_flag[ SchedSelIdx ] is equal to 1),

Floor( Δtg,90( n ) ) <= InitCpbRemovalDelay[ SchedSelIdx ] <= Ceil( Δtg,90( n ) ) (C‑16)

NOTE 4 – The exact number of bits in the CPB at the removal time of each picture may depend on which buffering period SEI message is selected to initialise the HRD. Encoders must take this into account to ensure that all specified constraints must be obeyed regardless of which buffering period SEI message is selected to initialise the HRD, as the HRD may be initialised at any one of the buffering period SEI messages.

1. A CPB overflow is specified as the condition in which the total number of bits in the CPB is larger than the CPB size. The CPB shall never overflow.
2. A CPB underflow is specified as the condition in which the nominal CPB removal time of decoding unit m tr,n( m ) is less than the final CPB arrival time of decoding unit m taf( m ) for any value of m. When low\_delay\_hrd\_flag is equal to 0, the CPB shall never underflow.
3. The nominal removal times of pictures from the CPB (starting from the second picture in decoding order), shall satisfy the constraints on tr,n( n ) and tr( n ) expressed in subclauses A.4.1 through A.4.2.
4. After the decoding process for reference picture set as specified in subclause 8.3.2 has been invoked, the number of decoded pictures in the DPB, not including the current picture, with temporal\_id lower than or equal to the temporal\_id of the current picture, shall be less than or equal to sps\_max\_dec\_pic\_buffering[ temporal\_id ].
5. All reference pictures shall be present in the DPB when needed for prediction. Each picture shall be present in the DPB at its DPB output time unless it is removed from the DPB before its output time by one of the processes specified in subclause C.3.
6. The value of Δto,dpb( n ) as given by Equation , which is the difference between the output time of a picture and that of the first picture following it in output order and having pic\_output\_flag equal to 1, shall satisfy the constraint expressed in subclause A.4.1 for the profile and level specified in the bitstream using the decoding process specified in clauses 2-9.
   1. Decoder conformance

A decoder conforming to this Recommendation | International Standard shall fulfil the following requirements.

A decoder claiming conformance to a specific profile and level shall be able to decode successfully all conforming bitstreams specified for decoder conformance in subclause C.4, provided that all sequence parameter sets and picture parameter sets referred to in the VCL NAL units, and appropriate buffering period and picture timing SEI messages are conveyed to the decoder, in a timely manner, either in the bitstream (by non-VCL NAL units), or by external means not specified by this Recommendation | International Standard.

There are two types of conformance that can be claimed by a decoder: output timing conformance and output order conformance.

To check conformance of a decoder, test bitstreams conforming to the claimed profile and level, as specified by subclause C.4 are delivered by a hypothetical stream scheduler (HSS) both to the HRD and to the decoder under test (DUT). All pictures output by the HRD shall also be output by the DUT and, for each picture output by the HRD, the values of all samples that are output by the DUT for the corresponding picture shall be equal to the values of the samples output by the HRD.

For output timing decoder conformance, the HSS operates as described above, with delivery schedules selected only from the subset of values of SchedSelIdx for which the bit rate and CPB size are restricted as specified in Annex A for the specified profile and level, or with "interpolated" delivery schedules as specified below for which the bit rate and CPB size are restricted as specified in Annex A. The same delivery schedule is used for both the HRD and DUT.

When the HRD parameters and the buffering period SEI messages are present with cpb\_cnt\_minus1 greater than 0, the decoder shall be capable of decoding the bitstream as delivered from the HSS operating using an "interpolated" delivery schedule specified as having peak bit rate r, CPB size c( r ), and initial CPB removal delay ( f( r )r ) as follows:

 = ( r − BitRate[ SchedSelIdx − 1 ] )  ( BitRate[ SchedSelIdx ] − BitRate[ SchedSelIdx − 1 ] ), (C‑17)

c( r ) =  \* CpbSize[ SchedSelIdx ] + (1 −  \* CpbSize[ SchedSelIdx−1 ], (C‑18)

f( r ) = InitCpbRemovalDelay[ SchedSelIdx ] \* BitRate[ SchedSelIdx ] +   
 ( 1 −  InitCpbRemovalDelay[ SchedSelIdx − 1 ] \* BitRate[ SchedSelIdx − 1 ] (C‑19)

for any SchedSelIdx > 0 and r such that BitRate[ SchedSelIdx − 1 ] <= r <= BitRate[ SchedSelIdx ] such that r and c( r ) are within the limits as specified in Annex A for the maximum bit rate and buffer size for the specified profile and level.

NOTE 1 – InitCpbRemovalDelay[ SchedSelIdx ] can be different from one buffering period to another and have to be re‑calculated.

For output timing decoder conformance, an HRD as described above is used and the timing (relative to the delivery time of the first bit) of picture output is the same for both HRD and the DUT up to a fixed delay.

For output order decoder conformance, the HSS delivers the bitstream to the DUT "by demand" from the DUT, meaning that the HSS delivers bits (in decoding order) only when the DUT requires more bits to proceed with its processing.

NOTE 2 – This means that for this test, the coded picture buffer of the DUT could be as small as the size of the largest access unit.

A modified HRD as described below is used, and the HSS delivers the bitstream to the HRD by one of the schedules specified in the bitstream such that the bit rate and CPB size are restricted as specified in Annex A. The order of pictures output shall be the same for both HRD and the DUT.

For output order decoder conformance, the HRD CPB size is equal to CpbSize[ SchedSelIdx ] for the selected schedule and the DPB size is equal to MaxDpbSize. Removal time from the CPB for the HRD is equal to final bit arrival time and decoding is immediate. The operation of the DPB of this HRD is described below.

* + 1. Operation of the output order DPB

The decoded picture buffer contains picture storage buffers. Each of the picture storage buffers contains a decoded picture that is marked as "used for reference" or is held for future output. At HRD initialization, the DPB is empty. The following steps happen in the order as listed below.

* + 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) happens instantaneously when the first decoding unit of the access unit containing the current picture is removed from the CPB and proceeds as follows.

The decoding process for reference picture set as specified in subclause 8.3.2 is invoked.

– If the current picture is an IDR or a BLA picture, the following applies.

1. When the IDR or BLA picture is not the first IDR or BLA picture decoded and the value of pic\_width\_in\_luma\_samples or pic\_height\_in\_luma\_samples or sps\_max\_dec\_pic\_buffering[ i ] for any possible value of i 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 sps\_max\_dec\_pic\_buffering[ i ] 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 picture or DPB size changes more gracefully than the HRD in regard to changes in pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples or sps\_max\_dec\_pic\_buffering[ i ].

1. When no\_output\_of\_prior\_pics\_flag is equal to 1 or is inferred to be equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain.

– Otherwise (the current picture is not an IDR or a BLA picture), picture storage buffers containing a picture which are marked as "not needed for output" and "unused for reference" are emptied (without output). When one or more of the following conditions are true, the "bumping" process specified in subclause  is invoked repeatedly until there is an empty picture storage buffer to store the current decoded picture.

1. The number of pictures in the DPB that are marked as "needed for output" is greater than sps\_num\_reorder\_pics[ temporal\_id ],
2. The number of pictures in the DPB with temporal\_id lower than or equal to the temporal\_id of the current picture is equal to sps\_max\_dec\_pic\_buffering[ temporal\_id ] + 1.

When the current picture is an IDR or a BLA 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. Picture storage buffers containing a picture that is marked as "not needed for output" and "unused for reference" are emptied (without output).

2. All non-empty picture storage buffers in the DPB are emptied by repeatedly invoking the "bumping" process specified in subclause .

* + - 1. "Bumping" process

The "bumping" process is invoked in the following cases.

– The current picture is an IDR or a BLA 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 .

– The number of pictures in the DPB that are marked "needed for output" is greater than sps\_num\_reorder\_pics[ temporal\_id ], as specified in subclause .

– The number of pictures in the DPB with temporal\_id lower than or equal to the temporal\_id of the current picture is equal to sps\_max\_dec\_pic\_buffering[ temporal\_id ] + 1, as specified in subclause .

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 picture storage buffer that included the picture that was cropped and output contains a picture marked as "unused for reference", the picture storage buffer is emptied.
   * 1. Picture decoding, marking and storage

The following happens instantaneously when the last decoding unit of access unit n containing the current picture is removed from the CPB.

The current picture is considered as decoded after the last decoding unit of the picture is decoded. The current decoded picture is stored in an empty picture storage buffer in the DPB, and the following applies.

– If the current decoded picture has PicOutputFlag equal to 1, it is marked as "needed for output".

– Otherwise (the current decoded picture has PicOutputFlag 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  for the specification of conformance). Some SEI message information is required to check bitstream conformance and for output timing decoder conformance.

In Annex , 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  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

|  |  |
| --- | --- |
| sei\_payload( payloadType, payloadSize ) { | Descriptor |
| if( payloadType = = 0 ) |  |
| buffering\_period( payloadSize ) |  |
| else if( payloadType = = 1 ) |  |
| pic\_timing( payloadSize ) |  |
| else if( payloadType = = 2 ) |  |
| pan\_scan\_rect( payloadSize ) |  |
| else if( payloadType = = 3 ) |  |
| filler\_payload( payloadSize ) |  |
| else if( payloadType = = 4 ) |  |
| user\_data\_registered\_itu\_t\_t35( payloadSize ) |  |
| else if( payloadType = = 5 ) |  |
| user\_data\_unregistered( payloadSize ) |  |
| else if( payloadType = = 6 ) |  |
| recovery\_point( payloadSize ) |  |
| else if( payloadType = = 9 ) |  |
| scene\_info( payloadSize ) |  |
| else if( payloadType = = 15 ) |  |
| full\_frame\_snapshot( payloadSize ) |  |
| else if( payloadType = = 16 ) |  |
| progressive\_refinement\_segment\_start( payloadSize ) |  |
| else if( payloadType = = 17 ) |  |
| progressive\_refinement\_segment\_end( payloadSize ) |  |
| else if( payloadType = = 19 ) |  |
| film\_grain\_characteristics( payloadSize ) |  |
| else if( payloadType = = 20 ) |  |
| deblocking\_filter\_display\_preference( payloadSize ) |  |
| else if( payloadType = = 22 ) |  |
| post\_filter\_hint( payloadSize ) |  |
| else if( payloadType = = 23 ) |  |
| tone\_mapping\_info( payloadSize ) |  |
| else if( payloadType = = 45 ) |  |
| frame\_packing\_arrangement( payloadSize ) |  |
| else if( payloadType = = 47 ) |  |
| display\_orientation( payloadSize ) |  |
| else if( payloadType = = 128 ) |  |
| sop\_description( payloadSize ) |  |
| else if( payloadType = = 129 ) |  |
| field\_indication( payloadSize ) |  |
| else if( payloadType = = 130) |  |
| decoded\_picture\_hash( payloadSize ) |  |
| else |  |
| reserved\_sei\_message( payloadSize ) |  |
| if( !byte\_aligned( ) ) { |  |
| **bit\_equal\_to\_one** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **bit\_equal\_to\_zero** /\* equal to 0 \*/ | f(1) |
| } |  |
| } |  |

* + 1. Buffering period SEI message syntax

|  |  |
| --- | --- |
| buffering\_period( payloadSize ) { | **Descriptor** |
| **seq\_parameter\_set\_id** | ue(v) |
| if( !sub\_pic\_cpb\_params\_present\_flag ) |  |
| **rap\_cpb\_params\_present\_flag** | u(1) |
| if( NalHrdBpPresentFlag ) { |  |
| for( SchedSelIdx = 0; SchedSelIdx <= cpb\_cnt\_minus1; SchedSelIdx++ ) { |  |
| **initial\_cpb\_removal\_delay[** SchedSelIdx **]** | u(v) |
| **initial\_cpb\_removal\_delay\_offset[** SchedSelIdx **]** | u(v) |
| if( sub\_pic\_cpb\_params\_present\_flag | |  rap\_cpb\_params\_present\_flag ) { |  |
| **initial\_alt\_cpb\_removal\_delay[** SchedSelIdx **]** | u(v) |
| **initial\_alt\_cpb\_removal\_delay\_offset[** SchedSelIdx **]** | u(v) |
| } |  |
| } |  |
| } |  |
| if( VclHrdBpPresentFlag ) { |  |
| for( SchedSelIdx = 0; SchedSelIdx <= cpb\_cnt\_minus1; SchedSelIdx++ ) { |  |
| **initial\_cpb\_removal\_delay[** SchedSelIdx **]** | u(v) |
| **initial\_cpb\_removal\_delay\_offset[** SchedSelIdx **]** | u(v) |
| if( sub\_pic\_cpb\_params\_present\_flag | |  rap\_cpb\_params\_present\_flag) { |  |
| **initial\_alt\_cpb\_removal\_delay[** SchedSelIdx **]** | u(v) |
| **initial\_alt\_cpb\_removal\_delay\_offset[** SchedSelIdx **]** | u(v) |
| } |  |
| } |  |
| } |  |
| } |  |

* + 1. Picture timing SEI message syntax

|  |  |
| --- | --- |
| pic\_timing( payloadSize ) { | Descriptor |
| if( CpbDpbDelaysPresentFlag ) { |  |
| **cpb\_removal\_delay** | u(v) |
| **dpb\_output\_delay** | u(v) |
| if( sub\_pic\_cpb\_params\_present\_flag ) { |  |
| **num\_decoding\_units\_minus1** | ue(v) |
| for( i = 0; i <= num\_decoding\_units\_minus1; i++ ) { |  |
| **num\_nalus\_in\_du\_minus1[**i**]** | ue(v) |
| **du\_cpb\_removal\_delay[**i**]** | u(v) |
| } |  |
| } |  |
| } |  |
| } |  |

* + 1. Pan-scan rectangle SEI message syntax

The syntax table is specifed in subclause D.1.3 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Filler payload SEI message syntax

The syntax table is specifed in subclause D.1.4 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. User data registered by Rec. ITU-T T.35 SEI message syntax

The syntax table is specifed in subclause D.1.5 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. User data unregistered SEI message syntax

The syntax table is specifed in subclause D.1.6 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Recovery point SEI message syntax

|  |  |
| --- | --- |
| recovery\_point( payloadSize ) { | Descriptor |
| **recovery\_poc\_cnt** | ue(v) |
| **exact\_match\_flag** | u(1) |
| **broken\_link\_flag** | u(1) |
| } |  |

[Ed. (YK): The broken\_link\_flag is not useful anymore after the introduction of BLA pictures, and therefore should be removed from the recovery point SEI message.]

* + 1. Scene information SEI message syntax

The syntax table is specifed in subclause D.1.10 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Full-frame snapshot SEI message syntax

The syntax table is specifed in subclause D.1.16 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Progressive refinement segment start SEI message syntax

The syntax table is specifed in subclause D.1.17 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Progressive refinement segment end SEI message syntax

The syntax table is specifed in subclause D.1.18 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Film grain characteristics SEI message syntax

The syntax table is specifed in subclause D.1.20 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Deblocking filter display preference SEI message syntax

The syntax table is specifed in subclause D.1.21 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Post-filter hint SEI message syntax

The syntax table is specifed in subclause D.1.23 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Tone mapping information SEI message syntax

The syntax table is specifed in subclause D.1.24 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Frame packing arrangement SEI message syntax

The syntax table is specifed in subclause D.1.25 of Rec. ITU-T H.264 | ISO/IEC 14496-10.

* + 1. Display orientation SEI message syntax

|  |  |
| --- | --- |
| display\_orientation( payloadSize ) { | **Descriptor** |
| **display\_orientation\_cancel\_flag** | u(1) |
| if( !display\_orientation\_cancel\_flag ) { |  |
| **hor\_flip** | u(1) |
| **ver\_flip** | u(1) |
| **anticlockwise\_rotation** | u(16) |
| **display\_orientation\_repetition\_period** | ue(v) |
| **display\_orientation\_extension\_flag** | u(1) |
| } |  |
| } |  |

* + 1. SOP description SEI message syntax

|  |  |
| --- | --- |
| sop\_description( payloadSize ) { | Descriptor |
| **sps\_id** | ue(v) |
| **num\_pics\_in\_sop\_minus1** | ue(v) |
| for( i = 0; i <= num\_pics\_in\_sop\_minus1; i++ ) { |  |
| **sop\_desc\_nal\_ref\_flag**[ i ] | u(1) |
| **sop\_desc\_temporal\_id**[ i ] | u(3) |
| **st\_rps\_idx**[ i ] | ue(v) |
| if( i > 0 ) |  |
| **poc\_delta**[ i ] | se(v) |
| } |  |
| } |  |

* + 1. Field indication SEI message syntax

|  |  |
| --- | --- |
| field\_indication( payloadSize ) { | Descriptor |
| **field\_pic\_flag** | u(1) |
| **progressive\_source\_flag** | u(1) |
| **duplicate\_flag** | u(1) |
| if( field\_pic\_flag ) |  |
| **bottom\_field\_flag** | u(1) |
| else if( !progressive\_source\_flag ) |  |
| **top\_field\_first\_flag** | u(1) |
| else |  |
| **reserved\_zero\_1bit** /\* equal to 0 \***/** | u(1) |
| **reserved\_zero\_4bits** /\* equal to 0 \***/** | u(4) |
| } |  |

* + 1. Decoded picture hash SEI message syntax

|  |  |
| --- | --- |
| decoded\_picture\_hash( payloadSize ) { | Descriptor |
| **hash\_type** | u(8) |
| for( cIdx = 0; cIdx < (chroma\_format\_idc = = 0) ? 1 : 3; cIdx++ ) |  |
| if( hash\_type = = 0 ) |  |
| for( i = 0; i < 16; i++) |  |
| **picture\_md5**[ cIdx ][ i ] | b(8) |
| else if( hash\_type = = 1 ) |  |
| **picture\_crc**[ cIdx ] | u(16) |
| else if( hash\_type = = 2 ) |  |
| **picture\_checksum**[ cIdx ] | u(32) |
| } |  |

* + 1. Reserved SEI message syntax

|  |  |
| --- | --- |
| reserved\_sei\_message( payloadSize ) { | Descriptor |
| for( i = 0; i < payloadSize; i++ ) |  |
| **reserved\_sei\_message\_payload\_byte** | b(8) |
| } |  |

* 1. SEI payload semantics
     1. Buffering period SEI message semantics

When NalHrdBpPresentFlag or VclHrdBpPresentFlag are equal to 1, a buffering period SEI message can be associated with any access unit in the bitstream, and a buffering period SEI message shall be associated with each RAP access unit, and with each access unit associated with a recovery point SEI message.

NOTE – For some applications, the frequent presence of a buffering period SEI message may be desirable.

A buffering period is specified as the set of access units between two instances of the buffering period SEI message in decoding order.

**seq\_parameter\_set\_id** specifies the sequence parameter set that contains the sequence HRD attributes. The value of seq\_parameter\_set\_id shall be equal to the value of seq\_parameter\_set\_id in the picture parameter set referenced by the primary coded picture associated with the buffering period SEI message. The value of seq\_parameter\_set\_id shall be in the range of 0 to 31, inclusive.

**rap\_cpb\_params\_present\_flag** equal to 1 specifies the presence of the initial\_alt\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_alt\_cpb\_removal\_delay\_offset[ SchedSelIdx ] syntax elements. When not present, the value of rap\_cpb\_params\_present\_flag is inferred to be equal to 0. When the associated picture is neither a CRA picture nor a BLA picture, the value of rap\_cpb\_params\_present\_flag shall be equal to 0.

**initial\_cpb\_removal\_delay**[ SchedSelIdx ] and **initial\_alt\_cpb\_removal\_delay**[ SchedSelIdx ] specify the initial CPB removal delays for the SchedSelIdx-th CPB. The syntax elements have a length in bits given by initial\_cpb\_removal\_delay\_length\_minus1 + 1, and are in units of a 90 kHz clock. The values of the syntax elements shall not be equal to 0 and shall not exceed 90000 \* ( CpbSize[ SchedSelIdx ] ÷ BitRate[ SchedSelIdx ] ), the time-equivalent of the CPB size in 90 kHz clock units.

**initial\_cpb\_removal\_delay\_offset**[ SchedSelIdx ] and **initial\_alt\_cpb\_removal\_delay\_offset**[ SchedSelIdx ] are used for the SchedSelIdx-th CPB to specify the initial delivery time of coded data units to the CPB. The syntax elements have a length in bits given by initial\_cpb\_removal\_delay\_length\_minus1 + 1 and are in units of a 90 kHz clock. These syntax elements are not used by decoders and may be needed only for the delivery scheduler (HSS) specified in Annex C.

Over the entire coded video sequence, the sum of initial\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_cpb\_removal\_delay\_offset[ SchedSelIdx ] shall be constant for each value of SchedSelIdx, and the sum of initial\_alt\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_alt\_cpb\_removal\_delay\_offset[ SchedSelIdx ] shall be constant for each value of SchedSelIdx.

* + 1. Picture timing SEI message semantics

NOTE 1 – The syntax of the picture timing SEI message is dependent on the content of the sequence parameter set that is active for the coded picture associated with the picture timing SEI message. However, unless the picture timing SEI message of an IDR or BLA access unit is preceded by a buffering period SEI message within the same access unit, the activation of the associated sequence parameter set (and, for IDR or BLA pictures that are not the first picture in the bitstream, the determination that the coded picture is an IDR picture or a BLA picture) does not occur until the decoding of the first coded slice NAL unit of the coded picture. Since the coded slice NAL unit of the coded picture follows the picture timing SEI message in NAL unit order, there may be cases in which it is necessary for a decoder to store the RBSP containing the picture timing SEI message until determining the parameters of the sequence parameter that will be active for the coded picture, and then perform the parsing of the picture timing SEI message.

The presence of picture timing SEI message in the bitstream is specified as follows.

– If CpbDpbDelaysPresentFlag is equal to 1, one picture timing SEI message shall be present in every access unit of the coded video sequence.

– Otherwise (CpbDpbDelaysPresentFlag is equal to 0), no picture timing SEI messages shall be present in any access unit of the coded video sequence.

**cpb\_removal\_delay** specifies how many clock ticks to wait after removal from the CPB of the access unit associated with the most recent buffering period SEI message in a preceding access unit before removing from the buffer the access unit data associated with the picture timing SEI message. This value is also used to calculate an earliest possible time of arrival of access unit data into the CPB for the HSS. The syntax element is a fixed length code whose length in bits is given by cpb\_removal\_delay\_length\_minus1 + 1. The cpb\_removal\_delay is the remainder of a modulo 2(cpb\_removal\_delay\_length\_minus1 + 1) counter.

NOTE 2 – The value of cpb\_removal\_delay\_length\_minus1 that determines the length (in bits) of the syntax element cpb\_removal\_delay is the value of cpb\_removal\_delay\_length\_minus1 coded in the sequence parameter set that is active for the primary coded picture associated with the picture timing SEI message, although cpb\_removal\_delay specifies a number of clock ticks relative to the removal time of the preceding access unit containing a buffering period SEI message, which may be an access unit of a different coded video sequence.

**dpb\_output\_delay** is used to compute the DPB output time of the picture. It specifies how many clock ticks to wait after removal of the last decoding unit in an access unit from the CPB before the decoded picture is output from the DPB.

NOTE 3 – A picture is not removed from the DPB at its output time when it is still marked as "used for short-term reference" or "used for long-term reference".

NOTE 4 – Only one dpb\_output\_delay is specified for a decoded picture.

The length of the syntax element dpb\_output\_delay is given in bits by dpb\_output\_delay\_length\_minus1 + 1. When sps\_max\_dec\_pic\_buffering[ max\_temporal\_layers\_minus1 ] is equal to 0, dpb\_output\_delay shall be equal to 0.

The output time derived from the dpb\_output\_delay of any picture that is output from an output timing conforming decoder shall precede the output time derived from the dpb\_output\_delay of all pictures in any subsequent coded video sequence in decoding order.

The picture output order established by the values of this syntax element shall be the same order as established by the values of PicOrderCntVal.

For pictures that are not output by the "bumping" process because they precede, in decoding order, an IDR or BLA picture with no\_output\_of\_prior\_pics\_flag equal to 1 or inferred to be equal to 1, the output times derived from dpb\_output\_delay shall be increasing with increasing value of PicOrderCntVal relative to all pictures within the same coded video sequence.

**num\_decoding\_units\_minus1** plus 1 specifies the number of decoding units in the access unit the picture timing SEI message is associated with. The value of num\_decoding\_units\_minus1 shall be in the range of 0 to PicWidthInCtbs \* PicHeightInCtbs – 1, inclusive.

**num\_nalus\_in\_du\_minus1[**i**]** plus 1 specifies the number of NAL units in the i-th decoding unit of the access unit the picture timing SEI message is associated with. The value of num\_nalus\_in\_du\_minus1[ i ] shall be in the range of 0 to PicWidthInCtbs \* PicHeightInCtbs – 1, inclusive.

The first decoding unit of the access unit consists of the first num\_nalus\_in\_du\_minus1[ 0 ] + 1 consecutive NAL units in decoding order in the access unit. The i-th (with i greater than 0) decoding unit of the access unit consists of the num\_nalus\_in\_du\_minus1[ i ] + 1 consecutive NAL units immediately following the last NAL unit in the previous decoding unit of the access unit, in decoding order. There shall be at least one VCL NAL unit in each decoding unit. All non-VCL NAL units associated with a VCL NAL unit shall be included in the same decoding unit.

**du\_cpb\_removal\_delay[**i**]** specifies how many sub-picture clock ticks to wait after removal from the CPB of the first decoding unit in the access unit associated with the most recent buffering period SEI message in a preceding access unit before removing from the CPB the i-th decoding unit in the access unit associated with the picture timing SEI message. This value is also used to calculate an earliest possible time of arrival of decoding unit data into the CPB for the HSS, as specified in Annex C. The syntax element is a fixed length code whose length in bits is given by cpb\_removal\_delay\_length\_minus1 + 1. The du\_cpb\_removal\_delay[ i ] is the remainder of a modulo 2(cpb\_removal\_delay\_length\_minus1 + 1) counter.

NOTE 5 – The value of cpb\_removal\_delay\_length\_minus1 that determines the length (in bits) of the syntax element du\_cpb\_removal\_delay[ i ] is the value of cpb\_removal\_delay\_length\_minus1 coded in the sequence parameter set that is active for the coded picture associated with the picture timing SEI message, although du\_cpb\_removal\_delay[ i ] specifies a number of sub-picture clock ticks relative to the removal time of the first decoding unit in the preceding access unit containing a buffering period SEI message, which may be an access unit of a different coded video sequence.

* + 1. Pan-scan rectangle SEI message semantics

The semantics specifed in subclause D.2.3 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Filler payload SEI message semantics

The semantics specifed in subclause D.2.4 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. User data registered by ITU-T Rec. T.35 SEI message semantics

The semantics specifed in subclause D.2.5 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. User data unregistered SEI message semantics

The semantics specifed in subclause D.2.6 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Recovery point SEI message semantics

The recovery point SEI message assists a decoder in determining when the decoding process will produce acceptable pictures for display after the decoder initiates random access or after the encoder indicates a broken link in the coded video sequence. When the decoding process is started with the access unit in decoding order associated with the recovery point SEI message, all decoded pictures at or subsequent to the recovery point in output order specified in this SEI message are indicated to be correct or approximately correct in content. Decoded pictures produced by random access at or before the picture associated with the recovery point SEI message need not be correct in content until the indicated recovery point, and the operation of the decoding process starting at the picture associated with the recovery point SEI message may contain references to pictures unavailable in the decoded picture buffer.

In addition, by use of the broken\_link\_flag, the recovery point SEI message can indicate to the decoder the location of some pictures in the bitstream that can result in serious visual artefacts when displayed, even when the decoding process was begun at the location of a previous IDR access unit in decoding order.

NOTE 1 – The broken\_link\_flag can be used by encoders to indicate the location of a point after which the decoding process for the decoding of some pictures may cause references to pictures that, though available for use in the decoding process, are not the pictures that were used for reference when the bitstream was originally encoded (e.g., due to a splicing operation performed during the generation of the bitstream).

The recovery point is specified as a count in units of PicOrderCntVal increments subsequent to PicOrderCntVal of the current access unit at the position of the SEI message.

When random access is performed to start decoding from the access unit associated with the recovery point SEI message, picture, the associated picture is considered as the first picture in the bitstream, and the variables prevPicOrderCntLsb and prevPicOrderCntMsb used in derivation of PicOrderCntVal are both set to be equal to 0.

NOTE 2 – When HRD information is present in the bitstream, a buffering period SEI message should be associated with the access unit associated with the recovery point SEI message in order to establish initialisation of the HRD buffer model after a random access.

Any sequence, picture or adaptation parameter set RBSP that is referred to by a picture associated with a recovery point SEI message or by any picture following such a picture in decoding order shall be available to the decoding process prior to its activation, regardless of whether or not the decoding process is started at the beginning of the bitstream or with the access unit, in decoding order, that is associated with the recovery point SEI message.

**recovery\_frame\_cnt** specifies the recovery point of output pictures in output order. All decoded pictures in output order are indicated to be correct or approximately correct in content starting at the output order position of the reference picture having the frame\_num equal to the frame\_num of the VCL NAL units for the current access unit incremented by recovery\_frame\_cnt in modulo MaxFrameNum arithmetic. recovery\_frame\_cnt shall be in the range of 0 to MaxFrameNum − 1, inclusive.

**recovery\_poc\_cnt** specifies the recovery point of output pictures in output order. All decoded pictures in output order are indicated to be correct or approximately correct in content starting at the output order position of the first picture after the current picture in decoding order that has PicOrderCntVal equal to the PicOrderCntVal of the current picture incremented by recovery\_poc\_cnt. The value of recovery\_poc\_cnt shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive.

**exact\_match\_flag** indicates whether decoded pictures at and subsequent to the specified recovery point in output order derived by starting the decoding process at the access unit associated with the recovery point SEI message shall be an exact match to the pictures that would be produced by starting the decoding process at the location of a previous RAP access unit, if any, in the bitstream. The value 0 indicates that the match does not need to be exact and the value 1 indicates that the match shall be exact.

NOTE 3 – When performing random access, decoders should infer all references to unavailable reference pictures as references to pictures containing only intra coding blocks and having sample values given by Y equal to ( 1 << ( BitDepthY − 1 ) ), Cb and Cr both equal to ( 1 << ( BitDepthC − 1 ) ) (mid-level grey), regardless of the value of exact\_match\_flag.

When exact\_match\_flag is equal to 0, the quality of the approximation at the recovery point is chosen by the encoding process and is not specified by this Recommendation | International Standard.

**broken\_link\_flag** indicates the presence or absence of a broken link in the NAL unit stream at the location of the recovery point SEI message and is assigned further semantics as follows:

– If broken\_link\_flag is equal to 1, pictures produced by starting the decoding process at the location of a previous RAP access unit may contain undesirable visual artefacts to the extent that decoded pictures at and subsequent to the access unit associated with the recovery point SEI message in decoding order should not be displayed until the specified recovery point in output order.

– Otherwise (broken\_link\_flag is equal to 0), no indication is given regarding any potential presence of visual artefacts.

[Ed. (YK): The broken\_link\_flag is not useful anymore after the introduction of BLA pictures, and therefore should be removed from the recovery point SEI message.]

Regardless of the value of the broken\_link\_flag, pictures subsequent to the specified recovery point in output order are specified to be correct or approximately correct in content.

* + 1. Scene information SEI message semantics

The semantics specifed in subclause D.2.10 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Full-frame snapshot SEI message semantics

The semantics specifed in subclause D.2.16 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Progressive refinement segment start SEI message semantics

The semantics specifed in subclause D.2.17 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Progressive refinement segment end SEI message semantics

The semantics specifed in subclause D.2.18 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Film grain characteristics SEI message semantics

The semantics specifed in subclause D.2.20 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Deblocking filter display preference SEI message semantics

The semantics specifed in subclause D.2.21 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Post-filter hint SEI message semantics

The semantics specifed in subclause D.2.23 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Tone mapping information SEI message semantics

The semantics specifed in subclause D.2.24 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Frame packing arrangement SEI message semantics

The semantics specifed in subclause D.2.25 of ITU-T Rec. H.264 | ISO/IEC 14496-10 apply.

* + 1. Display orientation SEI message semantics

This SEI message informs the decoder of a transformation that is recommended to be applied to the output decoded and cropped picture prior to display.

**display\_orientation\_cancel\_flag** equal to 1 indicates that the SEI message cancels the persistence of any previous display orientation SEI message in output order. display\_orientation\_cancel\_flag equal to 0 indicates that display orientation information follows.

**hor\_flip** equal to 1 indicates that the cropped decoded picture should be flipped horizontally for display. hor\_flip equal to 0 indicates that the decoded picture should not be flipped horizontally.

When hor\_flip is equal to 1, the cropped decoded picture should be flipped as follows for each colour component Z = L, Cb, and Cr, letting dZ be the final cropped array of output samples for the component Z:

for( x = 0; x < croppedWidthInSamplesZ; x++ ) [Ed. (GJS): More properly account for cropping.]  
 for( y = 0; y < croppedHeightInSamplesZ; y++ )  
 dZ[ x, y ] = Z[ croppedWidthInSamplesZ − x − 1, y ]

**ver\_flip** equal to 1 indicates that the cropped decoded picture should be flipped vertically (in addition to any horizontal flipping when hor\_flip is equal to 1) for display. ver\_flip equal to 0 indicates that the decoded picture should not be flipped vertically.

When ver\_flip is equal to 1, the cropped decoded picture should be flipped as follows for each colour component Z = L, Cb, and Cr, letting dZ be the final cropped array of output samples for the component Z:

for( x = 0; x < croppedWidthInSamplesZ; x++ )  
 for( y = 0; y < croppedHeightInSamplesZ; y++ )  
 dZ[ x, y ] = Z[ x, croppedWidthInSamplesZ − y − 1 ]

**anticlockwise\_rotation** specifies the recommended anticlockwise rotation of the decoded picture (after applying horizontal and/or vertical flipping when hor\_flip or ver\_flip is set) prior to display. The decoded picture should be rotated by 360 \* anticlockwise\_rotation ÷ 216 degrees (2 \* π \* anticlockwise\_rotation ÷ 216 radians, where π is Archimedes' Constant (3.141 592 653 589 793 …) in the anticlockwise direction prior to display. For example, anticlockwise\_rotation equal to 0 indicates no rotation and anticlockwise\_rotation equal to 16 384 indicates 90 degrees (π ÷ 2 radians) rotation in the anticlockwise direction.

NOTE – It is possible for equivalent transformations to be expressed in multiple ways using these syntax elements. For example, the combination of having both hor\_flip and ver\_flip equal to 1 with anticlockwise\_rotation equal to 0 can alternatively be expressed by having both hor\_flip and ver\_flip equal to 1 with anticlockwise\_rotation equal to 0x8000000, and the combination of hor\_flip equal to 1 with ver\_flip equal to 0 and anticlockwise\_rotation equal to 0 can alternatively be expressed by having hor\_flip equal to 0 with ver\_flip equal to 1 and anticlockwise\_rotation equal to 0x8000000.

**display\_orientation\_repetition\_period** specifies the persistence of the display orientation SEI message and may specify a picture order count interval within which another display orientation SEI message or the end of the coded video sequence shall be present in the bitstream. The value of display\_orientation\_repetition\_period shall be in the range 0 to 16 384, inclusive.

**display\_orientation\_repetition\_period** equal to 0 specifies that the display orientation SEI message applies to the current decoded picture only.

**display\_orientation\_repetition\_period** equal to 1 specifies that the display orientation SEI message persists in output order until one or more of the following conditions are true:

– A new coded video sequence begins.

– A picture in an access unit containing a display orientation SEI message is output having PicOrderCnt( ) greater than PicOrderCnt( CurrPic ).

display\_orientation\_repetition\_period greater than 1 specifies that the display orientation SEI message persists until one or more of the following conditions are true:

– A new coded video sequence begins.

– A picture in an access unit containing a display orientation SEI message is output having PicOrderCnt( ) greater than PicOrderCnt( CurrPic ) and less than or equal to PicOrderCnt( CurrPic ) + display\_orientation\_repetition\_period.

**display\_orientation\_repetition\_period** greater than 1 indicates that another display orientation SEI message shall be present for a picture in an access unit that is output having PicOrderCnt( ) greater than PicOrderCnt( CurrPic ) and less than or equal to PicOrderCnt( CurrPic ) + display\_orientation\_repetition\_period; unless the bitstream ends or a new coded video sequence begins without output of such a picture. [Ed. (GJS): Check POC variable use.]

**display\_orientation\_extension\_flag** equal to 0 indicates that no additional data follows within the post-filter hint SEI message. The value of display\_orientation\_extension\_flag shall be equal to 0. The value of 1 for display\_orientation\_extension\_flag is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all data that follows the value of 1 for display\_orientation\_extension\_flag in a display orientation SEI message.

* + 1. SOP description SEI message semantics

The SOP description SEI message indicates constraints that apply in the SOP that starts with the current access unit. [Ed.(GJS): Define SOP.]

The SOP description SEI message shall not be present in any access unit with temporal\_id greater than 0.

**sps\_id** specifies the seq\_parameter\_set\_id value identifying a sequence parameter set.

**num\_pics\_in\_sop\_minus1** + 1 specifies the number of pictures in the SOP.

**sop\_desc\_nal\_ref\_flag[** i ] specifies the nal\_ref\_flag value of the i-th picture in decoding order within the SOP.

**sop\_desc\_temporal\_id[** i ] specifies the temporal\_id value of the i-th picture in decoding order within the SOP.

**st\_rps\_idx**[ i ] specifies the short-term reference picture set included in the sequence parameter set identified by sps\_id and used by the i-th picture in decoding order within the SOP.

**poc\_delta**[ i ] specifies the value of DiffPicOrderCnt( the i-th picture in decoding order within the SOP, the (i-1)th picture in decoding order within the SOP ). [Ed. (GJS): Improve phrasing – e.g., use variables.]

The bitstream may not contain all the pictures described in a SOP description. For example, the bitstream may have been subject to temporal\_id based sub-bitstream extraction, but pictures that have temporal\_id values no longer existing in the extracted bitstream may still be present in the SOP description SEI message. The following constraints specify that the pictures that are present in the bitstream must match to the information given in the SOP description SEI message. [Ed. (GJS): This paragraph seems normative. Improve phrasing. Does it contradict other statements within the annex?]

The variable picOrderCntExp[ i ] is derived as follows.

picOrderCntExp[ 0 ] = PicOrderCntVal of the current picture  
for( i = 1; i <= num\_pics\_in\_sop\_minus1; i++ )  
 picOrderCntExp[ i ] = picOrderCntExp[ i − 1 ] + poc\_delta[ i ]

Let nalRefFlag[ j ], tId[ j ], stRpsIdx[ j ], and picOrderCntVal[ j ] be the values of nal\_ref\_flag, temporal\_id, st\_rps\_idx, and PicOrderCntVal that are in effect for the j-th picture in decoding order starting from j equal to 0 indicating the first picture of the SOP where this SEI message resides. Let currSeqParamSet be the previous seq\_parameter\_set\_rbsp with seq\_parameter\_set\_id equal to sps\_id in decoding order. long\_term\_ref\_pics\_present\_flag shall be equal to 0 in currSeqParamSet. [Ed. (GJS): This seems to introduce some confusion with respect to uses of the same variable names in other places in the text. I suggest using subscripts rather than array symbols here.]

It is a requirement of bitstream conformance that when the SOP description SEI message, the following constraints shall apply for each picture i from picture 0 to picture num\_pics\_in\_sop\_minus1 when picOrderCntExp[ i ] is equal to picOrderCntVal[ j ], where j is greater than 0 and picOrderCntVal[ j − 1 ] is less than or equal to picOrderCntExp[ num\_pics\_in\_sop\_minus1 ]:

– nalRefFlag[ j ] shall be equal to sop\_desc\_nal\_ref\_flag[ i ]

– tId[ j ] shall be equal to sop\_desc\_temporal\_id[ i ]

– stRpsIdx[ j ] shall be equal to st\_rps\_idx[ i ]

– currSeqParamSet shall be the active sequence parameter set RBSP for picture j

* + 1. Field indication SEI message semantics

The field indication SEI message applies to the current access unit only.

The presence of the field indication SEI message in the bitstream is specified as follows.

– If field\_seq\_flag is equal to 1, one field indication SEI message shall be present in every access unit of the current coded video sequence.

– Otherwise (field\_seq\_flag is equal to 0), one field indication SEI message may be present in any access unit of the current coded video sequence.

NOTE 1 – In the absence of a field indication SEI message or other indication to the contrary, the current picture should be inferred to be a progressive frame coded as a frame picture.

The nominal vertical and horizontal sampling locations of samples in top and bottom fields for 4:2:0, 4:2:2, and 4:4:4 chroma formats are shown in , , and , respectively.

**field\_pic\_flag** equal to 0 indicates that the associated picture represents a video frame. field\_pic\_flag equal to 1 indicates that the associated picture represents a video field. The value of field\_pic\_flag shall be equal to field\_seq\_flag. When the field indication SEI message is not present, the value of field\_pic\_flag should be inferred to be equal to 0.

NOTE 2 – The specified decoding process does not treat access units conveying pictures that represent fields or frames differently. A sequence of pictures that represent fields would therefore be coded with the picture dimensions of an individual field. For example, access units containing pictures that represent 1080i fields would commonly have cropped output dimensions of 1920x540, while the sequence picture rate would commonly express the rate of the source fields (typically between 50 and 60 Hz), instead of the source frame rate (typically between 25 and 30 Hz).

**progressive\_source\_flag** equal to 1 indicates that the scan type of the associated picture should be interpreted as progressive. progressive\_source\_flag equal to 0 indicates that the scan type of the associated picture should be interpreted as interlaced. When the field indication SEI message is not present, the value of progressive\_source\_flag should be inferred to be equal to 1.

The interpretation of combinations of field\_pic\_flag and progressive\_source\_flag values is defined in .

Table D‑1 – Indicated interpretation of field\_pic\_flag and progressive\_source\_flag

|  |  |  |
| --- | --- | --- |
| Interpretation | field\_pic\_flag | progressive\_source\_flag |
| Picture is progressive frame | 0 | 1 |
| Picture is interleaved fields coded in frame picture | 0 | 0 |
| Picture is field | 1 | 0 |
| Picture is field extracted from progressive frame | 1 | 1 |

**duplicate\_flag** equal to 1 indicates that the current picture is indicated to be a duplicate of a previous picture in output order. duplicate\_flag equal to 0 indicates that the current picture is not indicated to be a duplicate picture.

NOTE 3 – This flag should be used to mark coded pictures known to have originated from a repetition process such as 3:2 pull-down or other duplication and interpolation methods. This flag would commonly be used when a video feed is encoded in a "transport pass-through" fashion, with known duplicate pictures tagged by setting duplicate\_flag equal to 1.

NOTE 4 – When field\_pic\_flag is equal to 1 and duplicate\_flag is equal to 1, it is assumed the access unit contains a duplicated field of the previous field in output order with the same parity as the current field.

**bottom\_field\_flag** indicates the parity of the field contained within the access unit when field\_pic\_flag is equal to 1. bottom\_field\_flag equal to 1 indicates bottom field parity. bottom\_field\_flag equal to 0 indicates top field parity.

**top\_field\_first\_flag** indicates the preferred field output order for display purposes when fields have been interleaved to form frames in a sequence of coded frames. If top\_field\_first\_flag is equal to 1, the top field is indicated to be temporally first, followed by the bottom field. Otherwise (top\_field\_first\_flag is equal to 0), the bottom field is indicated to be temporally first, followed by the top field.

NOTE 5 – The display process, when applicable, is outside the scope of this Specification.

**reserved\_zero\_1bit** shall be equal to 0. The value 1 for reserved\_zero\_1bit is reserved for future backward-compatible use by ITU-T | ISO/IEC. Decoders shall ignore the value of reserved\_zero\_1bit.

**reserved\_zero\_4bits** shall be equal to 0. Other values for reserved\_zero\_4bits are reserved for future backward-compatible use by ITU-T | ISO/IEC. Decoders shall ignore the value of reserved\_zero\_4bits.



Figure D‑1 – Nominal vertical and horizontal sampling locations of 4:2:0 samples in top and bottom fields



Figure D‑2 – Nominal vertical and horizontal sampling locations of 4:2:2 samples in top and bottom fields



Figure D‑3 – Nominal vertical and horizontal sampling locations of 4:4:4 samples in top and bottom fields

* + 1. Decoded picture hash SEI message semantics

This message provides a hash for each colour component of the decoded picture in the current access unit.

Prior to computing the hash, the decoded picture data is arranged into one or three strings of bytes called pictureData[ cIdx ] of lengths pictureDataLen[ cIdx ] according to the following pseudocode process:

for( cIdx = 0; cIdx < (chroma\_format\_idc = = 0) ? 1 : 3; cIdx++ ) {  
 if (cIdx = = 0) {  
 compWidth = pic\_width\_in\_luma\_samples  
 compHeight = pic\_height\_in\_luma\_samples  
 bitDepth = BitDepthY  
 }  
 else {  
 compWidth = pic\_width\_in\_luma\_samples / SubWidthC  
 componentWidth = pic\_height\_in\_luma\_samples / SubWidthC  
 bitDepth = BitDepthC  
 }  
 dataLen = 0  
 for( i = 0; i < compWidth \* compHeight; i++ ) {  
 pictureData[ cIdx ][ dataLen++ ] = component[ cIdx ][ i ] & 0xFF  
 if( bitDepth > 8 )   
 pictureData[ cIdx ][ dataLen++ ] = component[ cIdx ][ i ] >> 8  
 }  
 pictureDataLen[ cIdx ] = dataLen  
 }

where component[ cIdx ][ i ] is an array in raster scan order of decoded sample values in two's complement representation.

[Ed. (YK): The variable SubWidthC is not defined when chroma\_format\_idc is equal to 3 and separate\_colour\_plane\_flag is equal to 1.]

**hash\_type** indicates the method used to calculate the checksum according to .

Table D‑2 – Interpretation of hash\_type

|  |  |
| --- | --- |
| **hash\_type** | **Method** |
| 0 | MD5 (RFC 1321) |
| 1 | CRC |
| 2 | Checksum |
| 3..255 | Reserved |

[Ed. (GJS): Add normative reference to RFC 1321, and clarify informative reference to H.271 to indicate that it is an informative reference.]

**picture\_md5[** cIdx **][** i **]** is the 16-byte MD5 hash of the colour component cIdx of the decoded picture. [Ed. (YK): Is colour component cIdx a precise description?] The value of picture\_md5[ cIdx ][ i ] shall be equal to the value of digestVal[ cIdx ] obtained by performing the following pseudocode process using the MD5 functions defined in RFC 1321:

MD5Init( context )  
 MD5Update( context, pictureData[ cIdx ], pictureDataLen[ cIdx ] )  
 MD5Final( digestVal[ cIdx ], context )

**picture\_crc[** cIdx **]** is the cyclic redundancy check (CRC) of the colour component cIdx of the decoded picture. The value of picture\_crc[ cIdx ] shall be equal to the value of crcVal[ cIdx ] obtained by performing the following pseudocode process:

crc = 0xFFFF  
 pictureData[ cIdx ][ pictureDataLen[ cIdx ] ] = 0  
 pictureData[ cIdx ][ pictureDataLen[ cIdx ] + 1 ] = 0  
 for( bitIdx = 0; bitIdx < ( pictureDataLen[ cIdx ]  + 2 ) \* 8; bitIdx++ ) {  
 dataByte = pictureData[ cIdx ][ bitIdx >> 3 ]  
 crcMsb = ( crc >> 15 ) & 1  
 bitVal = ( dataByte >> (7 – ( bitIdx & 7 ) ) ) & 1  
 crc = ( ( ( crc << 1 ) + bitVal ) & 0xFFFF ) ^ ( crcMsb \* 0x1021 )  
 }  
 crcVal[ cIdx ] = crc

NOTE – The same CRC specification is found in Rec. ITU-T H.271.

**picture\_checksum**[ cIdx ] is the checksum of the colour component cIdx of the decoded picture. The value of picure\_checksum[ cIdx ] shall be equal to the value of checksum[ cIdx ] obtained by performing the following pseudo code process.

sum = 0  
 for( y = 0; y < componentHeight; y++ ) {  
 for( x = 0; x < componentWidth; x++ ) {  
 xorMask = ( x & 0xFF ) ^ ( y & 0xFF ) ^ ( x >> 8 ) ^ ( y >> 8 )  
 sum = ( sum + ( ( component[ cIdx ][ y \* compWidth + x ] & 0xFF ) ^ xorMask ) ) & 0xFFFFFFFF  
 if( BitDepth > 8 )  
 sum = ( sum + ( ( component[ cIdx ][ y \* compWidth + x ] >> 8 ) ^ xorMask ) ) & 0xFFFFFFFF  
 }  
 }  
 checksum[ cIdx ] = sum

* + 1. Reserved SEI message semantics

This SEI message consists of data reserved for future backward-compatible use by ITU-T | ISO/IEC. It is a requirement of bitstream conformance that bitstreams shall not contain reserved SEI messages until and unless the use of such messages has been specified by ITU-T | ISO/IEC. Decoders that encounter reserved SEI messages shall discard their content without effect on the decoding process, except as specified in the future by ITU-T | ISO/IEC.

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 C for the specification of output order conformance). Some VUI parameters are required to check bitstream conformance and for output timing decoder conformance.

In Annex , 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  and 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

|  |  |
| --- | --- |
| vui\_parameters( ) { | Descriptor |
| **aspect\_ratio\_info\_present\_flag** | u(1) |
| if( aspect\_ratio\_info\_present\_flag ) { |  |
| **aspect\_ratio\_idc** | u(8) |
| if( aspect\_ratio\_idc = = Extended\_SAR ) { |  |
| **sar\_width** | u(16) |
| **sar\_height** | u(16) |
| } |  |
| } |  |
| **overscan\_info\_present\_flag** | u(1) |
| if( overscan\_info\_present\_flag ) |  |
| **overscan\_appropriate\_flag** | u(1) |
| **video\_signal\_type\_present\_flag** | u(1) |
| if( video\_signal\_type\_present\_flag ) { |  |
| **video\_format** | u(3) |
| **video\_full\_range\_flag** | u(1) |
| **colour\_description\_present\_flag** | u(1) |
| if( colour\_description\_present\_flag ) { |  |
| **colour\_primaries** | u(8) |
| **transfer\_characteristics** | u(8) |
| **matrix\_coefficients** | u(8) |
| } |  |
| } |  |
| **chroma\_loc\_info\_present\_flag** | u(1) |
| if( chroma\_loc\_info\_present\_flag ) { |  |
| **chroma\_sample\_loc\_type\_top\_field** | ue(v) |
| **chroma\_sample\_loc\_type\_bottom\_field** | ue(v) |
| } |  |
| **neutral\_chroma\_indication\_flag** | u(1) |
| **field\_seq\_flag** | u(1) |
| **timing\_info\_present\_flag** | u(1) |
| if( timing\_info\_present\_flag ) { |  |
| **num\_units\_in\_tick** | u(32) |
| **time\_scale** | u(32) |
| **fixed\_pic\_rate\_flag** | u(1) |
| } |  |
| **nal\_hrd\_parameters\_present\_flag** | u(1) |
| if( nal\_hrd\_parameters\_present\_flag ) |  |
| hrd\_parameters( ) |  |
| **vcl\_hrd\_parameters\_present\_flag** | u(1) |
| if( vcl\_hrd\_parameters\_present\_flag ) |  |
| hrd\_parameters( ) |  |
| if( nal\_hrd\_parameters\_present\_flag | | vcl\_hrd\_parameters\_present\_flag ) |  |
| **low\_delay\_hrd\_flag** | u(1) |
| **sub\_pic\_cpb\_params\_present\_flag** | u(1) |
| if( sub\_pic\_cpb\_params\_present\_flag ) |  |
| **num\_units\_in\_sub\_tick** | u(32) |
| } |  |
| **bitstream\_restriction\_flag** | u(1) |
| if( bitstream\_restriction\_flag ) { |  |
| **tiles\_fixed\_structure\_flag** | u(1) |
| **motion\_vectors\_over\_pic\_boundaries\_flag** | u(1) |
| **max\_bytes\_per\_pic\_denom** | ue(v) |
| **max\_bits\_per\_mincu\_denom** | ue(v) |
| **log2\_max\_mv\_length\_horizontal** | ue(v) |
| **log2\_max\_mv\_length\_vertical** | ue(v) |
| } |  |
| } |  |

* + 1. HRD parameters syntax

|  |  |
| --- | --- |
| hrd\_parameters( ) { | Descriptor |
| **cpb\_cnt\_minus1** | ue(v) |
| **bit\_rate\_scale** | u(4) |
| **cpb\_size\_scale** | u(4) |
| for( SchedSelIdx = 0; SchedSelIdx <= cpb\_cnt\_minus1; SchedSelIdx++ ) { |  |
| **bit\_rate\_value\_minus1[** SchedSelIdx **]** | ue(v) |
| **cpb\_size\_value\_minus1[** SchedSelIdx **]** | ue(v) |
| **cbr\_flag[** SchedSelIdx **]** | u(1) |
| } |  |
| **initial\_cpb\_removal\_delay\_length\_minus1** | u(5) |
| **cpb\_removal\_delay\_length\_minus1** | u(5) |
| **dpb\_output\_delay\_length\_minus1** | u(5) |
| **time\_offset\_length** | u(5) |
| } |  |

* 1. VUI semantics
     1. VUI parameters semantics

**aspect\_ratio\_info\_present\_flag** equal to 1 specifies that aspect\_ratio\_idc is present. aspect\_ratio\_info\_present\_flag equal to 0 specifies that aspect\_ratio\_idc is not present.

**aspect\_ratio\_idc** specifies the value of the sample aspect ratio of the luma samples. shows the meaning of the code. When aspect\_ratio\_idc indicates Extended\_SAR, the sample aspect ratio is represented by sar\_width : sar\_height. When the aspect\_ratio\_idc syntax element is not present, aspect\_ratio\_idc value is inferred to be equal to 0.

Table E‑1 – Meaning of sample aspect ratio indicator

|  |  |  |
| --- | --- | --- |
| **aspect\_ratio\_idc** | **Sample aspect ratio** | **(informative) Examples of use** |
| 0 | Unspecified |  |
| 1 | 1:1 ("square") | 1280x720 16:9 frame without horizontal overscan 1920x1080 16:9 frame without horizontal overscan (cropped from 1920x1088) 640x480 4:3 frame without horizontal overscan |
| 2 | 12:11 | 720x576 4:3 frame with horizontal overscan 352x288 4:3 frame without horizontal overscan |
| 3 | 10:11 | 720x480 4:3 frame with horizontal overscan 352x240 4:3 frame without horizontal overscan |
| 4 | 16:11 | 720x576 16:9 frame with horizontal overscan 528x576 4:3 frame without horizontal overscan |
| 5 | 40:33 | 720x480 16:9 frame with horizontal overscan 528x480 4:3 frame without horizontal overscan |
| 6 | 24:11 | 352x576 4:3 frame without horizontal overscan 480x576 16:9 frame with horizontal overscan |
| 7 | 20:11 | 352x480 4:3 frame without horizontal overscan 480x480 16:9 frame with horizontal overscan |
| 8 | 32:11 | 352x576 16:9 frame without horizontal overscan |
| 9 | 80:33 | 352x480 16:9 frame without horizontal overscan |
| 10 | 18:11 | 480x576 4:3 frame with horizontal overscan |
| 11 | 15:11 | 480x480 4:3 frame with horizontal overscan |
| 12 | 64:33 | 528x576 16:9 frame without horizontal overscan |
| 13 | 160:99 | 528x480 16:9 frame without horizontal overscan |
| 14 | 4:3 | 1440x1080 16:9 frame without horizontal overscan |
| 15 | 3:2 | 1280x1080 16:9 frame without horizontal overscan |
| 16 | 2:1 | 960x1080 16:9 frame without horizontal overscan |
| 17..254 | Reserved |  |
| 255 | Extended\_SAR |  |

NOTE 1 – For the examples in , the term "without horizontal overscan" refers to display processes in which the display area matches the area of the cropped decoded pictures and the term "with horizontal overscan" refers to display processes in which some parts near the left and/or right border of the cropped decoded pictures are not visible in the display area. As an example, the entry "720x576 4:3 frame with horizontal overscan" for aspect\_ratio\_idc equal to 2 refers to having an area of 704x576 luma samples (which has an aspect ratio of 4:3) of the cropped decoded frame (720x576 luma samples) that is visible in the display area.

**sar\_width** indicates the horizontal size of the sample aspect ratio (in arbitrary units).

**sar\_height** indicates the vertical size of the sample aspect ratio (in the same arbitrary units as sar\_width).

sar\_width and sar\_height shall be relatively prime or equal to 0. When aspect\_ratio\_idc is equal to 0 or sar\_width is equal to 0 or sar\_height is equal to 0, the sample aspect ratio shall be considered unspecified by this Recommendation | International Standard.

**overscan\_info\_present\_flag** equal to 1 specifies that the overscan\_appropriate\_flag is present. When overscan\_info\_present\_flag is equal to 0 or is not present, the preferred display method for the video signal is unspecified.

**overscan\_appropriate\_flag** equal to 1 indicates that the cropped decoded pictures output are suitable for display using overscan. overscan\_appropriate\_flag equal to 0 indicates that the cropped decoded pictures output contain visually important information in the entire region out to the edges of the cropping rectangle of the picture, such that the cropped decoded pictures output should not be displayed using overscan. Instead, they should be displayed using either an exact match between the display area and the cropping rectangle, or using underscan. As used in this paragraph, the term "overscan" refers to display processes in which some parts near the borders of the cropped decoded pictures are not visible in the display area. The term "underscan" describes display processes in which the entire cropped decoded pictures are visible in the display area, but they do not cover the entire display area. For display processes that neither use overscan nor underscan, the display area exactly matches the area of the cropped decoded pictures.

NOTE 2 – For example, overscan\_appropriate\_flag equal to 1 might be used for entertainment television programming, or for a live view of people in a videoconference, and overscan\_appropriate\_flag equal to 0 might be used for computer screen capture or security camera content.

**video\_signal\_type\_present\_flag** equal to 1 specifies that video\_format, video\_full\_range\_flag and colour\_description\_present\_flag are present. video\_signal\_type\_present\_flag equal to 0, specify that video\_format, video\_full\_range\_flag and colour\_description\_present\_flag are not present.

**video\_format** indicates the representation of the pictures as specified in , before being coded in accordance with this Recommendation | International Standard. When the video\_format syntax element is not present, video\_format value is inferred to be equal to 5.

Table E‑2 – Colour primaries

|  |  |  |
| --- | --- | --- |
| Value | Primaries | Informative Remark |
| 0 | Reserved | For future use by ITU‑T | ISO/IEC |
| 1 | primary x y  green 0.300 0.600  blue 0.150 0.060  red 0.640 0.330  white D65 0.3127 0.3290 | Rec. ITU‑R BT.709-5  ITU-R Rec. BT.1361 conventional colour gamut system and extended colour gamut system  IEC 61966-2-4  Society of Motion Picture and Television Engineers RP 177 (1993) Annex B |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU‑T | ISO/IEC |
| 4 | primary x y  green 0.21 0.71  blue 0.14 0.08  red 0.67 0.33  white C 0.310 0.316 | Rec. ITU‑R BT.470‑6 System M (historical)  United States National Television System Committee 1953 Recommendation for transmission standards for colour television  United States Federal Communications Commission Title 47 Code of Federal Regulations (2003) 73.682 (a) (20) |
| 5 | primary x y  green 0.29 0.60  blue 0.15 0.06  red 0.64 0.33  white D65 0.3127 0.3290 | Rec. ITU‑R BT.470‑6 System B, G (historical)  Rec. ITU‑R BT.601‑6 625  Rec. ITU‑R BT.1358 625  Rec. ITU‑R BT.1700 625 PAL and 625 SECAM |
| 6 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | Rec. ITU‑R BT.601‑6 525  Rec. ITU‑R BT.1358 525  Rec. ITU‑R BT.1700 NTSC  Society of Motion Picture and Television Engineers 170M (2004)  (functionally the same as the value 7) |
| 7 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | Society of Motion Picture and Television Engineers 240M (1999)  (functionally the same as the value 6) |
| 8 | primary x y  green 0.243 0.692 (Wratten 58)  blue 0.145 0.049 (Wratten 47)  red 0.681 0.319 (Wratten 25)  white C 0.310 0.316 | Generic film (colour filters using Illuminant C) |
| 9..255 | Reserved | For future use by ITU‑T | ISO/IEC |

**transfer\_characteristics** indicates the opto-electronic transfer characteristic of the source picture as specified in as a function of a linear optical intensity input Lc with a nominal real-valued range of 0 to 1.

When the transfer\_characteristics syntax element is not present, the value of transfer\_characteristics is inferred to be equal to 2 (the transfer characteristics are unspecified or are determined by the application).

Table E‑3 – Transfer characteristics

| Value | Transfer Characteristic | Informative Remark |
| --- | --- | --- |
| 0 | Reserved | For future use by ITU‑T | ISO/IEC |
| 1 | V = 1.099 \* Lc0.45 − 0.099 for 1 >= Lc >= 0.018  V = 4.500 \* Lc for 0.018 > Lc >= 0 | Rec. ITU‑R BT.709-5  Rec. ITU‑R BT.1361 conventional colour gamut system  (functionally the same as the value 6) |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU‑T | ISO/IEC |
| 4 | Assumed display gamma 2.2 | Rec. ITU‑R BT.470‑6 System M (historical)  United States National Television System Committee 1953 Recommendation for transmission standards for colour television  United States Federal Communications Commission Title 47 Code of Federal Regulations (2003) 73.682 (a) (20)  Rec. ITU‑R BT.1700 (2007 revision) 625 PAL and 625 SECAM |
| 5 | Assumed display gamma 2.8 | Rec. ITU‑R BT.470-6 System B, G (historical) |
| 6 | V = 1.099 \* Lc0.45 − 0.099 for 1 >= Lc >= 0.018  V = 4.500 \* Lc for 0.018 > Lc >= 0 | Rec. ITU‑R BT.601‑6 525 or 625  Rec. ITU‑R BT.1358 525 or 625  Rec. ITU‑R BT.1700 NTSC  Society of Motion Picture and Television Engineers 170M (2004)  (functionally the same as the value 1) |
| 7 | V = 1.1115 \* Lc0.45 − 0.1115 for 1 >= Lc >= 0.0228  V = 4.0 \* Lc for 0.0228 > Lc >= 0 | Society of Motion Picture and Television Engineers 240M (1999) |
| 8 | V = Lc for 1 > Lc >= 0 | Linear transfer characteristics |
| 9 | V = 1.0 + Log10( Lc ) ÷ 2 for 1 >= Lc >= 0.01  V = 0.0 for 0.01 > Lc >= 0 | Logarithmic transfer characteristic (100:1 range) |
| 10 | V = 1.0 + Log10( Lc ) ÷ 2.5 for 1 >= Lc >= Sqrt( 10 ) ÷ 1000  V = 0.0 for Sqrt( 10 ) ÷ 1000 > Lc >= 0 | Logarithmic transfer characteristic (100 \* Sqrt( 10 ) : 1 range) |
| 11 | V = 1.099 \* Lc0.45 − 0.099 for Lc >= 0.018  V = 4.500 \* Lc for 0.018 > Lc > −0.018  V = −1.099 \* ( −Lc )0.45 + 0.099 for −0.018 >= Lc | IEC 61966-2-4 |
| 12 | V = 1.099 \* Lc0.45 − 0.099 for 1.33 > Lc >= 0.018  V = 4.500 \* Lc for 0.018 > Lc >= −0.0045  V = −( 1.099 \* ( −4 \* Lc )0.45 − 0.099 ) ÷ 4 for −0.0045 > Lc >= −0.25 | Rec. ITU‑R BT.1361 extended colour gamut system |
| 13..255 | Reserved | For future use by ITU‑T | ISO/IEC |

**matrix\_coefficients** describes the matrix coefficients used in deriving luma and chroma signals from the green, blue, and red primaries, as specified in .

matrix\_coefficients shall not be equal to 0 unless one or more of the following conditions are true:

– BitDepthC is equal to BitDepthY,

– chroma\_format\_idc is equal to 3 (4:4:4).

The specification of the use of matrix\_coefficients equal to 0 under all other conditions is reserved for future use by ITU‑T | ISO/IEC.

matrix\_coefficients shall not be equal to 8 unless one or more of the following conditions are true:

– BitDepthC is equal to BitDepthY,

– BitDepthC is equal to BitDepthY + 1 and chroma\_format\_idc is equal to 3 (4:4:4).

The specification of the use of matrix\_coefficients equal to 8 under all other conditions is reserved for future use by ITU‑T | ISO/IEC.

When the matrix\_coefficients syntax element is not present, the value of matrix\_coefficients is inferred to be equal to 2 (unspecified).

For the interpretation of matrix\_coefficients, the following is specified:

– The range of E′R, E′G, and E′B is specified as follows:

– If transfer\_characteristics is not equal to 11 or 12, E′R, E′G, and E′B are real numbers with values in the range of 0 to 1.

– Otherwise (transfer\_characteristics is equal to 11 (IEC 61966-2-4) or 12 (Rec. ITU-R BT.1361 extended colour gamut system)), E′R, E′G and E′B are real numbers with a larger range not specified in this Recommendation.

– Nominal white is specified as having E′R equal to 1, E′G equal to 1, and E′B equal to 1.

– Nominal black is specified as having E′R equal to 0, E′G equal to 0, and E′B equal to 0.

The interpretation of matrix\_coefficients is specified as follows:

– If video\_full\_range\_flag is equal to 0, the following applies:

– If matrix\_coefficients is equal to 1, 4, 5, 6, or 7, the following equations apply:

Y = Clip1Y( Round( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′Y + 16 ) ) ) (E‑1)

Cb = Clip1C( Round( ( 1 << ( BitDepthC − 8 ) ) \* ( 224 \* E′PB + 128 ) ) ) (E‑2)

Cr = Clip1C( Round( ( 1 << ( BitDepthC − 8 ) ) \* ( 224 \* E′PR + 128 ) ) ) (E‑3)

– Otherwise, if matrix\_coefficients is equal to 0 or 8, the following equations apply:

R = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′R + 16 ) ) (E‑4)

G = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′G + 16 ) ) (E‑5)

B = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′B + 16 ) ) (E‑6)

– Otherwise, if matrix\_coefficients is equal to 2, the interpretation of the matrix\_coefficients syntax element is unknown or is determined by the application.

– Otherwise (matrix\_coefficients is not equal to 0, 1, 2, 4, 5, 6, 7, or 8), the interpretation of the matrix\_coefficients syntax element is reserved for future definition by ITU‑T | ISO/IEC.

– Otherwise (video\_full\_range\_flag is equal to 1), the following applies:

– If matrix\_coefficients is equal to 1, 4, 5, 6, or 7, the following equations apply:

Y = Clip1Y( Round( ( ( 1 << BitDepthY ) − 1 ) \* E′Y ) ) (E‑7)

Cb = Clip1C( Round( ( ( 1 << BitDepthC ) − 1 ) \* E′PB + ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑8)

Cr = Clip1C( Round( ( ( 1 << BitDepthC ) − 1 ) \* E′PR + ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑9)

– Otherwise, if matrix\_coefficients is equal to 0 or 8, the following equations apply:

R = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′R ) (E‑10)

G = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′G ) (E‑11)

B = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′B ) (E‑12)

– Otherwise, if matrix\_coefficients is equal to 2, the interpretation of the matrix\_coefficients syntax element is unknown or is determined by the application.

– Otherwise (matrix\_coefficients is not equal to 0, 1, 2, 4, 5, 6, 7, or 8), the interpretation of the matrix\_coefficients syntax element is reserved for future definition by ITU‑T | ISO/IEC.

The variables E′Y, E′PB, and E′PR (for matrix\_coefficients not equal to 0 or 8) or Y, Cb, and Cr (for matrix\_coefficients equal to 0 or 8) are specified as follows:

– If matrix\_coefficients is not equal to 0 or 8, the following equations apply:

E′Y = KR \* E′R + ( 1 − KR − KB ) \* E′G + KB \* E′B (E‑13)

E′PB = 0.5 \* ( E′B − E′Y ) ÷ ( 1 − KB ) (E‑14)

E′PR = 0.5 \* ( E′R − E′Y ) ÷ ( 1 − KR ) (E‑15)

NOTE 3 – E′Y is a real number with the value 0 associated with nominal black and the value 1 associated with nominal white. E′PB and E′PR are real numbers with the value 0 associated with both nominal black and nominal white. When transfer\_characteristics is not equal to 11 or 12, E′Y is a real number with values in the range of 0 to 1. When transfer\_characteristics is not equal to 11 or 12, E′PB and E′PR are real numbers with values in the range of −0.5 to 0.5. When transfer\_characteristics is equal to 11 (IEC 61966-2-4), or 12 (ITU‑R BT.1361 extended colour gamut system), E′Y, E′PB and E′PR are real numbers with a larger range not specified in this Recommendation.

– Otherwise, if matrix\_coefficients is equal to 0, the following equations apply:

Y = Round( G ) (E‑16)

Cb = Round( B ) (E‑17)

Cr = Round( R ) (E‑18)

– Otherwise (matrix\_coefficients is equal to 8), the following applies:

– If BitDepthC is equal to BitDepthY, the following equations apply:

Y = Round( 0.5 \* G + 0.25 \* ( R + B ) ) (E‑19)

Cb = Round( 0.5 \* G − 0.25 \* ( R + B ) ) + ( 1 << ( BitDepthC − 1 ) ) (E‑20)

Cr = Round( 0.5 \* (R − B ) ) + ( 1 << ( BitDepthC − 1 ) ) (E‑21)

NOTE 4 – For purposes of the YCgCo nomenclature used in , Cb and Cr of Equations  and may be referred to as Cg and Co, respectively. The inverse conversion for the above three equations should be computed as:

t = Y − ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) (E‑22)

G = Clip1Y( Y + ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑23)

B = Clip1Y( t − ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑24)

R = Clip1Y( t + ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑25)

– Otherwise (BitDepthC is not equal to BitDepthY), the following equations apply:

Cr = Round( R ) − Round( B ) + ( 1 << ( BitDepthC − 1 ) ) (E‑26)

t = Round( B ) + ( ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑27)

Cb = Round( G ) − t + ( 1 << ( BitDepthC − 1 ) ) (E‑28)

Y = t + ( ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑29)

NOTE 5 – For purposes of the YCgCo nomenclature used in , Cb and Cr of Equations  and may be referred to as Cg and Co, respectively. The inverse conversion for the above four equations should be computed as.

t = Y − ( ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑30)

G = Clip1Y( t + ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑31)

B = Clip1Y( t − ( ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) ) (E‑32)

R = Clip1Y( B + ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑33)

Table E‑4 – Matrix coefficients

|  |  |  |
| --- | --- | --- |
| Value | Matrix | Informative remark |
| 0 | GBR | Typically referred to as RGB; see Equations  to |
| 1 | KR = 0.2126; KB = 0.0722 | ITU‑R Rec. BT.709-5  ITU‑R Rec. BT.1361 conventional colour gamut system and extended colour gamut system  IEC 61966-2-4 xvYCC709  Society of Motion Picture and Television Engineers RP 177 (1993) Annex B |
| 2 | Unspecified | Image characteristics are unknown or are determined by the application. |
| 3 | Reserved | For future use by ITU‑T | ISO/IEC |
| 4 | KR = 0.30; KB = 0.11 | United States Federal Communications Commission Title 47 Code of Federal Regulations (2003) 73.682 (a) (20) |
| 5 | KR = 0.299; KB = 0.114 | ITU‑R Rec. BT.470‑6 System B, G (historical)  ITU‑R Rec. BT.601‑6 625  ITU‑R Rec. BT.1358 625  ITU‑R Rec. BT.1700 625 PAL and 625 SECAM  IEC 61966-2-4 xvYCC601  (functionally the same as the value 6) |
| 6 | KR = 0.299; KB = 0.114 | ITU‑R Rec. BT.601‑6 525  ITU‑R Rec. BT.1358 525  ITU‑R Rec. BT.1700 NTSC  Society of Motion Picture and Television Engineers 170M (2004)  (functionally the same as the value 5) |
| 7 | KR = 0.212; KB = 0.087 | Society of Motion Picture and Television Engineers 240M (1999) |
| 8 | YCgCo | See Equations  to |
| 9..255 | Reserved | For future use by ITU‑T | ISO/IEC |

**chroma\_loc\_info\_present\_flag** equal to 1 specifies that chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field are present. chroma\_loc\_info\_present\_flag equal to 0 specifies that chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field are not present.

When chroma\_format\_idc is not equal to 1, chroma\_loc\_info\_present\_flag should be equal to 0.

**chroma\_sample\_loc\_type\_top\_field** and **chroma\_sample\_loc\_type\_bottom\_field** specify the location of chroma samples as follows:

– If chroma\_format\_idc is equal to 1 (4:2:0 chroma format), chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field specify the location of chroma samples for the top field and the bottom field, respectively, as shown in .

– Otherwise (chroma\_format\_idc is not equal to 1), the values of the syntax elements chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field shall be ignored. When chroma\_format\_idc is equal to 2 (4:2:2 chroma format) or 3 (4:4:4 chroma format), the location of chroma samples is specified in subclause . When chroma\_format\_idc is equal to 0, there is no chroma sample array.

The value of chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field shall be in the range of 0 to 5, inclusive. When the chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field are not present, the values of chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field is inferred to be equal to 0.

NOTE 6 – When coding progressive source material, chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field should have the same value.



Figure E‑1 – Location of chroma samples for top and bottom fields for chroma\_format\_idc equal to 1 (4:2:0 chroma format) as a function of chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field

**neutral\_chroma\_indication\_flag** equal to 1 indicates that the value of all decoded chroma samples is equal to 1 << ( BitDepthC − 1 ). neutral\_chroma\_indication\_flag equal to 0 provides no indication of decoded chroma sample values. When neutral\_chroma\_indication\_flag is equal to 1, it is a requirement of bitstream conformance that the value of all decoded chroma samples produced by the decoding process shall be equal to  1 << ( BitDepthC − 1 ). When neutral\_chroma\_indication\_flag is not present, it is inferred to be equal to 0.

NOTE 7 – When neutral\_chroma\_indication\_flag is equal to 1, it is not necessary for the decoder to apply the specified decoding process in order to determine the value of the decoded chroma samples.

**field\_seq\_flag** equal to 1 indicates that the coded video sequence conveys pictures that represent fields, and specifies that a field indication SEI message shall be present in every access unit of the current coded video sequence. field\_seq\_flag equal to 0 indicates that the coded video sequence conveys pictures that represent frames and that a field indication SEI message may or may not be present in any access unit of the current coded video sequence. When field\_seq\_flag is not present, it is inferred to be equal to 0.

NOTE 8 – The specified decoding process does not treat access units conveying pictures that represent fields or frames differently. A sequence of pictures that represent fields would therefore be coded with the picture dimensions of an individual field. For example, access units containing pictures that represent 1080i fields would commonly have cropped output dimensions of 1920x540, while the sequence picture rate would commonly express the rate of the source fields (typically between 50 and 60 Hz), instead of the source frame rate (typically between 25 and 30 Hz).

**timing\_info\_present\_flag** equal to 1 specifies that num\_units\_in\_tick, time\_scale and fixed\_pic\_rate\_flag are present in the bitstream. timing\_info\_present\_flag equal to 0 specifies that num\_units\_in\_tick, time\_scale and fixed\_pic\_rate\_flag are not present in the bitstream.

**num\_units\_in\_tick** is the number of time units of a clock operating at the frequency time\_scale Hz that corresponds to one increment (called a clock tick) of a clock tick counter. num\_units\_in\_tick shall be greater than 0. A clock tick is the minimum interval of time that can be represented in the coded data when sub\_pic\_cpb\_params\_present\_flag is equal to 0. For example, when the picture rate of a video signal is 30 000 ÷ 1001 Hz, time\_scale may be equal to 60 000 and num\_units\_in\_tick may be equal to 1001. See Equation .

**time\_scale** is the number of time units that pass in one second. For example, a time coordinate system that measures time using a 27 MHz clock has a time\_scale of 27 000 000. time\_scale shall be greater than 0.

**fixed\_pic\_rate\_flag** equal to 1 indicates that the temporal distance between the HRD output times of any two consecutive pictures in output order is constrained as follows. fixed\_pic\_rate\_flag equal to 0 indicates that no such constraints apply to the temporal distance between the HRD output times of any two consecutive pictures in output order.

When fixed\_pic\_rate\_flag is not present, it is inferred to be equal to 0.

When fixed\_pic\_rate\_flag is equal to 1 for a coded video sequence containing picture n, the value computed for Δto,dpb( n ) as specified in Equation  shall be equal to tc as specified in Equation  (using the value of tc for the coded video sequence containing picture n) when one or more of the following conditions are true for the following picture nn that is specified for use in Equation :

– picture nn is in the same coded video sequence as picture n.

– picture nn is in a different coded video sequence and fixed\_pic\_rate\_flag is equal to 1 in the coded video sequence containing picture nn and the value of num\_units\_in\_tick ÷ time\_scale is the same for both coded video sequences.

**nal\_hrd\_parameters\_present\_flag** equal to 1 specifies that NAL HRD parameters (pertaining to Type II bitstream conformance) are present. nal\_hrd\_parameters\_present\_flag equal to 0 specifies that NAL HRD parameters are not present.

NOTE 9 – When nal\_hrd\_parameters\_present\_flag is equal to 0, the conformance of the bitstream cannot be verified without provision of the NAL HRD parameters and all buffering period and picture timing SEI messages, by some means not specified in this Recommendation | International Standard.

When nal\_hrd\_parameters\_present\_flag is equal to 1, NAL HRD parameters (subclauses  and ) immediately follow the flag.

The variable NalHrdBpPresentFlag is derived as follows:

– If any of the following is true, the value of NalHrdBpPresentFlag shall be set equal to 1:

– nal\_hrd\_parameters\_present\_flag is present in the bitstream and is equal to 1,

– the need for presence of buffering periods for NAL HRD operation to be present in the bitstream in buffering period SEI messages is determined by the application, by some means not specified in this Recommendation | International Standard.

– Otherwise, the value of NalHrdBpPresentFlag shall be set equal to 0.

**vcl\_hrd\_parameters\_present\_flag** equal to 1 specifies that VCL HRD parameters (pertaining to all bitstream conformance) are present. vcl\_hrd\_parameters\_present\_flag equal to 0 specifies that VCL HRD parameters are not present.

NOTE 10 – When vcl\_hrd\_parameters\_present\_flag is equal to 0, the conformance of the bitstream cannot be verified without provision of the VCL HRD parameters and all buffering period and picture timing SEI messages, by some means not specified in this Recommendation | International Standard.

When vcl\_hrd\_parameters\_present\_flag is equal to 1, VCL HRD parameters (subclauses  and ) immediately follow the flag.

The variable VclHrdBpPresentFlag is derived as follows:

– If any of the following is true, the value of VclHrdBpPresentFlag shall be set equal to 1:

– vcl\_hrd\_parameters\_present\_flag is present in the bitstream and is equal to 1,

– the need for presence of buffering periods for VCL HRD operation to be present in the bitstream in buffering period SEI messages is determined by the application, by some means not specified in this Recommendation | International Standard.

– Otherwise, the value of VclHrdBpPresentFlag shall be set equal to 0.

The variable CpbDpbDelaysPresentFlag is derived as follows:

– If any of the following is true, the value of CpbDpbDelaysPresentFlag shall be set equal to 1:

– nal\_hrd\_parameters\_present\_flag is present in the bitstream and is equal to 1,

– vcl\_hrd\_parameters\_present\_flag is present in the bitstream and is equal to 1,

– the need for presence of CPB and DPB output delays to be present in the bitstream in picture timing SEI messages is determined by the application, by some means not specified in this Recommendation | International Standard.

– Otherwise, the value of CpbDpbDelaysPresentFlag shall be set equal to 0.

**low\_delay\_hrd\_flag** specifies the HRD operational mode as specified in Annex . When fixed\_pic\_rate\_flag is equal to 1, low\_delay\_hrd\_flag shall be equal to 0. When low\_delay\_hrd\_flag is not present, its value is inferred to be equal to 1 − fixed\_pic\_rate\_flag.

NOTE 11 – When low\_delay\_hrd\_flag is equal to 1, "big pictures" that violate the nominal CPB removal times due to the number of bits used by an access unit are permitted. It is expected, but not required, that such "big pictures" occur only occasionally.

**sub\_pic\_cpb\_params\_present\_flag** equal to 1 specifies that sub-picture level CPB removal delay parameters are present and the CPB may operate at access unit level or sub-picture level. sub\_pic\_cpb\_params\_present\_flag equal to 0 specifies that sub-picture level CPB removal delay parameters are not present and the CPB operates at access unit level. When sub\_pic\_cpb\_params\_present\_flag is not present, its value is inferred to be equal to 0.

**num\_units\_in\_sub\_tick** is the number of time units of a clock operating at the frequency time\_scale Hz that corresponds to one increment (called a sub-picture clock tick) of a sub-picture clock tick counter. num\_units\_in\_sub\_tick shall be greater than 0. A sub-picture clock tick is the minimum interval of time that can be represented in the coded data when sub\_pic\_cpb\_params\_present\_flag is equal to 1.

**bitstream\_restriction\_flag** equal to 1, specifies that the following coded video sequence bitstream restriction parameters are present. bitstream\_restriction\_flag equal to 0, specifies that the following coded video sequence bitstream restriction parameters are not present.

**tiles\_fixed\_structure\_flag** equal to 1 indicates that each picture parameter set that is active in the coded video sequence has the same value of the syntax elements num\_tile\_columns\_minus1, num\_tile\_rows\_minus1, uniform\_spacing\_flag, column\_width[ i ], row\_height[ i ] and loop\_filter\_across\_tiles\_enabled\_flag, when present. tiles\_fixed\_structure\_flag equal to 0 indicates that tiles syntax elements in different picture parameter sets may or may not have the same value. When the tiles\_fixed\_structure\_flag syntax element is not present, it is inferred to be equal to 0.

NOTE 12 – The signalling of tiles\_fixed\_structure\_flag equal to 1 is a guarantee to a decoder that each picture in the coded video sequence has the same number of tiles distributed in the same way which might be useful for workload allocation in the case of multi-threaded decoding.

**motion\_vectors\_over\_pic\_boundaries\_flag** equal to 0 indicates that no sample outside the picture boundaries and no sample at a fractional sample position for which the sample value is derived using one or more samples outside the picture boundaries is used for inter prediction of any sample. motion\_vectors\_over\_pic\_boundaries\_flag equal to 1 indicates that one or more samples outside picture boundaries may be used in inter prediction. When the motion\_vectors\_over\_pic\_boundaries\_flag syntax element is not present, motion\_vectors\_over\_pic\_boundaries\_flag value is inferred to be equal to 1.

**max\_bytes\_per\_pic\_denom** indicates a number of bytes not exceeded by the sum of the sizes of the VCL NAL units associated with any coded picture in the coded video sequence.

The number of bytes that represent a picture in the NAL unit stream is specified for this purpose as the total number of bytes of VCL NAL unit data (i.e., the total of the NumBytesInNALunit variables for the VCL NAL units) for the picture. The value of max\_bytes\_per\_pic\_denom shall be in the range of 0 to 16, inclusive.

Depending on max\_bytes\_per\_pic\_denom the following applies:

– If max\_bytes\_per\_pic\_denom is equal to 0, no limits are indicated.

– Otherwise (max\_bytes\_per\_pic\_denom is not equal to 0), it is a requirement of bitstream conformance that no coded picture shall be represented in the coded video sequence by more than the following number of bytes.

( PicSizeInMinCbs \* RawMinCUBits ) ÷ ( 8 \* max\_bytes\_per\_pic\_denom ) (E‑34)

When the max\_bytes\_per\_pic\_denom syntax element is not present, the value of max\_bytes\_per\_pic\_denom is inferred to be equal to 2.

**max\_bits\_per\_mincu\_denom** indicates an upper bound for the number of coded bits of coding\_unit( ) data for any coding block in any picture of the coded video sequence. The value of max\_bits\_per\_mincu\_denom shall be in the range of 0 to 16, inclusive.

Depending on max\_bits\_per\_mincu\_denom the following applies:

– If max\_bits\_per\_mincu\_denom is equal to 0, no limit is specified by this syntax element.

– Otherwise (max\_bits\_per\_mincu\_denom is not equal to 0), it is a requirement of bitstream conformance that no coded coding\_unit( ) shall be represented in the bitstream by more than the following number of bits.

( 128 + RawMinCUBits ) ÷ max\_bits\_per\_mincu\_denom \* ( 2 << log2CbSize − log2MinCbSize ) (E‑35)

where log2CbSize is the value of log2CbSize for the given coding block and the number of bits of coding\_unit( ) data for the same coding block is given by the number of times read\_bits( 1 ) is called in subclauses 9.2.3.2.2 and 9.2.3.2.3.

When the max\_bits\_per\_mincu\_denom is not present,thevalue of max\_bits\_per\_mincu\_denom is inferred to be equal to 1.

**log2\_max\_mv\_length\_horizontal** and **log2\_max\_mv\_length\_vertical** indicate the maximum absolute value of a decoded horizontal and vertical motion vector component, respectively, in ¼ luma sample units, for all pictures in the coded video sequence. A value of n asserts that no value of a motion vector component shall exceed the range from −2n to 2n − 1, inclusive, in units of ¼ luma sample displacement. The value of log2\_max\_mv\_length\_horizontal shall be in the range of 0 to 16, inclusive. The value of log2\_max\_mv\_length\_vertical shall be in the range of 0 to 16, inclusive. When log2\_max\_mv\_length\_horizontal is not present, the values of log2\_max\_mv\_length\_horizontal and log2\_max\_mv\_length\_vertical is inferred to be equal to 16.

NOTE 13 – The maximum absolute value of a decoded vertical or horizontal motion vector component is also constrained by profile and level limits as specified in Annex .

* + 1. HRD parameters semantics

**cpb\_cnt\_minus1** plus 1 specifies the number of alternative CPB specifications in the bitstream. The value of cpb\_cnt\_minus1 shall be in the range of 0 to 31, inclusive. When low\_delay\_hrd\_flag is equal to 1, cpb\_cnt\_minus1 shall be equal to 0. When cpb\_cnt\_minus1 is not present, it is inferred to be equal to 0.

**bit\_rate\_scale** (together with bit\_rate\_value\_minus1[ SchedSelIdx ]) specifies the maximum input bit rate of the SchedSelIdx-th CPB.

**cpb\_size\_scale** (together with cpb\_size\_value\_minus1[ SchedSelIdx ]) specifies the CPB size of the SchedSelIdx-th CPB.

**bit\_rate\_value\_minus1[** SchedSelIdx **]** (together with bit\_rate\_scale) specifies the maximum input bit rate for the SchedSelIdx‑th CPB. bit\_rate\_value\_minus1[ SchedSelIdx ] shall be in the range of 0 to 232 − 2, inclusive. For any SchedSelIdx > 0, bit\_rate\_value\_minus1[ SchedSelIdx ] shall be greater than bit\_rate\_value\_minus1[ SchedSelIdx − 1 ]. The bit rate in bits per second is given by

BitRate[ SchedSelIdx ] = ( bit\_rate\_value\_minus1[ SchedSelIdx ] + 1 ) \* 2(6 + bit\_rate\_scale) (E‑36)

When the bit\_rate\_value\_minus1[ SchedSelIdx ] syntax element is not present, the value of BitRate[ SchedSelIdx ] is inferred as follows:

– If profile\_idc is equal to 100, BitRate[ SchedSelIdx ] is inferred to be equal to 1000 \* MaxBR for VCL HRD parameters and to be equal to 1200 \* MaxBR for NAL HRD parameters, where MaxBR is specified in subclause .

– Otherwise, BitRate[ SchedSelIdx ] is inferred to be equal to cpbBrVclFactor \* MaxBR for VCL HRD parameters and to be equal to cpbBrNalFactor \* MaxBR for NAL HRD parameters, where MaxBR, cpbBrVclFactor and cpbBrNalFactor are specified in subclause .

**cpb\_size\_value\_minus1[** SchedSelIdx **]** is used together with cpb\_size\_scale to specify the SchedSelIdx-th CPB size. cpb\_size\_value\_minus1[ SchedSelIdx ] shall be in the range of 0 to 232 − 2, inclusive. For any SchedSelIdx greater than 0, cpb\_size\_value\_minus1[ SchedSelIdx ] shall be less than or equal to cpb\_size\_value\_minus1[ SchedSelIdx −1 ].

The CPB size in bits is given by

CpbSize[ SchedSelIdx ] = ( cpb\_size\_value\_minus1[ SchedSelIdx ] + 1 ) \* 2(4 + cpb\_size\_scale) (E‑)

When the cpb\_size\_value\_minus1[ SchedSelIdx ] syntax element is not present, the value of CpbSize[ SchedSelIdx ] is inferred as follows:

– If profile\_idc is equal to 100, CpbSize[ SchedSelIdx ] is inferred to be equal to 1000 \* MaxCPB for VCL HRD parameters and to be equal to 1200 \* MaxCPB for NAL HRD parameters, where MaxCPB is specified in subclause . [Ed. (GJS): Looks wrong, and why are there two items here? Why not just let Annex A take care of this?]

– Otherwise, CpbSize[ SchedSelIdx ] is inferred to be equal to cpbBrVclFactor \* MaxCPB for VCL HRD parameters and to be equal to cpbBrNalFactor \* MaxCPB for NAL HRD parameters, where MaxCPB, cpbBrVclFactor and cpbBrNalFactor are specified in subclause .

**cbr\_flag[** SchedSelIdx **]** equal to 0 specifies that to decode this bitstream by the HRD using the SchedSelIdx-th CPB specification, the hypothetical stream delivery scheduler (HSS) operates in an intermittent bit rate mode. cbr\_flag[ SchedSelIdx ] equal to 1 specifies that the HSS operates in a constant bit rate (CBR) mode. When the cbr\_flag[ SchedSelIdx ] syntax element is not present, the value of cbr\_flag is inferred to be equal to 0.

**initial\_cpb\_removal\_delay\_length\_minus1** specifies the length in bits of the initial\_cpb\_removal\_delay[ SchedSelIdx ] and initial\_cpb\_removal\_delay\_offset[ SchedSelIdx ] syntax elements of the buffering period SEI message. The length of initial\_cpb\_removal\_delay[ SchedSelIdx ] and of initial\_cpb\_removal\_delay\_offset[ SchedSelIdx ] is initial\_cpb\_removal\_delay\_length\_minus1 + 1. When the initial\_cpb\_removal\_delay\_length\_minus1 syntax element is present in more than one hrd\_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the initial\_cpb\_removal\_delay\_length\_minus1 parameters shall be equal in both hrd\_parameters( ) syntax structures. When the initial\_cpb\_removal\_delay\_length\_minus1 syntax element is not present, it is inferred to be equal to 23.

**cpb\_removal\_delay\_length\_minus1** specifies the length in bits of the cpb\_removal\_delay syntax element. The length of the cpb\_removal\_delay syntax element of the picture timing SEI message is cpb\_removal\_delay\_length\_minus1 + 1. When the cpb\_removal\_delay\_length\_minus1 syntax element is present in more than one hrd\_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the cpb\_removal\_delay\_length\_minus1 parameters shall be equal in both hrd\_parameters( ) syntax structures. When the cpb\_removal\_delay\_length\_minus1 syntax element is not present, it is inferred to be equal to 23.

**dpb\_output\_delay\_length\_minus1** specifies the length in bits of the dpb\_output\_delay syntax element. The length of the dpb\_output\_delay syntax element of the picture timing SEI message is dpb\_output\_delay\_length\_minus1 + 1. When the dpb\_output\_delay\_length\_minus1 syntax element is present in more than one hrd\_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the dpb\_output\_delay\_length\_minus1 parameters shall be equal in both hrd\_parameters( ) syntax structures. When the dpb\_output\_delay\_length\_minus1 syntax element is not present, it is inferred to be equal to 23.

**time\_offset\_length** greater than 0 specifies the length in bits of the time\_offset syntax element. time\_offset\_length equal to 0 specifies that the time\_offset syntax element is not present. When the time\_offset\_length syntax element is present in more than one hrd\_parameters( ) syntax structure within the VUI parameters syntax structure, the value of the time\_offset\_length parameters shall be equal in both hrd\_parameters( ) syntax structures. When the time\_offset\_length syntax element is not present, it is inferred to be equal to 24.