**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 N18370**

**March 2019, Geneva, CH**

|  |  |
| --- | --- |
| **Source** | **JVET** |
| **Status** | **Approved** |
| **Title** | **Working Draft 5 of Versatile Video Coding** |

|  |  |
| --- | --- |
| **Joint Video Experts Team (JVET)**  **of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11**  14th Meeting: Geneva, CH, 19–27 Mar. 2019 | Document: JVET-N1001-v10 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **Versatile Video Coding (Draft 5)** | | |
| *Status:* | Output document approved by JVET | | |
| *Purpose:* | Draft text of video coding specification | | |
| *Author(s) or Contact(s):* | Benjamin Bross Jianle Chen Shan Liu | Email: | firstname.lastname@hhi.fraunhofer.de jianle.chen@futurewei.com shanl@tencent.com |
| *Source:* | Editors | | |

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

# Abstract

Draft 5 of Versatile Video Coding.

Ed. Notes:

* Incorporated JVET-N0067: NAL unit header.
* Incorporated JVET-N0278: Layer concept without inter-layer referencing.
* Incorporated JVET-N0349: Decoding parameter set (DPS).
* Incorporated JVET-N0869: Semantics of DPS ID equal to 0.
* Incorporated JVET-N0805: APS types (with loop).
* Incorporated JVET-N0100: Signalling information about long-term reference picture POC LSB.
* Incorporated JVET-N0100: Conditional signalling of num\_ref\_idx\_active\_overwrite\_flag.
* Incorporated JVET-N0288: Inference rule for num\_bricks\_in\_slice\_minus1.
* Incorporated JVET-N0124: Inference rule for single\_brick\_per\_slice\_flag and signalling the bottom right brick index as a delta from top left brick index.
* Incorporated JVET-N0047: Ref pic list for IDR.
* Incorporated JVET-N0276: 5 new constraint flags.
* Incorporated JVET-N0276: Sub-profile indication.
* Incorporated JVET-N0865: Gradual random access (GRA).
* Incorporated JVET-N0101: CRA decoding process.
* Incorporated JVET-N0070: Ref pic wraparound cleanups.
* Incorporated JVET-N0352: Conformance window.
* Incorporated JVET-N0438: Loop filtering disabling aross virtual boundaries.
* Incorporated JVET-N0498: Uniform tile partitioning.
* Incorporated JVET-N0150: WPP with 1-CTU lag.
* Incorporated JVET-N0120: Editorial improvements to the general decoding process to clarify the difference between a CVS and a bitstream.
* Incorporated JVET-N0857: Tile and brick partitioning.
* Incorporated JVET-N0706: Decoded picture hash SEI message.
* Incorporated JVET-N0494: Dependent RAP indication SEI message.
* Incorporated JVET-N0353: Buffering period & picture timing SEI messages.
* Incorporated JVET-N0867: Temporal scalability HRD parameters.
* Incorporated JVET-N0063: VUI design.
* Incorporated JVET-N0423 & N0350: HRD starting point.
* Incorporated JVET-N0266: Remove 4x4 unipred, and 4x8/8x4 bipred regular inter modes, and remove the use of for shared merge candidates in the regular merge list.
* Incorporated JVET-N0821: 6-tap interpolation filter for affine MC.
* Incorporated JVET-N0068: Restriction of memory bandwidth consumption of affine MC.
* Incorporated JVET-N0481: BCW index inheritance for constructed affine merge candidate.
* Incorporated JVET-N0334: MV clipping for MMVD, DMVR and CPMV of constructed affine merge candidadates.
* Incorporated JVET-N0407: Disable 8x8/4xN CUs for DMVR.
* Incorporated JVET-N0178: Implicitly split BDOF application region along 16x16 boundaries.
* Incorporated JVET-N0146: Disabling BDOF for BCW and WP, and align the condidtion of DMVR and BIO.
* Incorporated JVET-N0444: Condition check of block height is not equal to 4 for BDOF.
* Incorporated JVET-N0325: 8-bit fixed precision for BDOF calculations.
* Incorporated JVET-N0127: MMVD enabling signalling in SPS
* Incorporated JVET-N0332: disable MVD scaling for MMVD for LTRP.
* Incorporated JVET-N0448/JVET-N0380: infer the number of MMVD base merge candidate to 1.
* Incorporated JVET-N0302: CIIP with position-independent weights.
* Incorporated JVET-N0324: Signal a regular merge flag right after the merge flag and skip flag.
* Incorporated JVET-N0851/N0447/N0400/N0500: align triangle merge candidate number and regular merge candidate number.
* Incorporated JVET-N0340: regular merge candidate list is re-used for triangle.
* Incorporated JVET-N0483: disable sub-block transform when triangle mode is used.
* Incorporated JVET-M0335: Round MVs toward zero.
* Incorporated JVET-N0235: Add an SPS-level flag to turn on and off symmetric MVD
* Incorporated JVET-N0213: Remove TMVP from merge and AMVP list for 4x8 and 8x4 CUs
* Incorporated JVET-N0868: Revised spec text for DMVR reconciling with software ticket [#214](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/214)
* Fixed minor bugs and typos reported in [#94](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/94), [#221](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/221), [#227](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/227), [#248](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/248).
* Fixed bug [#232](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/232): Mismatch between specification and software for triangle merge mode weighted prediction.
* Fixed minor bugs and typos reported in [#249](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/249), [#253](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/253), [#256](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/256).
* Fixed bug [#247](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/247): Fix of size constraint for triangle merge mode when MMVD is disabled in SPS (also in VTM).
* Incorporated JVET-M0471/N0473: Deblocking with long tap filters (including ISP and SBT transform boundaries).
* Incorporated JVET-M0908: Deblocking of CIIP boundaries.
* Incorporated JVET-M0277: Apply pcm\_loop\_filter\_disabled\_flag for ALF.
* Incorporated JVET-N0242: Non-linear ALF with clipping.
* Incorporated fixes for JVET-N0278: Layer concept without inter-layer referencing.
* Fixed some bugs related to reference picture list, tiles and bricks (including the bug reported in [#261](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/261)), presence of the VUI syntax in the SPS, CABAC initialization, HRD, semantics of the picture timing SEI message, etc.
* Incorporated JVET-N0415: CTU-adaptive ALF with fixed filter.
* Incorporated JVET-N0180: ALF line buffer reduction.
* Incorporated JVET-N0220: Simplification of LMCS and other fixes.
* Incorporated JVET-N0477: LMCS with no chroma residue scaling in case of CBF=0.
* Incorporated JVET-N0383/N0251: IBC search range constraint.
* Incorporated JVET-N0175/N0251/N0384: IBC search range increase for CTUs less than 128x128.
* Incorporated JVET-N0843: IBC motion vector prediction unification for merge and MVP mode.
* Incorporated JVET-N0317: Put (0,0) vector as default in IBC merge list.
* Incorporated JVET-N0318/N0427: Disable IBC for 128x128 blocks.
* Fixed bug [#262](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/262): Missing text for IBC with shared merge list.
* Incorporated JVET-N0185: Unified MPM list for intra mode coding.
* Incorporated JVET-N0137: Disable 2x2/4x2/2x4 in dual tree only.
* Incorporated JVET-N0308: Restrict the maximum CU size for ISP to be 64x64.
* Incorporated JVET-N0435: Harmonization between WAIP and intra smoothing filters.
* Incorporated JVET-N0271: CCLM derived from four neighbouring samples.
* Fixed bug [#223](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/223): Fixed ISP interpolation filter conditions.
* Fixed bug [#165](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/165): Removed top-left sample availability derivation process in for CCLM.
* Added SPS flag for ISP.
* Added SPS flag for MRL.
* Incorporated JVET-N0217: Matrix-based intra prediction (MIP).
* Fixed bug [#276](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/276): Added regular\_merge\_flag to context derivation table.
* Fixed bug [#277](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/277): Replaced all occurances of slice-level slice\_loop\_filter\_across\_slices\_enabled\_flag with PPS loop\_filter\_across\_slices\_enabled\_flag.
* Incorporated JVET-N0470: Consider SMVD flag for BDOF condition.
* Added SPS flag for MIP.
* Fixed bugs [#279](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/279), [#280](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/280), [#282](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/282), [#283](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/283), [#284](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/284).
* Incorporated JVET-N0054: Joint coding of chrominance residuals.
* Incorporated JVET-N0188: Unified rice parameter derivation for coefficient level coding.
* Incorporated JVET-N0492: Chroma TU CBF dependent luma TU CBF signalling.
* Incorporated JVET-N0103: Determine coefficient group size based on TB size instead of colour component.
* Incorporated JVET-N0600: Context reduction for amvr\_flag syntax.
* Incorporated JVET-N0286: Context reduction for bcw\_idx syntax.
* Incorporated JVET-N0194: Context selection of last x/y syntax based on non-reduced TU size in case of zero-out.
* Incorporated JVET-N0280: Transform skip residual coding.
* Incorporated JVET-N0413: Block-based quantized residual domain DPCM (BDPCM).
* Fixed bug [#292](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/292): Minor transform skip residual coding issues.
* Fixed bug [#270](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/270): (Bug fix) Changed nuh\_layer\_id to nuh\_layer\_id\_plus1.
* Fixed bug [#298](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/298): typo in IBC BV candidate list derivation.
* Fixed bug [#275](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/275): Wrong bdofUtilizationFlag indexing.
* Fixed bug [#274](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/274): Undefined variables sGxGym and sGxGys.
* Fixed bug [#273](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/273): Wrong HMVP table size.
* Fixed bug [#299](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/299): Minor intra prediction issues.
* Fixed bug [#302](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/302): Fix CBF flags syntax and semantics for implicit TU split.
* Fixed bugs [#307](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/307), [#308](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/308), [#316](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/316): Deblocking issues.
* Fixed the bug that DeltaPocMsbCycleLt was used without being defined.
* Incorporated JVET-N0193: Low frequency non-separable transform (LFNST).
* Incorporated JVET-N0105: Remove mode dependency in LFNST ctx derivation.
* Incorporated JVET-N0866: Unification of implicit transform selection for ISP and SBT.
* Incorporated JVET-N0246: Include square root of 2 factor in levelScale values.
* Incorporated JVET-N0671: Support of 4:4:4 and 4:2:2 chroma formats.
* Incorporated JVET-N0225: Separate color plane id.
* Incorporated JVET-N0847: Default and user-defined scaling matrices
* Incorporated DMVR fixes and editorial improvements.
* Fixed bug [#245](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/245): wrong handling of boundary partitioning for a corner case.
* Fixed bug [#341](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/341): coefficients outside the first coefficient group are considered in 8x8 LFNST.

Draft 4 of Versatile Video Coding.

Ed. Notes:

* Incorporated JVET-M0102: Intra subpartitions (ISP).
* Incorporated JVET-M0142: chroma format dependent CCLM downsampling filter.
* Incorporated JVET-M0064: table reduction in CCLM model parameter calculation.
* Incorporated JVET-M0092: intra reference sample filtering cleanup.
* Incorporated JVET-M0238: PDPC linear interpolation on the secondary boundary for adjacent angular modes is changed to nearest neighbour.
* Incorporated JVET-M0407: CPR search range.
* Fixed bug [#154](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/154) Availability check for CPR/IBC chroma CU reference block is missing.
* Aligned PDPC filtering for INTRA\_ANGULAR18 and INTRA\_ANGULAR50 to VTM.
* Fixed bug [#167](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/167) on PDPC size condition.
* Incorporated JVET-M0497: Fast DST-7/DCT-8.
* Incorporated JVET-M0303: Shape adaptive transform selection.
* Incorporated JVET-M0464/JVET-M0201: Combined transform skip(TS) and MTS syntax plus extended TS sizes.
* Incorporated JVET-M0297: Zero-out of last 16 samples for 32 samples DST-7/DCT-8.
* Incorporated JVET-M0140: Subblock transform for inter CUs.
* Incorporated JVET-M0251/M0257: Zero-out of last 32 samples for 64 samples DCT-2 fix.
* Incorporated JVET-M0273/M0240/M0116/M0338/M0204: only using left neighbor for SbTMVP fetching
* Incorporated JVET-M0246: AMVR for affine
* Incorporated JVET-M0145: affine sub-block MV clipping
* Incorporated JVET-M0166/M0228/M0477: remove MV comparison for constructed merge candidates
* Incorporated JVET-M0170: Parallel processing for merge mode
* Incorporated JVET-M0147: Decoder side motion vector refinement
* Incorporated JVET-M0361: fix of cu\_cbf for merge mode
* Incorporated JVET-M0487: using integer samples instead of bilinear interpolation for extended region of BDOF.
* Incorporated JVET-M0483: IBC signalled as a separate CU prediction mode.
* Incorporated JVET-M0063: Generalization of BDOF bit-depth.
* Incorporated triangular modifications including:
  + JVET-M0118/M0185/M0190/M207(test 1)/M0216(the first aspect)/M0234 (change corresponding to the result table 7 and 8)/M0317(section 2.2)/M0328: Do not signal the triangular prediction mode flag in cases where the combination is not allowed (MMVD, CIIP),
  + JVET-M0328: always use second weight group in triangular prediction.
* Incorporated JVET-M0883: signaling change of triangular merging candidate which does not need LUT.
* Incorporated JVET-M0193: pairwise average merging candidate reduction.
* Incorporated HMVP modifications including:
  + JVET-M0436 reduce HMVP number from 6 to 5,
  + JVET-M0300 HMVP initialization for parallel processing with tiles,
  + JVET-M0264 BCW weight is also stored in HMVP,
  + JVET-M0126 reduced HMVP candidate pruning.
* Incorporated JVET-M0255: MMVD mode without fractional sample offsets for screen content coding.
* Incorporated JVET-M0171/M0068: remove redundant MV scaling in MMVD.
* Incorporated JVET-M0444: symmetrical MVD coding for L0 to L1.
* Incorporated JVET-M0479: MV clipping to 18 bits.
* Incorporated JVET-M0512: TMVP storage reduction.
* Incorporated JVET-M0192: subblock chroma MV derivation for affine from two luma MVs.
* Incorporated JVET-M0111: weighted prediction (WP) and disable BCW signalling if WP is enabled.
* Incorporated JVET-M0281/M0117: modified AMVP pruning with rounding.
* Fixed bug [#175](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/175) incorrect derivation of CCLM parameter b.
* Incorporated JVET-M0128: Reference picture management
* Incorporated JVET-M0132: Adaptation parameter set (APS)
* Incorporated JVET-M0853: Adding the support of rectangular slices in addition to the existing raster-scan slices, and enabling extraction of MCTSs without changing VLC NAL units
* Incorporated JVET-M0160: Adding loop\_filter\_across\_slice\_enabled\_flag to the PPS
* Incorporated JVET-M0101:
  + Replace the existing IRAP\_NUT with 3 new NAL unit types: IDR\_W\_RADL, IDR\_N\_LP, CRA\_NUT (from JVET-M0101).
  + Add external means flag HandleCraAsCvsStartFlag, with similar text as in HEVC. Text provided in a v3 of JVET-M0101.
  + Add a NUT value for step-wise temporal access STSA (from JVET-M0101).
  + Add a NUT value for AUD (from JVET-M0101).
  + Add sps\_max\_sub\_layers\_minus1 syntax element to SPS, and decoding process in 8.1.1, 8.1.2 and 8.1.3 of JVET-M0101.
  + Add text of sections 7.4.2.4 to 7.4.2.4.5 on NAL unit order and AU boundary detection from JVET-M0101, which is primarily editorial, but has some technical aspects.
  + Add profile\_tier\_level( ) syntax structure which includes sub-layer level idc (similar to HEVC but without sub-layer-specific profiles).
  + Add general\_non\_packed\_constraint\_flag with semantics as in JVET-M0101 (rename the flag to display\_suitability\_flag? – that's editorial).
  + Add the temporal scalability sub-bitstream extraction process in JVET-M0101.
  + Add RASL and RADL NUTs
* Incorporated JVET-M0451: Add new constraint flags corresponding to VVC WD 3 tools.
* Incorporated JVET-M0415: Change the sps\_ref\_wraparound\_offset to sps\_ref\_wraparound\_offset\_minus1 and changing the units to be MinCbSizeY as in option 1 (minor cleanup).
* Incorporated JVET-M0381: Reduce merge idx ctx coded bins (test 2.2.2a).
* Incorporated JVET-M0502: Add one context for pred\_mode\_flag (method 2).
* Incorporated JVET-M0453: Modified CABAC probability estimation (5.1.13\* + init from 5.1.2).
* Fixed bug [#147](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/147) on coefficient coding.
* Incorporated JVET-M0470: Limited EGk for abs\_rem/ dec\_abs\_level.
* Incorporated JVET-M0173: Move rem\_abs\_gt3\_flag into first coding pass.
* Incorporated JVET-M0119: Modified dequantization scaling for TS.
* Incorporated JVET-M0685: QP prediction fix for parallel encoding.
* Incorporated JVET-M0113/M0188: Bug fix for quantization group QP signalling.
* Incorporated JVET-M0421: Split-first signalling for partitioning.
* Incorporated JVET-M0446/M0888/M0905: Inferred QT split to avoid 32x128/128x32 partitions at picture boundaries.
* Incorporated JVET-M0427: Picture reconstruction with luma mapping and chroma scaling (LMCS).

Draft 3 of Versatile Video Coding.

Ed. Notes:

* Fixed bug [#92](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/92) missing MV clipping in interplolation process.
* Incorporated JVET-L0664: disabling 5x5 ALF for luma component.
* Incorporated JVET-L0082/JVET-L0083: reduction of bits for ALF coefficients.
* Incorporated JVET-L0147: subsampling of ALF classifiers.
* Incorporated SAO as found in HEVC.
* Incorporated deblocking filter as found in HEVC with the following modifications:
  + adapt processing to dual tree CTU partitioning,
  + replace RQT based transfrom block processing with implicit TU split for large blocks (JVET-K0307, JVET-K0237, JVET-K0369, JVET-K0232, JVET-K0315),
  + replace prediction blocks with coding subblocks (JVET-L0074),
  + only apply deblocking on transfrom block / coding subblock edges if CU is aligned on 8x8 grid in the respecitive direction.
* Incorporated JVET-L0410: tc table fix
* Incorporated JVET-L0414: luma adaptive deblocking filter.
* Fixed bug [#97](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/97) mismatch between agreed ALF text JVET-K0564 and draft by removing alf\_chroma\_ctb\_present\_flag.
* Incorporated JVET-L0285: 8-bit transform matrix.
* Incorporated JVET-L0118: MTS index fix.
* Incorporated JVET-L0059: remove dependency on number non-zero coeff. for MTS index signalling.
* Incorporated JVET-L0362: QP signalling.
* Incorporated JVET-L0428: delta QP and chroma QP offset for dual tree.
* Incorporated JVET-K0251: increase maximum QP value from 51 to 63 (including fix from JVET-L0553).
* Incorporated JVET-L0217: fix relation between QT/BT/TT syntax elements.
* Incorporated JVET-L0678: QT/BT/TT syntax overriding in slice header.
* Incorporated JVET-L0081: BT/TT constraint.
* Incorporated JVET-L0191: cclm simplification using min and max values.
* Incorporated JVET-L0136: cclm with line buffer restriction.
* Incorporated JVET-L0340: multi-directional cclm.
* Incorporated JVET-L0053/JVET-L0272: modified chroma direct mode.
* Incorporated JVET-L0628: intra 4-tap interpolation filter.
* Incorporated JVET-L0165: intra 6 MPMs.
* Incorporated JVET-L0283: multi-line intra prediction.
* Incorporated JVET-L0279: unification of intra angular prediction.
* Incorporated PCM mode from HEVC and JVET-L0209: PCM mode with dual tree partition.
* Incorporated JVET-L0694: combination of affine mode and subblock temporal merging candidate modifications including:
  + JVET-L0142/JVET-L0366/JVET-L0632: affine merge candidate list (CE4 4.2.6.d as modifed in L0632).
  + JVET-L0369: moving subblock temporal merging candidate into affine merge candidate list (CE4 4.2.8).
  + JVET-L0045: line buffer reduction for affine inherited candidates, location 1 (CE4 4.1.11.a).
  + JVET-L0047: modification of affine control point motion vector storage (method 1).
  + JVET-L0271: Simplification of affine motion vector predictor candidate list construction (CE4 4.1.6.a).
* Incorporated JVET-L0265: change chroma subblock size to 4x4 instead of 2x2 for affine motion compensation.
* Incorporated JVET-L0198: fix subblock size to 8x8 for subblock TMVP (JVET-L0468/JVET-L0104).
* Incorporated JVET-L0198: use first spatial neighbouring MV for collocated subblock TMVP position
* Incorporated JVET-L0055: restrict subblock TMVP to CUs with width >= 8 and height >=8.
* Fixed bug [#120](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/120) clipping in vertical subblock TMVP location goes beyond the current CTU row.
* Incorporated JVET-L0274: reduce number of context coded bins (CE7 7.1.3.b).
* Incorporated missing context derivation for all CABAC-coded syntax elements.
* Incorporated JVET-L0361: context modeling of CU split modes.
* Incorporated JVET-L0194: use one context for the first bin of merge\_idx and bypass coding for the others.
* Incorporated JVET-L0266: History-based motion vector prediction.
* Incorporated JVET-L0158: Reset the HMVP FIFO list for each CTU row.
* Incorporated JVET-L0104: Prohibit 4x4 bi-prediction for inter CU.
* Incorporated JVET-L0054: Merge with MVD (MMVD) (CE4 4.5.4.b).
* Incorporated JVET-L0100: Combined inter merge / intra prediction (CIIP).
* Incorporated JVET-L0090: Pairwise average merging candidates.
* Incorporated JVET-L0646: Bi-prediction with CU weights.
* Incorporated JVET-L0256: Bidirectional optical flow (BDOF).
* Incorporated JVET-L0231: Horizontal MV wrap-around.
* Fixes and cleanups:
  + fixed minor typos,
  + resolved open issues from editors notes,
  + reordered SPS flags to be more aligned with VTM,
  + put all skip/merge related CU syntax in a merge\_data( ) syntax structure (editorial),
  + fixed order of merge syntax by moving merge\_idx and after combined merge/intra syntax,
  + conditioned combined merge/intra syntax on sps\_ciip\_enabled\_flag.
* Incorporated JVET-L0124: Triangular inter-picture prediction mode.
* Incorporated JVET-L0293: Current picture referencing (CPR).
* Fixed bug [#142](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/142) corrected order of derivation process for subblock-based temporal merging base motion data.
* Fixed bug [#137](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/137) wrong CCLM parameter derivation condition.
* Fixed issues related to triangular partitions.
* Incorporated JVET-L0064: Agreement in principle on the need of CRA and its signalling in NAL unit header
* Incorporated JVET-L0248: A restriction on the TemporalId values of the current picture and the active PPS
* Incorporated JVET-L0249: POC signalling and derivation
* Incorporated JVET-L0449: POC LSB in the slice headers for all picture types
* Incorporated JVET-L0686: Tiling and tiling grouping; removed traditional slices
* Incorporated JVET-L0696: Interoperability point and constraint flags

Draft 2 of Versatile Video Coding.

Ed. Notes:

* Incorporated JVET-K0230: Separate trees for intra slices (without multi-DMs) with an implicit split to 64x64;
* Incorporated JVET-K0556: Prohibit ternary split of something bigger than 64 in width or height (and not send the bit to indicate ternary type at that level).
* Incorporated JVET-K0351 (test c): Keep only the TT restriction (preventing binary split with same orientation in center partition of the ternary split)
* Incorporated JVET-K0554: Implicit splitting at picture boundaries and ensure MinQTSize at boundary splits
* Fixed bug [#65](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/65) typos and unused variables in section 6.4
* Fixed bug [#67](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/67) implicit vertical BT split at picture boundary issue
* Incorporated JVET-K0072: Dependent quantization with fallback switch at the picture level and modified entropy coding supporting dependent quantization including:
  + adapted scaling to non-square transform blocks,
  + added binarization process for abs\_remainder,
  + specified CoeffMin and CoeffMax with fixed values,
  + added 0-th order Exp-Golomb code parsing process.
* Incorporated JVET-K0310: Sign data hiding (can only be used when dependent quantization is disabled).
* Incorporated JVET-K0529: Intra prediction using 3MPM on 67 prediction modes (Planar, DC and 65 angular modes)
* Incorporated JVET-K0122: DC prediction without division.
* Incorporated JVET-K0500: Wide-angle intra prediction.
* Incorporated JVET-K0063: Position-dependent intra prediction combination
* Incorporated JVET-K0190: Cross-component linear model intra prediction
* Incorporated multiple transfrom selection (MTS) for both intra and inter, each controlled by an SPS flag.
* Incorporated transform skip.
* Fixed bug [#68](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/68) various typos
* Fixed bug [#71](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/71) various typos
* Fixed bug [#72](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/72) on CCLM
* Incorporated JVET-K0357: adaptive motion vector resolution (AMVR)
* Incorpor,ated JVET-K0565: affine motion compensation (MC) including:
  + JVET-K0052: Affine merge bug fix
  + JVET-K0184: Affine MC (CE4.1.1a 4x4 fixed subblock size).
  + JVET-K0337: Affine MC coding and models (4.1.3a, affine MVP list construction, and 4.1.3b, MV difference coding, and 4.1.3c, 4/6 parameter model, no slice level switch).
  + JVET-K0367/JVET-K0052/JVET-K0103: Restriction of affine merge mode to CU sizes >= 8x8
* Incorporated 1/16 motion compensation (MC) including:
  + 1/16 MV storage
  + 1/16 merge and affine MVs
  + MVDs in AMVR accuracy (1/4,1,4) shifted to 1/16
  + Inter MVP candidates rounded to AMVR accuracy (1/4,1,4) and shifted to 1/16
  + 1/16 luma and 1/32 chroma interpolation filters
* Incorporated subblock-based temporal merging candidates with 8x8 motion vector storage (JVET-K0346).
* Incorporated JVET-K0371: 4x4 block classification based Adaptive Loop Filter (ALF).
* Fixed bug [#75](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/75) regarding a bottom and right boundary partition issue.
* Fixed bug [#90](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/90) typos in copying the control point vectors to temporal notion vectors.
* Fixed bug [#86](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/86) in intra reference sample filtering.
* Incorporated JVET-K0325: High Level Syntax (HLS) starting point.
* Fixed bug [#82](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/82) on zeroing-out high frequency transform coefficients for larger TUs (>32x32).
* Fixed bug [#85](https://jvet.hhi.fraunhofer.de/trac/vvc/ticket/85) on MTS index coding.

Draft 1 of Versatile Video Coding.

Ed. Notes:

* Incorporated basic definitions, abbreviations and conventions
* Incorporated a basic high-level syntax (HLS) with NAL units, SPS, PPS and slice header.
* Incorporated block partitioning by a quadtree with nested multi-type tree using binary and ternary splits with
  + CU leaf nodes
  + Prediction at CU level
  + Transform at CU level
  + Minimum CU size with 4x4 luma coding block and corresponding chroma coding blocks (2x2 for 4:2:0)
  + Maximum TU size with 64x64 luma transform block and corresponding chroma transform blocks (32x32 for 4:2:0)
  + Minimum TU size with 4x4 luma transform block and corresponding chroma transform blocks (2x2 for 4:2:0)
  + Single tree for luma and chroma

**CONTENTS**

*Page*

Abstract i

1 Scope 1

2 Normative references 1

2.1 Identical Recommendations | International Standards 1

2.2 Paired Recommendations | International Standards equivalent in technical content 1

2.3 Additional references 1

3 Definitions 1

4 Abbreviations 8

5 Conventions 10

5.1 General 10

5.2 Arithmetic operators 10

5.3 Logical operators 10

5.4 Relational operators 10

5.5 Bit-wise operators 10

5.6 Assignment operators 11

5.7 Range notation 11

5.8 Mathematical functions 11

5.9 Order of operation precedence 12

5.10 Variables, syntax elements and tables 13

5.11 Text description of logical operations 14

5.12 Processes 15

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 16

6.1 Bitstream formats 16

6.2 Source, decoded and output picture formats 16

6.3 Partitioning of pictures, slices, tiles, bricks, and CTUs 18

6.3.1 Partitioning of pictures into slices, tiles, and bricks 18

6.3.2 Block, quadtree and multi-type tree structures 20

6.3.3 Spatial or component-wise partitionings 21

6.4 Availability processes 21

6.4.1 Allowed quad split process 21

6.4.2 Allowed binary split process 22

6.4.3 Allowed ternary split process 24

6.4.4 Derivation process for neighbouring block availability 24

6.5 Scanning processes 25

6.5.1 CTB raster scanning, tile scanning, and brick scanning process 25

6.5.2 Up-right diagonal and raster scan order array initialization process 27

7 Syntax and semantics 29

7.1 Method of specifying syntax in tabular form 29

7.2 Specification of syntax functions and descriptors 30

7.3 Syntax in tabular form 31

7.3.1 NAL unit syntax 31

7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 32

7.3.3 Profile, tier, and level syntax 40

7.3.4 Supplemental enhancement information message syntax 42

7.3.5 Slice header syntax 42

7.3.6 Reference picture list structure syntax 49

7.3.7 Slice data syntax 50

7.4 Semantics 69

7.4.1 General 69

7.4.2 NAL unit semantics 69

7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 74

7.4.4 Profile, tier, and level semantics 93

7.4.5 Supplemental enhancement information message semantics 95

7.4.6 Slice header semantics 95

7.4.7 Reference picture list structure semantics 109

7.4.8 Slice data semantics 110

8 Decoding process 127

8.1 General decoding process 127

8.2 NAL unit decoding process 129

8.3 Slice decoding process 129

8.3.1 Decoding process for picture order count 129

8.3.2 Decoding process for reference picture lists construction 130

8.3.3 Decoding process for reference picture marking 131

8.3.4 Decoding process for generating unavailable reference pictures 131

8.3.5 Decoding process for symmetric motion vector difference reference indices 132

8.4 Decoding process for coding units coded in intra prediction mode 133

8.4.1 General decoding process for coding units coded in intra prediction mode 133

8.4.2 Derivation process for MIP mode 134

8.4.3 Derivation process for luma intra prediction mode 137

8.4.4 Derivation process for chroma intra prediction mode 140

8.4.5 Decoding process for intra blocks 142

8.5 Decoding process for coding units coded in inter prediction mode 168

8.5.1 General decoding process for coding units coded in inter prediction mode 168

8.5.2 Derivation process for motion vector components and reference indices 172

8.5.3 Decoder side motion vector refinement process 191

8.5.4 Derivation process for triangle motion vector components and reference indices 196

8.5.5 Derivation process for subblock motion vector components and reference indices 198

8.5.6 Decoding process for inter blocks 221

8.5.7 Decoding process for triangle inter blocks 236

8.5.8 Decoding process for the residual signal of coding blocks coded in inter prediction mode 242

8.6 Decoding process for coding units coded in IBC prediction mode 242

8.6.1 General decoding process for coding units coded in IBC prediction mode 242

8.6.2 Derivation process for motion vector components for IBC blocks 245

8.6.3 Decoding process for ibc blocks 250

8.7 Scaling, transformation and array construction process 252

8.7.1 Derivation process for quantization parameters 252

8.7.2 Scaling and transformation process 254

8.7.3 Scaling process for transform coefficients 254

8.7.4 Transformation process for scaled transform coefficients 256

8.7.5 Picture reconstruction process 276

8.8 In-loop Filter Process 278

8.8.1 General 278

8.8.2 Picture inverse mapping process for luma samples 279

8.8.3 Deblocking filter process 280

8.8.4 Sample adaptive offset process 304

8.8.5 Adaptive loop filter process 306

9 Parsing process 314

9.1 General 314

9.2 Parsing process for k-th order Exp-Golomb codes 314

9.2.1 General 314

9.2.2 Mapping process for signed Exp-Golomb codes 315

9.3 Parsing process for truncated unary codes 316

9.4 Parsing process for truncated binary codes 316

9.5 CABAC parsing process for slice data 317

9.5.1 General 317

9.5.2 Initialization process 318

9.5.3 Binarization process 325

9.5.4 Decoding process flow 335

10 Sub-bitstream extraction process 349

Annex A Profiles, tiers and levels 350

A.1 Overview of profiles, tiers and levels 350

Annex B Byte stream format 351

B.1 General 351

Annex C Hypothetical reference decoder 352

C.1 General 352

C.2 Operation of coded picture buffer 355

C.2.1 General 355

C.2.2 Timing of decoding unit arrival 355

C.2.3 Timing of access unit removal and decoding of access unit 356

C.3 Operation of the decoded picture buffer 358

C.3.1 General 358

C.3.2 Removal of pictures from the DPB before decoding of the current picture 358

C.3.3 Picture output 358

C.3.4 Current decoded picture marking and storage 359

C.4 Bitstream conformance 359

C.5 Decoder conformance 360

C.5.1 General 360

C.5.2 Operation of the output order DPB 361

Annex D HRD related supplemental enhancement information 364

D.1 General 364

D.2 SEI payload syntax 364

D.2.1 General SEI message syntax 364

D.2.2 Buffering period SEI message syntax 365

D.2.3 Picture timing SEI message syntax 366

D.3 SEI payload semantics 366

D.3.1 General SEI payload semantics 366

D.3.2 Buffering period SEI message semantics 367

D.3.3 Picture timing SEI message semantics 369

Annex E Video usability information 374

E.1 General 374

E.2 VUI syntax 374

E.2.1 VUI parameters syntax 374

E.2.2 HRD parameters syntax 375

E.2.3 Sub-layer HRD parameters syntax 375

E.3 VUI semantics 375

E.3.1 VUI parameters semantics 375

E.3.2 HRD parameters semantics 389

E.3.3 Sub-layer HRD parameters semantics 391

Annex F Non-HRD related supplemental enhancement information 392

F.1 General 392

F.2 Non-HRD related SEI payload syntax 392

F.2.1 Decoded picture hash SEI message syntax 392

F.2.2 Dependent random access point indication SEI message syntax 392

F.3 Non-HRD related SEI payload semantics 392

F.3.1 Decoded picture hash SEI message semantics 392

F.3.2 Dependent random access point indication SEI message semantics 394

# 

INTERNATIONAL STANDARD

ISO/IEC VVC

ITU-T Rec. H.VVC

ITU-T RECOMMENDATION

Versatile video coding

# Scope

This Recommendation | International Standard specifies versatile video coding.

# Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

## Identical Recommendations | International Standards

– None

## Paired Recommendations | International Standards equivalent in technical content

– None

## Additional references

– [Ed. (BB): Add references as needed.]

# Definitions

[Ed. (BB) included basic definitions to be updated.]

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

* 1. **access unit**: A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order,* and contain exactly one *coded picture* for each present *layer access unit*.
  2. **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.
  3. **ALF APS**: An *APS* that has aps\_params\_type equal to ALF\_APS.
  4. **adaptation parameter set (APS)**: A *syntax structure* containing *syntax elements* that apply to zero or more *slices* as determined by zero or more *syntax elements* found in *slice headers.*
  5. **associated IRAP picture**: The previous *IRAP picture* in *decoding order* (when present).
  6. **associated non-VCL NAL unit**: A *non-VCL NAL unit* (when present) for a *VCL NAL unit* where the *VCL NAL unit* is the *associated VCL NAL unit* of the *non-VCL NAL unit*.
  7. **associated VCL NAL unit**: The preceding *VCL NAL unit* in *decoding order* for a *non-VCL NAL unit* with NalUnitType equal to EOS\_NUT, EOB\_NUT, SUFFIX\_SEI\_NUT, RSV\_NVCL\_7, RSV\_NVCL\_23, or in the range of UNSPEC30..UNSPEC31; or otherwise the next *VCL NAL unit* in *decoding order*.
  8. **bin**: One bit of a *bin string*.
  9. **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
  10. **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
  11. **binary split**: A split of a rectangular MxN *block* of samples into two *blocks* where a vertical split results in a first (M / 2)xN *block* and a second (M / 2)xN *block*, and a horizontal split results in a first Mx(N / 2) *block* and a second Mx(N / 2) *block*.
  12. **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
  13. **bi-predictive (B) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.
  14. **bitstream**: A sequence of bits, in the form of a *NAL unit stream* or a *byte stream*, that forms the representation of *coded pictures* and associated data forming one or more coded video sequences *(CVSs)*.
  15. **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
  16. **brick**: A rectangular region of *CTU* rows within a particular *tile* in a *picture*.

NOTE – A tile may be partitioned into multiple bricks, each of which consisting of one or more CTU rows within the tile. A tile that is not partitioned into multiple bricks is also referred to as a brick. However, a brick that is a true subset of a tile is not referred to as a tile.

* 1. **brick scan**: A specific sequential ordering of *CTUs* *partitioning* a *picture* in which the *CTUs* are ordered consecutively in *CTU* *raster scan* in a *brick*, *bricks* within a *tile* are ordered consecutively in a *raster scan* of the *bricks* of the *tile*, and *tiles* in a *picture* are ordered consecutively in a *raster scan* of the *tiles* of the *picture*.
  2. **byte**: A sequence of 8 bits, within which, when written or read as a sequence of bit values, the left-most and right-most bits represent the most and least significant bits, respectively.
  3. **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*, and 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.
  4. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex B.
  5. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  6. **chroma**: An adjective, represented by the symbols Cb and Cr, specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours.

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**: An *IRAP picture* for which each *VCL NAL unit* has NalUnitType equal to CRA\_NUT.

NOTE – A CRA picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be the first picture in the bitstream in decoding order, or may appear later in the bitstream. A CRA picture may have associated RADL or RASL pictures. When a CRA picture has NoIncorrectPicOutputFlag equal to 1, the associated RASL pictures are not output by the decoder, because they may not be decodable, as they may contain references to pictures that are not present in the bitstream.

* 1. **coded layer video sequence (CLVS)**: A sequence of *layer* *access units* that consists, in *decoding order*, of a *CLVSS layer access unit*, followed by zero or more *layer* *access* *units* that are not *CLVSS layer access units*, including all subsequent *layer* *access units* up to but not including any subsequent *layer* *access unit* that is a *CLVSS layer* *access unit*.

NOTE – A CLVSS layer access unit may be an IDR layer access unit, a CRA layer access unit, or a GRA layer access unit. The value of NoIncorrectPicOutputFlag is equal to 1 for each IDR layer access unit, and each CRA layer access unit that has HandleCraAsCvsStartFlag equal to 1, and each CRA or GRA layer access unit that is the first layer access unit in the layer of the bitstream in decoding order or the first layer access unit in the layer of the bitstream that follows an end of sequence NAL unit in decoding order.

* 1. **coded layer video sequence start (CLVSS) layer access unit**: A *layer* *access unit* in which the *coded picture* is a *CLVSS picture*.
  2. **coded layer video sequence start (CLVSS) picture**: A *coded picture* that is an *IRAP picture* with NoIncorrectPicOutputFlag equal to 1 or a *GRA picture* with NoIncorrectPicOutputFlag equal to 1.
  3. **coded picture**: A *coded representation* of a *picture* comprising *VCL NAL units* with a particular value of NuhLayerId within an *access unit* and containing all *CTUs* of the *picture*.
  4. **coded representation**: A data element as represented in its coded form.
  5. **coded video sequence (CVS)**: A sequence of *access units* that consists, in *decoding order*, of a *CVSS access unit*, followed by zero or more *access* *units* that are not *CVSS access units*, including all subsequent *access units* up to but not including any subsequent *access unit* that is a *CVSS access unit*.
  6. **coded video sequence start (CVSS) access unit**: An *access unit* in which the *coded picture* in each present *layer* *access unit* is a *CLVSS picture*.
  7. **coding block**: An MxN *block* of samples for some values of M and N such that the division of a *CTB* into *coding blocks* is a *partitioning*.
  8. **coding tree block (CTB)**: An NxN *block* of samples for some value of N such that the division of a *component* into *CTBs* is a *partitioning*.
  9. **coding tree unit (CTU)**: A *CTB* of *luma* samples, two corresponding *CTBs* of *chroma* samples of a *picture* that has three sample arrays, or a *CTB* 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.
  10. **coding unit (CU)**: 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.
  11. **component**: An array or single sample from one of the three arrays (*luma* and two *chroma*) that compose 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 compose a *picture* in monochrome format.
  12. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
  13. **decoded picture**: A *decoded picture* is derived by decoding a *coded picture*.
  14. **decoder**: An embodiment of a *decoding process*.
  15. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
  16. **decoding process**: The process specified in this Specification that reads a *bitstream* and derives *decoded* *pictures* from it.
  17. **emulation prevention byte**: A *byte* equal to 0x03 that is present within a *NAL unit* when the *syntax elements* of the *bitstream* form certain patterns of *byte* values in a manner that ensures that no sequence of consecutive *byte-aligned* *bytes* in the *NAL unit* can contain a *start code prefix*.
  18. **encoder**: An embodiment of an *encoding process*.
  19. **encoding process**: A process not specified in this Specification that produces a *bitstream* conforming to this Specification.
  20. **flag**: A variable or single-bit *syntax element* that can take one of the two possible values: 0 and 1.
  21. **frequency index**: A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to the application of a *transform* in the *decoding process.*
  22. **gradual random access (GRA) layer access unit**: A *layer* *access unit* in which the *coded picture* is a GRA picture.
  23. **gradual random access (GRA) picture:** A *picture* for which each VCL NAL unit has NalUnitType equal to GRA\_NUT.
  24. **informative**: A term used to refer to content provided in this Specification that does not establish any mandatory requirements for conformance to this Specification and thus is not considered an integral part of this Specification.
  25. **instantaneous decoding refresh (IDR) layer access unit**: A *layer* *access unit* in which the *coded picture* is an *IDR picture*.
  26. **instantaneous decoding refresh (IDR) picture**: An *IRAP* *picture* for which each *VCL NAL unit* has NalUnitType equal to IDR\_W\_RADL or IDR\_N\_LP.

NOTE – An IDR picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be the first picture in the bitstream in decoding order, or may appear later in the bitstream. Each IDR picture is the first picture of a CVS in decoding order. When an IDR picture for which each VCL NAL unit has NalUnitType equal to IDR\_W\_RADL, it may have associated RADL pictures. When an IDR picture for which each VCL NAL unit has NalUnitType equal to IDR\_N\_LP, it does not have any associated leading pictures. An IDR picture does not have associated RASL pictures.

* 1. **inter coding**: Coding of a *coding block*, *slice*, or *picture* that uses *inter prediction*.
  2. **inter prediction**: A *prediction* derived in a manner that is dependent on data elements (e.g., sample values or motion vectors) of one or more *reference* *pictures*.

NOTE – A prediction from a reference picture that is the current picture itself is also inter prediction.

* 1. **intra block copy (IBC) prediction**: A *prediction* derived in a manner that is dependent on data elements (e.g., sample values or block vectors) of the same decoded *slice* without referring to a *reference picture*.
  2. **intra coding**: Coding of a *coding block, slice*, or *picture* that uses *intra prediction*.
  3. **intra prediction**: A *prediction* derived from only data elements (e.g., sample values) of the same decoded *slice* without referring to a *reference picture*.
  4. **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* in each present *layer access unit* is an *IRAP picture*.
  5. **intra random access point (IRAP) layer access unit**: A *layer access unit* in which the *coded picture* is an *IRAP picture*.
  6. **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has NalUnitType in the range of IDR\_W\_RADL to CRA\_NUT, inclusive.

NOTE – An IRAP picture does not refer to any pictures other than itself for inter prediction in its decoding process, and may be a CRA picture or an IDR picture. The first picture in the bitstream in decoding order must be an IRAP picture. Provided the necessary parameter sets are available when they need to be activated, the IRAP picture and all subsequent non-RASL pictures in decoding order can be correctly decoded without performing the decoding process of any pictures that precede the IRAP picture in decoding order.

* 1. **intra (I) slice**: A *slice* that is decoded using *intra prediction* only.
  2. **layer**: A set of *VCL NAL units* that all have a particular value of NuhLayerId and the *associated non-VCL NAL units*.
  3. **layer access unit**: A set of *NAL units* for which the VCL NAL units all have a particular value of NuhLayerId, that are associated with each other according to a specified classification rule, that are consecutive in *decoding order,* and that contain exactly one *coded picture.*
  4. **leading picture**: A *picture* that that is in the same *layer* as the *associated IRAP picture* and precedes the *associated* *IRAP picture* in *output order*.
  5. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  6. **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Specification, or the value of a *transform coefficient* prior to *scaling*.

NOTE – 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 the specified constraints, support a different level for each supported profile.

* 1. **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into *reference picture list 0* (*list 1*).
  2. **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*).
  3. **LMCS APS**: An *APS* that has aps\_params\_type equal to LMCS\_APS.
  4. **long-term reference picture (LTRP)**: A *picture* that is marked as "used for long-term reference".
  5. **luma**: An adjective, represented by the symbol or subscript Y or L, specifying that a sample array or single sample is representing the monochrome signal related to the primary colours.

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 that is used to refer to behaviour that is allowed, but not necessarily required*.*

NOTE – 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.

* 1. **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*.
  2. **multi-type tree**: A *tree* in which a parent node can be split either into two child nodes using a *binary split* or into three child nodes using a *ternary split*, each of which may become parent node for another split into either two or three child nodes.
  3. **must**: A term that is used in expressing an observation about a requirement or an implication of a requirement that is specified elsewhere in this Specification (used exclusively in an *informative* context).
  4. **network abstraction layer (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. **network abstraction layer (NAL) unit stream**: A sequence of *NAL units*.
  6. **non-IDR picture**: A *coded picture* that is not an *IDR picture*.
  7. **non-IRAP picture**: A *coded picture* that is not an *IRAP picture*.
  8. **non-VCL NAL unit**: A *NAL unit* that is not a *VCL NAL unit*.
  9. **note**: A term that is used to prefix *informative* remarks (used exclusively in an *informative* context).
  10. **output order**: The order in which the *decoded* *pictures* are output from the *decoded picture buffer* (for the *decoded pictures* that are to be output from the *decoded picture buffer*).
  11. **parameter**: A *syntax element* of a *sequence parameter set (SPS)* or *picture parameter set (PPS)*, or the second word of the defined term *quantization parameter*.
  12. **partitioning**: The division of a set into subsets such that each element of the set is in exactly one of the subsets.
  13. **picture**: An array of *luma* samples in monochrome format or an array of *luma* samples and two corresponding arrays of *chroma* samples in 4:2:0, 4:2:2, and 4:4:4 colour format.

NOTE – A picture may be either a frame or a field. However, in one CVS, either all pictures are frames or all pictures are fields.

* 1. **picture parameter set (PPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded pictures* as determined by a *syntax element* found in each *slice header.*
  2. **picture order count (POC)**: A variable that is associated with each *picture*, uniquely identifies the associated *picture* among all *pictures* in the *CLVS*, and, when the associated *picture* is to be output from the *decoded picture buffer*, indicates the position of the associated *picture* in *output order* relative to the *output order* positions of the other *pictures* in the same *CLVS* that are to be output from the *decoded picture buffer*.
  3. **prediction**: An embodiment of the *prediction process*.
  4. **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.
  5. **predictive (P) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.
  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 Specification.
  8. **pulse code modulation (PCM)**: Coding of the samples of a *block* by directly representing the sample values without *prediction* or application of a *transform*.
  9. **quadtree**: A *tree* in which a parent node can be split into four child nodes, each of which may become parent node for another split into four child nodes.
  10. **quantization parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
  11. **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
  12. **random access decodable leading (RADL) layer access unit**: A *layer* *access unit* in which the *coded picture* is a *RADL picture*.
  13. **random access decodable leading (RADL) picture**: A *coded picture* for which each *VCL NAL unit* has NalUnitType equal to RADL\_NUT.

NOTE – All RADL pictures are leading pictures. RADL pictures are not used as reference pictures for the decoding process of trailing pictures of the same associated IRAP picture. When present, all RADL pictures precede, in decoding order, all trailing pictures of the same associated IRAP picture.

* 1. **random access skipped leading (RASL) layer access unit**: A *layer* *access unit* in which the *coded picture* is a *RASL picture.*
  2. **random access skipped leading (RASL) picture**: A *coded picture* for which each *VCL NAL unit* has NalUnitType equal to RASL\_NUT.

NOTE – All RASL pictures are leading pictures of an associated CRA picture. When the associated CRA picture has NoIncorrectPicOutputFlag equal to 1, the RASL picture is not output and may not be correctly decodable, as the RASL picture may contain references to pictures that are not present in the bitstream. RASL pictures are not used as reference pictures for the decoding process of non-RASL pictures. When present, all RASL pictures precede, in decoding order, all trailing pictures of the same associated CRA picture.

* 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* and that is either empty or has the form of a *string of data bits* containing *syntax elements* followed by an *RBSP stop bit* and 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*, for which the location of the end 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. **reference index**: An index into a *reference picture list*.
  5. **reference picture**: A *picture* that is a *short-term reference picture* or a *long-term reference picture*.

NOTE – A reference picture contains samples that may be used for inter prediction in the decoding process of subsequent pictures in decoding order.

* 1. **reference picture list**: A list of *reference pictures* that is used for *inter prediction* of a *P* or *B slice.*

NOTE – Two reference picture lists, reference picture list 0 and reference picture list 1, are generated for each slice of a non-IDR picture. The set of unique pictures referred to by all entries in the two reference picture lists associated with a picture consists of all reference pictures that may be used for inter prediction of the associated picture or any picture following the associated picture in decoding order. For the decoding process of a P slice, only reference picture list 0 is used for inter prediction. For the decoding process of a B slice, both reference picture list 0 and reference picture list 1 are used for inter prediction. For decoding the slice data of an I slice, no reference picture list is used for for inter prediction.

* 1. **reference picture list 0**: The *reference picture list* used for *inter prediction* of a *P* or the first *reference picture list* used for *inter prediction* of a *B* *slice*.
  2. **reference picture list 1**: The second *reference picture list* used for *inter prediction* of a *B slice*.
  3. **reserved**: A term that may be used to specify that some values of a particular *syntax element* are for future use by ITU-T | ISO/IEC and shall not be used in *bitstreams* conforming to this version of this Specification, but may be used in bitstreams conforming to future extensions of this Specification by ITU‑T | ISO/IEC.
  4. **residual**: The decoded difference between a *prediction* of a sample or data element and its decoded value.
  5. **scaling**: The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.
  6. **sequence parameter set (SPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *CVSs* as determined by the content of a *syntax element* found in the *PPS* referred to by a *syntax element* found in each *slice header.*
  7. **shall**: A term used to express mandatory requirements for conformance to this Specification.

NOTE – 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 cropped decoded pictures to those output from the decoding process described in this Specification conforms to the decoding process requirements of this Specification.

* 1. **short-term reference picture (STRP)**: A *picture* that is marked as "used for short-term reference".
  2. **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 Specification.
  3. **slice**: An integer number of *bricks* of a *picture* that are exclusivelycontained in a single *NAL unit*.

NOTE – A slice consists of either a number of complete tiles or only a consecutive sequence of complete bricks of one tile.

* 1. **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *bricks* represented in the *slice*.
  2. **source**: A term used to describe the video material or some of its attributes before encoding.
  3. **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

NOTE – 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.

* 1. **step-wise temporal sub-layer access (STSA) layer access unit**: A *layer* *access unit* in which the *coded picture* is an *STSA picture*.
  2. **step-wise temporal sub-layer access (STSA) picture**: A *coded picture* for which each *VCL NAL unit* has NalUnitType equal to STSA\_NUT.

NOTE – An STSA picture does not use pictures with the same TemporalId as the STSA picture for inter prediction reference. Pictures following an STSA picture in decoding order with the same TemporalId as the STSA picture do not use pictures prior to the STSA picture in decoding order with the same TemporalId as the STSA picture for inter prediction reference. An STSA picture enables up-switching, at the STSA picture, to the sub-layer containing the STSA picture, from the immediately lower sub-layer. STSA pictures must have TemporalId greater than 0.

* 1. **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*, where 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.
  2. **sub-bitstream extraction process**: A specified process by which *NAL units* in a *bitstream* that do not belong to a target set, determined by a target highest TemporalId and a target LayerId, are removed from the *bitstream*, with the output sub-bitstream consisting of the NAL units in the *bitstream* that belong to the target set.
  3. **sub-layer**: A temporal scalable layer of a temporal scalable *bitstream*, consisting of *VCL NAL units* with a particular value of the TemporalId variable and the associated *non-VCL NAL units*.
  4. **syntax element**: An element of data represented in the *bitstream*.
  5. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  6. **ternary split**: A split of a rectangular MxN *block* of samples into three *blocks* where a vertical split results in a first (M / 4)xN *block*, a second (M / 2)xN *block*, a third (M / 4)xN *block*, and a horizontal split results in a first Mx(N / 4) *block*, a second Mx(N / 2) *block*, a third Mx(N / 4) *block*.
  7. **tier**: A specified category of *level* constraints imposed on values of the *syntax elements* in the *bitstream*, where the *level* constraints are nested within a *tier* and a *decoder* conforming to a certain *tier* and *level* would be capable of decoding all *bitstreams* that conform to the same *tier* or the lower *tier* of that *level* or any *level* below it.
  8. **tile**: A rectangular region of *CTUs* within a particular *tile column* and a particular *tile row* in a *picture*.
  9. **tile column**: A rectangular region of *CTUs* having a height equal to the height of the *picture* and a width specified by *syntax elements* in the *picture parameter set*.
  10. **tile row**: A rectangular region of *CTUs* having a height specified by *syntax elements* in the *picture parameter set* and a width equal to the width of the *picture*.
  11. **tile scan**: A specific sequential ordering of *CTUs* *partitioning* a *picture* in which the *CTUs* are ordered consecutively in *CTU* *raster scan* in a *tile* whereas *tiles* in a *picture* are ordered consecutively in a *raster scan* of the *tiles* of the *picture*.
  12. **trailing picture**: A non-IRAP *picture* that follows the *associated IRAP picture* in *output order* and that is not an STSA picture.

NOTE – Trailing pictures associated with an IRAP picture also follow the IRAP picture in decoding order. Pictures that follow the associated IRAP picture in output order and precede the associated IRAP picture in decoding order are not allowed.

* 1. **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.
  2. **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.
  3. **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 a *transform* in the *decoding process*.
  4. **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.
  5. **transform unit (TU)**: A *transform block* of *luma* samples and two corresponding *transform blocks* of *chroma* samples of a *picture* and *syntax structures* used to transform the *transform block* samples.
  6. **tree**: A tree is a finite set of nodes with a unique root node.
  7. **unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.
  8. **video coding layer (VCL) NAL unit**: A collective term for *coded slice NAL units* and the subset of *NAL units* that have *reserved* values of NalUnitType that are classified as VCL NAL units in this Specification.

# Abbreviations

[Ed. (BB) included some basic definitions (some of which are not currently used), to be updated.]

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

ALF Adaptive Loop Filter

AMVR Adaptive Motion Vector Resolution

APS Adaptation Parameter Set

B Bi-predictive

BCW Bi-prediction with CU-level Weights

BDPCM Block-based Delta Pulse Code Modulation

CABAC Context-based Adaptive Binary Arithmetic Coding

CB Coding Block

CBR Constant Bit Rate

CPB Coded Picture Buffer

CRA Clean Random Access

CRC Cyclic Redundancy Check

CTB Coding Tree Block

CTU Coding Tree Unit

CU Coding Unit

CVS Coded Video Sequence

DPB Decoded Picture Buffer

DPS Decoding Parameter Set

DRAP Dependent Random Access Point

EG Exponential-Golomb

EGk k-th order Exponential-Golomb

FCC Federal Communications Commission (of the United States)

FIFO First-In, First-Out

FIR Finite Impulse Response

FL Fixed-Length

GBR Green, Blue and Red

GRA Gradual Random Access

I Intra

IBC Intra Block Copy

IDR Instantaneous Decoding Refresh

IRAP Intra Random Access Point

LFNST Low Frequency Non-Separable Transform

LPS Least Probable Symbol

LSB Least Significant Bit

LTRP Long-Term Reference Picture

LMCS Luma Mapping with Chroma Scaling

MIP Matrix-based Intra Prediction

MPS Most Probable Symbol

MSB Most Significant Bit

MTS Multiple Transform Selection

MVP Motion Vector Prediction

NAL Network Abstraction Layer

NTSC National Television System Committee (of the United States)

P Predictive

PCM Pulse Code Modulation

POC Picture Order Count

PPS Picture Parameter Set

QP Quantization Parameter

RADL Random Access Decodable Leading (Picture)

RASL Random Access Skipped Leading (Picture)

RBSP Raw Byte Sequence Payload

RGB Same as GBR

RPS Reference Picture Set

SAO Sample Adaptive Offset

SAR Sample Aspect Ratio

SEI Supplemental Enhancement Information

SMPTE Society of Motion Picture and Television Engineers

SODB String Of Data Bits

SPS Sequence Parameter Set

STRP Short-Term Reference Picture

STSA Step-wise Temporal Sub-layer Access

TR Truncated Rice

UCS Universal Coded Character Set

UTF UCS Transmission Format

VBR Variable Bit Rate

VCL Video Coding Layer

VPS Video Parameter Set

# Conventions

## General

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, e.g., "the first" is equivalent to the 0-th, "the second" is equivalent to the 1-th, etc.

## 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 |
| xy | 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 most significant bits (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 least significant bits (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 and z being greater than y.

## Mathematical functions

The following mathematical functions are defined:

Abs( x ) = (5‑1)

Asin( x ) the trigonometric inverse sine function, operating on an argument x that is  
in the range of −1.0 to 1.0, inclusive, with an output value in the range of   
−π÷2 to π÷2, inclusive, in units of radians (5‑2)

Atan( x ) the trigonometric inverse tangent function, operating on an argument x, with  
an output value in the range of −π÷2 to π÷2, inclusive, in units of radians (5‑3)

Atan2( y, x ) = (5‑4)

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

Clip1Y( x ) = Clip3( 0, ( 1 << BitDepthY ) − 1, x ) (5‑6)

Clip1C( x ) = Clip3( 0, ( 1 << BitDepthC ) − 1, x ) (5‑7)

Clip3( x, y, z ) = (5‑8)

ClipH( o, W, x ) = (5‑9)

Cos( x ) the trigonometric cosine function operating on an argument x in units of radians. (5‑10)

Floor( x ) the largest integer less than or equal to x. (5‑11)

GetCurrMsb( a, b, c, d ) = (5‑12)

Ln( x ) the natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...). (5‑13)

Log2( x ) the base-2 logarithm of x. (5‑14)

Log10( x ) the base-10 logarithm of x. (5‑15)

Min( x, y ) = (5‑16)

Max( x, y ) = (5‑17)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑18)

Sign( x ) = (5‑19)

Sin( x ) the trigonometric sine function operating on an argument x in units of radians (5‑20)

Sqrt( x ) = (5‑21)

Swap( x, y ) = ( y, x ) (5‑22)

Tan( x ) the trigonometric tangent function operating on an argument x in units of radians (5‑23)

## Order of operation precedence

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

– Operations of a higher precedence are evaluated before any operation of a lower precedence.

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

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

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

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

|  |
| --- |
| **operations (with operands x, y, and z)** |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| xy |
| "x \* y", "x / y", "x ÷ y", "", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x  <<  y", "x  >>  y" |
| "x < y", "x  <=  y", "x > y", "x  >=  y" |
| "x  = =  y", "x  !=  y" |
| "x & y" |
| "x | y" |
| "x  &&  y" |
| "x  | |  y" |
| "x ? y : z" |
| "x..y" |
| "x = y", "x  +=  y", "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), and one descriptor 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 clause 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 clause 7.2 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 clause 5.8) 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. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[ x ].

A specification of values of the entries in rows and columns of an array may be denoted by { {...} {...} }, where each inner pair of brackets specifies the values of the elements within a row in increasing column order and the rows are ordered in increasing row order. Thus, setting a matrix s equal to { { 1 6 } { 4 9 }} specifies that s[ 0 ][ 0 ] is set equal to 1, s[ 1 ][ 0 ] is set equal to 6, s[ 0 ][ 1 ] is set equal to 4, and s[ 1 ][ 1 ] is set equal to 9.

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 mathematically in the following form:

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 mathematically in the following form:

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 mathematically in the following form:

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 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.

# Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships

## Bitstream formats

This clause specifies the relationship between the network abstraction layer (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 Specification. The byte stream format is specified in Annex B.

## Source, decoded and output picture formats

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

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

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

– Luma (Y) only (monochrome).

– Luma and two chroma (YCbCr or YCgCo).

– Green, blue, and red (GBR, also known as RGB).

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

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

The variables SubWidthC and SubHeightC are specified in Table 6‑1, depending on the chroma format sampling structure, which is specified through chroma\_format\_idc and separate\_colour\_plane\_flag. 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 | 1 |
| 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 | 1 | 1 |

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 16, inclusive, and the number of bits used in the luma array may differ from the number of bits used in the chroma arrays.

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



Figure 6‑1 – Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

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



Figure 6‑2 – Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

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



Figure 6‑3 – Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture

## Partitioning of pictures, slices, tiles, bricks, and CTUs

### Partitioning of pictures into slices, tiles, and bricks

This subclause specifies how a picture is partitioned into slices, tiles, and bricks.

A picture is divided into one or more tile rows and one or more tile columns. A tile is a sequence of CTUs that covers a rectangular region of a picture.

A tile is divided into one or more bricks, each of which consisting of a number of CTU rows within the tile.

A tile that is not partitioned into multiple bricks is also referred to as a brick. However, a brick that is a true subset of a tile is not referred to as a tile.

A slice either contains a number of tiles of a picture or a number of bricks of a tile.

Two modes of slices are supported, namely the raster-scan slice mode and the rectangular slice mode. In the raster-scan slice mode, a slice contains a sequence of tiles in a tile raster scan of a picture. In the rectangular slice mode, a slice contains a number of bricks of a picture that collectively form a rectangular region of the picture. The bricks within a rectangular slice are in the order of brick raster scan of the slice.

Figure 6‑4 shows an example of raster-scan slice partitioning of a picture, where the picture is divided into 12 tiles and 3 raster-scan slices.

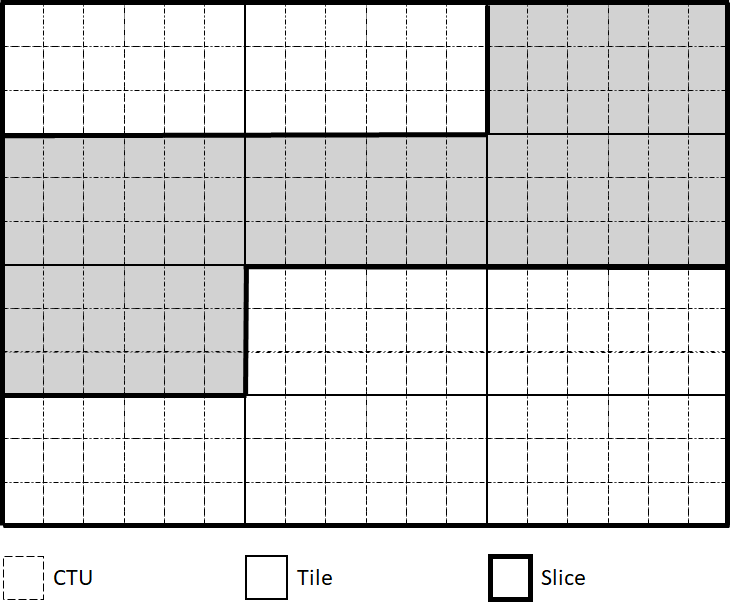


Figure 6‑4 – A picture with 18 by 12 luma CTUs that is partitioned into 12 tiles and 3 raster-scan slices (informative)

Figure 6‑5 shows an example of rectangular slice partitioning of a picture, where the picture is divided into 24 tiles (6 tile columns and 4 tile rows) and 9 rectangular slices.

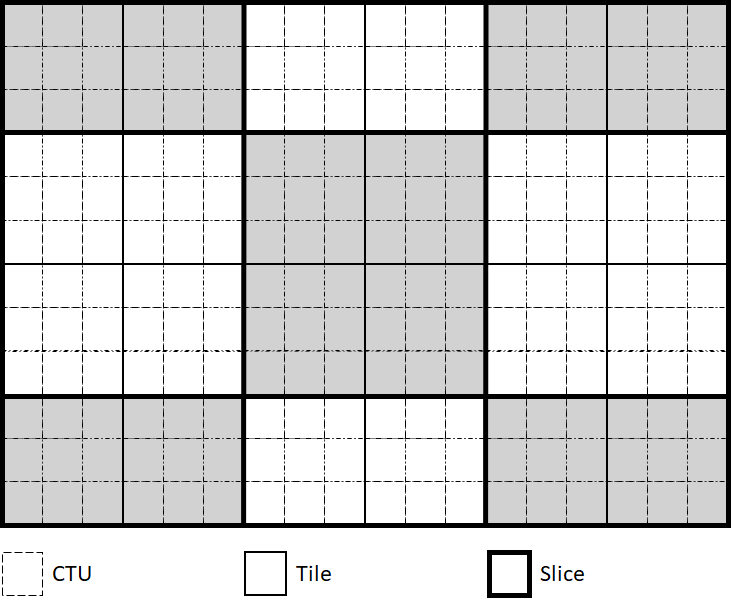


Figure 6‑5 – A picture with 18 by 12 luma CTUs that is partitioned into 24 tiles and 9 rectangular slices (informative)

Figure 6‑6 shows an example of a picture partitioned into tiles, bricks, and rectangular slices, where the picture is divided into 4 tiles (2 tile columns and 2 tile rows), 11 bricks (the top-left tile contains 1 brick, the top-right tile contains 5 bricks, the bottom-left tile contains 2 bricks, and the bottom-right tile contain 3 bricks), and 4 rectangular slices.



Figure 6‑6 – A picture that is partitioned into 4 tiles, 11 bricks, and 4 rectangular slices (informative)

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a slice contains only CTUs 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 a picture may be interleaved with each other under the constraint that for each value of colour\_plane\_id, the coded slice NAL units with that value of colour\_plane\_id shall be in the order of increasing CTU address in brick scan order for the first CTU of each coded slice NAL unit.

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

### Block, quadtree and multi-type tree structures

The samples are processed in units of CTBs. The array size for each luma CTB in both width and height is CtbSizeY in units of samples. The width and height of the array for each chroma CTB are CtbWidthC and CtbHeightC, respectively, in units of samples.

[Ed. (BB): Revise the following for QT+MTT.]

Each CTB 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 CTB. The quadtree is split until a leaf is reached, which is referred to as the quadtree leaf. When the component width is not an integer number of the CTB size, the CTBs at the right component boundary are incomplete. When the component height is not an integer multiple of the CTB size, the CTBs at the bottom component 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 grouped into "unit" structures as follows:

– One transform block (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three transform blocks (luma and chroma components of a picture in 4:2:0, 4:2:2 or 4:4:4 colour format) and the associated transform syntax structures units are associated with 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 transform units are associated with a coding unit.

– One CTB (monochrome picture or separate\_colour\_plane\_flag is equal to 1) or three CTBs (luma and chroma), the associated coding tree syntax structures and the associated coding units are associated with a CTU.

### Spatial or component-wise partitionings

The following divisions of processing elements of this Specification form spatial or component-wise partitioning:

– The division of each picture into components

– The division of each component into CTBs

– The division of each picture into tile columns

– The division of each picture into tile rows

– The division of each tile column into tiles

– The division of each tile row into tiles

– The division of each tile into bricks

– The division of each tile into CTUs

– The division of each brick into CTUs

– The division of each picture into slices

– The division of each slice into bricks

– The division of each slice into CTUs

– The division of each CTU into CTBs

– The division of each CTB into coding blocks, except that the CTBs are incomplete at the right component boundary when the component width is not an integer multiple of the CTB size and the CTBs are incomplete at the bottom component boundary when the component height is not an integer multiple of the CTB size

– The division of each CTU into coding units, except that the CTUs are incomplete at the right picture boundary when the picture width in luma samples is not an integer multiple of the luma CTB size and the CTUs are incomplete at the bottom picture boundary when the picture height in luma samples is not an integer multiple of the luma CTB size

– The division of each coding unit into transform units

– The division of each coding unit into coding blocks

– The division of each coding block into transform blocks

– The division of each transform unit into transform blocks.

## Availability processes

[Ed. (BB): Define appropriate availability checking process.]

### Allowed quad split process

Inputs to this process are:

* a coding block size cbSize in luma samples,
* a multi-type tree depth mttDepth,
* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed.

Output of this process is the variable allowSplitQt.

The variable allowSplitQt is derived as follows:

* If one or more of the following conditions are true, allowSplitQt is set equal to FALSE:
* treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA and cbSize is less than or equal to MinQtSizeY
* treeType is equal to DUAL\_TREE\_CHROMA and cbSize / SubWidthC is less than or equal to MinQtSizeC
* mttDepth is not equal to 0
* treeType is equal to DUAL\_TREE\_CHROMA and ( cbSize / SubWidthC ) is less than or equal to 4 [Ed. (SL): is “less than or” needed here?]
* Otherwise, allowSplitQt is set equal to TRUE.

### Allowed binary split process

Inputs to this process are:

* a binary split mode btSplit,
* a coding block width cbWidth in luma samples,
* a coding block height cbHeight in luma samples,
* a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture,
* a multi-type tree depth mttDepth,
* a maximum multi-type tree depth with offset maxMttDepth,
* a maximum binary tree size maxBtSize,
* a minimium quadtree size minQtSize,
* a partition index partIdx,
* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed.

Output of this process is the variable allowBtSplit.

Table 6‑2 – Specification of parallelTtSplit and cbSize based on btSplit.

|  |  |  |
| --- | --- | --- |
|  | **btSplit = = SPLIT\_BT\_VER** | **btSplit = = SPLIT\_BT\_HOR** |
| **parallelTtSplit** | SPLIT\_TT\_VER | SPLIT\_TT\_HOR |
| **cbSize** | cbWidth | cbHeight |

The variables parallelTtSplit and cbSize are derived as specified in  Table 6‑2.

The variable allowBtSplit is derived as follows:

* If one or more of the following conditions are true, allowBtSplit is set equal to FALSE:
* cbSize is less than or equal to MinBtSizeY
* cbWidth is greater than maxBtSize
* cbHeight is greater than maxBtSize
* mttDepth is greater than or equal to maxMttDepth
* treeType is equal to DUAL\_TREE\_CHROMA and ( cbWidth / SubWidthC ) \* ( cbHeight / SubHeightC ) is less than or equal to 16 [Ed. (SL): is “less than or” needed here?]
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_VER
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_VER
* cbHeight is greater than MaxTbSizeY
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_HOR
* cbWidth is greater than MaxTbSizeY
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples
* cbWidth is greater than minQtSize
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_HOR
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* y0 + cbHeight is less than or equal to pic\_height\_in\_luma\_samples
* Otherwise, if all of the following conditions are true, allowBtSplit is set equal to FALSE:
* mttDepth is greater than 0
* partIdx is equal to 1
* MttSplitMode[ x0 ][ y0 ][ mttDepth − 1 ] is equal to parallelTtSplit
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_VER
* cbWidth is less than or equal to MaxTbSizeY
* cbHeight is greater than MaxTbSizeY
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE
* btSplit is equal to SPLIT\_BT\_HOR
* cbWidth is greater than MaxTbSizeY
* cbHeight is less than or equal to MaxTbSizeY

– Otherwise, allowBtSplit is set equal to TRUE.

### Allowed ternary split process

Inputs to this process are:

* a ternary split mode ttSplit,
* a coding block width cbWidth in luma samples,
* a coding block height cbHeight in luma samples,
* a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture,
* a multi-type tree depth mttDepth
* a maximum multi-type tree depth with offset maxMttDepth,
* a maximum ternary tree size maxTtSize,
* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed.

Output of this process is the variable allowTtSplit.

Table 6‑3 – Specification of cbSize based on ttSplit.

|  |  |  |
| --- | --- | --- |
|  | **ttSplit = = SPLIT\_TT\_VER** | **ttSplit = = SPLIT\_TT\_HOR** |
| **cbSize** | cbWidth | cbHeight |

The variable cbSize is derived as specified in Table 6‑3.

The variable allowTtSplit is derived as follows:

* If one or more of the following conditions are true, allowTtSplit is set equal to FALSE:
* cbSize is less than or equal to 2 \* MinTtSizeY
* cbWidth is greater than Min( MaxTbSizeY, maxTtSize )
* cbHeight is greater than Min( MaxTbSizeY, maxTtSize )
* mttDepth is greater than or equal to maxMttDepth
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples
* treeType is equal to DUAL\_TREE\_CHROMA and ( cbWidth / SubWidthC ) \* ( cbHeight / SubHeightC ) is less than or equal to 32

– Otherwise, allowTtSplit is set equal to TRUE.

### Derivation process for neighbouring block availability

Inputs to this process are:

* the luma location ( xCurr, yCurr ) of the top-left sample of the current block relative to the top-left luma sample of the current picture,
* the luma location ( xNbY, yNbY ) covered by a neighbouring block relative to the top-left luma sample of the current picture.

Output of this process is the availability of the neighbouring block covering the location ( xNbY, yNbY ), denoted as availableN.

The neighbouring block availability availableN is derived as follows:

– If entropy\_coding\_sync\_enabled\_flag is equal to 1 and xNbY  >>  Log2CTBSize is greater than or equal to ( xCurr  >>  Log2CTBSize ) + 1, availableN is set equal to FALSE.

– Otherwise, if the neighbouring block is contained in a different brick than the current block, availableN is set equal to FALSE. [Ed. (YK): The case outside of a slice boundary is covered by this sentence as each slice always contains an integer number of complete bricks.]

– Otherwise, [Ed. (CJ): the derivation process of availableN is to be specified.]

## Scanning processes

### CTB raster scanning, tile scanning, and brick scanning process

The list colWidth[ i ] for i ranging from 0 to num\_tile\_columns\_minus1, inclusive, specifying the width of the i-th tile column in units of CTBs, is derived, and when uniform\_tile\_spacing\_flag is equal to 1, the value of num\_tile\_columns\_minus1 is inferred, as follows:

if( uniform\_tile\_spacing\_flag ) {  
 remainingWidthInCtbsY = PicWidthInCtbsY  
 i = 0  
 while( remainingWidthInCtbsY > ( tile\_cols\_width\_minus1 + 1 ) ) {  
 colWidth[ i++ ] = tile\_cols\_width\_minus1 + 1  
 remainingWidthInCtbsY −= ( tile\_cols\_width\_minus1 + 1 )  
 }  
 colWidth[ i ] = remainingWidthInCtbsY  
 num\_tile\_columns\_minus1 = i  
} else {  
 colWidth[ num\_tile\_columns\_minus1 ] = PicWidthInCtbsY (6‑1)  
 for( i = 0; i < num\_tile\_columns\_minus1; i++ ) {  
 colWidth[ i ] = tile\_column\_width\_minus1[ i ] + 1  
 colWidth[ num\_tile\_columns\_minus1 ] −= colWidth[ i ]  
 }  
}

The list RowHeight[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, specifying the height of the j-th tile row in units of CTBs, is derived, and when uniform\_tile\_spacing\_flag is equal to 1, the value of num\_tile\_rows\_minus1 is inferred, as follows:

if( uniform\_tile\_spacing\_flag ) {  
 remainingHeightInCtbsY = PicHeightInCtbsY  
 i = 0  
 while( remainingHeightInCtbsY > ( tile\_rows\_height\_minus1 + 1 ) ) {  
 RowHeight[ i++ ] = tile\_rows\_height\_minus1 + 1  
 remainingHeightInCtbsY −= ( tile\_rows\_height\_minus1 + 1 )  
 }  
 RowHeight[ i ] = remainingHeightInCtbsY  
 num\_tile\_rows\_minus1 = i  
} else {  
 RowHeight[ num\_tile\_rows\_minus1 ] = PicHeightInCtbsY (6‑2)  
 for( j = 0; j < num\_tile\_rows\_minus1; j++ ) {  
 RowHeight[ j ] = tile\_row\_height\_minus1[ j ] + 1  
 RowHeight[ num\_tile\_rows\_minus1 ] −= RowHeight[ j ]  
 }  
}

The list tileColBd[ i ] for i ranging from 0 to num\_tile\_columns\_minus1 + 1, inclusive, specifying the location of the i-th tile column boundary in units of CTBs, is derived as follows:

for( tileColBd[ 0 ] = 0, i = 0; i <= num\_tile\_columns\_minus1; i++ )  
 tileColBd[ i + 1 ] = tileColBd[ i ] + colWidth[ i ] (6‑3)

The list tileRowBd[ j ] for j ranging from 0 to num\_tile\_rows\_minus1 + 1, inclusive, specifying the location of the j-th tile row boundary in units of CTBs, is derived as follows:

for( tileRowBd[ 0 ] = 0, j = 0; j <= num\_tile\_rows\_minus1; j++ )  
 tileRowBd[ j + 1 ] = tileRowBd[ j ] + RowHeight[ j ] (6‑4)

The variable NumBricksInPic, specifying the number of bricks in a picture referring to the PPS, and the lists BrickColBd[ brickIdx ], BrickRowBd[ brickIdx ], BrickWidth[ brickIdx ], and BrickHeight[ brickIdx ] for brickIdx ranging from 0 to NumBricksInPic − 1, inclusive, specifying the locations of the vertical brick boundaries in units of CTBs, the locations of the horizontal brick boundaries in units of CTBs, the widths of the bricks in units of CTBs, and the heights of bricks in units of CTBs, are derived, and for each i ranging from 0 to NumTilesInPic − 1, inclusive, when uniform\_brick\_spacing\_flag[ i ] is equal to 1, the value of num\_brick\_rows\_minus1[ i ] is inferred, as follows:

for ( brickIdx = 0, i = 0; i < NumTilesInPic; i++ ) {  
 tileX = i % ( num\_tile\_columns\_minus1 + 1 )  
 tileY = i / ( num\_tile\_columns\_minus1 + 1 )  
 if( !brick\_split\_flag[ i ] ) {  
 BrickColBd[ brickIdx ] = tileColBd[ tileX ]  
 BrickRowBd[ brickIdx ] = tileRowBd[ tileY ]  
 BrickWidth[ brickIdx ] = colWidth[ tileX ]  
 BrickHeight[ brickIdx ] = RowHeight[ tileY ] (6‑5)  
 brickIdx++  
 } else {  
 if( uniform\_brick\_spacing\_flag[ i ] ) {  
 remainingHeightInCtbsY = RowHeight[ tileY ]  
 j = 0  
 while( remainingHeightInCtbsY > ( brick\_height\_minus1[ i ] + 1 ) ) {  
 rowHeight2[ j++ ] = brick\_height\_minus1[ i ] + 1  
 remainingHeightInCtbsY −= ( brick\_height\_minus1[ i ] + 1 )  
 }  
 rowHeight2[ j ] = remainingHeightInCtbsY  
 num\_brick\_rows\_minus1[ i ] = j  
 } else {  
 rowHeight2[ num\_brick\_rows\_minus1[ i ] ] = RowHeight[ tileY ]  
 for( j = 0; j < num\_brick\_rows\_minus1[ i ]; j++ ) {  
 rowHeight2[ j ] = brick\_row\_height\_minus1[ i ][ j ]+ 1  
 rowHeight2[ num\_brick\_rows\_minus1[ i ] ] −= rowHeight2[ j ]  
 }  
 }  
 for( rowBd2[ 0 ] = 0, j = 0; j <= num\_brick\_rows\_minus1[ i ]; j++ )  
 rowBd2[ j + 1 ] = rowBd2[ j ] + rowHeight2[ j ]  
 for( j = 0; j < num\_brick\_rows\_minus1[ i ] + 1; j++ ) {  
 BrickColBd[ brickIdx ] = tileColBd[ tileX ]  
 BrickRowBd[ brickIdx ] = tileRowBd[ tileY ] + rowBd2[ j ]  
 BrickWidth[ brickIdx ] = colWidth[ tileX ]  
 BrickHeight[ brickIdx ] = rowHeight2[ j ]  
 brickIdx++  
 }  
 }  
}  
NumBricksInPic = brickIdx

The list CtbAddrRsToBs[ ctbAddrRs ] for ctbAddrRs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in CTB raster scan of a picture to a CTB address in brick scan, is derived as follows:

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ ) {  
 tbX = ctbAddrRs % PicWidthInCtbsY  
 tbY = ctbAddrRs / PicWidthInCtbsY  
 brickFound = FALSE  
 for( bkIdx = NumBricksInPic − 1, i = 0; i < NumBricksInPic − 1 && !brickFound; i++ ) {  
 brickFound = tbX < ( BrickColBd[ i ] + BrickWidth[ i ] ) &&  
 tbY < ( BrickRowBd[ i ] + BrickHeight[ i ] )  
 if( brickFound ) (6‑6)  
 bkIdx = i  
 }  
 CtbAddrRsToBs[ ctbAddrRs ] = 0  
 for( i = 0; i < bkIdx; i++ )  
 CtbAddrRsToBs[ ctbAddrRs ] += BrickHeight[ i ] \* BrickWidth[ i ]  
 CtbAddrRsToBs[ ctbAddrRs ] +=  
 ( tbY − BrickRowBd[ bkIdx ] ) \* BrickWidth[ bkIdx ] + tbX − BrickColBd[ bkIdx ]  
}

The list CtbAddrBsToRs[ ctbAddrBs ] for ctbAddrBs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in brick scan to a CTB address in CTB raster scan of a picture, is derived as follows:

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ ) (6‑7)  
 CtbAddrBsToRs[ CtbAddrRsToBs[ ctbAddrRs ] ] = ctbAddrRs

The list BrickId[ ctbAddrBs ] for ctbAddrBs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in brick scan to a brick ID, is derived as follows:

for( i = 0, i < NumBricksInPic; i++ )  
 for( y = BrickRowBd[ i ]; y < BrickRowBd[ i ] + BrickHeight[ i ]; y++ ) (6‑8)  
 for( x = BrickColBd[ i ]; x < BrickColBd[ i ] + BrickWidth[ i ]; x++ )  
 BrickId[ CtbAddrRsToBs[ y \* PicWidthInCtbsY+ x ] ] = i

The list NumCtusInBrick[ brickIdx ] for brickIdx ranging from 0 to NumBricksInPic − 1, inclusive, specifying the conversion from a brick index to the number of CTUs in the brick, is derived as follows:

for( i = 0; i < NumBricksInPic; i++ ) (6‑9)  
 NumCtusInBrick[ i ] = BrickWidth[ i ] \* BrickHeight[ i ]

The list FirstCtbAddrBs[ brickIdx ] for brickIdx ranging from 0 to NumBricksInPic − 1, inclusive, specifying the conversion from a brick ID to the CTB address in brick scan of the first CTB in the brick are derived as follows:

for( ctbAddrBs = 0, brickIdx = 0, brickStartFlag = 1; ctbAddrBs < PicSizeInCtbsY; ctbAddrBs++ ) {  
 if( brickStartFlag ) {  
 FirstCtbAddrBs[ brickIdx ] = ctbAddrBs (6‑10)  
 brickStartFlag = 0  
 }  
 brickEndFlag = ( ctbAddrBs = = PicSizeInCtbsY − 1 ) | |  
 ( BrickId[ ctbAddrBs + 1 ] != BrickId[ ctbAddrBs ] )  
 if( brickEndFlag ) {  
 brickIdx++  
 brickStartFlag = 1  
 }  
}

### Up-right diagonal and raster scan order array initialization process

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

Output of this process are the arrays diagScan[ sPos ][ sComp ] and rasterScan[ sPos ]. 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 value of blkWidth and blkHeight, the array diagScan is derived as follows:

i = 0  
x = 0  
y = 0  
stopLoop = FALSE  
while( !stopLoop ) {  
 while( y >= 0 ) {  
 if( x < blkWidth && y < blkHeight ) { (6‑11)  
 diagScan[ i ][ 0 ] = x  
 diagScan[ i ][ 1 ] = y  
 rasterScan[ x \* blkWidth + y] = i  
 i++  
 }  
 y− −  
 x++  
 }  
 y = x  
 x = 0  
 if( i >= blkWidth \* blkHeight )  
 stopLoop = TRUE  
}

# 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 some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this Specification.

The following table lists examples of the syntax specification format. 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.

|  |  |
| --- | --- |
|  | Descriptor |
| /\* A statement can be a syntax element with an associated 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** | ue(k) |
| 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 are expressed in terms of the value of a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte\_aligned( ) is specified as follows:

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

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

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

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

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

more\_data\_in\_payload( ) is specified as follows:

– If byte\_aligned( ) is equal to TRUE and the current position in the sei\_payload( ) syntax structure is 8 \* payloadSize bits from the beginning of the sei\_payload( ) syntax structure, the return value of more\_data\_in\_payload( ) is equal to FALSE.

– Otherwise, the return value of more\_data\_in\_payload( ) is equal to TRUE.

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

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

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

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

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

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

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

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

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

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

payload\_extension\_present( ) is specified as follows:

– If the current position in the sei\_payload( ) syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 \* payloadSize bits from the beginning of the syntax structure (i.e., the position of the payload\_bit\_equal\_to\_one syntax element), the return value of payload\_extension\_present( ) is equal to TRUE.

– Otherwise, the return value of payload\_extension\_present( ) is equal to FALSE.

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 clause 9.5.

– 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 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

– st(v): null-terminated string encoded as universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646. The parsing process is specified as follows: st(v) begins at a byte-aligned position in the bitstream and reads and returns a series of bytes from the bitstream, beginning at the current position and continuing up to but not including the next byte-aligned byte that is equal to 0x00, and advances the bitstream pointer by ( stringLength + 1 ) \* 8 bit positions, where stringLength is equal to the number of bytes returned.

NOTE – The st(v) syntax descriptor is only used in this Specification when the current position in the bitstream is a byte-aligned position.

– tb(v): truncated binary using up to maxVal bits with maxVal defined in the semantics of the symtax element. The parsing process for this descriptor is specified in clause 9.4.

– tu(v): truncated unary using up to maxVal bits with maxVal defined in the semantics of the symtax element. The parsing process for this descriptor is specified in clause 9.3.

– 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 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

– uek(v): unsigned integer k-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k defined in the semantics of the symtax element.

## Syntax in tabular form

### NAL unit syntax

#### General NAL unit syntax

|  |  |
| --- | --- |
| nal\_unit( NumBytesInNalUnit ) { | Descriptor |
| nal\_unit\_header( ) |  |
| 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) |
| } |  |

#### NAL unit header syntax

|  |  |
| --- | --- |
| nal\_unit\_header( ) { | Descriptor |
| **zero\_tid\_required\_flag** | u(1) |
| **nuh\_temporal\_id\_plus1** | u(3) |
| **nal\_unit\_type\_lsb** | u(4) |
| **nuh\_layer\_id\_plus1** | u(7) |
| **nuh\_reserved\_zero\_bit** | u(1) |
| } |  |

[Ed. (KS/YK): Changing nuh\_layer\_id to nuh\_layer\_id\_plus is a bug fix that was made during the editing period of JVET-N1001. It'd be good to have a discussion at a next JVET meeting on whether this is the best way to fix that bug: the last byte in the NAL unit header should be be equal to 0x00, which may lead to a start code emulation, because the NAL unit header is not included in the start code emulation prevention.]

### Raw byte sequence payloads, trailing bits and byte alignment syntax

#### Decoding parameter set syntax

|  |  |
| --- | --- |
| decoding\_parameter\_set\_rbsp( ) { | **Descriptor** |
| **dps\_decoding\_parameter\_set\_id** | u(4) |
| **dps\_max\_sub\_layers\_minus1** | u(3) |
| **dps\_reserved\_zero\_bit** | u(1) |
| profile\_tier\_level( dps\_max\_sub\_layers\_minus1 ) |  |
| **dps\_extension\_flag** | u(1) |
| if( dps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **dps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Video parameter set syntax

|  |  |
| --- | --- |
| video\_parameter\_set\_rbsp( ) { | Descriptor |
| **vps\_video\_parameter\_set\_id** | u(4) |
| **vps\_max\_layers\_minus1** | u(8) |
| for( i = 0; i <= vps\_max\_layers\_minus1; i++ ) { |  |
| **vps\_included\_layer\_id**[ i ] | u(7) |
| **vps\_reserved\_zero\_bit** | u(1) |
| } |  |
| **vps\_constraint\_info\_present\_flag** | u(1) |
| **vps\_reserved\_zero\_7bits** | u(7) |
| if( vps\_constraint\_info\_present\_flag ) |  |
| general\_constraint\_info( ) |  |
| **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 |
| **sps\_decoding\_parameter\_set\_id** | u(4) |
| **sps\_video\_parameter\_set\_id** | u(4) |
| **sps\_max\_sub\_layers\_minus1** | u(3) |
| **sps\_reserved\_zero\_5bits** | u(5) |
| profile\_tier\_level( sps\_max\_sub\_layers\_minus1 ) |  |
| **gra\_enabled\_flag** | u(1) |
| **sps\_seq\_parameter\_set\_id** | ue(v) |
| **chroma\_format\_idc** | ue(v) |
| if( chroma\_format\_idc = = 3 ) |  |
| **separate\_colour\_plane\_flag** | u(1) |
| **pic\_width\_in\_luma\_samples** | ue(v) |
| **pic\_height\_in\_luma\_samples** | ue(v) |
| **conformance\_window\_flag** | u(1) |
| if( conformance\_window\_flag ) { |  |
| **conf\_win\_left\_offset** | ue(v) |
| **conf\_win\_right\_offset** | ue(v) |
| **conf\_win\_top\_offset** | ue(v) |
| **conf\_win\_bottom\_offset** | ue(v) |
| } |  |
| **bit\_depth\_luma\_minus8** | ue(v) |
| **bit\_depth\_chroma\_minus8** | ue(v) |
| **log2\_max\_pic\_order\_cnt\_lsb\_minus4** | ue(v) |
| **sps\_sub\_layer\_ordering\_info\_present\_flag** | u(1) |
| for( i = ( sps\_sub\_layer\_ordering\_info\_present\_flag ? 0 : sps\_max\_sub\_layers\_minus1 );  i <= sps\_max\_sub\_layers\_minus1; i++ ) { |  |
| **sps\_max\_dec\_pic\_buffering\_minus1**[ i ] | ue(v) |
| **sps\_max\_num\_reorder\_pics**[ i ] | ue(v) |
| **sps\_max\_latency\_increase\_plus1**[ i ] | ue(v) |
| } |  |
| **long\_term\_ref\_pics\_flag** | u(1) |
| **sps\_idr\_rpl\_present\_flag** | u(1) |
| **rpl1\_same\_as\_rpl0\_flag** | u(1) |
| for( i = 0; i < !rpl1\_same\_as\_rpl0\_flag ? 2 : 1; i++ ) { |  |
| **num\_ref\_pic\_lists\_in\_sps**[ i ] | ue(v) |
| for( j = 0; j < num\_ref\_pic\_lists\_in\_sps[ i ]; j++) |  |
| ref\_pic\_list\_struct( i, j ) |  |
| } |  |
| if( ChromaArrayType != 0 ) |  |
| **qtbtt\_dual\_tree\_intra\_flag** | u(1) |
| **log2\_ctu\_size\_minus2** | ue(v) |
| **log2\_min\_luma\_coding\_block\_size\_minus2** | ue(v) |
| **partition\_constraints\_override\_enabled\_flag** | u(1) |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma** | ue(v) |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_inter\_slice** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_luma** | ue(v) |
| if( sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_luma != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma** | ue(v) |
| } |  |
| if( sps\_max\_mtt\_hierarchy\_depth\_inter\_slices != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice** | ue(v) |
| } |  |
| if( qtbtt\_dual\_tree\_intra\_flag ) { |  |
| **sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma** | ue(v) |
| **sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_chroma** | ue(v) |
| if ( sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_chroma != 0 ) { |  |
| **sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma** | ue(v) |
| **sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma** | ue(v) |
| } |  |
| } |  |
| **sps\_sao\_enabled\_flag** | u(1) |
| **sps\_alf\_enabled\_flag** | u(1) |
| **sps\_pcm\_enabled\_flag** | u(1) |
| if( sps\_pcm\_enabled\_flag ) { |  |
| **pcm\_sample\_bit\_depth\_luma\_minus1** | u(4) |
| **pcm\_sample\_bit\_depth\_chroma\_minus1** | u(4) |
| **log2\_min\_pcm\_luma\_coding\_block\_size\_minus3** | ue(v) |
| **log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size** | ue(v) |
| **pcm\_loop\_filter\_disabled\_flag** | u(1) |
| } |  |
| if( ( CtbSizeY / MinCbSizeY + 1) <= ( pic\_width\_in\_luma\_samples / MinCbSizeY − 1 ) ) { |  |
| **sps\_ref\_wraparound\_enabled\_flag** | u(1) |
| if( sps\_ref\_wraparound\_enabled\_flag ) |  |
| **sps\_ref\_wraparound\_offset\_minus1** | ue(v) |
| } |  |
| **sps\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **sps\_sbtmvp\_enabled\_flag** | u(1) |
| **sps\_amvr\_enabled\_flag** | u(1) |
| **sps\_bdof\_enabled\_flag** | u(1) |
| **sps\_smvd\_enabled\_flag** | u(1) |
| **sps\_affine\_amvr\_enabled\_flag** | u(1) |
| **sps\_dmvr\_enabled\_flag** | u(1) |
| **sps\_mmvd\_enabled\_flag** | u(1) |
| **sps\_isp\_enabled\_flag** | u(1) |
| **sps\_mrl\_enabled\_flag** | u(1) |
| **sps\_mip\_enabled\_flag** | u(1) |
| if( ChromaArrayType != 0 ) |  |
| **sps\_cclm\_enabled\_flag** | u(1) |
| if( sps\_cclm\_enabled\_flag && chroma\_format\_idc = = 1 ) [Ed. (JC): should sps\_cclm\_colocated\_chroma\_flag also be signalled for 422 case since it’s used in the decoding process, to be confirmed] |  |
| **sps\_cclm\_colocated\_chroma\_flag** | u(1) |
| **sps\_mts\_enabled\_flag** | u(1) |
| if( sps\_mts\_enabled\_flag ) { |  |
| **sps\_explicit\_mts\_intra\_enabled\_flag** | u(1) |
| **sps\_explicit\_mts\_inter\_enabled\_flag** | u(1) |
| } |  |
| **sps\_sbt\_enabled\_flag** | u(1) |
| if( sps\_sbt\_enabled\_flag ) |  |
| **sps\_sbt\_max\_size\_64\_flag** | u(1) |
| **sps\_affine\_enabled\_flag** | u(1) |
| if( sps\_affine\_enabled\_flag ) |  |
| **sps\_affine\_type\_flag** | u(1) |
| **sps\_bcw\_enabled\_flag** | u(1) |
| **sps\_ibc\_enabled\_flag** | u(1) |
| **sps\_ciip\_enabled\_flag** | u(1) |
| if( sps\_mmvd\_enabled\_flag ) |  |
| **sps\_fpel\_mmvd\_enabled\_flag** | u(1) |
| **sps\_triangle\_enabled\_flag** | u(1) |
| **sps\_lmcs\_enabled\_flag** | u(1) |
| **sps\_lfnst\_enabled\_flag** | u(1) |
| **sps\_ladf\_enabled\_flag** | u(1) |
| if ( sps\_ladf\_enabled\_flag ) { |  |
| **sps\_num\_ladf\_intervals\_minus2** | u(2) |
| **sps\_ladf\_lowest\_interval\_qp\_offset** | se(v) |
| for( i = 0; i < sps\_num\_ladf\_intervals\_minus2 + 1; i++ ) { |  |
| **sps\_ladf\_qp\_offset**[ i ] | se(v) |
| **sps\_ladf\_delta\_threshold\_minus1**[ i ] | ue(v) |
| } |  |
| } |  |
| **scaling\_list\_enabled\_flag** | u(1) |
| if( scaling\_list\_enabled\_flag ) { |  |
| **sps\_scaling\_list\_data\_present\_flag** | u(1) |
| if( sps\_scaling\_list\_data\_present\_flag ) |  |
| scaling\_list\_data( ) |  |
| } |  |
| **timing\_info\_present\_flag** | u(1) |
| if( timing\_info\_present\_flag ) { |  |
| **num\_units\_in\_tick** | u(32) |
| **time\_scale** | u(32) |
| **hrd\_parameters\_present\_flag** | u(1) |
| if( hrd\_parameters\_present\_flag ) |  |
| hrd\_parameters( sps\_max\_sub\_layers\_minus1 ) |  |
| } |  |
| **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 |
| **pps\_pic\_parameter\_set\_id** | ue(v) |
| **pps\_seq\_parameter\_set\_id** | ue(v) |
| **output\_flag\_present\_flag** | u(1) |
| **single\_tile\_in\_pic\_flag** | u(1) |
| if( !single\_tile\_in\_pic\_flag ) { |  |
| **uniform\_tile\_spacing\_flag** | u(1) |
| if( uniform\_tile\_spacing\_flag ) { |  |
| **tile\_cols\_width\_minus1** | ue(v) |
| **tile\_rows\_height\_minus1** | ue(v) |
| } else { |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| for( i = 0; i < num\_tile\_columns\_minus1; i++ ) |  |
| **tile\_column\_width\_minus1**[ i ] | ue(v) |
| for( i = 0; i < num\_tile\_rows\_minus1; i++ ) |  |
| **tile\_row\_height\_minus1**[ i ] | ue(v) |
| } |  |
| **brick\_splitting\_present\_flag** | u(1) |
| for( i = 0; brick\_splitting\_present\_flag && i < NumTilesInPic; i++ ) { |  |
| **brick\_split\_flag**[ i ] | u(1) |
| if( brick\_split\_flag[ i ] ) { |  |
| **uniform\_brick\_spacing\_flag**[ i ] | u(1) |
| if( uniform\_brick\_spacing\_flag[ i ] ) |  |
| **brick\_height\_minus1**[ i ] | ue(v) |
| else { |  |
| **num\_brick\_rows\_minus1**[ i ] | ue(v) |
| for( j = 0; j < num\_brick\_rows\_minus1[ i ]; j++ ) |  |
| **brick\_row\_height\_minus1**[ i ][ j ] | ue(v) |
| } |  |
| } |  |
| } |  |
| **single\_brick\_per\_slice\_flag** | u(1) |
| if( !single\_brick\_per\_slice\_flag ) |  |
| **rect\_slice\_flag** | u(1) |
| if( rect\_slice\_flag && !single\_brick\_per\_slice\_flag ) { |  |
| **num\_slices\_in\_pic\_minus1** | ue(v) |
| for( i = 0; i <= num\_slices\_in\_pic\_minus1; i++ ) { |  |
| if( i > 0 ) |  |
| **top\_left\_brick\_idx**[ i ] | u(v) |
| **bottom\_right\_brick\_idx\_delta**[ i ] | u(v) |
| } |  |
| } |  |
| **loop\_filter\_across\_bricks\_enabled\_flag** | u(1) |
| if( loop\_filter\_across\_bricks\_enabled\_flag ) |  |
| **loop\_filter\_across\_slices\_enabled\_flag** | u(1) |
| } |  |
| if( rect\_slice\_flag ) { |  |
| **signalled\_slice\_id\_flag** | u(1) |
| if( signalled\_slice\_id\_flag ) { |  |
| **signalled\_slice\_id\_length\_minus1** | ue(v) |
| for( i = 0; i <= num\_slices\_in\_pic\_minus1; i++ ) |  |
| **slice\_id**[ i ] | u(v) |
| } |  |
| } |  |
| **entropy\_coding\_sync\_enabled\_flag** | u(1) |
| **cabac\_init\_present\_flag** | u(1) |
| for( i = 0; i < 2; i++ ) |  |
| **num\_ref\_idx\_default\_active\_minus1**[ i ] | ue(v) |
| **rpl1\_idx\_present\_flag** | u(1) |
| **init\_qp\_minus26** | se(v) |
| **transform\_skip\_enabled\_flag** | u(1) |
| if( transform\_skip\_enabled\_flag ) |  |
| **log2\_transform\_skip\_max\_size\_minus2** | ue(v) |
| **cu\_qp\_delta\_enabled\_flag** | u(1) |
| if( cu\_qp\_delta\_enabled\_flag ) |  |
| **cu\_qp\_delta\_subdiv** | ue(v) |
| **pps\_cb\_qp\_offset** | se(v) |
| **pps\_cr\_qp\_offset** | se(v) |
| **pps\_joint\_cbcr\_qp\_offset** | se(v) |
| **pps\_slice\_chroma\_qp\_offsets\_present\_flag** | u(1) |
| **weighted\_pred\_flag** | u(1) |
| **weighted\_bipred\_flag** | u(1) |
| **deblocking\_filter\_control\_present\_flag** | u(1) |
| if( deblocking\_filter\_control\_present\_flag ) { |  |
| **deblocking\_filter\_override\_enabled\_flag** | u(1) |
| **pps\_deblocking\_filter\_disabled\_flag** | u(1) |
| if( !pps\_deblocking\_filter\_disabled\_flag ) { |  |
| **pps\_beta\_offset\_div2** | se(v) |
| **pps\_tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| **pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag** | u(1) |
| if( pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag ) { |  |
| **pps\_num\_ver\_virtual\_boundaries** | u(2) |
| for( i = 0; i < pps\_num\_ver\_virtual\_boundaries; i++ ) |  |
| **pps\_virtual\_boundaries\_pos\_x**[ i ] | u(v) |
| **pps\_num\_hor\_virtual\_boundaries** | u(2) |
| for( i = 0; i < pps\_num\_hor\_virtual\_boundaries; i++ ) |  |
| **pps\_virtual\_boundaries\_pos\_y**[ i ] | u(v) |
| } |  |
| **pps\_scaling\_list\_data\_present\_flag** | u(1) |
| if( pps\_scaling\_list\_data\_present\_flag ) |  |
| scaling\_list\_data( ) |  |
| **pps\_extension\_flag** | u(1) |
| if( pps\_extension\_flag ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **pps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Adaptation parameter set syntax

|  |  |
| --- | --- |
| adaptation\_parameter\_set\_rbsp( ) { | Descriptor |
| **adaptation\_parameter\_set\_id** | u(5) |
| **aps\_params\_type** | u(3) |
| if( aps\_params\_type = = ALF\_APS ) |  |
| alf\_data( adaptation\_parameter\_set\_id ) |  |
| else if( aps\_params\_type = = LMCS\_APS ) |  |
| lmcs\_data( ) |  |
| **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( ) |  |
| } |  |

#### Access unit delimiter RBSP syntax

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

#### End of sequence RBSP syntax

|  |  |
| --- | --- |
| end\_of\_seq\_rbsp( ) { | Descriptor |
| } |  |

#### End of bitstream RBSP syntax

|  |  |
| --- | --- |
| end\_of\_bitstream\_rbsp( ) { | Descriptor |
| } |  |

#### Slice layer RBSP syntax

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

#### RBSP slice trailing bits syntax

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

#### RBSP trailing bits syntax

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

#### Byte alignment syntax

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

#### Scaling list data syntax

|  |  |
| --- | --- |
| scaling\_list\_data( ) { | **Descriptor** |
| for( sizeId = 1; sizeId < 7; sizeId++ ) |  |
| for( matrixId = 0; matrixId < 6; matrixId ++ ) { |  |
| if( ! ( ( ( sizeId = = 1 ) && ( matrixId % 3 = = 0 ) ) | |   ( ( sizeId = = 6 ) && ( matrixId % 3 != 0 ) ) ) ) { |  |
| **scaling\_list\_pred\_mode\_flag**[ sizeId ][ matrixId ] | u(1) |
| if( !scaling\_list\_pred\_mode\_flag[ sizeId ][ matrixId ] ) |  |
| **scaling\_list\_pred\_matrix\_id\_delta**[ sizeId ][ matrixId ] | ue(v) |
| else { |  |
| nextCoef = 8 |  |
| coefNum = Min( 64, ( 1  <<  ( sizeId  <<  1 ) ) ) |  |
| if( sizeId > 3 ) { |  |
| **scaling\_list\_dc\_coef\_minus8**[ sizeId − 4 ][ matrixId ] | se(v) |
| nextCoef = scaling\_list\_dc\_coef\_minus8[ sizeId − 4 ][ matrixId ] + 8 |  |
| } |  |
| for( i = 0; i < coefNum; i++ ) { |  |
| x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] |  |
| y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] |  |
| if ( !( sizeId = = 6 && x >= 4 && y >= 4) ) { |  |
| **scaling\_list\_delta\_coef** | se(v) |
| nextCoef = ( nextCoef + scaling\_list\_delta\_coef + 256 ) % 256 |  |
| ScalingList[ sizeId ][ matrixId ][ i ] = nextCoef |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

### Profile, tier, and level syntax

#### General profile, tier, and level syntax

|  |  |
| --- | --- |
| profile\_tier\_level( maxNumSubLayersMinus1 ) { | **Descriptor** |
| **general\_profile\_idc** | u(7) |
| **general\_tier\_flag** | u(1) |
| **general\_sub\_profile\_idc** | u(24) |
| general\_constraint\_info( ) |  |
| **general\_level\_idc** | u(8) |
| for( i = 0; i < maxNumSubLayersMinus1; i++ ) |  |
| **sub\_layer\_level\_present\_flag**[ i ] | u(1) |
| while( !byte\_aligned( ) ) |  |
| **ptl\_alignment\_zero\_bit** | f(1) |
| for( i = 0; i < maxNumSubLayersMinus1; i++ ) |  |
| if( sub\_layer\_level\_present\_flag[ i ] ) |  |
| **sub\_layer\_level\_idc**[ i ] | u(8) |
| } |  |

#### General constraint information syntax

|  |  |
| --- | --- |
| general\_constraint\_info( ) { | **Descriptor** |
| **general\_progressive\_source\_flag** | u(1) |
| **general\_interlaced\_source\_flag** | u(1) |
| **general\_non\_packed\_constraint\_flag** | u(1) |
| **general\_frame\_only\_constraint\_flag** | u(1) |
| **intra\_only\_constraint\_flag** | u(1) |
| **max\_bitdepth\_constraint\_idc** | u(4) |
| **max\_chroma\_format\_constraint\_idc** | u(2) |
| **frame\_only\_constraint\_flag** | u(1) |
| **no\_qtbtt\_dual\_tree\_intra\_constraint\_flag** | u(1) |
| **no\_partition\_constraints\_override\_constraint\_flag** | u(1) |
| **no\_sao\_constraint\_flag** | u(1) |
| **no\_alf\_constraint\_flag** | u(1) |
| **no\_pcm\_constraint\_flag** | u(1) |
| **no\_ref\_wraparound\_constraint\_flag** | u(1) |
| **no\_temporal\_mvp\_constraint\_flag** | u(1) |
| **no\_sbtmvp\_constraint\_flag** | u(1) |
| **no\_amvr\_constraint\_flag** | u(1) |
| **no\_bdof\_constraint\_flag** | u(1) |
| **no\_dmvr\_constraint\_flag** | u(1) |
| **no\_cclm\_constraint\_flag** | u(1) |
| **no\_mts\_constraint\_flag** | u(1) |
| **no\_sbt\_constraint\_flag** | u(1) |
| **no\_affine\_motion\_constraint\_flag** | u(1) |
| **no\_bcw\_constraint\_flag** | u(1) |
| **no\_ibc\_constraint\_flag** | u(1) |
| **no\_ciip\_constraint\_flag** | u(1) |
| **no\_fpel\_mmvd\_constraint\_flag** | u(1) |
| **no\_triangle\_constraint\_flag** | u(1) |
| **no\_ladf\_constraint\_flag** | u(1) |
| **no\_transform\_skip\_constraint\_flag** | u(1) |
| **no\_qp\_delta\_constraint\_flag** | u(1) |
| **no\_dep\_quant\_constraint\_flag** | u(1) |
| **no\_sign\_data\_hiding\_constraint\_flag** | u(1) |
| // ADD reserved bits for future extensions |  |
| while( !byte\_aligned( ) ) |  |
| **gci\_alignment\_zero\_bit** | f(1) |
| } |  |

### Supplemental enhancement information message syntax

|  |  |
| --- | --- |
| sei\_message( ) { | **Descriptor** |
| payloadType = 0 |  |
| do { |  |
| **payload\_type\_byte** | u(8) |
| payloadType += payload\_type\_byte |  |
| } while (payload\_type\_byte = = 0xFF ) |  |
| payloadSize = 0 |  |
| do { |  |
| **payload\_size\_byte** | u(8) |
| payloadSize += payload\_size\_byte |  |
| } while (payload\_size\_byte = = 0xFF ) |  |
| sei\_payload( payloadType, payloadSize ) |  |
| } |  |

### Slice header syntax

#### General slice header syntax

|  |  |
| --- | --- |
| slice\_header( ) { | Descriptor |
| **slice\_pic\_parameter\_set\_id** | ue(v) |
| if( rect\_slice\_flag | | NumBricksInPic > 1 ) |  |
| **slice\_address** | u(v) |
| if( !rect\_slice\_flag && !single\_brick\_per\_slice\_flag ) |  |
| **num\_bricks\_in\_slice\_minus1** | ue(v) |
| **slice\_type** | ue(v) |
| if( separate\_colour\_plane\_flag = = 1 ) |  |
| **colour\_plane\_id** | u(2) |
| if( NalUnitType = = GRA\_NUT ) |  |
| **recovery\_poc\_cnt** | se(v) |
| **slice\_pic\_order\_cnt\_lsb** | u(v) |
| if( NalUnitType = = IDR\_W\_RADL | | NalUnitType = = IDR\_N\_LP | |  NalUnitType = = CRA\_NUT ) |  |
| **no\_output\_of\_prior\_pics\_flag** | u(1) |
| if( output\_flag\_present\_flag ) |  |
| **pic\_output\_flag** | u(1) |
| if( ( NalUnitType != IDR\_W\_RADL && NalUnitType != IDR\_N\_LP ) | |  sps\_idr\_rpl\_present\_flag ) { |  |
| for( i = 0; i < 2; i++ ) { |  |
| if( num\_ref\_pic\_lists\_in\_sps[ i ] > 0 &&  ( i = = 0 | | ( i = = 1 && rpl1\_idx\_present\_flag ) ) ) |  |
| **ref\_pic\_list\_sps\_flag**[ i ] | u(1) |
| if( ref\_pic\_list\_sps\_flag[ i ] ) { |  |
| if( num\_ref\_pic\_lists\_in\_sps[ i ] > 1 && |  |
| ( i = = 0 | | ( i = = 1 && rpl1\_idx\_present\_flag ) ) ) |  |
| **ref\_pic\_list\_idx**[ i ] | u(v) |
| } else |  |
| ref\_pic\_list\_struct( i, num\_ref\_pic\_lists\_in\_sps[ i ] ) |  |
| for( j = 0; j < NumLtrpEntries[ i ][ RplsIdx[ i ] ]; j++ ) { |  |
| if( ltrp\_in\_slice\_header\_flag[ i ][ RplsIdx[ i ] ] ) |  |
| **slice\_poc\_lsb\_lt**[ i ][ j ] | u(v) |
| **delta\_poc\_msb\_present\_flag**[ i ][ j ] | u(1) |
| if( delta\_poc\_msb\_present\_flag[ i ][ j ] ) |  |
| **delta\_poc\_msb\_cycle\_lt**[ i ][ j ] | ue(v) |
| } |  |
| } |  |
| if( ( slice\_type != I && num\_ref\_entries[ 0 ][ RplsIdx[ 0 ] ] > 1 ) | |  ( slice\_type = = B && num\_ref\_entries[ 1 ][ RplsIdx[ 1 ] ] > 1 ) ) { |  |
| **num\_ref\_idx\_active\_override\_flag** | u(1) |
| if( num\_ref\_idx\_active\_override\_flag ) |  |
| for( i = 0; i < ( slice\_type = = B ? 2: 1 ); i++ ) |  |
| if( num\_ref\_entries[ i ][ RplsIdx[ i ] ] > 1 ) |  |
| **num\_ref\_idx\_active\_minus1**[ i ] | ue(v) |
| } |  |
| } |  |
| if( partition\_constraints\_override\_enabled\_flag ) { |  |
| **partition\_constraints\_override\_flag** | ue(v) |
| if( partition\_constraints\_override\_flag ) { |  |
| **slice\_log2\_diff\_min\_qt\_min\_cb\_luma** | ue(v) |
| **slice\_max\_mtt\_hierarchy\_depth\_luma** | ue(v) |
| if( slice\_max\_mtt\_hierarchy\_depth\_luma != 0 ) |  |
| **slice\_log2\_diff\_max\_bt\_min\_qt\_luma** | ue(v) |
| **slice\_log2\_diff\_max\_tt\_min\_qt\_luma** | ue(v) |
| } |  |
| if( slice\_type = = I && qtbtt\_dual\_tree\_intra\_flag ) { |  |
| **slice\_log2\_diff\_min\_qt\_min\_cb\_chroma** | ue(v) |
| **slice\_max\_mtt\_hierarchy\_depth\_chroma** | ue(v) |
| if( slice\_max\_mtt\_hierarchy\_depth\_chroma != 0 ) |  |
| **slice\_log2\_diff\_max\_bt\_min\_qt\_chroma** | ue(v) |
| **slice\_log2\_diff\_max\_tt\_min\_qt\_chroma** | ue(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| if ( slice\_type != I ) { |  |
| if( sps\_temporal\_mvp\_enabled\_flag ) |  |
| **slice\_temporal\_mvp\_enabled\_flag** | u(1) |
| if( slice\_type = = B ) |  |
| **mvd\_l1\_zero\_flag** | u(1) |
| if( cabac\_init\_present\_flag ) |  |
| **cabac\_init\_flag** | u(1) |
| if( slice\_temporal\_mvp\_enabled\_flag ) { |  |
| if( slice\_type = = B ) |  |
| **collocated\_from\_l0\_flag** | u(1) |
| } |  |
| if( ( weighted\_pred\_flag && slice\_type = = P ) | |  ( weighted\_bipred\_flag && slice\_type = = B ) ) |  |
| pred\_weight\_table( ) |  |
| **six\_minus\_max\_num\_merge\_cand** | ue(v) |
| if( sps\_affine\_enabled\_flag ) |  |
| **five\_minus\_max\_num\_subblock\_merge\_cand** | ue(v) |
| if( sps\_fpel\_mmvd\_enabled\_flag ) |  |
| **slice\_fpel\_mmvd\_enabled\_flag** | u(1) |
| if( sps\_triangle\_enabled\_flag && MaxNumMergeCand >= 2 ) |  |
| **max\_num\_merge\_cand\_minus\_max\_num\_triangle\_cand** | ue(v) |
| } else if ( sps\_ibc\_enabled\_flag ) |  |
| **six\_minus\_max\_num\_merge\_cand** | ue(v) |
| **slice\_qp\_delta** | se(v) |
| if( pps\_slice\_chroma\_qp\_offsets\_present\_flag ) { |  |
| **slice\_cb\_qp\_offset** | se(v) |
| **slice\_cr\_qp\_offset** | se(v) |
| **slice\_joint\_cbcr\_qp\_offset** | se(v) |
| } |  |
| if( sps\_sao\_enabled\_flag ) { |  |
| **slice\_sao\_luma\_flag** | u(1) |
| if( ChromaArrayType != 0 ) |  |
| **slice\_sao\_chroma\_flag** | u(1) |
| } |  |
| if( sps\_alf\_enabled\_flag ) { |  |
| **slice\_alf\_enabled\_flag** | u(1) |
| if( slice\_alf\_enabled\_flag ) { |  |
| **slice\_num\_alf\_aps\_ids\_luma** | tb(v) |
| for( i = 0; i < slice\_num\_alf\_aps\_ids\_luma; i++ ) |  |
| **slice\_alf\_aps\_id\_luma**[ i ] | u(5) |
| if( ChromaArrayType != 0 ) |  |
| **slice\_alf\_chroma\_idc** | tu(v) |
| if( slice\_alf\_chroma\_idc && ( slice\_type != I | | slice\_num\_alf\_aps\_ids\_luma != 1) ) |  |
| **slice\_alf\_aps\_id\_chroma** | u(5) |
| } |  |
| } |  |
| **dep\_quant\_enabled\_flag** | u(1) |
| if( !dep\_quant\_enabled\_flag ) |  |
| **sign\_data\_hiding\_enabled\_flag** | u(1) |
| if( deblocking\_filter\_override\_enabled\_flag ) |  |
| **deblocking\_filter\_override\_flag** | u(1) |
| if( deblocking\_filter\_override\_flag ) { |  |
| **slice\_deblocking\_filter\_disabled\_flag** | u(1) |
| if( !slice\_deblocking\_filter\_disabled\_flag ) { |  |
| **slice\_beta\_offset\_div2** | se(v) |
| **slice\_tc\_offset\_div2** | se(v) |
| } |  |
| } |  |
| if( sps\_lmcs\_enabled\_flag ) { |  |
| **slice\_lmcs\_enabled\_flag** | u(1) |
| if( slice\_lmcs\_enabled\_flag ) { |  |
| **slice\_lmcs\_aps\_id** | u(5) |
| if( !( qtbtt\_dual\_tree\_intra\_flag && slice\_type = = I ) ) |  |
| **slice\_chroma\_residual\_scale\_flag** | u(1) |
| } |  |
| if ( entropy\_coding\_sync\_enabled\_flag ) |  |
| **num\_entry\_point\_offsets** | ue(v) |
| if( NumEntryPoints > 0 ) { |  |
| **offset\_len\_minus1** | ue(v) |
| for( i = 0; i < NumEntryPoints; i++ ) |  |
| **entry\_point\_offset\_minus1**[ i ] | u(v) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

#### Weighted prediction parameters syntax

|  |  |
| --- | --- |
| pred\_weight\_table( ) { | Descriptor |
| **luma\_log2\_weight\_denom** | ue(v) |
| if( ChromaArrayType != 0 ) |  |
| **delta\_chroma\_log2\_weight\_denom** | se(v) |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) |  |
| **luma\_weight\_l0\_flag**[ i ] | u(1) |
| if( ChromaArrayType != 0 ) |  |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) |  |
| **chroma\_weight\_l0\_flag**[ i ] | u(1) |
| for( i = 0; i < NumRefIdxActive[ 0 ]; i++ ) { |  |
| if( luma\_weight\_l0\_flag[ i ] ) { |  |
| **delta\_luma\_weight\_l0**[ i ] | se(v) |
| **luma\_offset\_l0**[ i ] | se(v) |
| } |  |
| if( chroma\_weight\_l0\_flag[ i ] ) |  |
| 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 < NumRefIdxActive[ 1 ]; i++ ) |  |
| **luma\_weight\_l1\_flag**[ i ] | u(1) |
| if( ChromaArrayType != 0 ) |  |
| for( i = 0; i < NumRefIdxActive[ 1 ]; i++ ) |  |
| **chroma\_weight\_l1\_flag**[ i ] | u(1) |
| for( i = 0; i < NumRefIdxActive[ 1 ]; i++ ) { |  |
| if( luma\_weight\_l1\_flag[ i ] ) { |  |
| **delta\_luma\_weight\_l1**[ i ] | se(v) |
| **luma\_offset\_l1**[ i ] | se(v) |
| } |  |
| if( chroma\_weight\_l1\_flag[ i ] ) |  |
| for( j = 0; j < 2; j++ ) { |  |
| **delta\_chroma\_weight\_l1**[ i ][ j ] | se(v) |
| **delta\_chroma\_offset\_l1**[ i ][ j ] | se(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Adaptive loop filter data syntax

|  |  |
| --- | --- |
| alf\_data( adaptation\_parameter\_set\_id ) { | Descriptor |
| **alf\_luma\_filter\_signal\_flag** | u(1) |
| **alf\_chroma\_filter\_signal\_flag** | u(1) |
| if( alf\_luma\_filter\_signal\_flag ) { |  |
| **alf\_luma\_clip\_flag** | u(1) |
| **alf\_luma\_num\_filters\_signalled\_minus1** | tb(v) |
| if( alf\_luma\_num\_filters\_signalled\_minus1 > 0 ) { |  |
| for( filtIdx = 0; filtIdx < NumAlfFilters; filtIdx++ ) |  |
| **alf\_luma\_coeff\_delta\_idx**[ filtIdx ] | tb(v) |
| } |  |
| **alf\_luma\_use\_fixed\_filter\_flag** | u(1) |
| if( **alf\_luma\_use\_fixed\_filter\_flag** ) { |  |
| **alf\_luma\_fixed\_filter\_set\_idx** | tb(v) |
| **alf\_luma\_fixed\_filter\_pred\_present\_flag** | u(1) |
| if( alf\_luma\_fixed\_filter\_pred\_present\_flag ) { |  |
| for( i = 0; i < NumAlfFilters; i++ ) |  |
| **alf\_luma\_fixed\_filter\_pred\_flag**[ i ] | u(1) |
| } |  |
| } |  |
| **alf\_luma\_coeff\_delta\_flag** | u(1) |
| if( !alf\_luma\_coeff\_delta\_flag && alf\_luma\_num\_filters\_signalled\_minus1 > 0 ) |  |
| **alf\_luma\_coeff\_delta\_prediction\_flag** | u(1) |
| **alf\_luma\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < 3; i++ ) |  |
| **alf\_luma\_eg\_order\_increase\_flag**[ i ] | u(1) |
| if( alf\_luma\_coeff\_delta\_flag ) { |  |
| for( sfIdx = 0; sfIdx <= alf\_luma\_num\_filters\_signalled\_minus1; sfIdx++ ) |  |
| **alf\_luma\_coeff\_flag**[ sfIdx ] | u(1) |
| } |  |
| for( sfIdx = 0; sfIdx <= alf\_luma\_num\_filters\_signalled\_minus1; sfIdx++ ) { |  |
| if( alf\_luma\_coeff\_flag[ sfIdx ] ) { |  |
| for ( j = 0; j < 12; j++ ) { |  |
| **alf\_luma\_coeff\_delta\_abs**[ sfIdx ][ j ] | uek(v) |
| if( alf\_luma\_coeff\_delta\_abs[ sfIdx ][ j ] ) |  |
| **alf\_luma\_coeff\_delta\_sign**[ sfIdx ][ j ] | u(1) |
| } |  |
| } |  |
| } |  |
| if( alf\_luma\_clip\_flag ) { |  |
| **alf\_luma\_clip\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < 3; i++ ) |  |
| **alf\_luma\_clip\_eg\_order\_increase\_flag**[ i ] | u(1) |
| for( sfIdx = 0; sfIdx <= alf\_luma\_num\_filters\_signalled\_minus1; sfIdx++ ) { |  |
| if( alf\_luma\_coeff\_flag[ sfIdx ] ) { |  |
| for ( j = 0; j < 12; j++ ) { |  |
| if( filtCoeff[ sfIdx ][ j ] ) |  |
| **alf\_luma\_clip\_idx**[ sfIdx ][ j ] | uek(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| if( alf\_chroma\_filter\_signal\_flag ) { |  |
| **alf\_chroma\_clip\_flag** | u(1) |
| **alf\_chroma\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < 2; i++ ) |  |
| **alf\_chroma\_eg\_order\_increase\_flag**[ i ] | u(1) |
| for( j = 0; j < 6; j++ ) { |  |
| **alf\_chroma\_coeff\_abs**[ j ] | uek(v) |
| if( alf\_chroma\_coeff\_abs[ j ] > 0 ) |  |
| **alf\_chroma\_coeff\_sign**[ j ] | u(1) |
| } |  |
| if( alf\_chroma\_clip\_flag ) { |  |
| **alf\_chroma\_clip\_min\_eg\_order\_minus1** | ue(v) |
| for( i = 0; i < 2; i++ ) |  |
| **alf\_chroma\_clip\_eg\_order\_increase\_flag**[ i ] | u(1) |
| for( j = 0; j < 6; j++ ) { |  |
| if( **alf\_chroma\_coeff\_abs**[ j ] ) |  |
| **alf\_chroma\_clip\_idx**[ j ] | uek(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### Luma mapping with chroma scaling data syntax

|  |  |
| --- | --- |
| lmcs\_data () { | **Descriptor** |
| **lmcs\_min\_bin\_idx** | ue(v) |
| **lmcs\_delta\_max\_bin\_idx** | ue(v) |
| **lmcs\_delta\_cw\_prec\_minus1** | ue(v) |
| for ( i = lmcs\_min\_bin\_idx; i <= LmcsMaxBinIdx; i++ ) { |  |
| **lmcs\_delta\_abs\_cw**[ i ] | u(v) |
| if ( lmcs\_delta\_abs\_cw[ i ] ) > 0 ) |  |
| **lmcs\_delta\_sign\_cw\_flag**[ i ] | u(1) |
| } |  |
| } |  |

### Reference picture list structure syntax

|  |  |
| --- | --- |
| ref\_pic\_list\_struct( listIdx, rplsIdx ) { | Descriptor |
| **num\_ref\_entries**[ listIdx ][ rplsIdx ] | ue(v) |
| if( long\_term\_ref\_pics\_flag ) |  |
| **ltrp\_in\_slice\_header\_flag**[ listIdx ][ rplsIdx ] | u(1) |
| for( i = 0, j = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++) { |  |
| if( long\_term\_ref\_pics\_flag ) |  |
| **st\_ref\_pic\_flag**[ listIdx ][ rplsIdx ][ i ] | u(1) |
| if( st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) { |  |
| **abs\_delta\_poc\_st**[ listIdx ][ rplsIdx ][ i ] | ue(v) |
| if( abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] > 0 ) |  |
| **strp\_entry\_sign\_flag**[ listIdx ][ rplsIdx ][ i ] | u(1) |
| } else if( !ltrp\_in\_slice\_header\_flag[ listIdx ][ rplsIdx ] ) |  |
| **rpls\_poc\_lsb\_lt**[ listIdx ][ rplsIdx ][ j++ ] | u(v) |
| } |  |
| } |  |

### Slice data syntax

#### General slice data syntax

|  |  |
| --- | --- |
| slice\_data( ) { | Descriptor |
| for( i = 0; i < NumBricksInCurrSlice; i++ ) { |  |
| CtbAddrInBs = FirstCtbAddrBs[ SliceBrickIdx[ i ] ] |  |
| for( j = 0; j < NumCtusInBrick[ SliceBrickIdx[ i ] ]; j++, CtbAddrInBs++ ) { |  |
| if( ( j % BrickWidth[ SliceBrickIdx[ i ] ] ) = = 0 ) { |  |
| NumHmvpCand = 0 |  |
| NumHmvpIbcCand = 0 |  |
| } |  |
| CtbAddrInRs = CtbAddrBsToRs[ CtbAddrInBs ] |  |
| coding\_tree\_unit( ) |  |
| if( entropy\_coding\_sync\_enabled\_flag &&  ( ( j + 1 ) % BrickWidth[ SliceBrickIdx[ i ] ] = = 0 ) ) { |  |
| **end\_of\_subset\_one\_bit** /\* equal to 1 \*/ | ae(v) |
| if( j < NumCtusInBrick[ SliceBrickIdx[ i ] ] − 1 ) |  |
| byte\_alignment( ) |  |
| } |  |
| } |  |
| if( !entropy\_coding\_sync\_enabled\_flag ) { |  |
| **end\_of\_brick\_one\_bit** /\* equal to 1 \*/ | ae(v) |
| if( i < NumBricksInCurrSlice − 1 ) |  |
| byte\_alignment( ) |  |
| } |  |
| } |  |
| } |  |

#### Coding tree unit syntax

|  |  |
| --- | --- |
| coding\_tree\_unit( ) { | Descriptor |
| xCtb = ( CtbAddrInRs % PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| yCtb = ( CtbAddrInRs / PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| if( slice\_sao\_luma\_flag | | slice\_sao\_chroma\_flag ) |  |
| sao( xCtb  >>  CtbLog2SizeY, yCtb  >>  CtbLog2SizeY ) |  |
| if( slice\_alf\_enabled\_flag ){ |  |
| **alf\_ctb\_flag**[ 0 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| if( alf\_ctb\_flag[ 0 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] ) { |  |
| if( slice\_num\_alf\_aps\_ids\_luma > 0) |  |
| **alf\_ctb\_use\_first\_aps\_flag** | ae(v) |
| if( !alf\_ctb\_use\_first\_aps\_flag ) { |  |
| if( slice\_num\_alf\_aps\_ids\_luma > 1 ) |  |
| **alf\_use\_aps\_flag** | ae(v) |
| if( alf\_use\_aps\_flag ) |  |
| **alf\_luma\_prev\_filter\_idx\_minus1** | ae(v) |
| else |  |
| **alf\_luma\_fixed\_filter\_idx** | ae(v) |
| } |  |
| } |  |
| if( slice\_alf\_chroma\_idc  = =  1  | |  slice\_alf\_chroma\_idc  = =  3 ) |  |
| **alf\_ctb\_flag**[ 1 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| if( slice\_alf\_chroma\_idc  = =  2  | |  slice\_alf\_chroma\_idc  = =  3 ) |  |
| **alf\_ctb\_flag**[ 2 ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] | ae(v) |
| } |  |
| if( slice\_type = = I  &&  qtbtt\_dual\_tree\_intra\_flag ) |  |
| dual\_tree\_implicit\_qt\_split ( xCtb, yCtb, CtbSizeY, 0 ) |  |
| else |  |
| coding\_tree( xCtb, yCtb, CtbSizeY, CtbSizeY, 1, 0, 0, 0, 0, 0, SINGLE\_TREE ) |  |
| } |  |

|  |  |
| --- | --- |
| dual\_tree\_implicit\_qt\_split( x0, y0, cbSize, cqtDepth ) { | Descriptor |
| cbSubdiv = 2 \* cqtDepth |  |
| if( cbSize > 64 ) { |  |
| if( cu\_qp\_delta\_enabled\_flag && cbSubdiv <= cu\_qp\_delta\_subdiv ) { |  |
| IsCuQpDeltaCoded = 0 |  |
| CuQpDeltaVal = 0 |  |
| CuQgTopLeftX = x0 |  |
| CuQgTopLeftY = y0 |  |
| } |  |
| x1 = x0 + ( cbSize / 2 ) |  |
| y1 = y0 + ( cbSize / 2 ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y0, cbSize / 2, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y0, cbSize / 2, cqtDepth + 1 ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x0, y1, cbSize / 2, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples && y1 < pic\_height\_in\_luma\_samples ) |  |
| dual\_tree\_implicit\_qt\_split( x1, y1, cbSize / 2, cqtDepth + 1 ) |  |
| } else { |  |
| coding\_tree( x0, y0, cbSize, cbSize, 1, cbSubdiv, cqtDepth, 0, 0, 0,   DUAL\_TREE\_LUMA ) |  |
| coding\_tree( x0, y0, cbSize, cbSize, 0, cbSubdiv, cqtDepth, 0, 0, 0,   DUAL\_TREE\_CHROMA ) |  |
| } |  |
| } |  |

#### Sample adaptive offset syntax

|  |  |
| --- | --- |
| sao( rx, ry ) { | Descriptor |
| if( rx > 0 ) { |  |
| leftCtbInBrick = BrickId[ CtbAddrInBs ] = =  BrickId[ CtbAddrRsToBs[ CtbAddrInRs − 1 ] ] |  |
| if( leftCtbInBrick ) |  |
| **sao\_merge\_left\_flag** | ae(v) |
| } |  |
| if( ry > 0 && !sao\_merge\_left\_flag ) { |  |
| upCtbInBrick = BrickId[ CtbAddrInBs ] = =   BrickId[ CtbAddrRsToBs[ CtbAddrInRs − PicWidthInCtbsY ] ] |  |
| if( upCtbInBrick ) |  |
| **sao\_merge\_up\_flag** | ae(v) |
| } |  |
| if( !sao\_merge\_up\_flag && !sao\_merge\_left\_flag ) |  |
| for( cIdx = 0; cIdx < ( ChromaArrayType != 0 ? 3 : 1 ); cIdx++ ) |  |
| if( ( slice\_sao\_luma\_flag && cIdx = = 0 ) | |  ( slice\_sao\_chroma\_flag && cIdx > 0 ) ) { |  |
| if( cIdx = = 0 ) |  |
| **sao\_type\_idx\_luma** | ae(v) |
| else if( cIdx = = 1 ) |  |
| **sao\_type\_idx\_chroma** | ae(v) |
| if( SaoTypeIdx[ cIdx ][ rx ][ ry ] != 0 ) { |  |
| for( i = 0; i < 4; i++ ) |  |
| **sao\_offset\_abs**[ cIdx ][ rx ][ ry ][ i ] | ae(v) |
| if( SaoTypeIdx[ cIdx ][ rx ][ ry ] = = 1 ) { |  |
| for( i = 0; i < 4; i++ ) |  |
| if( sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] != 0 ) |  |
| **sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ] | ae(v) |
| **sao\_band\_position**[ cIdx ][ rx ][ ry ] | ae(v) |
| } else { |  |
| if( cIdx = = 0 ) |  |
| **sao\_eo\_class\_luma** | ae(v) |
| if( cIdx = = 1 ) |  |
| **sao\_eo\_class\_chroma** | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

[Ed. (BB): Adapt syntax once tiles are integrated.]

#### Coding tree syntax

|  |  |
| --- | --- |
| coding\_tree( x0, y0, cbWidth, cbHeight, qgOn, cbSubdiv, cqtDepth, mttDepth, depthOffset, partIdx,    treeType ) { | Descriptor |
| if( ( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor | | allowSplitQT )   &&( x0 + cbWidth <= pic\_width\_in\_luma\_samples )   && (y0 + cbHeight <= pic\_height\_in\_luma\_samples ) ) |  |
| **split\_cu\_flag** | ae(v) |
| if( cu\_qp\_delta\_enabled\_flag && qgOn && cbSubdiv  <=  cu\_qp\_delta\_subdiv ) { |  |
| IsCuQpDeltaCoded = 0 |  |
| CuQpDeltaVal = 0 |  |
| CuQgTopLeftX = x0 |  |
| CuQgTopLeftY = y0 |  |
| } |  |
| if( split\_cu\_flag ) { |  |
| if( ( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor ) &&   allowSplitQT ) |  |
| **split\_qt\_flag** | ae(v) |
| if( !split\_qt\_flag ) { |  |
| if( ( allowSplitBtHor | | allowSplitTtHor ) &&   ( allowSplitBtVer | | allowSplitTtVer ) ) |  |
| **mtt\_split\_cu\_vertical\_flag** | ae(v) |
| if( ( allowSplitBtVer && allowSplitTtVer && mtt\_split\_cu\_vertical\_flag ) | |   ( allowSplitBtHor && allowSplitTtHor && !mtt\_split\_cu\_vertical\_flag ) ) |  |
| **mtt\_split\_cu\_binary\_flag** | ae(v) |
| if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_VER ) { |  |
| depthOffset  +=  ( x0 + cbWidth  >  pic\_width\_in\_luma\_samples ) ? 1 : 0 |  |
| x1 = x0 + ( cbWidth / 2 ) |  |
| coding\_tree( x0, y0, cbWidth / 2, cbHeight, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeightY, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_HOR ) { |  |
| depthOffset  +=  ( y0 + cbHeight  >  pic\_height\_in\_luma\_samples ) ? 1 : 0 |  |
| y1 = y0 + ( cbHeight / 2 ) |  |
| coding\_tree( x0, y0, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_tree( x0, y1, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_TT\_VER ) { |  |
| x1 = x0 + ( cbWidth / 4 ) |  |
| x2 = x0 + ( 3 \* cbWidth / 4 ) |  |
| qgOn = qgOn && ( cbSubdiv + 2 <= cu\_qp\_delta\_subdiv ) |  |
| coding\_tree( x0, y0, cbWidth / 4, cbHeight, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeight, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| coding\_tree( x2, y0, cbWidth / 4, cbHeight, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 2, treeType ) |  |
| } else { /\* SPLIT\_TT\_HOR \*/ |  |
| y1 = y0 + ( cbHeight / 4 ) |  |
| y2 = y0 + ( 3 \* cbHeight / 4 ) |  |
| qgOn = qgOn && ( cbSubdiv + 2 <= cu\_qp\_delta\_subdiv ) |  |
| coding\_tree( x0, y0, cbWidth, cbHeight / 4, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 0, treeType ) |  |
| coding\_tree( x0, y1, cbWidth, cbHeight / 2, qgOn, cbSubdiv + 1,   cqtDepth, mttDepth + 1, depthOffset, 1, treeType ) |  |
| coding\_tree( x0, y2, cbWidth, cbHeight / 4, qgOn, cbSubdiv + 2,   cqtDepth, mttDepth + 1, depthOffset, 2 , treeType) |  |
| } |  |
| } else { |  |
| x1 = x0 + ( cbWidth / 2 ) |  |
| y1 = y0 + ( cbHeight / 2 ) |  |
| coding\_tree( x0, y0, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1, 0, 0, 0, treeType ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y0, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1, 0, 0, 1, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_tree( x0, y1, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1,  0, 0, 2, treeType ) |  |
| if( y1 < pic\_height\_in\_luma\_samples && x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_tree( x1, y1, cbWidth / 2, cbHeight / 2, qgOn, cbSubdiv + 2,   cqtDepth + 1,  0, 0, 3, treeType ) |  |
| } |  |
| } else |  |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) |  |
| } |  |

#### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, cbWidth, cbHeight, treeType ) { | Descriptor |
| if( slice\_type != I | | sps\_ibc\_enabled\_flag ) { |  |
| if( treeType != DUAL\_TREE\_CHROMA &&   !( cbWidth = = 4 && cbHeight = = 4 && !sps\_ibc\_enabled\_flag ) ) |  |
| **cu\_skip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( cu\_skip\_flag[ x0 ][ y0 ] = = 0 && slice\_type != I   && !( cbWidth = = 4 && cbHeight = = 4 ) ) |  |
| **pred\_mode\_flag** | ae(v) |
| if( ( ( slice\_type = = I && cu\_skip\_flag[ x0 ][ y0 ] = =0 ) | |  ( slice\_type != I && ( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA | |   ( cbWidth = = 4 && cbHeight = = 4 && cu\_skip\_flag[ x0 ][ y0 ] = = 0 ) ) ) ) &&  sps\_ibc\_enabled\_flag && ( cbWidth != 128 | | cbHeight != 128 ) ) |  |
| **pred\_mode\_ibc\_flag** | ae(v) |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA ) { |  |
| if( sps\_pcm\_enabled\_flag &&  cbWidth >= MinIpcmCbSizeY && cbWidth <= MaxIpcmCbSizeY &&  cbHeight >= MinIpcmCbSizeY && cbHeight <= MaxIpcmCbSizeY ) |  |
| **pcm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( pcm\_flag[ x0 ][ y0 ] ) { |  |
| while( !byte\_aligned( ) ) |  |
| **pcm\_alignment\_zero\_bit** | f(1) |
| pcm\_sample( cbWidth, cbHeight, treeType) |  |
| } else { |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| if( cbWidth <= 32 && cbHeight <= 32 ) |  |
| **intra\_bdpcm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_bdpcm\_flag[ x0 ][ y0 ] ) |  |
| **intra\_bdpcm\_dir\_flag**[ x0 ][ y0 ] | ae(v) |
| else { |  |
| if( sps\_mip\_enabled\_flag &&   ( Abs( Log2( cbWidth ) − Log2( cbHeight ) ) <= 2 ) &&   cbWidth <= MaxTbSizeY && cbHeight <= MaxTbSizeY ) |  |
| **intra\_mip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_mip\_flag[ x0 ][ y0 ] ) { |  |
| **intra\_mip\_mpm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_mip\_mpm\_flag[ x0 ][ y0 ] ) |  |
| **intra\_mip\_mpm\_idx**[ x0 ][ y0 ] | ae(v) |
| else |  |
| **intra\_mip\_mpm\_remainder**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( sps\_mrl\_enabled\_flag && ( ( y0 % CtbSizeY )  >  0 ) ) |  |
| **intra\_luma\_ref\_idx**[ x0 ][ y0 ] | ae(v) |
| if ( sps\_isp\_enabled\_flag && intra\_luma\_ref\_idx[ x0 ][ y0 ] = = 0 &&   ( cbWidth <= MaxTbSizeY && cbHeight <= MaxTbSizeY ) &&   ( cbWidth \* cbHeight > MinTbSizeY \* MinTbSizeY ) ) |  |
| **intra\_subpartitions\_mode\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] = = 1 &&   cbWidth <= MaxTbSizeY && cbHeight <= MaxTbSizeY ) |  |
| **intra\_subpartitions\_split\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_luma\_ref\_idx[ x0 ][ y0 ] = = 0 &&   intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **intra\_luma\_mpm\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_luma\_mpm\_flag[ x0 ][ y0 ] ) { |  |
| if( intra\_luma\_ref\_idx[ x0 ][ y0 ] = = 0 ) |  |
| **intra\_luma\_not\_planar\_flag**[ x0 ][ y0 ] | ae(v) |
| if( intra\_luma\_not\_planar\_flag[ x0 ][ y0 ] ) |  |
| **intra\_luma\_mpm\_idx**[ x0 ][ y0 ] | ae(v) |
| } else |  |
| **intra\_luma\_mpm\_remainder**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| if( ( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA ) &&  ChromaArrayType != 0 ) |  |
| **intra\_chroma\_pred\_mode**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } else if( treeType != DUAL\_TREE\_CHROMA ) { /\* MODE\_INTER or MODE\_IBC \*/ |  |
| if( cu\_skip\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **general\_merge\_flag**[ x0 ][ y0 ] | ae(v) |
| if( general\_merge\_flag[ x0 ][ y0 ] ) { |  |
| merge\_data( x0, y0, cbWidth, cbHeight ) |  |
| } else if ( CuPredMode[ x0 ][ y0 ] = = MODE\_IBC ) { |  |
| mvd\_coding( x0, y0, 0, 0 ) |  |
| **mvp\_l0\_flag**[ x0 ][ y0 ] | ae(v) |
| if( sps\_amvr\_enabled\_flag &&   ( MvdL0[ x0 ][ y0 ][ 0 ] != 0 | | MvdL0[ x0 ][ y0 ][ 1 ] != 0 ) ) { |  |
| **amvr\_precision\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } else { |  |
| if( slice\_type = = B ) |  |
| **inter\_pred\_idc**[ x0 ][ y0 ] | ae(v) |
| if( sps\_affine\_enabled\_flag && cbWidth >= 16 && cbHeight >= 16 ) { |  |
| **inter\_affine\_flag**[ x0 ][ y0 ] | ae(v) |
| if( sps\_affine\_type\_flag && inter\_affine\_flag[ x0 ][ y0 ] ) |  |
| **cu\_affine\_type\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( sps\_smvd\_enabled\_flag && inter\_pred\_idc[ x0 ][ y0 ] = = PRED\_BI &&  !inter\_affine\_flag[ x0 ][ y0 ] && RefIdxSymL0 > −1 && RefIdxSymL1 > −1 ) |  |
| **sym\_mvd\_flag**[ x0 ][ y0 ] | ae(v) |
| if( inter\_pred\_idc[ x0 ][ y0 ] != PRED\_L1 ) { |  |
| if( NumRefIdxActive[ 0 ] > 1 && !sym\_mvd\_flag[ x0 ][ y0 ] ) |  |
| **ref\_idx\_l0**[ x0 ][ y0 ] | ae(v) |
| mvd\_coding( x0, y0, 0, 0 ) |  |
| if( MotionModelIdc[ x0 ][ y0 ] > 0 ) |  |
| mvd\_coding( x0, y0, 0, 1 ) |  |
| if(MotionModelIdc[ x0 ][ y0 ] > 1 ) |  |
| mvd\_coding( x0, y0, 0, 2 ) |  |
| **mvp\_l0\_flag**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| MvdL0[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL0[ x0 ][ y0 ][ 1 ] = 0 |  |
| } |  |
| if( inter\_pred\_idc[ x0 ][ y0 ] != PRED\_L0 ) { |  |
| if( NumRefIdxActive[ 1 ] > 1 && !sym\_mvd\_flag[ x0 ][ y0 ] ) |  |
| **ref\_idx\_l1**[ x0 ][ y0 ] | ae(v) |
| if( mvd\_l1\_zero\_flag && inter\_pred\_idc[ x0 ][ y0 ] = = PRED\_BI ) { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 0 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 0 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 1 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 1 ][ 1 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 2 ][ 0 ] = 0 |  |
| MvdCpL1[ x0 ][ y0 ][ 2 ][ 1 ] = 0 |  |
| } else { |  |
| if( sym\_mvd\_flag[ x0 ][ y0 ] ) { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = −MvdL0[ x0 ][ y0 ][ 0 ] |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = −MvdL0[ x0 ][ y0 ][ 1 ] |  |
| } else |  |
| mvd\_coding( x0, y0, 1, 0 ) |  |
| if( MotionModelIdc[ x0 ][ y0 ] > 0 ) |  |
| mvd\_coding( x0, y0, 1, 1 ) |  |
| if(MotionModelIdc[ x0 ][ y0 ] > 1 ) |  |
| mvd\_coding( x0, y0, 1, 2 ) |  |
| **mvp\_l1\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } else { |  |
| MvdL1[ x0 ][ y0 ][ 0 ] = 0 |  |
| MvdL1[ x0 ][ y0 ][ 1 ] = 0 |  |
| } |  |
| if( ( sps\_amvr\_enabled\_flag && inter\_affine\_flag[ x0 ][ y0 ] = = 0 &&  ( MvdL0[ x0 ][ y0 ][ 0 ] != 0 | | MvdL0[ x0 ][ y0 ][ 1 ] != 0 | |  MvdL1[ x0 ][ y0 ][ 0 ] != 0 | | MvdL1[ x0 ][ y0 ][ 1 ] != 0 ) ) | |  ( sps\_affine\_amvr\_enabled\_flag && inter\_affine\_flag[ x0 ][ y0 ] = = 1 &&  ( MvdCpL0[ x0 ][ y0 ][ 0 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 0 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 0 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 0 ] [ 1 ] != 0 | |  MvdCpL0[ x0 ][ y0 ][ 1 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 1 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 1 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 1 ] [ 1 ] != 0 | |  MvdCpL0[ x0 ][ y0 ][ 2 ] [ 0 ] != 0 | | MvdCpL0[ x0 ][ y0 ][ 2 ] [ 1 ] != 0 | |  MvdCpL1[ x0 ][ y0 ][ 2 ] [ 0 ] != 0 | | MvdCpL1[ x0 ][ y0 ][ 2 ] [ 1 ] != 0 ) ) { |  |
| **amvr\_flag**[ x0 ][ y0 ] | ae(v) |
| if( amvr\_flag[ x0 ][ y0 ] ) |  |
| **amvr\_precision\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( sps\_bcw\_enabled\_flag && inter\_pred\_idc[ x0 ][ y0 ] = = PRED\_BI &&  luma\_weight\_l0\_flag[ ref\_idx\_l0 [ x0 ][ y0 ] ] = = 0 &&  luma\_weight\_l1\_flag[ ref\_idx\_l1 [ x0 ][ y0 ] ] = = 0 &&  chroma\_weight\_l0\_flag[ ref\_idx\_l0 [ x0 ][ y0 ] ] = = 0 &&  chroma\_weight\_l1\_flag[ ref\_idx\_l1 [ x0 ][ y0 ] ] = = 0 &&  cbWidth \* cbHeight >= 256 ) |  |
| **bcw\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| if( !pcm\_flag[ x0 ][ y0 ] ) { |  |
| if( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA &&   general\_merge\_flag[ x0 ][ y0 ] = = 0 ) |  |
| **cu\_cbf** | ae(v) |
| if( cu\_cbf ) { |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTER && sps\_sbt\_enabled\_flag &&   !ciip\_flag[ x0 ][ y0 ] && !MergeTriangleFlag[ x0 ][ y0 ] ) { |  |
| if( cbWidth  <=  MaxSbtSize && cbHeight  <=  MaxSbtSize ) { |  |
| allowSbtVerH = cbWidth  >=  8 |  |
| allowSbtVerQ = cbWidth  >=  16 |  |
| allowSbtHorH = cbHeight  >=  8 |  |
| allowSbtHorQ = cbHeight  >=  16 |  |
| if( allowSbtVerH | | allowSbtHorH | | allowSbtVerQ | | allowSbtHorQ ) |  |
| **cu\_sbt\_flag** | ae(v) |
| } |  |
| if( cu\_sbt\_flag ) { |  |
| if( ( allowSbtVerH | | allowSbtHorH ) && ( allowSbtVerQ | | allowSbtHorQ) ) |  |
| **cu\_sbt\_quad\_flag** | ae(v) |
| if( ( cu\_sbt\_quad\_flag && allowSbtVerQ && allowSbtHorQ ) | |   ( !cu\_sbt\_quad\_flag && allowSbtVerH && allowSbtHorH ) ) |  |
| **cu\_sbt\_horizontal\_flag** | ae(v) |
| **cu\_sbt\_pos\_flag** | ae(v) |
| } |  |
| } |  |
| numSigCoeff = 0 |  |
| numZeroOutSigCoeff = 0 |  |
| transform\_tree( x0, y0, cbWidth, cbHeight, treeType ) |  |
| lfnstWidth = ( treeType = = DUAL\_TREE\_CHROMA )  ?  cbWidth / SubWidthC    :  cbWidth |  |
| lfnstHeight = ( treeType = = DUAL\_TREE\_CHROMA )  ?  cbHeight  /  SubHeightC    :  cbHeight |  |
| if( Min( lfnstWidth, lfnstHeight ) >= 4 && sps\_lfnst\_enabled\_flag = = 1 &&   CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA &&   IntraSubPartitionsSplitType = = ISP\_NO\_SPLIT &&   !intra\_mip\_flag[ x0 ][ y0 ] ) { |  |
| if( ( numSigCoeff > ( ( treeType  = =  SINGLE\_TREE )  ?  2  :  1 ) ) &&   numZeroOutSigCoeff = = 0 ) |  |
| **lfnst\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |

#### PCM sample syntax

|  |  |
| --- | --- |
| pcm\_sample( cbWidth, cbHeight, treeType ) { | **Descriptor** |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| for( i = 0; i < cbWidth \* cbHeight; i++ ) |  |
| **pcm\_sample\_luma**[ i ] | u(v) |
| } |  |
| if( ( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA) &&   ChromaArrayType != 0 ) { |  |
| for( i = 0; i < 2 \* ( ( cbWidth \* cbHeight ) / ( SubWidthC \* SubHeightC ) ); i++ ) |  |
| **pcm\_sample\_chroma**[ i ] | u(v) |
| } |  |
| } |  |

#### Merge data syntax

|  |  |
| --- | --- |
| merge\_data( x0, y0, cbWidth, cbHeight ) { | Descriptor |
| if ( CuPredMode[ x0 ][ y0 ] = = MODE\_IBC ) { |  |
| if( MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( sps\_mmvd\_enabled\_flag | | cbWidth \* cbHeight != 32 ) |  |
| **regular\_merge\_flag**[ x0 ][ y0 ] | ae(v) |
| if ( regular\_merge\_flag[ x0 ][ y0 ] = = 1 ){ |  |
| if( MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( sps\_mmvd\_enabled\_flag && cbWidth \* cbHeight != 32 ) |  |
| **mmvd\_merge\_flag**[ x0 ][ y0 ] | ae(v) |
| if( mmvd\_merge\_flag[ x0 ][ y0 ] = = 1 ) { |  |
| if( MaxNumMergeCand > 1 ) |  |
| **mmvd\_cand\_flag**[ x0 ][ y0 ] | ae(v) |
| **mmvd\_distance\_idx**[ x0 ][ y0 ] | ae(v) |
| **mmvd\_direction\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( MaxNumSubblockMergeCand > 0 && cbWidth >= 8 && cbHeight >= 8 ) |  |
| **merge\_subblock\_flag**[ x0 ][ y0 ] | ae(v) |
| if( merge\_subblock\_flag[ x0 ][ y0 ] = = 1 ) { |  |
| if( MaxNumSubblockMergeCand > 1 ) |  |
| **merge\_subblock\_idx**[ x0 ][ y0 ] | ae(v) |
| } else { |  |
| if( sps\_ciip\_enabled\_flag && cu\_skip\_flag[ x0 ][ y0 ] = = 0 &&   ( cbWidth \* cbHeight ) >= 64 && cbWidth < 128 && cbHeight < 128 ) { |  |
| **ciip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( ciip\_flag[ x0 ][ y0 ] && MaxNumMergeCand > 1 ) |  |
| **merge\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( MergeTriangleFlag[ x0 ][ y0 ] ) { |  |
| **merge\_triangle\_split\_dir**[ x0 ][ y0 ] | ae(v) |
| **merge\_triangle\_idx0**[ x0 ][ y0 ] | ae(v) |
| **merge\_triangle\_idx1**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

#### Motion vector difference syntax

|  |  |
| --- | --- |
| mvd\_coding( x0, y0, refList ,cpIdx ) { | **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) |
| } |  |
| } |  |

#### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0, y0, tbWidth, tbHeight , treeType) { | Descriptor |
| InferTuCbfLuma = 1 |  |
| if( IntraSubPartitionsSplitType = = NO\_ISP\_SPLIT && !cu\_sbt\_flag ) { |  |
| if( tbWidth > MaxTbSizeY | | tbHeight > MaxTbSizeY ) { |  |
| trafoWidth = ( tbWidth > MaxTbSizeY ) ? (tbWidth / 2) : tbWidth |  |
| trafoHeight = ( tbHeight > MaxTbSizeY ) ? (tbHeight / 2) : tbHeight |  |
| transform\_tree( x0, y0, trafoWidth,  trafoHeight ) |  |
| if( tbWidth > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0, trafoWidth, trafoHeight, treeType ) |  |
| if( tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0, y0 + trafoHeight, trafoWidth, trafoHeight, treeType ) |  |
| if( tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight, treeType ) |  |
| } else { |  |
| transform\_unit( x0, y0, tbWidth, tbHeight, treeType, 0 ) |  |
| } |  |
| } else if( cu\_sbt\_flag ) { |  |
| if( !cu\_sbt\_horizontal\_flag ) { |  |
| trafoWidth = tbWidth \* SbtNumFourthsTb0 / 4 |  |
| transform\_unit( x0, y0, trafoWidth, tbHeight, treeType , 0 ) |  |
| transform\_unit( x0 + trafoWidth, y0, tbWidth − trafoWidth , tbHeight, treeType , 1 ) |  |
| } else { |  |
| trafoHeight = tbHeight \* SbtNumFourthsTb0 / 4 |  |
| transform\_unit( x0, y0, tbWidth, trafoHeight, treeType , 0 ) |  |
| transform\_unit( x0, y0 + trafoHeight, tbWidth, tbHeight − trafoHeight, treeType , 1 ) |  |
| } |  |
| } else if( IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ) { |  |
| trafoHeight = tbHeight / NumIntraSubPartitions |  |
| for( partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++ ) |  |
| transform\_unit( x0, y0 + trafoHeight \* partIdx, tbWidth, trafoHeight, treeType, partIdx ) |  |
| } else if( IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ) { |  |
| trafoWidth = tbWidth / NumIntraSubPartitions |  |
| for( partIdx = 0; partIdx < NumIntraSubPartitions; partIdx++ ) |  |
| transform\_unit( x0 + trafoWidth \* partIdx, y0, trafoWidth, tbHeight, treeType, partIdx ) |  |
| } |  |
| } |  |

#### Transform unit syntax

|  |  |
| --- | --- |
| transform\_unit( x0, y0, tbWidth, tbHeight, treeType, subTuIndex ) { | Descriptor |
| if( ( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_CHROMA ) &&   ChromaArrayType != 0 ) { |  |
| if( ( IntraSubPartitionsSplitType =  = ISP\_NO\_SPLIT && !( cu\_sbt\_flag &&   ( ( subTuIndex  = = 0  &&  cu\_sbt\_pos\_flag )  | |    ( subTuIndex  = = 1  &&  !cu\_sbt\_pos\_flag ) ) ) )  | |    ( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   ( subTuIndex = = NumIntraSubPartitions − 1 ) ) ) { |  |
| **tu\_cbf\_cb**[ x0 ][ y0 ] | ae(v) |
| **tu\_cbf\_cr**[ x0 ][ y0 ] | ae(v) |
| } |  |
| } |  |
| if( treeType = = SINGLE\_TREE | | treeType = = DUAL\_TREE\_LUMA ) { |  |
| if( ( IntraSubPartitionsSplitType =  = ISP\_NO\_SPLIT && !( cu\_sbt\_flag &&   ( ( subTuIndex  = = 0  &&  cu\_sbt\_pos\_flag )  | |    ( subTuIndex  = = 1  &&  !cu\_sbt\_pos\_flag ) ) ) &&   ( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA | |   tu\_cbf\_cb[ x0 ][ y0 ] | | tu\_cbf\_cr[ x0 ][ y0 ] | |   CbWidth[ x0 ][ y0 ] > MaxTbSizeY | | CbHeight[ x0 ][ y0 ] > MaxTbSizeY ) )  | |    ( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   ( subTuIndex < NumIntraSubPartitions − 1 | | !InferTuCbfLuma ) ) ) |  |
| **tu\_cbf\_luma**[ x0 ][ y0 ] | ae(v) |
| if (IntraSubPartitionsSplitType != ISP\_NO\_SPLIT ) |  |
| InferTuCbfLuma = InferTuCbfLuma && !tu\_cbf\_luma[ x0 ][ y0 ] |  |
| } |  |
| if( IntraSubPartitionsSplitType != ISP\_NO\_SPLIT &&   treeType = = SINGLE\_TREE && subTuIndex = = NumIntraSubPartitions − 1 ) ) |  |
| xC = CbPosX[ x0 ][ y0 ] |  |
| yC = CbPosY[ x0 ][ y0 ] |  |
| wC = CbWidth[ x0 ][ y0 ] / SubWidthC |  |
| hC = CbHeight[ x0 ][ y0 ] / SubHeightC |  |
| } else |  |
| xC = x0 |  |
| yC = y0 |  |
| wC = tbWidth / SubWidthC |  |
| hC = tbHeight / SubHeightC |  |
| } |  |
| if( ( tu\_cbf\_luma[ x0 ][ y0 ] | | tu\_cbf\_cb[ x0 ][ y0 ] | | tu\_cbf\_cr[ x0 ][ y0 ] )  &&  treeType != DUAL\_TREE\_CHROMA ) { |  |
| if( cu\_qp\_delta\_enabled\_flag && !IsCuQpDeltaCoded ) { |  |
| **cu\_qp\_delta\_abs** | ae(v) |
| if( cu\_qp\_delta\_abs ) |  |
| **cu\_qp\_delta\_sign\_flag** | ae(v) |
| } |  |
| } |  |
| if( tu\_cbf\_luma[ x0 ][ y0 ]   &&  treeType != DUAL\_TREE\_CHROMA  &&  ( tbWidth  <=  32 )  &&  ( tbHeight  <=  32 )   &&  ( IntraSubPartitionsSplit[ x0 ][ y0 ]  = =  ISP\_NO\_SPLIT ) &&  ( !cu\_sbt\_flag ) ) { |  |
| if( transform\_skip\_enabled\_flag && tbWidth <= MaxTsSize && tbHeight <= MaxTsSize ) |  |
| **transform\_skip\_flag**[ x0 ][ y0 ] | ae(v) |
| if( (( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA && sps\_explicit\_mts\_inter\_enabled\_flag )   | | ( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA && sps\_explicit\_mts\_intra\_enabled\_flag ))  && ( !transform\_skip\_flag[ x0 ][ y0 ] ) ) |  |
| **tu\_mts\_idx**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( tu\_cbf\_luma[ x0 ][ y0 ] ) { |  |
| if( !transform\_skip\_flag[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, Log2( tbWidth ), Log2( tbHeight ), 0 ) |  |
| else |  |
| residual\_ts\_coding( x0, y0, Log2( tbWidth ), Log2( tbHeight ), 0 ) |  |
| } |  |
| if( tu\_cbf\_cb[ x0 ][ y0 ] ) |  |
| residual\_coding( xC, yC, Log2( wC ), Log2( hC ), 1 ) |  |
| if( tu\_cbf\_cr[ x0 ][ y0 ] ) { |  |
| if( tu\_cbf\_cb[ x0 ][ y0 ] ) |  |
| **tu\_joint\_cbcr\_residual**[ x0 ][ y0 ] | ae(v) |
| if( !tu\_joint\_cbcr\_residual[ x0 ][ y0 ] ) |  |
| residual\_coding( xC, yC, Log2( wC ), Log2( hC ), 2 ) |  |
| } |  |
| } |  |

#### Residual coding syntax

|  |  |  |
| --- | --- | --- |
| residual\_coding( x0, y0, log2TbWidth, log2TbHeight, cIdx ) { | Descriptor | |
| if( ( tu\_mts\_idx[ x0 ][ y0 ] > 0 | |  ( cu\_sbt\_flag  &&  log2TbWidth < 6  &&  log2TbHeight < 6 ) ) && cIdx = = 0 && log2TbWidth > 4 ) |  | |
| log2ZoTbWidth = 4 |  | |
| else |  | |
| log2ZoTbWidth = Min( log2TbWidth, 5 ) |  | |
| MaxCcbs = 2 \* ( 1 << log2TbWidth ) \* ( 1<< log2TbHeight ) |  | |
| if( tu\_mts\_idx[ x0 ][ y0 ] > 0 | |  ( cu\_sbt\_flag  &&  log2TbWidth < 6  &&  log2TbHeight < 6 ) )  && cIdx = = 0 && log2TbHeight > 4 ) |  | |
| log2ZoTbHeight = 4 |  | |
| else |  | |
| log2ZoTbHeight = Min( log2TbHeight, 5 ) |  | |
| if( log2TbWidth > 0 ) |  | |
| **last\_sig\_coeff\_x\_prefix** | ae(v) |
| if( log2TbHeight > 0 ) |  |
| **last\_sig\_coeff\_y\_prefix** | ae(v) |
| if( last\_sig\_coeff\_x\_prefix > 3 ) |  |
| **last\_sig\_coeff\_x\_suffix** | ae(v) |
| if( last\_sig\_coeff\_y\_prefix > 3 ) |  |
| **last\_sig\_coeff\_y\_suffix** | ae(v) |
| log2TbWidth = log2ZoTbWidth |  |
| log2TbHeight = log2ZoTbHeight |  |
| log2SbW = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) |  |
| log2SbH = log2SbW |  |
| if( log2TbWidth + log2TbHeight > 3 ) { |  |
| if( log2TbWidth < 2 ) { |  |
| log2SbW = log2TbWidth |  |
| log2SbH = 4 − log2SbW |  |
| } else if( log2TbHeight < 2 ) { |  |
| log2SbH = log2TbHeight |  |
| log2SbW = 4 − log2SbH |  |
| } |  |
| } |  |
| numSbCoeff = 1 << ( log2SbW + log2SbH ) |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock = ( 1  <<  ( log2TbWidth + log2TbHeight − ( log2SbW + log2SbH ) ) ) − 1 |  |
| do { |  |
| if( lastScanPos = = 0 ) { |  |
| lastScanPos = numSbCoeff |  |
| lastSubBlock− − |  |
| } |  |
| lastScanPos− − |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ lastSubBlock ][ 1 ] |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ lastScanPos ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ lastScanPos ][ 1 ] |  |
| } while( ( xC != LastSignificantCoeffX ) | | ( yC != LastSignificantCoeffY ) ) |  |
| QState = 0 |  |
| for( i = lastSubBlock; i >= 0; i− − ) { |  |
| startQStateSb = QState |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ i ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbW ][ log2TbHeight − log2SbH ]  [ i ][ 1 ] |  |
| inferSbDcSigCoeffFlag = 0 |  |
| if( ( i < lastSubBlock ) && ( i > 0 ) ) { |  |
| **coded\_sub\_block\_flag**[ xS ][ yS ] | ae(v) |
| inferSbDcSigCoeffFlag = 1 |  |
| } |  |
| firstSigScanPosSb = numSbCoeff |  |
| lastSigScanPosSb = −1 |  |
| remBinsPass1 = ( ( log2SbW + log2SbH ) < 4 ? 8 : 32 ) |  |
| firstPosMode0 = ( i = = lastSubBlock ? lastScanPos : numSbCoeff − 1 ) |  |
| firstPosMode1 = −1 |  |
| for( n = firstPosMode0; n >= 0 && remBinsPass1 >= 4; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( coded\_sub\_block\_flag[ xS ][ yS ] && ( n > 0 | | !inferSbDcSigCoeffFlag ) &&   ( xC != LastSignificantCoeffX | | yC != Last SignificantCoeffY ) ) { |  |
| **sig\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| remBinsPass1− − |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) |  |
| inferSbDcSigCoeffFlag = 0 |  |
| } |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) { |  | |
| if( !transform\_skip\_flag[ x0 ][ y0 ] ) { |  | |
| numSigCoeff++ |  | |
| if( ( n  >=  8 && i = = 0 && ( log2TbWidth = = 2 | | log2TbWidth = = 3 ) && ( log2TbWidth = = log2TbHeight ) ) | | ( ( i = = 1 | | i = = 2 )   && log2TbWidth >= 3 && log2TbHeight >= 3 ) ) |  | |
| numZeroOutSigCoeff++ |  | |
| } |  | |
| **abs\_level\_gtx\_flag**[ n ][ 0 ] | ae(v) |
| remBinsPass1− − |  |
| if( abs\_level\_gtx\_flag[ n ][ 0 ] ) { |  |
| **par\_level\_flag**[ n ] | ae(v) |
| remBinsPass1− − |  |
| **abs\_level\_gtx\_flag**[ n ][ 1 ] | ae(v) |
| remBinsPass1− − |  |
| } |  |
| if( lastSigScanPosSb = = −1 ) |  |
| lastSigScanPosSb = n |  |
| firstSigScanPosSb = n |  |
| } |  |
| AbsLevelPass1[ xC ][ yC ] = sig\_coeff\_flag[ xC ][ yC ] + par\_level\_flag[ n ] +   abs\_level\_gtx\_flag[ n ][ 0 ] + 2 \* abs\_level\_gtx\_flag[ n ][ 1 ] |  |
| if( dep\_quant\_enabled\_flag ) |  |
| QState = QStateTransTable[ QState ][ AbsLevelPass1[ xC ][ yC ] & 1 ] |  |
| if( remBinsPass1 < 4 ) |  |
| firstPosMode1 = n − 1 |  |
| } |  |
| for( n = numSbCoeff − 1; n >= firstPosMode1; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( abs\_level\_gtx\_flag[ n ][ 1 ] ) |  |
| **abs\_remainder**[ n ] | ae(v) |
| AbsLevel[ xC ][ yC ] = AbsLevelPass1[ xC ][ yC ] +2 \* abs\_remainder[ n ] |  |
| } |  |
| for( n = firstPosMode1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| **dec\_abs\_level**[ n ] | ae(v) |
| if(AbsLevel[ xC ][ yC ] > 0 ) |  |
| firstSigScanPosSb = n |  |
| if( dep\_quant\_enabled\_flag ) |  |
| QState = QStateTransTable[ QState ][ AbsLevel[ xC ][ yC ] & 1 ] |  |
| } |  |
| if( dep\_quant\_enabled\_flag | | !sign\_data\_hiding\_enabled\_flag ) |  |
| signHidden = 0 |  |
| else |  |
| signHidden = ( lastSigScanPosSb − firstSigScanPosSb > 3 ? 1 : 0 ) |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( ( AbsLevel[ xC ][ yC ] > 0 ) &&   ( !signHidden | | ( n != firstSigScanPosSb ) ) ) |  |
| **coeff\_sign\_flag**[ n ] | ae(v) |
| } |  |
| if( dep\_quant\_enabled\_flag ) { |  |
| QState = startQStateSb |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( AbsLevel[ xC ][ yC ] > 0 ) |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =  ( 2 \* AbsLevel[ xC ][ yC ] − ( QState > 1 ? 1 : 0 ) ) \*  ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| QState = QStateTransTable[ QState ][ par\_level\_flag[ n ] ] |  |
| } else { |  |
| sumAbsLevel = 0 |  |
| for( n = numSbCoeff − 1; n >= 0; n− − ) { |  |
| xC = ( xS << log2SbW ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 0 ] |  |
| yC = ( yS << log2SbH ) + DiagScanOrder[ log2SbW ][ log2SbH ][ n ][ 1 ] |  |
| if( AbsLevel[ xC ][ yC ] > 0 ) { |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   AbsLevel[ xC ][ yC ] \* ( 1 − 2 \* coeff\_sign\_flag[ n ] ) |  |
| if( signHidden ) { |  |
| sumAbsLevel += AbsLevel[ xC ][ yC ] |  |
| if( ( n = = firstSigScanPosSb ) && ( sumAbsLevel % 2 ) = = 1 ) ) |  |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] =   −TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  | |

|  |  |
| --- | --- |
| residual\_ts\_coding( x0, y0, log2TbWidth, log2TbHeight, cIdx ) { | Descriptor |
| log2SbSize = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) |  |
| numSbCoeff = 1 << ( log2SbSize << 1 ) |  |
| lastSubBlock = ( 1  <<  ( log2TbWidth + log2TbHeight − 2 \* log2SbSize ) ) − 1 |  |
| inferSbCbf = 1 |  |
| MaxCcbs = 2 \* ( 1 << log2TbWidth ) \* ( 1<< log2TbHeight ) |  |
| for( i =0; i <= lastSubBlock; i++ ) { |  |
| xS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ][ i ][ 0 ] |  |
| yS = DiagScanOrder[ log2TbWidth − log2SbSize ][ log2TbHeight − log2SbSize ][ i ][ 1 ] |  |
| if( ( i != lastSubBlock | | !inferSbCbf ) { |  |
| **coded\_sub\_block\_flag**[ xS ][ yS ] | ae(v) |
| MaxCcbs− − |  |
| } |  |
| if( coded\_sub\_block\_flag[ xS ][ yS ] && i < lastSubBlock ) |  |
| inferSbCbf = 0 |  |
| /\* First scan pass \*/ |  |
| inferSbSigCoeffFlag = 1 |  |
| for( n = 0; n <= numSbCoeff − 1; n++ ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( coded\_sub\_block\_flag[ xS ][ yS ] &&   ( n != numSbCoeff − 1 | | !inferSbSigCoeffFlag ) ) { |  |
| **sig\_coeff\_flag**[ xC ][ yC ] | ae(v) |
| MaxCcbs− − |  |
| if( sig\_coeff\_flag[ xC ][ yC ] ) |  |
| inferSbSigCoeffFlag = 0 |  |
| } |  |
| if( sig\_coeff\_flag[ xC ][ yC ] { |  |
| **coeff\_sign\_flag**[ n ] | ae(v) |
| MaxCcbs− − |  |
| **abs\_level\_gtx\_flag**[ n ][ 0 ] | ae(v) |
| MaxCcbs− − |  |
| if( abs\_level\_gtx\_flag[ n ][ 0 ] ) { |  |
| **par\_level\_flag**[ n ] | ae(v) |
| MaxCcbs− − |  |
| } |  |
| } |  |
| AbsLevelPassX[ xC ][ yC ] =   sig\_coeff\_flag[ xC ][ yC ] + par\_level\_flag[ n ] + abs\_level\_gtx\_flag[ n ][ 0 ] |  |
| } |  |
| /\* Greater than X scan passes (numGtXFlags=5) \*/ |  |
| for( j = 1; j < 5; j++ ) { |  |
| for( n = 0; n <= numSbCoeff − 1; n++ ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( abs\_level\_gtx\_flag[ n ][ j − 1 ] ) |  |
| **abs\_level\_gtx\_flag**[ n ][ j ] | ae(v) |
| MaxCcbs− − |  |
| AbsLevelPassX[ xC ][ yC ] + = 2 \* abs\_level\_gtx\_flag[ n ][ j ] |  |
| } |  |
| } |  |
| /\* remainder scan pass \*/ |  |
| for( n = 0; n <= numSbCoeff − 1; n++ ) { |  |
| xC = ( xS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 0 ] |  |
| yC = ( yS << log2SbSize ) + DiagScanOrder[ log2SbSize ][ log2SbSize ][ n ][ 1 ] |  |
| if( abs\_level\_gtx\_flag[ n ][ 4 ] ) |  |
| **abs\_remainder**[ n ] | ae(v) |
| TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] = ( 1 − 2 \* coeff\_sign\_flag[ n ] ) \*   ( AbsLevelPassX[ xC ][ yC ] + abs\_remainder[ n ] ) |  |
| } |  |
| } |  |
| } |  |

## Semantics

### General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this clause. 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 Specification.

### NAL unit semantics

#### General NAL unit semantics

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

NOTE 1 – The video coding layer (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 specified in Annex B.

**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 string of data bits **(**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.

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 are 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 is as specified in Table 7‑1.

NOTE 2 – 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

#### NAL unit header semantics

**zero\_tid\_required\_flag** equal to 0 specifies that zero\_tid\_required\_flag does not impose any additional constraints on the value of nuh\_temporal\_id1\_plus1.

**nuh\_temporal\_id\_plus1** minus 1 specifies a temporal identifier for the NAL unit.

The value of nuh\_temporal\_id\_plus1 shall not be equal to 0. When zero\_tid\_required\_flag is equal to 1, the value of nuh\_temporal\_id\_plus1 shall be equal to 1.

The variable TemporalId is derived as follows:

TemporalId = nuh\_temporal\_id\_plus1 − 1 (7‑1)

NOTE 1 – NAL unit types in the range of 16 to 31, inclusive, have zero\_tid\_required\_flag equal to 1, and consequently have TemporalId equal to 0.

The value of TemporalId shall be the same for all VCL NAL units of a layer access unit. The value of TemporalId of a coded picture or a layer access unit is the value of the TemporalId of the VCL NAL units of the coded picture or the layer access unit. [Ed. (YK): Check whether all layer AUs in an AU should be required to have the same value of TemporalId.]

The value of TemporalId for non-VCL NAL units is constrained as follows:

– If NalUnitType is equal to SPS\_NUT, TemporalId is equal to 0 and the TemporalId of the layer access unit containing the NAL unit shall be equal to 0.

– Otherwise, if NalUnitType is equal to APS\_NUT, TemporalId shall be equal to that of the layer access unit containing the NAL unit.

– Otherwise, when NalUnitType is not equal to EOS\_NUT and not equal to EOB\_NUT, TemporalId shall be greater than or equal to the TemporalId of the layer access unit containing the NAL unit.

NOTE 2 – When the NAL unit is a non-VCL NAL unit, the value of TemporalId is equal to the minimum value of the TemporalId values of all layer access units to which the non-VCL NAL unit applies. When NalUnitType is equal to PPS\_NUT, TemporalId may be greater than or equal to the TemporalId of the containing layer access unit, as all picture parameter sets (PPSs) may be included in the beginning of a bitstream, wherein the first coded picture has TemporalId equal to 0. When NalUnitType is equal to PREFIX\_SEI\_NUT or SUFFIX\_SEI\_NUT, TemporalId may be greater than or equal to the TemporalId of the containing layer access unit, as an SEI NAL unit may contain information that applies to a bitstream subset that includes layer access units for which the TemporalId values are greater than the TemporalId of the layer access unit containing the SEI NAL unit.

**nal\_unit\_type\_lsb** specifies the least significant bits for the NAL unit type.

The variable NalUnitType, which specifies the NAL unit type, i.e., the type of RBSP data structure contained in the NAL unit as specified in Table 7‑1, is derived as follows:

NalUnitType = ( zero\_tid\_required\_flag << 4 ) + nal\_unit\_type\_lsb (7‑2)

NAL units that have NalUnitType in the range of UNSPEC28..UNSPEC31, inclusive, for which semantics are not specified, shall not affect the decoding process specified in this Specification.

NOTE 3 – NAL unit types in the range of UNSPEC28..UNSPEC31 may be used as determined by the application. No decoding process for these values of NalUnitType is specified in this Specification. Since different applications might use these NAL unit types for different purposes, particular care must be exercised in the design of encoders that generate NAL units with these NalUnitType values, and in the design of decoders that interpret the content of NAL units with these NalUnitType values. This Specification does not define any management for these values. These NalUnitType values might only be suitable for use in contexts in which "collisions" of usage (i.e., different definitions of the meaning of the NAL unit content for the same NalUnitType value) are unimportant, or not possible, or are managed – e.g., defined or managed in the controlling application or transport specification, or by controlling the environment in which bitstreams are distributed.

For purposes other than determining the amount of data in the decoding units of the bitstream (as specified in Annex C), decoders shall ignore (remove from the bitstream and discard) the contents of all NAL units that use reserved values of NalUnitType.

NOTE 4 – This requirement allows future definition of compatible extensions to this Specification.

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

|  |  |  |  |
| --- | --- | --- | --- |
| **NalUnitType** | **Name of NalUnitType** | **Content of NAL unit and RBSP syntax structure** | **NAL unit type class** |
| 0 | PPS\_NUT | Picture parameter set pic\_parameter\_set\_rbsp( ) | non-VCL |
| 1 | AUD\_NUT | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-VCL |
| 2 3 | PREFIX\_SEI\_NUT SUFFIX\_SEI\_NUT | Supplemental enhancement information sei\_rbsp( ) | non-VCL |
| 4 | APS\_NUT | Adaptation parameter set adaptation\_parameter\_set\_rbsp( ) | non-VCL |
| 6 5..7 | RSV\_NVCL65.. RSV\_NVCL7 | Reserved | non-VCL |
| 8 | TRAIL\_NUT | Coded slice of a non-STSA trailing picture slice\_layer\_rbsp( ) | VCL |
| 9 | STSA\_NUT | Coded slice of an STSA picture slice\_layer\_rbsp( ) | VCL |
| 10 | RADL\_NUT | Coded slice of a RADL picture slice\_layer\_rbsp( ) | VCL |
| 11 | RASL\_NUT | Coded slice of a RASL picture slice\_layer\_rbsp( ) | VCL |
| 12..15 | RSV\_VCL\_12.. RSV\_VCL\_15 | Reserved non-IRAP VCL NAL unit types | VCL |
| 16 | DPS\_NUT | Decoding parameter set decoding\_parameter\_set\_rbsp( ) | non-VCL |
| 17 | SPS\_NUT | Sequence parameter set seq\_parameter\_set\_rbsp( ) | non-VCL |
| 18 | EOS\_NUT | End of sequence end\_of\_seq\_rbsp( ) | non-VCL |
| 19 | EOB\_NUT | End of bitstream end\_of\_bitstream\_rbsp( ) | non-VCL |
| 20 | VPS\_NUT | Video parameter set video\_parameter\_set\_rbsp( ) | non-VCL |
| 21..23 | RSV\_NVCL21.. RSV\_NVCL23 | Reserved | non-VCL |
| 24 25 | IDR\_W\_RADL IDR\_N\_LP | Coded slice of an IDR picture slice\_layer\_rbsp( ) | VCL |
| 26 | CRA\_NUT | Coded slice of a CRA picture slice\_layer\_rbsp( ) | VCL |
| 27 | GRA\_NUT | Coded slice of a gradual random access picture  slice\_layer\_rbsp( ) | VCL |
| 28..31 | UNSPEC28.. UNSPEC31 | Unspecified | non-VCL |

NOTE 5 – A clean random access (CRA) picture may have associated RASL or RADL pictures present in the bitstream. [Ed. (YK): The NOTE index is one greater than the value it should be. Removing of the table would make it correct when updating the fields, but ....]

NOTE 6 – An instantaneous decoding refresh (IDR) picture having NalUnitType equal to IDR\_N\_LP does not have associated leading pictures present in the bitstream. An IDR picture having NalUnitType equal to IDR\_W\_RADL does not have associated RASL pictures present in the bitstream, but may have associated RADL pictures in the bitstream.

**nuh\_layer\_id\_plus1** minus 1 specifies the identifier of the layer to which a VCL NAL unit belongs or the identifier of a layer to which a non-VCL NAL unit applies. The value of nuh\_layer\_id\_plus1 shall be in the range of 1 to 126, inclusive. The value of 127 may be specified in the future by ITU-T | ISO/IEC. For purposes other than determining the amount of data in the decoding units of the bitstream, decoders shall ignore all data that follow the value 127 for nuh\_layer\_id\_plus1 in a NAL unit.

NOTE 7 – The value of 127 for nuh\_layer\_id\_plus1 may be used to indicate an extended layer identifier in a future extension of this Specification.

The variable NuhLayerId is derived as follows:

NuhLayerId = nuh\_layer\_id\_plus1 − 1 (7‑3)

The value of NuhLayerId shall be the same for all VCL NAL units of a coded picture. The value of NuhLayerId of a coded picture is the value of the NuhLayerId of the VCL NAL units of the coded picture.

**nuh\_reserved\_zero\_bit** shall be equal to '0'. The value 1 of nuh\_reserved\_zero\_bit 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 nuh\_reserved\_zero\_bit equal to '1'.

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

This clause does not form an integral part of this Specification.

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 greater 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 are searched for byte-aligned bits of the following binary patterns:

'00000000 00000000 000000xx' (where 'xx' represents any two-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 with the NAL unit header, within which the NalUnitType indicates the type of RBSP data structure in the NAL unit.

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 both of the following:

– 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, layer access units, access units, and coded video sequences

##### General

This clause 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 clauses 7.3, D.2, E.2, and F.2 specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

##### Order of DPS, VPS, SPS, PPS and APS RBSPs and their activation

TBD.

##### Order of access units and association to CVSs

A bitstream conforming to this Specification consists of one or more CVSs.

A CVS consists of one or more access units. The order of NAL units and coded pictures and their association to access units is described in clause 7.4.2.4.4.

The first access unit of a CVS is a CVSS access unit, wherein each present layer access unit is a CLVSS layer access unit, which is either an IRAP layer access unit with NoIncorrectPicOutputFlag equal to 1 or a GRA layer access unit with NoIncorrectPicOutputFlag equal to 1.

It is a requirement of bitstream conformance that, when present, each layer access unit in the next access unit after an access unit that contains an end of sequence NAL unit or an end of bitstream NAL unit shall be an IRAP layer access unit, which may be an IDR layer access unit or a CRA layer access unit, or a GRA layer access unit.

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

This clause specifies the order of NAL units and coded pictures and their association to layer access units and access units for CVSs that conform to one or more of the profiles specified in Annex A and that are decoded using the decoding process specified in clauses 2 through 10.

A layer access unit consists of one coded picture, zero or more VCL NAL units and zero or more non-VCL NAL units. The association of VCL NAL units to coded pictures is described in clause 7.4.2.4.5.

An access unit consists of one or more layer access units in increasing order of NuhLayerId.

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

Let firstVclNalUnitInAu be a VCL NAL unit that is the first VCL NAL unit of a coded picture and for which the derived PicOrderCntVal differs from the PicOrderCntVal of the previous coded picture. The first of any of the following NAL units preceding firstVclNalUnitInAu and succeeding the last VCL NAL unit preceding firstVclNalUnitInAu, if any, specifies the start of a new access unit:

– access unit delimiter NAL unit (when present),

– DPS NAL unit (when present),

– VPS NAL unit (when present),

– SPS NAL unit (when present),

– PPS NAL unit (when present),

– APS NAL unit (when present),

– Prefix SEI NAL unit (when present),

– NAL units with NalUnitType equal to RSV\_NVCL\_5, RSV\_NVCL\_6, RSV\_NVCL\_21, or RSV\_NVCL\_22 (when present),

– NAL units with NalUnitType in the range of UNSPEC28..UNSPEC29 (when present).

NOTE – The first NAL unit preceding firstVclNalUnitInAu and succeeding the last VCL NAL unit preceding firstVclNalUnitInAu, if any, can only be one of the above-listed NAL units.

When there is none of the above NAL units preceding firstVclNalUnitInAu and succeeding the last VCL NAL preceding firstVclNalUnitInAu, if any, firstVclNalUnitInAu starts a new access unit.

The order of the coded pictures and non-VCL NAL units within a layer access unit or an access unit shall obey the following constraints:

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

– When any DPS NAL units, VPS NAL units, SPS NAL units, PPS NAL units, APS NAL units, prefix SEI NAL units, NAL units with NalUnitType equal to RSV\_NVCL\_5, RSV\_NVCL\_6, RSV\_NVCL\_21, or RSV\_NVCL\_22, or NAL units with NalUnitType in the range of UNSPEC28..UNSPEC29 are present in a layer access unit, they shall not follow the last VCL NAL unit of the layer access unit.

– NAL units having NalUnitType equal to SUFFIX\_SEI\_NUT, RSV\_NVCL\_7, or RSV\_NVCL\_23, or in the range of UNSPEC30..UNSPEC31 in a layer access unit shall not precede the first VCL NAL unit of the layer access unit.

– When an end of sequence NAL unit is present in an access unit, it shall be the last NAL unit among all NAL units with in the access unit other than an end of bitstream NAL unit (when present).

– When an end of bitstream NAL unit is present in an access unit, it shall be the last NAL unit in the access unit.

[Ed. A Figure should be added here]

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

TBD.

### Raw byte sequence payloads, trailing bits and byte alignment semantics

#### Decoding parameter set RBSP semantics

NOTE 1 – DPS NAL units are required to be available to the decoding process prior to their activation (either in the bitstream or by external means), as specified in clause 7.4.2.4.2. However, the DPS RBSP contains information that is not necessary for operation of the decoding process specified in clauses 2 through 10 of this Specification.

**dps\_decoding\_parameter\_set\_id** identifies the DPS for reference by other syntax elements. The value of dps\_decoding\_parameter\_set\_id shall not be equal to 0.

**dps\_max\_sub\_layers\_minus1** plus 1 specifies the maximum number of temporal sub-layers that may be present in each CVS referring to the DPS. The value of dps\_max\_sub\_layers\_minus1 shall be in the range of 0 to 6, inclusive.

**dps\_reserved\_zero\_bit** shall be equal to 0 in bitstreams conforming to this version of this Specification. The value 0 for dps\_reserved\_zero\_bit is reserved for future use by ITU-T | ISO/IEC.

**dps\_extension\_flag** equal to 0 specifies that no dps\_extension\_data\_flag syntax elements are present in the DPS RBSP syntax structure. dps\_extension\_flag equal to 1 specifies that there are dps\_extension\_data\_flag syntax elements present in the DPS RBSP syntax structure.

**dps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in Annex A. Decoders conforming to this version of this Specification shall ignore all dps\_extension\_data\_flag syntax elements.

#### Video parameter set RBSP semantics

**vps\_video\_parameter\_set\_id** provides an identifier for the VPS for reference by other syntax elements.

**vps\_max\_layers\_minus1** plus 1 specifies the maximum allowed number of layers in each CVS referring to the VPS. vps\_max\_layers\_minus1 shall be less than 126 in bitstreams conforming to this version of this Specification. The value of 126 for vps\_max\_layers\_minus1 is reserved for future use by ITU-T | ISO/IEC. Although the value of vps\_max\_layers\_minus1 is required to be less than 126 in this version of this Specification, decoders shall allow the value of vps\_max\_layers\_minus1 equal to 126 to appear in the syntax.

NOTE 1 – The value of 126 for vps\_max\_layers\_minus1 may be used to indicate an extended number of layers in a future extension where more than 126 layers in a bitstream need to be supported.

The variable MaxLayersMinus1 is set equal to Min( 125, vps\_max\_layers\_minus1 ).

**vps\_included\_layer\_id**[ i ] specifies the NuhLayerId value of the i-th layer in each CVS referring to the VPS.

**vps\_reserved\_zero\_bit** shall be equal to 0 in bitstreams conforming to this version of this Specification. The value 1 for vps\_reserved\_zero\_bit is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of vps\_reserved\_zero\_bit.

NOTE 2 – The value of 1 for vps\_reserved\_zero\_bit may be used to indicate support for inter-layer prediction in a future extension.

**vps\_constraint\_info\_present\_flag** equal to 1 specifies that the general\_constraint\_info( ) syntax structure is present in the VPS. vps\_constraint\_info\_present\_flag equal to 0 specifies that the general\_constraint\_info( ) syntax structure is not present in the VPS.

**vps\_reserved\_zero\_7bits** shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for vps\_reserved\_zero\_7bits are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of vps\_reserved\_zero\_7bits.

**vps\_extension\_flag** equal to 0 specifies that no vps\_extension\_data\_flag syntax elements are present in the VPS RBSP syntax structure. vps\_extension\_flag equal to 1 specifies that there are vps\_extension\_data\_flag syntax elements present in the VPS RBSP syntax structure.

**vps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all vps\_extension\_data\_flag syntax elements.

#### Sequence parameter set RBSP semantics

**sps\_decoding\_parameter\_set\_id**, when greater than 0, specifies the value of dps\_decoding\_parameter\_set\_id for the DPS referred to by the SPS. When sps\_decoding\_parameter\_set\_id is equal to 0, the SPS does not refer to a DPS and no DPS is active when decoding each CVS referring to the SPS.

**sps\_video\_parameter\_set\_id**, when greater than 0, specifies the value of vps\_video\_parameter\_set\_id for the VPS referred to by the SPS. When sps\_video\_parameter\_set\_id is equal to 0, the SPS does not refer to a VPS and no VPS is active when decoding each CVS referring to the SPS.

**sps\_max\_sub\_layers\_minus1** plus 1 specifies the maximum number of temporal sub-layers that may be present in each CVS referring to the SPS. The value of sps\_max\_sub\_layers\_minus1 shall be in the range of 0 to 6, inclusive.

**sps\_reserved\_zero\_5bits** shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for sps\_reserved\_zero\_5bits are reserved for future use by ITU-T | ISO/IEC.

**gra\_enabled\_flag** equal to 1 specifies that GRA pictures may be present in CVSs referring to the SPS. gra\_enabled\_flag equal to 0 specifies that GRA pictures are not present in CVSs referring to the SPS.

**sps\_seq\_parameter\_set\_id** provides an identifier for the SPS for reference by other syntax elements. The value of sps\_seq\_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 clause 6.2. The value of chroma\_format\_idc shall be in the range of 0 to 3, inclusive.

**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) and uses the monochrome coding syntax. In this case, each colour plane is associated with a specific colour\_plane\_id value.

NOTE 1 – 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.

**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 MinCbSizeY.

**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 MinCbSizeY.

**conformance\_window\_flag** equal to 1 indicates that the conformance cropping window offset parameters follow next in the SPS. conformance\_window\_flag equal to 0 indicates that the conformance cropping window offset parameters are not present.

**conf\_win\_left\_offset**, **conf\_win\_right\_offset**, **conf\_win\_top\_offset**, and **conf\_win\_bottom\_offset** specify the samples of the pictures in the CVS that are output from the decoding process, in terms of a rectangular region specified in picture coordinates for output. When conformance\_window\_flag is equal to 0, the values of conf\_win\_left\_offset, conf\_win\_right\_offset, conf\_win\_top\_offset, and conf\_win\_bottom\_offset are inferred to be equal to 0.

The conformance cropping window contains the luma samples with horizontal picture coordinates from SubWidthC \* conf\_win\_left\_offset to pic\_width\_in\_luma\_samples − ( SubWidthC \* conf\_win\_right\_offset + 1 ) and vertical picture coordinates from SubHeightC \* conf\_win\_top\_offset to pic\_height\_in\_luma\_samples − ( SubHeightC \* conf\_win\_bottom\_offset + 1 ), inclusive.

The value of SubWidthC \* ( conf\_win\_left\_offset + conf\_win\_right\_offset ) shall be less than pic\_width\_in\_luma\_samples, and the value of SubHeightC \* ( conf\_win\_top\_offset + conf\_win\_bottom\_offset ) shall be less than pic\_height\_in\_luma\_samples.

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 2 – The conformance cropping window offset parameters are only applied at the output. All internal decoding processes are applied to the uncropped picture size.

**bit\_depth\_luma\_minus8** specifies the bit depth of the samples of the luma array BitDepthY and the value of the luma quantization parameter range offset QpBdOffsetY as follows:

BitDepthY = 8 + bit\_depth\_luma\_minus8 (7‑4)

QpBdOffsetY = 6 \* bit\_depth\_luma\_minus8 (7‑5)

bit\_depth\_luma\_minus8 shall be in the range of 0 to 8, inclusive.

**bit\_depth\_chroma\_minus8** specifies the bit depth of the samples of the chroma arrays BitDepthC and the value of the chroma quantization parameter range offset QpBdOffsetC as follows:

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (7‑6)

QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (7‑7)

bit\_depth\_chroma\_minus8 shall be in the range of 0 to 8, inclusive.

**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 ) (7‑8)

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

**sps\_sub\_layer\_ordering\_info\_present\_flag** equal to 1 specifies that sps\_max\_dec\_pic\_buffering\_minus1[ i ], sps\_max\_num\_reorder\_pics[ i ], and sps\_max\_latency\_increase\_plus1[ i ] are present for sps\_max\_sub\_layers\_minus1 + 1 sub-layers. sps\_sub\_layer\_ordering\_info\_present\_flag equal to 0 specifies that the values of sps\_max\_dec\_pic\_buffering\_minus1[ sps\_max\_sub\_layers\_minus1 ], sps\_max\_num\_reorder\_pics[ sps\_max\_sub\_layers\_minus1 ], and sps\_max\_latency\_increase\_plus1[ sps\_max\_sub\_layers\_minus1 ] apply to all sub-layers.

**sps\_max\_dec\_pic\_buffering\_minus1**[ i ] plus 1 specifies the maximum required size of the decoded picture buffer for the CVS in units of picture storage buffers when HighestTid is equal to i. The value of sps\_max\_dec\_pic\_buffering\_minus1[ i ] shall be in the range of 0 to MaxDpbSize − 1, inclusive, where MaxDpbSize is as specified somewhere else. [Ed. (YK): Change to reference to the clause specifying MaxDpbSize when available.] When i is greater than 0, sps\_max\_dec\_pic\_buffering\_minus1[ i ] shall be greater than or equal to sps\_max\_dec\_pic\_buffering\_minus1[ i − 1 ]. When sps\_max\_dec\_pic\_buffering\_minus1[ i ] is not present for i in the range of 0 to sps\_max\_sub\_layers\_minus1 − 1, inclusive, due to sps\_sub\_layer\_ordering\_info\_present\_flag being equal to 0, it is inferred to be equal to sps\_max\_dec\_pic\_buffering\_minus1[ sps\_max\_sub\_layers\_minus1 ].

**sps\_max\_num\_reorder\_pics**[ i ] indicates the maximum allowed number of pictures that can precede any picture in the CVS in decoding order and follow that picture in output order when HighestTid is equal to i. The value of sps\_max\_num\_reorder\_pics[ i ] shall be in the range of 0 to sps\_max\_dec\_pic\_buffering\_minus1[ i ], inclusive. When i is greater than 0, sps\_max\_num\_reorder\_pics[ i ] shall be greater than or equal to sps\_max\_num\_reorder\_pics[ i − 1 ]. When sps\_max\_num\_reorder\_pics[ i ] is not present for i in the range of 0 to sps\_max\_sub\_layers\_minus1 − 1, inclusive, due to sps\_sub\_layer\_ordering\_info\_present\_flag being equal to 0, it is inferred to be equal to sps\_max\_num\_reorder\_pics[ sps\_max\_sub\_layers\_minus1 ].

**sps\_max\_latency\_increase\_plus1**[ i ] not equal to 0 is used to compute the value of SpsMaxLatencyPictures[ i ], which specifies the maximum number of pictures that can precede any picture in the CVS in output order and follow that picture in decoding order when HighestTid is equal to i.

When sps\_max\_latency\_increase\_plus1[ i ] is not equal to 0, the value of SpsMaxLatencyPictures[ i ] is specified as follows:

SpsMaxLatencyPictures[ i ] = sps\_max\_num\_reorder\_pics[ i ] + sps\_max\_latency\_increase\_plus1[ i ] − 1 (7‑9)

When sps\_max\_latency\_increase\_plus1[ i ] is equal to 0, no corresponding limit is expressed.

The value of sps\_max\_latency\_increase\_plus1[ i ] shall be in the range of 0 to 232 − 2, inclusive. When sps\_max\_latency\_increase\_plus1[ i ] is not present for i in the range of 0 to sps\_max\_sub\_layers\_minus1 − 1, inclusive, due to sps\_sub\_layer\_ordering\_info\_present\_flag being equal to 0, it is inferred to be equal to sps\_max\_latency\_increase\_plus1[ sps\_max\_sub\_layers\_minus1 ].

**long\_term\_ref\_pics\_flag** equal to 0 specifies that no LTRP is used for inter prediction of any coded picture in the CVS. long\_term\_ref\_pics\_flag equal to 1 specifies that LTRPs may be used for inter prediction of one or more coded pictures in the CVS.

**sps\_idr\_rpl\_present\_flag** equal to 1 specifies that reference picture list syntax elements are present in slice headers of IDR pictures. sps\_idr\_rpl\_present\_flag equal to 0 specifies that reference picture list syntax elements are not present in slice headers of IDR pictures.

**rpl1\_same\_as\_rpl0\_flag** equal to 1 specifies that the syntax structures num\_ref\_pic\_lists\_in\_sps[ 1 ] and ref\_pic\_list\_struct( 1, rplsIdx ) are not present and the following applies:

– The value of num\_ref\_pic\_lists\_in\_sps[ 1 ] is inferred to be equal to the value of num\_ref\_pic\_lists\_in\_sps[ 0 ].

– The value of each of syntax elements in ref\_pic\_list\_struct( 1, rplsIdx ) is inferred to be equal to the value of corresponding syntax element in ref\_pic\_list\_struct( 0, rplsIdx ) for rplsIdx ranging from 0 to num\_ref\_pic\_lists\_in\_sps[ 0 ] − 1.

**num\_ref\_pic\_lists\_in\_sps**[ i ] specifies the number of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i included in the SPS. The value of num\_ref\_pic\_lists\_in\_sps[ i ] shall be in the range of 0 to 64, inclusive.

NOTE 3 – For each value of listIdx (equal to 0 or 1), a decoder should allocate memory for a total number of num\_ref\_pic\_lists\_in\_sps[ i ] + 1 ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures since there may be one ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure directly signalled in the slice headers of a current picture.

**qtbtt\_dual\_tree\_intra\_flag** equal to 1 specifies that for I slices, each CTU is split into coding units with 64x64 luma samples using an implicit quadtree split and that these coding units are the root of two separate coding\_tree syntax structure for luma and chroma. qtbtt\_dual\_tree\_intra\_flag equal to 0 specifies separate coding\_tree syntax structure is not used for I slices. When qtbtt\_dual\_tree\_intra\_flag is not present, it is inferred to be equal to 0.

**log2\_ctu\_size\_minus2** plus 2 specifies the luma coding tree block size of each CTU.

**log2\_min\_luma\_coding\_block\_size\_minus2** plus 2 specifies the minimum luma coding block size.

The variables CtbLog2SizeY, CtbSizeY, MinCbLog2SizeY, MinCbSizeY, MinTbLog2SizeY, MaxTbLog2SizeY, MinTbSizeY, MaxTbSizeY, PicWidthInCtbsY, PicHeightInCtbsY, PicSizeInCtbsY, PicWidthInMinCbsY, PicHeightInMinCbsY, PicSizeInMinCbsY, PicSizeInSamplesY, PicWidthInSamplesC and PicHeightInSamplesC are derived as follows:

CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 (7‑10)

CtbSizeY = 1  <<  CtbLog2SizeY (7‑11)

MinCbLog2SizeY = log2\_min\_luma\_coding\_block\_size\_minus2 + 2 (7‑12)

MinCbSizeY = 1  <<  MinCbLog2SizeY (7‑13)

MinTbLog2SizeY = 2 (7‑14)

MaxTbLog2SizeY = 6 (7‑15)

MinTbSizeY = 1  <<  MinTbLog2SizeY (7‑16)

MaxTbSizeY = 1  <<  MaxTbLog2SizeY (7‑17)

PicWidthInCtbsY = Ceil( pic\_width\_in\_luma\_samples ÷ CtbSizeY ) (7‑18)

PicHeightInCtbsY = Ceil( pic\_height\_in\_luma\_samples ÷ CtbSizeY ) (7‑19)

PicSizeInCtbsY = PicWidthInCtbsY \* PicHeightInCtbsY (7‑20)

PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY (7‑21)

PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY (7‑22)

PicSizeInMinCbsY = PicWidthInMinCbsY \* PicHeightInMinCbsY (7‑23)

PicSizeInSamplesY = pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples (7‑24)

PicWidthInSamplesC = pic\_width\_in\_luma\_samples / SubWidthC (7‑25)

PicHeightInSamplesC = pic\_height\_in\_luma\_samples / SubHeightC (7‑26)

[Ed. (BB): Currently the maximum transform size (64x64 luma samples and corresponding chroma sample size) and the minimum transform size (4x4 luma samples and corresponding chroma samples) is fixed, pending further specification development.]

The variables CtbWidthC and CtbHeightC, which specify the width and height, respectively, of the array for each chroma CTB, 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 follows:

CtbWidthC = CtbSizeY / SubWidthC (7‑27)

CtbHeightC = CtbSizeY / SubHeightC (7‑28)

For log2BlockWidth ranging from 0 to 4 and for log2BlockHeight ranging from 0 to 4, inclusive, the up-right diagonal and raster scan order array initialization process as specified in clause 6.5.2 is invoked with 1  <<  log2BlockWidth and 1  <<  log2BlockHeight as inputs, and the output is assigned to DiagScanOrder[ log2BlockWidth ][ log2BlockHeight ] and RasterScanOrder[ log2BlockWidth ][ log2BlockHeight ].

**partition\_constraints\_override\_enabled\_flag** equal to 1 specifies the presence of partition\_constraints\_override\_flag in the slice headers for slices referring to the SPS. partition\_constraints\_override\_enabled\_flag equal to 0 specifies the absence of partition\_constraints\_override\_flag in the slice headers for slices referring to the SPS.

**sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding block size in luma samples for luma CUs in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_min\_qt\_min\_cb\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

MinQtLog2SizeIntraY = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma + MinCbLog2SizeY (7‑29)

**sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum luma coding block size in luma samples for luma CUs in slices with slice\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_min\_qt\_min\_cb\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. The base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU is derived as follows:

MinQtLog2SizeInterY = sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice + MinCbLog2SizeY (7‑30)

**sps\_max\_mtt\_hierarchy\_depth\_inter\_slice** specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice\_max\_mtt\_hierarchy\_depth\_luma present in the slice header of the slices referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_inter\_slice shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

**sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_luma** specifies the default maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice\_max\_mtt\_hierarchy\_depth\_luma present in the slice header of the slices referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

**sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_bt\_min\_qt\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_tt\_min\_qt\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraY, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_bt\_min\_qt\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 0 (B) or 1 (P) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_tt\_min\_qt\_luma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeInterY, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice is inferred to be equal to 0.

**sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma** specifies the default difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL\_TREE\_CHROMA in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_min\_qt\_min\_cb\_chroma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma is inferred to be equal to 0. The base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a CTU with treeType equal to DUAL\_TREE\_CHROMA is derived as follows:

MinQtLog2SizeIntraC = sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma + MinCbLog2SizeY (7‑31)

**sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_chroma** specifies the default maximum hierarchy depth for chroma coding units resulting from multi-type tree splitting of a chroma quadtree leaf with treeType equal to DUAL\_TREE\_CHROMA in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_ flag is equal to 1, the default maximum hierarchy depth can be overridden by slice\_max\_mtt\_hierarchy\_depth\_chroma present in the slice header of the slices referring to the SPS. The value of sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_chroma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_bt\_min\_qt\_chroma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC, inclusive. When sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma is not present, the value of sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma is inferred to be equal to 0.

**sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma** specifies the default difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in slices with slice\_type equal to 2 (I) referring to the SPS. When partition\_constraints\_override\_flag is equal to 1, the default difference can be overridden by slice\_log2\_diff\_max\_tt\_min\_qt\_chroma present in the slice header of the slices referring to the SPS. The value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeIntraC, inclusive. When sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma is not present, the value of sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma is inferred to be equal to 0.

**sps\_sao\_enabled\_flag** equal to 1 specifies that the sample adaptive offset process is applied to the reconstructed picture after the deblocking filter process. sps\_sao\_enabled\_flag equal to 0 specifies that the sample adaptive offset process is not applied to the reconstructed picture after the deblocking filter process.

**sps\_alf\_enabled\_flag** equal to 0 specifies that the adaptive loop filter is disabled. sps\_alf\_enabled\_flag equal to 1 specifies that the adaptive loop filter is enabled.

**sps\_pcm\_enabled\_flag** equal to 0 specifies that PCM-related syntax (pcm\_sample\_bit\_depth\_luma\_minus1, pcm\_sample\_‌bit\_depth\_chroma\_minus1, log2\_min\_pcm\_luma\_coding\_block\_size\_minus3, log2\_diff\_max\_min\_pcm\_luma\_‌coding\_block\_size, pcm\_loop\_filter\_disabled\_flag, pcm\_flag, pcm\_alignment\_zero\_bit syntax elements and pcm\_sample( ) syntax structure) is not present in the CVS.

NOTE 4 – When MinCbLog2SizeY is equal to 6 and sps\_pcm\_enabled\_flag is equal to 1, PCM sample data-related syntax (pcm\_flag, pcm\_alignment\_zero\_bit syntax elements and pcm\_sample( ) syntax structure) is not present in the CVS, because the maximum size of coding blocks that can convey PCM sample data-related syntax is restricted to be less than or equal to Min( CtbLog2SizeY, 5 ). Hence, MinCbLog2SizeY equal to 6 with sps\_pcm\_enabled\_flag equal to 1 is not an appropriate setting to convey PCM sample data in the CVS.

**pcm\_sample\_bit\_depth\_luma\_minus1** specifies the number of bits used to represent each of PCM sample values of the luma component as follows:

PcmBitDepthY = pcm\_sample\_bit\_depth\_luma\_minus1 + 1 (7‑32)

The value of PcmBitDepthY shall be less than or equal to the value of BitDepthY.

**pcm\_sample\_bit\_depth\_chroma\_minus1** specifies the number of bits used to represent each of PCM sample values of the chroma components as follows:

PcmBitDepthC = pcm\_sample\_bit\_depth\_chroma\_minus1 + 1 (7‑33)

The value of PcmBitDepthC shall be less than or equal to the value of BitDepthC. When ChromaArrayType is equal to 0, pcm\_sample\_bit\_depth\_chroma\_minus1 is not used in the decoding process and decoders shall ignore its value.

**log2\_min\_pcm\_luma\_coding\_block\_size\_minus3** plus 3 specifies the minimum size of coding blocks with pcm\_flag equal to 1.

The variable Log2MinIpcmCbSizeY is set equal to log2\_min\_pcm\_luma\_coding\_block\_size\_minus3 + 3. The value of Log2MinIpcmCbSizeY shall be in the range of Min( MinCbLog2SizeY, 5 ) to Min( CtbLog2SizeY, 5 ), inclusive.

**log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size** specifies the difference between the maximum and minimum size of coding blocks with pcm\_flag equal to 1.

The variable Log2MaxIpcmCbSizeY is set equal to log2\_diff\_max\_min\_pcm\_luma\_coding\_block\_size + Log2MinIpcmCbSizeY. The value of Log2MaxIpcmCbSizeY shall be less than or equal to Min( CtbLog2SizeY, 5 ).

**pcm\_loop\_filter\_disabled\_flag** specifies whether the loop filter process is disabled on reconstructed samples in a coding unit with pcm\_flag equal to 1 as follows:

– If pcm\_loop\_filter\_disabled\_flag is equal to 1, the deblocking filter, sample adaptive offset filter, and adaptive loop filter processes on the reconstructed samples in a coding unit with pcm\_flag equal to 1 are disabled.

– Otherwise (pcm\_loop\_filter\_disabled\_flag value is equal to 0), the deblocking filter, sample adaptive offset filter, and adaptive loop filter processes on the reconstructed samples in a coding unit with pcm\_flag equal to 1 are not disabled.

When pcm\_loop\_filter\_disabled\_flag is not present, it is inferred to be equal to 0.

**sps\_ref\_wraparound\_enabled\_flag** equal to 1 specifies that horizontal wrap-around motion compensation is applied in inter prediction. sps\_ref\_wraparound\_enabled\_flag equal to 0 specifies that horizontal wrap-around motion compensation is not applied. When not present, the value of sps\_ref\_wraparound\_enabled\_flag is inferred to be equal to 0.

**sps\_ref\_wraparound\_offset\_minus1** plus 1 specifies the offset used for computing the horizontal wrap-around position in units of MinCbSizeY luma samples. The value of ref\_wraparound\_offset\_minus1 shall be in the range of ( CtbSizeY / MinCbSizeY ) + 1 to ( pic\_width\_in\_luma\_samples / MinCbSizeY ) − 1, inclusive.

**sps\_temporal\_mvp\_enabled\_flag** equal to 1 specifies that slice\_temporal\_mvp\_enabled\_flag is present in the slice headers of slices with slice\_type not equal to I in the CVS. sps\_temporal\_mvp\_enabled\_flag equal to 0 specifies that slice\_temporal\_mvp\_enabled\_flag is not present in slice headers and that temporal motion vector predictors are not used in the CVS.

**sps\_sbtmvp\_enabled\_flag** equal to 1 specifies that subblock-based temporal motion vector predictors may be used in decoding of pictures with all slices having slice\_type not equal to I in the CVS. sps\_sbtmvp\_enabled\_flag equal to 0 specifies that subblock-based temporal motion vector predictors are not used in the CVS. When sps\_sbtmvp\_enabled\_flag is not present, it is inferred to be equal to 0.

**sps\_amvr\_enabled\_flag** equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding. amvr\_enabled\_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding.

**sps\_bdof\_enabled\_flag** equal to 0 specifies that the bidirectional optical flow inter prediction is disabled. sps\_bdof\_enabled\_flag equal to 1 specifies that the bidirectional optical flow inter prediction is enabled.

**sps\_smvd\_enabled\_flag** equal to 1 specifies that symmetric motion vector difference may be used in motion vector decoding. sps\_smvd\_enabled\_flag equal to 0 specifies that symmetric motion vector difference is not used in motion vector coding.

**sps\_affine\_amvr\_enabled\_flag** equal to 1 specifies that adaptive motion vector difference resolution is used in motion vector coding of affine inter mode. sps\_affine\_amvr\_enabled\_flag equal to 0 specifies that adaptive motion vector difference resolution is not used in motion vector coding of affine inter mode.

**sps\_dmvr\_enabled\_flag** equal to 1 specifies that decoder motion vector refinement based inter bi-prediction is enabled. sps\_dmvr\_enabled\_flag equal to 0 specifies that decoder motion vector refinement based inter bi-prediction is disabled.

**sps\_mmvd\_enabled\_flag** equal to 1 specifies that merge mode with motion vector difference is enabled. sps\_mmvd\_enabled\_flag equal to 0 specifies that merge mode with motion vector difference is disabled.

**sps\_isp\_enabled\_flag** equal to 1 specifies that intra prediction with subpartitions is enabled. sps\_isp\_enabled\_flag equal to 0 specifies that intra prediction with subpartitions is disabled.

**sps\_mrl\_enabled\_flag** equal to 1 specifies that intra prediction with multiple reference lines is enabled. sps\_mrl\_enabled\_flag equal to 0 specifies that intra prediction with multiple reference lines is disabled.

**sps\_mip\_enabled\_flag** equal to 1 specifies that matrix-based intra prediction is enabled. sps\_mip\_enabled\_flag equal to 0 specifies that matrix-based intra prediction is disabled.

**sps\_cclm\_enabled\_flag** equal to 0 specifies that the cross-component linear model intra prediction from luma component to chroma component is disabled. sps\_cclm\_enabled\_flag equal to 1 specifies that the cross-component linear model intra prediction from luma component to chroma componenent is enabled. When sps\_cclm\_enabled\_flag is not present, it is inferred to be equal to 0.

**sps\_cclm\_colocated\_chroma\_flag** equal to 1 specifies that the top-left downsampled luma sample in cross-component linear model intra prediction is collocated with the top-left luma sample. sps\_cclm\_colocated\_chroma\_flag equal to 0 specifies that the top-left downsampled luma sample in cross-component linear model intra prediction is horizontally co-sited with the top-left luma sample but vertically shifted by 0.5 units of luma samples relatively to the top-left luma sample.

**sps\_mts\_enabled\_flag** equal to 1 specifies that sps\_explicit\_mts\_intra\_enabled\_flag is present in the sequence parameter set RBSP syntax and that sps\_explicit\_mts\_inter\_enabled\_flag is present in the sequence parameter set RBSP syntax. sps\_mts\_enabled\_flag equal to 0 specifies that sps\_explicit\_mts\_intra\_enabled\_flag is not present in the sequence parameter set RBSP syntax and that sps\_explicit\_mts\_inter\_enabled\_flag is not present in the sequence parameter set RBSP syntax.

**sps\_explicit\_mts\_intra\_enabled\_flag** equal to 1 specifies that tu\_mts\_idx may be present in the transform unit syntax for intra coding units. sps\_explicit\_mts\_intra\_enabled\_flag equal to 0 specifies that tu\_mts\_idx is not present in the transform unit syntax for intra coding units. When not present, the value of sps\_explicit\_mts\_intra\_enabled\_flag is inferred to be equal to 0.

**sps\_explicit\_mts\_inter\_enabled\_flag** equal to 1 specifies that tu\_mts\_idx may be present in the transform unit syntax for inter coding units. sps\_explicit\_mts\_inter\_enabled\_flag equal to 0 specifies that tu\_mts\_idx is not present in the transform unit syntax for inter coding units. When not present, the value of sps\_explicit\_mts\_inter\_enabled\_flag is inferred to be equal to 0.

**sps\_sbt\_enabled\_flag** equal to 0 specifies that subblock transform for inter-predicted CUs is disabled. sps\_sbt\_enabled\_flag equal to 1 specifies that subblock transform for inter-predicteds CU is enabled.

**sps\_sbt\_max\_size\_64\_flag** equal to 0 specifies that the maximum CU width and height for allowing subblock transform is 32 luma samples. sps\_sbt\_max\_size\_64\_flag equal to 1 specifies that the maximum CU width and height for allowing subblock transform is 64 luma samples.

MaxSbtSize = sps\_sbt\_max\_size\_64\_flag ? 64 : 32 (7‑34)

**sps\_affine\_enabled\_flag** specifies whether affine model based motion compensation can be used for inter prediction. If sps\_affine\_enabled\_flag is equal to 0, the syntax shall be constrained such that no affine model based motion compensation is used in the CVS, and inter\_affine\_flag and cu\_affine\_type\_flag are not present in coding unit syntax of the CVS. Otherwise (sps\_affine\_enabled\_flag is equal to 1), affine model based motion compensation can be used in the CVS.

**sps\_affine\_type\_flag** specifies whether 6-parameter affine model based motion compensation can be used for inter prediction. If sps\_affine\_type\_flag is equal to 0, the syntax shall be constrained such that no 6-parameter affine model based motion compensation is used in the CVS, and cu\_affine\_type\_flag is not present in coding unit syntax in the CVS. Otherwise (sps\_affine\_type\_flag is equal to 1), 6-parameter affine model based motion compensation can be used in the CVS. When not present, the value of sps\_affine\_type\_flag is inferred to be equal to 0.

**sps\_bcw\_enabled\_flag** specifies whether bi-prediction with CU weights can be used for inter prediction. If sps\_bcw\_enabled\_flag is equal to 0, the syntax shall be constrained such that no bi-prediction with CU weights is used in the CVS, and bcw\_idx is not present in coding unit syntax of the CVS. Otherwise (sps\_bcw\_enabled\_flag is equal to 1), bi-prediction with CU weights can be used in the CVS.

**sps\_ibc\_enabled\_flag** equal to 1 specifies that current picture referencing may be used in decoding of pictures in the CVS. sps\_ibc\_enabled\_flag equal to 0 specifies that current picture referencing is not used in the CVS. When sps\_ibc\_enabled\_flag is not present, it is inferred to be equal to 0.

**sps\_ciip\_enabled\_flag** specifies that ciip\_flag may be present in the coding unit syntax for inter coding units. sps\_ciip\_enabled\_flag equal to 0 specifies that ciip\_flag is not present in the coding unit syntax for inter coding units.

**sps\_fpel\_mmvd\_enabled\_flag** equal to 1 specifies that merge mode with motion vector difference is using integer sample precision. sps\_fpel\_mmvd\_enabled\_flag equal to 0 specifies that merge mode with motion vector difference can use fractional sample precision.

**sps\_triangle\_enabled\_flag** specifies whether triangular shape based motion compensation can be used for inter prediction. sps\_triangle\_enabled\_flag equal to 0 specifies that the syntax shall be constrained such that no triangular shape based motion compensation is used in the CVS, and merge\_triangle\_split\_dir, merge\_triangle\_idx0, and merge\_triangle\_idx1 are not present in coding unit syntax of the CVS. sps\_triangle\_enabled\_flag equal to 1 specifies that triangular shape based motion compensation can be used in the CVS.

**sps\_lmcs\_enabled\_flag** equal to 1 specifies that luma mapping with chroma scaling is used in the CVS. sps\_lmcs\_enabled\_flag equal to 0 specifies that luma mapping with chroma scaling is not used in the CVS.

**sps\_lfnst\_enabled\_flag** equal to 1 specifies that lfnst\_idx may be present in the residual coding syntax for intra coding units. sps\_lfnst\_enabled\_flag equal to 0 specifies that lfnst\_idx is not present in the residual coding syntax for intra coding units.

**sps\_ladf\_enabled\_flag** equal to 1, specifies that sps\_num\_ladf\_intervals\_minus2, sps\_ladf\_lowest\_interval\_qp\_offset, sps\_ladf\_qp\_offset[ i ], and sps\_ladf\_delta\_threshold\_minus1[ i ] are present in the SPS.

**sps\_num\_ladf\_intervals\_minus2** plus 1 specifies the number of sps\_ladf\_delta\_threshold\_minus1[ i ] and sps\_ladf\_qp\_offset[ i ] syntax elements that are present in the SPS. The value of sps\_num\_ladf\_intervals\_minus2 shall be in the range of 0 to 3, inclusive.

**sps\_ladf\_lowest\_interval\_qp\_offset** specifies the offset used to derive the variable qP as specified in clause 8.8.3.6.1. The value of sps\_ladf\_lowest\_interval\_qp\_offset shall be in the range of 0 to 63, inclusive.

**sps\_ladf\_qp\_offset**[ i ] specifies the offset array used to derive the variable qP as specified in clause 8.8.3.6.1. The value of sps\_ladf\_qp\_offset[ i ] shall be in the range of 0 to 63, inclusive.

**sps\_ladf\_delta\_threshold\_minus1**[ i ] is used to compute the values of SpsLadfIntervalLowerBound[ i ], which specifies the lower bound of the i-th luma intensity level interval. The value of sps\_ladf\_delta\_threshold\_minus1[ i ] shall be in the range of 0 to 2BitDepthY − 3, inclusive.

The value of SpsLadfIntervalLowerBound[ 0 ] is set equal to 0.

For each value of i in the range of 0 to sps\_num\_ladf\_intervals\_minus2, inclusive, the variable SpsLadfIntervalLowerBound[ i + 1 ] is derived as follows:

SpsLadfIntervalLowerBound[ i + 1 ] = SpsLadfIntervalLowerBound[ i ] (7‑35)  
 + sps\_ladf\_delta\_threshold\_minus1[ i ] + 1

**scaling\_list\_enabled\_flag** equal to 1 specifies that a scaling list is used for the scaling process for transform coefficients. scaling\_list\_enabled\_flag equal to 0 specifies that scaling list is not used for the scaling process for transform coefficients.

**sps\_scaling\_list\_data\_present\_flag** equal to 1 specifies that the scaling\_list\_data( ) syntax structure is present in the SPS. sps\_scaling\_list\_data\_present\_flag equal to 0 specifies that the scaling\_list\_data( ) syntax structure is not present in the SPS. When not present, the value of sps\_scaling\_list\_data\_present\_flag is inferred to be equal to 0.

**timing\_info\_present\_flag** equal to 1 specifies that the syntax elemnts num\_units\_in\_tick, time\_scale, and hrd\_parameters\_present\_flag are present in the SPS RBSP syntax structure. timing\_info\_present\_flag equal to 0 specifies that num\_units\_in\_tick, time\_scale, and hrd\_parameters\_present\_flag are not present in the SPS RBSP syntax structure.

**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, in units of seconds, is equal to the quotient of num\_units\_in\_tick divided by time\_scale. For example, when the picture rate of a video signal is 25 Hz, time\_scale may be equal to 27 000 000 and num\_units\_in\_tick may be equal to 1 080 000, and consequently a clock tick may be equal to 0.04 seconds.

**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. The value of time\_scale shall be greater than 0.

**hrd\_parameters\_present\_flag** equal to 1 specifies that the syntax structure hrd\_parameters( ) is present in the SPS RBSP syntax structure. hrd\_parameters\_present\_flag equal to 0 specifies that the syntax structure hrd\_parameters( ) is not present in the SPS RBSP syntax structure.

**vui\_parameters\_present\_flag** equal to 1 specifies that the syntax structure vui\_parameters( ) is present in the SPS RBSP syntax structure. vui\_parameters\_present\_flag equal to 0 specifies that the syntax structure vui\_parameters( ) is not present in the SPS RBSP syntax structure.

**sps\_extension\_flag** equal to 0 specifies that no sps\_extension\_data\_flag syntax elements are present in the SPS RBSP syntax structure. sps\_extension\_flag equal to 1 specifies that there are sps\_extension\_data\_flag syntax elements present in the SPS RBSP syntax structure.

**sps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all sps\_extension\_data\_flag syntax elements.

#### Picture parameter set RBSP semantics

**pps\_pic\_parameter\_set\_id** identifies the PPS for reference by other syntax elements. The value of pps\_pic\_parameter\_set\_id shall be in the range of 0 to 63, inclusive.

**pps\_seq\_parameter\_set\_id** specifies the value of sps\_seq\_parameter\_set\_id for the active SPS. The value of pps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

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

**single\_tile\_in\_pic\_flag** equal to 1 specifies that there is only one tile in each picture referring to the PPS. single\_tile\_in\_pic\_flag equal to 0 specifies that there is more than one tile in each picture referring to the PPS.

NOTE – In absence of further brick splitting within a tile, the whole tile is referred to as a brick. When a picture contains only a single tile without further brick splitting, it is referred to as a single brick.

It is a requirement of bitstream conformance that the value of single\_tile\_in\_pic\_flag shall be the same for all PPSs that are activated within a CVS.

**uniform\_tile\_spacing\_flag** equal to 1 specifies that tile column boundaries and likewise tile row boundaries are distributed uniformly across the picture and signalled using the syntax elements tile\_cols\_width\_minus1 and tile\_rows\_height\_minus1. uniform\_tile\_spacing\_flag equal to 0 specifies that tile column boundaries and likewise tile row boundaries may or may not be distributed uniformly across the picture and signalled using the syntax elements num\_tile\_columns\_minus1 and num\_tile\_rows\_minus1 and a list of syntax element pairs tile\_column\_width\_minus1[ i ] and tile\_row\_height\_minus1[ i ]. When not present, the value of uniform\_tile\_spacing\_flag is inferred to be equal to 1.

**tile\_cols\_width\_minus1** plus 1 specifies the width of the tile columns excluding the right-most tile column of the picture in units of CTBs when uniform\_tile\_spacing\_flag is equal to 1. The value of tile\_cols\_width\_minus1 shall be in the range of 0 to PicWidthInCtbsY − 1, inclusive. When not present, the value of tile\_cols\_width\_minus1 is inferred to be equal to PicWidthInCtbsY − 1.

**tile\_rows\_height\_minus1** plus 1 specifies the height of the tile rows excluding the bottom tile row of the picture in units of CTBs when uniform\_tile\_spacing\_flag is equal to 1. The value of tile\_rows\_height\_minus1 shall be in the range of 0 to PicHeightInCtbsY − 1, inclusive. When not present, the value of tile\_rows\_height\_minus1 is inferred to be equal to PicHeightInCtbsY − 1.

**num\_tile\_columns\_minus1** plus 1 specifies the number of tile columns partitioning the picture when uniform\_tile\_spacing\_flag is equal to 0. The value of num\_tile\_columns\_minus1 shall be in the range of 0 to PicWidthInCtbsY − 1, inclusive. If single\_tile\_in\_pic\_flag is equal to 1, the value of num\_tile\_columns\_minus1 is inferred to be equal to 0. Otherwise, when uniform\_tile\_spacing\_flag is equal to 1, the value of num\_tile\_columns\_minus1 is inferred as specified in clause 6.5.1.

**num\_tile\_rows\_minus1** plus 1 specifies the number of tile rows partitioning the picture when uniform\_tile\_spacing\_flag is equal to 0. The value of num\_tile\_rows\_minus1 shall be in the range of 0 to PicHeightInCtbsY − 1, inclusive. If single\_tile\_in\_pic\_flag is equal to 1, the value of num\_tile\_rows\_minus1 is inferred to be equal to 0. Otherwise, when uniform\_tile\_spacing\_flag is equal to 1, the value of num\_tile\_rows\_minus1 is inferred as specified in clause 6.5.1.

The variable NumTilesInPic is set equal to ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ).

When single\_tile\_in\_pic\_flag is equal to 0, NumTilesInPic shall be greater than 1.

**tile\_column\_width\_minus1**[ i ] plus 1 specifies the width of the i-th tile column in units of CTBs.

**tile\_row\_height\_minus1**[ i ] plus 1 specifies the height of the i-th tile row in units of CTBs.

**brick\_splitting\_present\_flag** equal to 1 specifies that one or more tiles of pictures referring to the PPS may be divided into two or more bricks. brick\_splitting\_present\_flag equal to 0 specifies that no tiles of pictures referring to the PPS are divided into two or more bricks.

**brick\_split\_flag**[ i ] equal to 1 specifies that the i-th tile is divided into two or more bricks. brick\_split\_flag[ i ] equal to 0 specifies that the i-th tile is not divided into two or more bricks. When not present, the value of brick\_split\_flag[ i ] is inferred to be equal to 0.

**uniform\_brick\_spacing\_flag**[ i ] equal to 1 specifies that horizontal brick boundaries are distributed uniformly across the i-th tile and signalled using the syntax element brick\_height\_minus1[ i ]. uniform\_brick\_spacing\_flag[ i ] equal to 0 specifies that horizontal brick boundaries may or may not be distributed uniformly across i-th tile and signalled using the syntax element num\_brick\_rows\_minus1[ i ] and a list of syntax elements brick\_row\_height\_minus1[ i ][ j ]. When not present, the value of uniform\_brick\_spacing\_flag[ i ] is inferred to be equal to 1.

**brick\_height\_minus1**[ i ] plus 1 specifies the height of the brick rows excluding the bottom brick in the i-th tile in units of CTBs when uniform\_brick\_spacing\_flag[ i ] is equal to 1. When present, the value of brick\_height\_minus1 shall be in the range of 0 to RowHeight[ i ] − 2, inclusive. When not present, the value of brick\_height\_minus1[ i ] is inferred to be equal to RowHeight[ i ] − 1.

**num\_brick\_rows\_minus1**[ i ] plus 1 specifies the number of bricks partitioning the i-th tile when uniform\_brick\_spacing\_flag[ i ] is equal to 0. When present, the value of num\_brick\_rows\_minus1[ i ] shall be in the range of 1 to RowHeight[ i ] − 1, inclusive. If brick\_split\_flag[ i ] is equal to 0, the value of num\_brick\_rows\_minus1[ i ] is inferred to be equal to 0. Otherwise, when uniform\_brick\_spacing\_flag[ i ] is equal to 1, the value of num\_brick\_rows\_minus1[ i ] is inferred as specified in 6.5.1.

**brick\_row\_height\_minus1**[ i ][ j ] plus 1 specifies the height of the j-th brick in the i-th tile in units of CTBs when uniform\_tile\_spacing\_flag is equal to 0.

The following variables are derived, and, when uniform\_tile\_spacing\_flag is equal to 1, the values of num\_tile\_columns\_minus1 and num\_tile\_rows\_minus1 are inferred, and, for each i ranging from 0 to NumTilesInPic − 1, inclusive, when uniform\_brick\_spacing\_flag[ i ] is equal to 1, the value of num\_brick\_rows\_minus1[ i ] is inferred, by invoking the CTB raster and brick scanning conversion process as specified in clause 6.5.1:

– the list RowHeight[ j ] for j ranging from 0 to num\_tile\_rows\_minus1, inclusive, specifying the height of the j-th tile row in units of CTBs,

– the list CtbAddrRsToBs[ ctbAddrRs ] for ctbAddrRs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in the CTB raster scan of a picture to a CTB address in the brick scan,

– the list CtbAddrBsToRs[ ctbAddrBs ] for ctbAddrBs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in the brick scan to a CTB address in the CTB raster scan of a picture,

– the list BrickId[ ctbAddrBs ] for ctbAddrBs ranging from 0 to PicSizeInCtbsY − 1, inclusive, specifying the conversion from a CTB address in brick scan to a brick ID,

– the list NumCtusInBrick[ brickIdx ] for brickIdx ranging from 0 to NumBricksInPic − 1, inclusive, specifying the conversion from a brick index to the number of CTUs in the brick,

– the list FirstCtbAddrBs[ brickIdx ] for brickIdx ranging from 0 to NumBricksInPic − 1, inclusive, specifying the conversion from a brick ID to the CTB address in brick scan of the first CTB in the brick.

**single\_brick\_per\_slice\_flag** equal to 1 specifies that each slice that refers to this PPS includes one brick. single\_brick\_per\_slice\_flag equal to 0 specifies that a slice that refers to this PPS may include more than one brick. When not present, the value of single\_brick\_per\_slice\_flag is inferred to be equal to 1.

**rect\_slice\_flag** equal to 0 specifies that bricks within each slice are in raster scan order and the slice information is not signalled in PPS. rect\_slice\_flag equal to 1 specifies that bricks within each slice cover a rectangular region of the picture and the slice information is signalled in the PPS. When not present, rect\_slice\_flag is inferred to be equal to 1.

**num\_slices\_in\_pic\_minus1** plus 1 specifies the number of slices in each picture referring to the PPS. The value of num\_slices\_in\_pic\_minus1 shall be in the range of 0 to NumBricksInPic − 1, inclusive. When not present and single\_brick\_per\_slice\_flag is equal to 1, the value of num\_slices\_in\_pic\_minus1 is inferred to be equal to NumBricksInPic − 1.

**top\_left\_brick\_idx**[ i ] specifies the brick index of the brick located at the top-left corner of the i-th slice. The value of top\_left\_brick\_idx[ i ] shall not be equal to the value of top\_left\_brick\_idx[ j ] for any i not equal to j. When not present, the value of top\_left\_brick\_idx[ i ] is inferred to be equal to i. The length of the top\_left\_brick\_idx[ i ] syntax element is Ceil( Log2( NumBricksInPic ) bits.

**bottom\_right\_brick\_idx\_delta**[ i ] specifies the difference between the brick index of the brick located at the bottom-right corner of the i-th slice and top\_left\_brick\_idx[ i ]. When single\_brick\_per\_slice\_flag is equal to 1, the value of bottom\_right\_brick\_idx\_delta[ i ] is inferred to be equal to 0. The length of the bottom\_right\_brick\_idx\_delta[ i ] syntax element is Ceil( Log2( NumBricksInPic − top\_left\_brick\_idx[ i ] ) ) bits.

It is a requirement of bitstream conformance that a slice shall include either a number of complete tiles or only a consecutive sequence of complete bricks of one tile.

The variable NumBricksInSlice[ i ] and BricksToSliceMap[ j ], which specify the number of bricks in the i-th slice and the mapping of bricks to slices, are derived as follows:

NumBricksInSlice[ i ] = 0  
botRightBkIdx = top\_left\_brick\_idx[ i ] + bottom\_right\_brick\_idx\_delta[ i ]  
for( j = 0; j < NumBricksInPic; j++) {  
 if( BrickColBd[ j ] >= BrickColBd[ top\_left\_brick\_idx[ i ] ] &&  
 BrickColBd[ j ] <= BrickColBd[ botRightBkIdx ] &&  
 BrickRowBd[ j ] >= BrickRowBd[ top\_left\_brick\_idx[ i ] ] && (7‑36)  
 BrickRowBd[ j ] <= BrickRowBd [ botRightBkIdx ] ) {  
 NumBricksInSlice[ i ]++  
 BricksToSliceMap[ j ] = i  
 }  
}

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

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

**signalled\_slice\_id\_flag** equal to 1 specifies that the slice ID for each slice is signalled. signalled\_slice\_id\_flag equal to 0 specifies that slice IDs are not signalled. When rect\_slice\_flag is equal to 0, the value of signalled\_slice\_id\_flag is inferred to be equal to 0.

**signalled\_slice\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element slice\_id[ i ] when present, and the syntax element slice\_address in slice headers. The value of signalled\_slice\_id\_length\_minus1 shall be in the range of 0 to 15, inclusive. When not present, the value of signalled\_slice\_id\_length\_minus1 is inferred to be equal to Ceil( Log2( num\_slices\_in\_pic\_minus1 + 1 ) ) − 1.

**slice\_id**[ i ] specifies the slice ID of the i-th slice. The length of the slice\_id[ i ] syntax element is signalled\_slice\_id\_length\_minus1 + 1 bits. When not present, the value of slice\_id[ i ] is inferred to be equal to i, for each i in the range of 0 to num\_slices\_in\_pic\_minus1, inclusive.

**entropy\_coding\_sync\_enabled\_flag** equal to 1 specifies that a specific synchronization process for context variables is invoked before decoding the CTU that includes the first CTB of a row of CTBs in each brick in each picture referring to the PPS, and a specific storage process for context variables is invoked after decoding the CTU that includes the first CTB of a row of CTBs in each brick in each picture referring to the PPS. entropy\_coding\_sync\_enabled\_flag equal to 0 specifies that no specific synchronization process for context variables is required to be invoked before decoding the CTU that includes the first CTB of a row of CTBs in each brick in each picture referring to the PPS, and no specific storage process for context variables is required to be invoked after decoding the CTU that includes the first CTB of a row of CTBs in each brick in each picture referring to the PPS.

It is a requirement of bitstream conformance that the value of entropy\_coding\_sync\_enabled\_flag shall be the same for all PPSs that are activated within a CVS.

**cabac\_init\_present\_flag** equal to 1 specifies that cabac\_init\_flag is present in slice headers referring to the PPS. cabac\_init\_present\_flag equal to 0 specifies that cabac\_init\_flag is not present in slice headers referring to the PPS.

**num\_ref\_idx\_default\_active\_minus1**[ i ] plus 1, when i is equal to 0, specifies the inferred value of the variable NumRefIdxActive[ 0 ] for P or B slices with num\_ref\_idx\_active\_override\_flag equal to 0, and, when i is equal to 1, specifies the inferred value of NumRefIdxActive[ 1 ] for B slices with num\_ref\_idx\_active\_override\_flag equal to 0. The value of num\_ref\_idx\_default\_active\_minus1[ i ] shall be in the range of 0 to 14, inclusive.

**rpl1\_idx\_present\_flag** equal to 0 specifies that ref\_pic\_list\_sps\_flag[ 1 ] and ref\_pic\_list\_idx[ 1 ] are not present in slice headers. rpl1\_idx\_present\_flag equal to 1 specifies that ref\_pic\_list\_sps\_flag[ 1 ] and ref\_pic\_list\_idx[ 1 ] may be present in slice headers.

**init\_qp\_minus26** plus 26 specifies the initial value of SliceQpY for each slice referring to the PPS. The initial value of SliceQpY is modified at the slice layer when a non-zero value of slice\_qp\_delta is decoded. The value of init\_qp\_minus26 shall be in the range of −( 26 + QpBdOffsetY ) to +37, inclusive.

**transform\_skip\_enabled\_flag** equal to 1 specifies that transform\_skip\_flag may be present in the transform unit syntax. transform\_skip\_enabled\_flag equal to 0 specifies that transform\_skip\_flag is not present in the transform unit syntax.

**log2\_transform\_skip\_max\_size\_minus2** specifies the maximum block size used for transform skip, and shall be in the range of 0 to 3.

When not present, the value of log2\_transform\_skip\_max\_size\_minus2 is inferred to be equal to 0.

The variable MaxTsSize is set equal to 1 << ( log2\_transform\_skip\_max\_size\_minus2 + 2 ).

**cu\_qp\_delta\_enabled\_flag** equal to 1 specifies that the cu\_qp\_delta\_subdiv syntax element is present in the PPS and that cu\_qp\_delta\_abs may be present in the transform unit syntax. cu\_qp\_delta\_enabled\_flag equal to 0 specifies that the cu\_qp\_delta\_subdiv syntax element is not present in the PPS and that cu\_qp\_delta\_abs is not present in the transform unit syntax.

**cu\_qp\_delta\_subdiv** specifies the maximum cbSubdiv value of coding units that convey cu\_qp\_delta\_abs and cu\_qp\_delta\_sign\_flag. The value range of cu\_qp\_delta\_subdiv is specified as follows:

* If slice\_type is equal to I, the value of cu\_qp\_delta\_subdiv shall be in the range of 0 to 2 \* ( log2\_ctu\_size\_minus2 − log2\_min\_qt\_size\_intra\_slice\_minus2 + MaxMttDepthY ), inclusive.
* Otherwise (slice\_type is not equal to I), the value of cu\_qp\_delta\_subdiv shall be in the range of 0 to 2\* ( log2\_ctu\_size\_minus2 − log2\_min\_qt\_size\_inter\_slice\_minus2 + MaxMttDepthY ), inclusive.

When not present, the value of cu\_qp\_delta\_subdiv is inferred to be equal to 0.

[Ed. (BB): The issue here is that MaxMttDepthY is derived on slice level. In case of partition\_constraints\_override\_enabled\_flag equal to 1, one would need to parse the slice header in order to know the value of MaxMttDepthY.]

**pps\_cb\_qp\_offset** and **pps\_cr\_qp\_offset** specify the offsets to the luma quantization parameter Qp′Y used for deriving Qp′Cb and Qp′Cr, respectively. The values of pps\_cb\_qp\_offset and pps\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive. When ChromaArrayType is equal to 0, pps\_cb\_qp\_offset and pps\_cr\_qp\_offset are not used in the decoding process and decoders shall ignore their value.

**pps\_joint\_cbcr\_qp\_offset** specifies the offset to the luma quantization parameter Qp′Y used for deriving Qp′CbCr. The value of pps\_joint\_cbcr\_qp\_offset shall be in the range of −12 to +12, inclusive. When ChromaArrayType is equal to 0, pps\_joint\_cbcr\_qp\_offset is not used in the decoding process and decoders shall ignore its value.

**pps\_slice\_chroma\_qp\_offsets\_present\_flag** equal to 1 indicates that the slice\_cb\_qp\_offset and slice\_cr\_qp\_offset syntax elements are present in the associated slice headers. pps\_slice\_chroma\_qp\_offsets\_present\_flag equal to 0 indicates that these syntax elements are not present in the associated slice headers. When ChromaArrayType is equal to 0, pps\_slice\_chroma\_qp\_offsets\_present\_flag shall be equal to 0.

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

**weighted\_bipred\_flag** equal to 0 specifies that the default weighted prediction is applied to B slices. weighted\_bipred\_flag equal to 1 specifies that weighted prediction is applied to B slices.

**deblocking\_filter\_control\_present\_flag** equal to 1 specifies the presence of deblocking filter control syntax elements in the PPS. deblocking\_filter\_control\_present\_flag equal to 0 specifies the absence of deblocking filter control syntax elements in the PPS.

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

**pps\_deblocking\_filter\_disabled\_flag** equal to 1 specifies that the operation of deblocking filter is not applied for slices referring to the PPS in which slice\_deblocking\_filter\_disabled\_flag is not present. pps\_deblocking\_filter\_disabled\_flag equal to 0 specifies that the operation of the deblocking filter is applied for slices referring to the PPS in which slice\_deblocking\_filter\_disabled\_flag is not present. When not present, the value of pps\_deblocking\_filter\_disabled\_flag is inferred to be equal to 0.

**pps\_beta\_offset\_div2** and **pps\_tc\_offset\_div2** specify the default deblocking parameter offsets for β and tC (divided by 2) that are applied for slices referring to the PPS, unless the default deblocking parameter offsets are overridden by the deblocking parameter offsets present in the slice headers of the slices referring to the PPS. The values of pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2 shall both be in the range of −6 to 6, inclusive. When not present, the value of pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2 are inferred to be equal to 0.

**pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag** equal to 1 specifies that the in-loop filtering operations are disabled across the virtual boundaries in pictures referring to the PPS. pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flagequal to 0 specifies that no such disabling of in-loop filtering operations is applied in pictures referring to the PPS. The in-loop filtering operations include the deblocking filter, sample adaptive offset filter, and adaptive loop filter operations. When not present, the value of pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is inferred to be equal to 0.

**pps\_num\_ver\_virtual\_boundaries** specifies the number of pps\_virtual\_boundaries\_pos\_x[ i ] syntax elements that are present in the PPS. When pps\_num\_ver\_virtual\_boundaries is not present, it is inferred to be equal to 0.

**pps\_virtual\_boundaries\_pos\_x**[ i ] is used to compute the value of PpsVirtualBoundariesPosX[ i ], whichspecifies the location of the i-th vertical virtual boundary in units of luma samples. The number of bits used to represent pps\_virtual\_boundaries\_pos\_x[ i ] is Ceil( Log2( pic\_width\_in\_luma\_samples ) − 3 ). pps\_virtual\_boundaries\_pos\_x[ i ] shall be in the range of 1 to Ceil( pic\_width\_in\_luma\_samples ÷ 8 ) − 1, inclusive.

The location of the vertical virtual boundary PpsVirtualBoundariesPosX[ i ] is derived as follows:

PpsVirtualBoundariesPosX[ i ] = pps\_virtual\_boundaries\_pos\_x[ i ] \* 8 (7‑37)

The distance between any two vertical virtual boundaries shall be greater than or equal to CtbSizeY luma samples.

**pps\_num\_hor\_virtual\_boundaries** specifies the number of pps\_virtual\_boundaries\_pos\_y[ i ] syntax elements that are present in the PPS. When pps\_num\_hor\_virtual\_boundaries is not present, it is inferred to be equal to 0.

**pps\_virtual\_boundaries\_pos\_y**[ i ] is used to compute the value of PpsVirtualBoundariesPosY[ i ], whichspecifies the location of the i-th horizontal virtual boundary in units of luma samples. The number of bits used to represent pps\_virtual\_boundaries\_pos\_y[ i ] is Ceil( Log2( pic\_height\_in\_luma\_samples ) − 3 ). pps\_virtual\_boundaries\_pos\_y[ i ] shall be in the range of 1 to Ceil( pic\_height\_in\_luma\_samples ÷ 8 ) − 1, inclusive.

The location of the horizontal virtual boundary PpsVirtualBoundariesPosY[ i ] is derived as follows:

PpsVirtualBoundariesPosY[ i ] = pps\_virtual\_boundaries\_pos\_y[ i ] \* 8 (7‑38)

The distance between any two horizontal virtual boundaries shall be greater than or equal to CtbSizeY luma samples.

**pps\_scaling\_list\_data\_present\_flag** equal to 1 specifies that the scaling list data used for the pictures referring to the PPS are derived based on the scaling lists specified by the active SPS and the scaling lists specified by the PPS. pps\_scaling\_list\_data\_present\_flag equal to 0 specifies that the scaling list data used for the pictures referring to the PPS are inferred to be equal to those specified by the active SPS. When scaling\_list\_enabled\_flag is equal to 0, the value of pps\_scaling\_list\_data\_present\_flag shall be equal to 0. When scaling\_list\_enabled\_flag is equal to 1, sps\_scaling\_list\_data\_present\_flag is equal to 0 and pps\_scaling\_list\_data\_present\_flag is equal to 0, the default scaling list data are used to derive the array ScalingFactor as as specified in clause 7.4.3.14 the scaling list data semantics.

**pps\_extension\_flag** equal to 0 specifies that no pps\_extension\_data\_flag syntax elements are present in the PPS RBSP syntax structure. pps\_extension\_flag equal to 1 specifies that there are pps\_extension\_data\_flag syntax elements present in the PPS RBSP syntax structure.

**pps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all pps\_extension\_data\_flag syntax elements.

#### Adaptation parameter set semantics

**adaptation\_parameter\_set\_id** provides an identifier for the APS for reference by other syntax elements.

NOTE – APSs can be shared across pictures and can be different in different slices within a picture.

**aps\_params\_type** specifies the type of APS parameters carried in the APS as specified in Table 7‑2.

Table 7‑2 – APS parameters type codes and types of APS parameters

|  |  |  |
| --- | --- | --- |
| **aps\_params\_type** | **Name of aps\_params\_type** | **Type of APS parameters** |
| 0 | ALF\_APS | ALF parameters |
| 1 | LMCS\_APS | LMCS parameters |
| 2..7 | Reserved | Reserved |

**aps\_extension\_flag** equal to 0 specifies that no aps\_extension\_data\_flag syntax elements are present in the APS RBSP syntax structure. aps\_extension\_flag equal to 1 specifies that there are aps\_extension\_data\_flag syntax elements present in the APS RBSP syntax structure.

**aps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this Specification. Decoders conforming to this version of this Specification shall ignore all aps\_extension\_data\_flag syntax elements.

#### 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. An SEI RBSP contains one or more SEI messages.

#### Access unit delimiter RBSP semantics

The access unit delimiter may be used to indicate the type of slices present in the coded pictures in the access unit containing the access unit delimiter NAL unit 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 pictures in the access unit containing the access unit delimiter NAL unit are members of the set listed in Table 7‑3 for the given value of pic\_type. The value of pic\_type shall be equal to 0, 1 or 2 in bitstreams conforming to this version of this Specification. Other values of pic\_type are reserved for future use by ITU‑T | ISO/IEC. Decoders conforming to this version of this Specification shall ignore reserved values of pic\_type.

Table 7‑3 – Interpretation 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 |

#### End of sequence RBSP semantics

When present, the end of sequence RBSP specifies that the current access unit is the last access unit in the coded video sequence in decoding order and the next subsequent access unit in the bitstream in decoding order (if any) is an IRAP access unit. The syntax content of the SODB and RBSP for the end of sequence RBSP are empty.

#### End of bitstream RBSP semantics

The end of bitstream RBSP indicates that no additional NAL units are present in the bitstream that are subsequent to the end of bitstream RBSP in decoding order. The syntax content of the SODB and RBSP for the end of bitstream RBSP are empty.

#### Slice layer RBSP semantics

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

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

Let the variable RawMinCuBits be derived as follows:

RawMinCuBits = MinCbSizeY \* MinCbSizeY \*  
 ( BitDepthY + 2 \* BitDepthC / ( SubWidthC \* SubHeightC ) ) (7‑39)

The value of BinCountsInNalUnits shall be less than or equal to ( 32 ÷ 3 ) \* NumBytesInVclNalUnits + ( RawMinCuBits \* PicSizeInMinCbsY ) ÷ 32.

NOTE – The constraint on the maximum number of bins resulting from decoding the contents of the coded slice 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

**alignment\_bit\_equal\_to\_one** shall be equal to 1.

**alignment\_bit\_equal\_to\_zero** shall be equal to 0.

#### Scaling list data semantics

**scaling\_list\_pred\_mode\_flag**[ sizeId ][ matrixId ] equal to 0 specifies that the values of the scaling list are the same as the values of a reference scaling list. The reference scaling list is specified by scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ]. scaling\_list\_pred\_mode\_flag[ sizeId ][ matrixId ] equal to 1 specifies that the values of the scaling list are explicitly signalled.

**scaling\_list\_pred\_matrix\_id\_delta**[ sizeId ][ matrixId ] specifies the reference scaling list used to derive ScalingList[ sizeId ][ matrixId ], the derivation of ScalingList[ sizeId ][ matrixId ] is based on scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] as follows:

* If scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] is equal to 0, the scaling list is inferred from the default scaling list ScalingList[ sizeId ][ matrixId ][ i ] as specified in Table 7‑6, Table 7‑7, Table 7‑8 and Table 7‑9 for i = 0..Min( 63, ( 1  <<  ( sizeId  <<  1 ) ) − 1 ).
* Otherwise, the scaling list is inferred from the reference scaling list as follows:

refMatrixId = matrixId −   
 scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] \* ( sizeId  = =  6 ? 3 : 1 ) (7‑40)

ScalingList[ sizeId ][ matrixId ][ i ] = ScalingList[ sizeId ][ refMatrixId ][ i ]  
 with i =0..Min( 63, ( 1  <<  ( sizeId  <<  1 ) ) − 1 ) (7‑41)

If sizeId is equal to 1, the value of refMatrixId shall not be equal to 0 or 3. Otherwise, if sizeId is less than or equal to 5, the value of scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] shall be in the range of 0 to matrixId, inclusive. Otherwise (sizeId is equal to 6), the value of scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] shall be in the range of 0 to matrixId / 3, inclusive.

Table 7‑4 – Specification of sizeId

|  |  |
| --- | --- |
| **Size of quantization matrix** | **sizeId** |
| 1x1 | 0 |
| 2x2 | 1 |
| 4x4 | 2 |
| 8x8 | 3 |
| 16x16 | 4 |
| 32x32 | 5 |
| 64x64 | 6 |

Table 7‑5 – Specification of matrixId according to sizeId, prediction mode and colour component

|  |  |  |  |
| --- | --- | --- | --- |
| **sizeId** | **CuPredMode** | **cIdx (Colour component)** | **matrixId** |
| 2, 3, 4, 5, 6 | MODE\_INTRA, MODE\_IBC | 0 (Y) | 0 |
| 1, 2, 3, 4, 5, 6 | MODE\_INTRA, MODE\_IBC | 1 (Cb) | 1 |
| 1, 2, 3, 4, 5, 6 | MODE\_INTRA, MODE\_IBC | 2 (Cr) | 2 |
| 2, 3, 4, 5, 6 | MODE\_INTER | 0 (Y) | 3 |
| 1, 2, 3, 4, 5, 6 | MODE\_INTER | 1 (Cb) | 4 |
| 1, 2, 3, 4, 5, 6 | MODE\_INTER | 2 (Cr) | 5 |

**scaling\_list\_dc\_coef\_minus8**[ sizeId −4 ][ matrixId ] plus 8 specifies the value of the variable ScalingFactor[ 4 ][ matrixId ][ 0 ][ 0 ] for the scaling list for the 16x16 size when sizeId is equal to 4 and specifies the value of ScalingFactor[ 5 ][ matrixId ][ 0 ][ 0 ] for the scaling list for the 32x32 size when sizeId is equal to 5, and specifies the value of ScalingFactor[ 6 ][ matrixId ][ 0 ][ 0 ] for the scaling list for the 64x64 size when sizeId is equal to 6. The value of scaling\_list\_dc\_coef\_minus8[ sizeId −4 ][ matrixId ] shall be in the range of −7 to 247, inclusive.

When scaling\_list\_pred\_mode\_flag[ sizeId ][ matrixId ] is equal to 0, scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] is equal to 0 and sizeId is greater than 3, the value of scaling\_list\_dc\_coef\_minus8[ sizeId −4 ][ matrixId ] is inferred to be equal to 8.

When scaling\_list\_pred\_matrix\_id\_delta[ sizeId ][ matrixId ] is not equal to 0 and sizeId is greater than 3, the value of scaling\_list\_dc\_coef\_minus8[ sizeId −4 ][ matrixId ] is inferred to be equal to scaling\_list\_dc\_coef\_minus8[ sizeId −4 ][ refMatrixId ], where the value of refMatrixId is given by Equation (7–39).

**scaling\_list\_delta\_coef** specifies the difference between the current matrix coefficient ScalingList[ sizeId ][ matrixId ][ i ] and the previous matrix coefficient ScalingList[ sizeId ][ matrixId ][ i − 1 ], when scaling\_list\_pred\_mode\_flag[ sizeId ][ matrixId ] is equal to 1. The value of scaling\_list\_delta\_coef shall be in the range of −128 to 127, inclusive. The value of ScalingList[ sizeId ][ matrixId ][ i ] shall be greater than 0. When scaling\_list\_pred\_mode\_flag[ sizeId ][ matrixId ] is equal to 1 and scaling\_list\_delta\_coef is not present,the value of ScalingList[ sizeId ][ matrixId ][ i ] is inferred to be 0.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** |
| **ScalingList[ 1 ][ 1,2,4,5 ][ i ]** | 16 | 16 | 16 | 16 |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 2 ][ 0..5 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

Table 7‑8 – Specification of default values of ScalingList[ 3..5 ][ matrixId ][ i ] with i = 0..63

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 3..5 ][ 0..5 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 16** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 3..5 ][ 0..5 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 32** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 3..5 ][ 0..5 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 48** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 3..5 ][ 0..5 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

Table 7‑9 – Specification of default values of ScalingList[ 6 ][ matrixId ][ i ] with i = 0..63

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 6 ][ 0, 3 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 16** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 6 ][ 0, 3 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 32** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 6 ][ 0, 3 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| **i − 48** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **ScalingList[ 6 ][ 0, 3 ][ i ]** | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

The five-dimensional array ScalingFactor[ sizeId ][sizeId][ matrixId ][ x ][ y ], with x, y = 0..( 1  <<  sizeId ) **−** 1, specifies the array of scaling factors according to the variables sizeId specified in Table 7‑4 and matrixId specified in Table 7‑4.

The elements of the quantization matrix of size 2x2, ScalingFactor[ 1 ][ 1 ][ matrixId ][ x ][ y ], with i = 0..3, matrixId = 1, 2, 4, 5, are derived as follows:

x = DiagScanOrder[ 1 ][ 1 ][ i ][ 0 ] (7‑42)

y = DiagScanOrder[ 1 ][ 1 ][ i ][ 1 ] (7‑43)

ScalingFactor[ 1 ][ 1 ][ matrixId ][ x ][ y ] = ScalingList[ 1 ][ matrixId ][ i ] (7‑44)

The elements of the quantization matrix of size 4x4, ScalingFactor[ 2 ][ 2 ][ matrixId ][ x ][ y ], with i = 0..15, matrixId = 0..5, are derived as follows:

x = DiagScanOrder[ 2 ][ 2 ][ i ][ 0 ] (7‑45)

y = DiagScanOrder[ 2 ][ 2 ][ i ][ 1 ] (7‑46)

ScalingFactor[ 2 ][ 2 ][ matrixId ][ x ][ y ] = ScalingList[ 2 ][ matrixId ][ i ] (7‑47)

The elements of the quantization matrix of size 8x8, ScalingFactor[ 3 ][ 3 ][ matrixId ][ x ][ y ], with i = 0..63, matrixId = 0..5, are derived as follows:

x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] (7‑48)

y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] (7‑49)

ScalingFactor[ 3 ][ 3 ][ matrixId ][ x ][ y ] = ScalingList[ 3 ][ matrixId ][ i ] (7‑50)

The elements of the quantization matrix of size 16x16, ScalingFactor[ 4 ][ 4 ][ matrixId ][ x ][ y ], with i = 0..63, j = 0..1, k = 0..1, matrixId = 0..5, are derived as follows:

x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] (7‑51)

y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] (7‑52)

ScalingFactor[ 4 ][ 4 ][ matrixId ][ x \* 2 + k ][ y \* 2 + j ] = ScalingList[ 4 ][ matrixId ][ i ] (7‑53)

ScalingFactor[ 4 ][ 4 ][ matrixId ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 0 ][ matrixId ] + 8 (7‑54)

The elements of the quantization matrix of size 32x32, ScalingFactor[ 5 ][ 5 ][ matrixId ][ x ][ y ], with i = 0..63, j = 0..3, k = 0..3, matrixId = 0..5, are derived as follows:

x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] (7‑55)

y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] (7‑56)

ScalingFactor[ 5 ][ 5 ][ matrixId ][ x \* 4 + k ][ y \* 4 + j ] = ScalingList[ 5 ][ matrixId ][ i ] (7‑57)

ScalingFactor[ 5 ][ 5 ][ matrixId ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 1 ][ matrixId ] + 8 (7‑58)

The elements of the quantization matrix of size 64x64, ScalingFactor[ 6 ][ 6 ][ matrixId ][ x ][ y ], i = 0..63, j = 0..7, k = 0..7, matrixId = 0, 3, are derived as follows:

x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] (7‑59)

y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] (7‑60)

ScalingFactor[ 6 ][ 6 ][ matrixId ][ x \* 8 + k ][ y \* 8 + j ] = ScalingList[ 6 ][ matrixId ][ i ] (7‑61)

ScalingFactor[ 6 ][ 6 ][ matrixId ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 2 ][ matrixId ] + 8 (7‑62)

When ChromaArrayType is equal to 3, the elements of the chroma quantization matrix of size 64x64, ScalingFactor[ 6 ][ 6 ][ matrixId ][ x ][ y ], with i = 0..63, j = 0..7, k = 0..7, matrixId = 1, 2, 4 and 5 are derived as follows:

x = DiagScanOrder[ 3 ][ 3 ][ i ][ 0 ] (7‑63)

y = DiagScanOrder[ 3 ][ 3 ][ i ][ 1 ] (7‑64)

ScalingFactor[ 6 ][ 6 ][ matrixId ][ x \* 8 + k ][ y \* 8 + j ] = ScalingList[ 5 ][ matrixId ][ i ] (7‑65)

ScalingFactor[ 6 ][ 6][ matrixId ][ 0 ][ 0 ] = scaling\_list\_dc\_coef\_minus8[ 1 ][ matrixId ] + 8 (7‑66)

For a quantization matrix with rectangular size, the five-dimensional array ScalingFactor[ wId ][hId][ matrixId ][ x ][ y ], with wId = 0..6, hId = 0..6, matrixId = 0..5, x = 0..( 1  <<  wId ) − 1, y = 0..( 1  << hId ) − 1, and wId != hId, specifies the array of scaling factors for size (1 << wId)x(1 << hId). The elements of ScalingFactor[ wId ][hId][ matrixId ][ x ][ y ] are generated by using ScalingList[ maxSizeId ][ matrixId ][ i ] with maxSizeId = max( wId, hId ), as follows:

k = min( maxSizeId, 3 ),   
x = DiagScanOrder[ k ][ k ][ i ][ 0 ]  
y = DiagScanOrder[ k ][ k ][ i ][ 1 ]  
ratioW = ( 1 << wId )  /  ( 1 << k )  
ratioH = ( 1 << hId )  /  ( 1 << k )  
diffWH = 1 << abs(wId − hId) (7‑67)  
if( wId > hId )  
 ScalingFactor[ wId ][ hId ][ matrixId ][ x ][ y ] =  
 ScalingList[ maxSizeId ][ matrixId ]  
 [ RasterScanOrder[ k ][ k ][ ( 1 << k ) \* ( ( y \* diffWH ) / ratioW ) + x / ratioW ] ]  
else( wId < hId )  
 ScalingFactor[ wId ][ hId ][ matrixId ][ x ][ y ] =  
 ScalingList[ maxSizeId ][ matrixId ]  
 [ RasterScanOrder[ k ][ k ][ ( 1 << k ) \* ( y / ratioH ) + (x \* diffWH) / ratioH ] ]

NOTE – A quantization matrix element ScalingFactor[ wId ][ hId ][ matrixId ][ x ][ y ] can be zeroed out when any of the following conditions is true

* x is greater than 32
* y is greater than 32
* The decoded tu is not coded by default transform mode (i.e. transform type is not equal to 0 ) and x is greater than 16
* The decoded tu is not coded by default transform mode (i.e. transform type is not equal to 0 ) and y is greater than 16

### Profile, tier, and level semantics

#### General profile, tier, and level semantics

When the profile\_tier\_level( ) structure is included in a DPS, the BitstreamInScope is the entire bitstream for which the DPS is the active DPS. When the profile\_tier\_level( ) structure is included in an SPS, the BitstreamInScope is the CVS for which the SPS is the active SPS.

**general\_profile\_idc** indicates a profile to which BitstreamInScope conforms as specified in Annex A. Bitstreams shall not contain values of general\_profile\_idc other than those specified in Annex A. Other values of general\_profile\_idc are reserved for future use by ITU-T | ISO/IEC.

**general\_tier\_flag** specifies the tier context for the interpretation of general\_level\_idc as specified in Annex A.

**general\_sub\_profile\_idc** indicates interoperability metadata registered as specified by X Recommendation ITU-T T.35, the contents of which are not specified in this Specification.

**general\_level\_idc** indicates a level to which BitstreamInScope conforms as specified in Annex A. Bitstreams shall not contain values of general\_level\_idc other than those specified in Annex A. Other values of general\_level\_idc are reserved for future use by ITU-T | ISO/IEC.

NOTE 1 – A greater value of general\_level\_idc indicates a higher level. The maximum level signalled in the DPS for BitstreamInScope may be higher than the level signalled in the SPS for a CVS contained within BitstreamInScope.

NOTE 2 – When BitstreamInScope conforms to multiple profiles, general\_profile\_idc should indicate the profile that provides the preferred decoded result or the preferred bitstream identification, as determined by the encoder (in a manner not specified in this Specification).

NOTE 3 – When the profile\_tier\_level( ) syntax structure is included in a DPS and CVSs of BitstreamInScope conform to different profiles, general\_profile\_idc and level\_idc should indicate the profile and level for a decoder that is capable of decoding BitstreamInScope.

**sub\_layer\_level\_present\_flag**[ i ] equal to 1 specifies that level information is present in the profile\_tier\_level( ) syntax structure for the sub-layer representation with TemporalId equal to i. sub\_layer\_level\_present\_flag[ i ] equal to 0 specifies that level information is not present in the profile\_tier\_level( ) syntax structure for the sub-layer representation with TemporalId equal to i.

**ptl\_alignment\_zero\_bits** shall be equal to 0.

The semantics of the syntax element **sub\_layer\_level\_idc**[ i ] is, apart from the specification of the inference of not present values, the same as the syntax element general\_level\_idc, but apply to the sub-layer representation with TemporalId equal to i.

#### General constraint information semantics

**general\_progressive\_source\_flag** and **general\_interlaced\_source\_flag** are interpreted as follows:

– If general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 0, the source scan type of the pictures in BitstreamInScope should be interpreted as progressive only.

– Otherwise, if general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 1, the source scan type of the pictures in BitstreamInScope should be interpreted as interlaced only.

– Otherwise, if general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 0, the source scan type of the pictures in BitstreamInScope should be interpreted as unknown or unspecified.

– Otherwise (general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 1), the source scan type of each picture in BitstreamInScope should be interpreted as unknown or unspecified.

[Ed. If picture timing SEI message is specified for VVC, general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 1 can be specified to indicate the source scan type of each picture at the picture level using the syntax element source\_scan\_type in a picture timing SEI message.]

NOTE 1 – Decoders may ignore the values of general\_progressive\_source\_flag and general\_interlaced\_source\_flag. Moreover, the actual source scan type of the pictures is outside the scope of this Specification and the method by which the encoder selects the values of general\_progressive\_source\_flag and general\_interlaced\_source\_flag is unspecified.

**general\_non\_packed\_constraint\_flag** equal to 1 indicates that the cropped output pictures of the decoded bitstream are suitable to be displayed. general\_non\_packed\_constraint\_flag equal to 0 indicates that the cropped output pictures of the decoded bitstream may require further processing to be suitable for displaying. [Ed. Consider a better naming of this constraint flag, e.g., display\_suitability\_flag or something like that.]

NOTE 2 – Examples for content that requires additional processing before displaying are bitstreams containing one or more frame packing arrangement SEI messages, equirectangular projection SEI messages, or cubemap projection SEI messages. [Ed. This NOTE should be updated at a later phase of VVC standardization based on which SEI messages are adopted into the VVC design.]

**general\_frame\_only\_constraint\_flag** equal to 1 specifies that BitstreamInScope conveys pictures that represent frames. general\_frame\_only\_constraint\_flag equal to 0 specifies that BitstreamInScope conveys pictures that may or may not represent frames.

NOTE 3 – Decoders may ignore the value of general\_frame\_only\_constraint\_flag, as there are no decoding process requirements associated with it.

**intra\_only\_constraint\_flag** equal to 1 specifies that slice\_type shall be equal to I. intra\_only\_constraint\_flag equal to 0 does not impose a constraint.

**max\_bitdepth\_constraint\_idc** specifies that bit\_depth\_luma\_minus8 and bit\_depth\_chroma\_minus8 shall be in the range of 0 to max\_bitdepth\_constraint\_idc, inclusive.

**max\_chroma\_format\_constraint\_idc** specifies that chroma\_format\_idc shall be in the range of 0 to max\_chroma\_format\_constraint\_idc, inclusive.

**frame\_only\_constraint\_flag** equal to 1 specifies that the CVS conveys pictures that represent frames. frame\_only\_constraint\_flag equal to 0 does not impose a constraint.

**no\_qtbtt\_dual\_tree\_intra\_constraint\_flag** equal to 1 specifies that qtbtt\_dual\_tree\_intra\_flag shall be equal to 0. no\_qtbtt\_dual\_tree\_intra\_constraint\_flag equal to 0 does not impose a constraint.

**no\_partition\_constraints\_override\_constraint\_flag** equal to 1 specifies that partition\_constraints\_override\_enabled\_flag shall be equal to 0. no\_partition\_constraints\_override\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sao\_constraint\_flag** equal to 1 specifies that sps\_sao\_enabled\_flag shall be equal to 0. no\_sao\_constraint\_flag equal to 0 does not impose a constraint.

**no\_alf\_constraint\_flag** equal to 1 specifies that sps\_alf\_enabled\_flag shall be equal to 0. no\_alf\_constraint\_flag equal to 0 does not impose a constraint.

**no\_pcm\_constraint\_flag** equal to 1 specifies that sps\_pcm\_enabled\_flag shall be equal to 0. no\_pcm\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ref\_wraparound\_constraint\_flag** equal to 1 specifies that sps\_ref\_wraparound\_enabled\_flag shall be equal to 0. no\_ref\_wraparound\_constraint\_flag equal to 0 does not impose a constraint.

**no\_temporal\_mvp\_constraint\_flag** equal to 1 specifies that sps\_temporal\_mvp\_enabled\_flag shall be equal to 0. no\_temporal\_mvp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sbtmvp\_constraint\_flag** equal to 1 specifies that sps\_sbtmvp\_enabled\_flag shall be equal to 0. no\_sbtmvp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_amvr\_constraint\_flag** equal to 1 specifies that sps\_amvr\_enabled\_flag shall be equal to 0. no\_amvr\_constraint\_flag equal to 0 does not impose a constraint.

**no\_bdof\_constraint\_flag** equal to 1 specifies that sps\_bdof\_enabled\_flag shall be equal to 0. no\_bdof\_constraint\_flag equal to 0 does not impose a constraint.

**no\_dmvr\_constraint\_flag** equal to 1 specifies that sps\_dmvr\_enabled\_flag shall be equal to 0. no\_dmvr\_constraint\_flag equal to 0 does not impose a constraint.

**no\_cclm\_constraint\_flag** equal to 1 specifies that sps\_cclm\_enabled\_flag shall be equal to 0. no\_cclm\_constraint\_flag equal to 0 does not impose a constraint.

**no\_mts\_constraint\_flag** equal to 1 specifies that sps\_mts\_enabled\_flag shall be equal to 0. no\_mts\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sbt\_constraint\_flag** equal to 1 specifies that sps\_sbt\_enabled\_flag shall be equal to 0. no\_sbt\_constraint\_flag equal to 0 does not impose a constraint.

**no\_affine\_motion\_constraint\_flag** equal to 1 specifies that sps\_affine\_enabled\_flag  shall be equal to 0. no\_affine\_motion\_constraint\_flag equal to 0 does not impose a constraint.

**no\_bcw\_constraint\_flag** equal to 1 specifies that sps\_bcw\_enabled\_flag shall be equal to 0. no\_bcw\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ibc\_constraint\_flag** equal to 1 specifies that sps\_ibc\_enabled\_flag shall be equal to 0. no\_ibc\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ciip\_constraint\_flag** equal to 1 specifies that sps\_ciip\_enabled\_flag shall be equal to 0. no\_cipp\_constraint\_flag equal to 0 does not impose a constraint.

**no\_fpel\_mmvd\_constraint\_flag** equal to 1 specifies that sps\_fpel\_mmvd\_enabled\_flag shall be equal to 0. no\_fpel\_mmvd\_constraint\_flag equal to 0 does not impose a constraint.

**no\_triangle\_constraint\_flag** equal to 1 specifies that sps\_triangle\_enabled\_flag shall be equal to 0. no\_triangle\_constraint\_flag equal to 0 does not impose a constraint.

**no\_ladf\_constraint\_flag** equal to 1 specifies that sps\_ladf\_enabled\_flag shall be equal to 0. no\_ladf\_constraint\_flag equal to 0 does not impose a constraint.

**no\_transform\_skip\_constraint\_flag** equal to 1 specifies that sps\_transfrom\_skip\_enabled\_flag shall be equal to 0. no\_transform\_skip\_constraint\_flag equal to 0 does not impose a constraint.

**no\_qp\_delta\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that cu\_qp\_delta\_enabled\_flag shall be equal to 0. no\_qp\_delta\_constraint\_flag equal to 0 does not impose a constraint.

**no\_dep\_quant\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that dep\_quant\_enabled\_flag shall be equal to 0. no\_dep\_quant\_constraint\_flag equal to 0 does not impose a constraint.

**no\_sign\_data\_hiding\_constraint\_flag** equal to 1 specifies that it is a requirement of bitstream conformance that sign\_data\_hiding\_enabled\_flag shall be equal to 0. no\_sign\_data\_hiding\_constraint\_flag equal to 0 does not impose a constraint.

**gci\_alignment\_zero\_bits** shall be equal to 0.

### Supplemental enhancement information message semantics

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

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

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

### Slice header semantics

#### General slice header semantics

When present, the value of each of the slice header syntax elements slice\_pic\_parameter\_set\_id, slice\_pic\_order\_cnt\_lsb, and slice\_temporal\_mvp\_enabled\_flag shall be the same in all slice headers of a coded picture.

**slice\_pic\_parameter\_set\_id** specifies the value of pps\_pic\_parameter\_set\_id for the PPS in use. The value of slice\_pic\_parameter\_set\_id shall be in the range of 0 to 63, inclusive.

It is a requirement of bitstream conformance that the value of TemporalId of the current picture shall be greater than or equal to the value of TemporalId of the PPS that has pps\_pic\_parameter\_set\_id equal to slice\_pic\_parameter\_set\_id.

**slice\_address** specifies the slice address of the slice. When not present, the value of slice\_address is inferred to be equal to 0.

If rect\_slice\_flag is equal to 0, the following applies:

* The slice address is the brick ID as specified by Equation 6‑8.
* The length of slice\_address is Ceil( Log2 ( NumBricksInPic ) ) bits.
* The value of slice\_address shall be in the range of 0 to NumBricksInPic − 1, inclusive.

Otherwise (rect\_slice\_flag is equal to 1), the following applies:

* The slice address is the slice ID of the slice.
* The length of slice\_address is signalled\_slice\_id\_length\_minus1 + 1 bits.
* If signalled\_slice\_id\_flag is equal to 0, the value of slice\_address shall be in the range of 0 to num\_slices\_in\_pic\_minus1, inclusive. Otherwise, the value of slice\_address shall be in the range of 0 to 2( signalled\_slice\_id\_length\_minus1 + 1 ) − 1, inclusive.

It is a requirement of bitstream conformance that the following constraints apply:

* The value of slice\_address shall not be equal to the value of slice\_address of any other coded slice NAL unit of the same coded picture.
* The slices of a picture shall be in increasing order of their slice\_address values.
* The shapes of the slices of a picture shall be such that each brick, when decoded, shall have its entire left boundary and entire top boundary consisting of a picture boundary or consisting of boundaries of previously decoded brick(s).

**num\_bricks\_in\_slice\_minus1**, when present, specifies the number of bricks in the slice minus 1. The value of num\_bricks\_in\_slice\_minus1 shall be in the range of 0 to NumBricksInPic − 1, inclusive. When rect\_slice\_flag is equal to 0 and single\_brick\_per\_slice\_flag is equal to 1, the value of num\_bricks\_in\_slice\_minus1 is inferred to be equal to 0.

The variable NumBricksInCurrSlice, which specifies the number of bricks in the current slice, and SliceBrickIdx[ i ], which specifies the brick index of the i-th brick in the current slice, are derived as follows:

if( rect\_slice\_flag ) {  
 sliceIdx = 0  
 while( slice\_address != slice\_id[ sliceIdx ] )  
 sliceIdx++  
 NumBricksInCurrSlice = NumBricksInSlice[ sliceIdx ]  
 brickIdx = top\_left\_brick\_idx[ sliceIdx ]  
 botBrickIdx = brickIdx + bottom\_right\_brick\_idx\_delta[ sliceIdx ]  
 for( bIdx = 0; brickIdx <= botBrickIdx; brickIdx++ ) (7‑68)  
 if( BricksToSliceMap[ brickIdx ] = = sliceIdx )  
 SliceBrickIdx[ bIdx++ ] = brickIdx  
} else {  
 NumBricksInCurrSlice = num\_bricks\_in\_slice\_minus1 + 1  
 SliceBrickIdx[ 0 ] = slice\_address  
 for( i = 1; i < NumBricksInCurrSlice; i++ )  
 SliceBrickIdx[ i ] = SliceBrickIdx[ i − 1 ] + 1  
}

**slice\_type** specifies the coding type of the slice according to Table 7‑10.

Table 7‑10 – Name association to slice\_type

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

When NalUnitType is a value of NalUnitType in the range of IDR\_W\_RADL to CRA\_NUT, inclusive, 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 values 0, 1 and 2 correspond to the Y, Cb and Cr planes, respectively.

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

**recovery\_poc\_cnt** specifies the recovery point of decoded pictures in output order. If there is a picture picA that follows the current GRA picture in decoding order in the CVS and that has PicOrderCntVal equal to the PicOrderCntVal of the current GRA picture plus the value of recovery\_poc\_cnt, the picture picA is referred to as the recovery point picture. Otherwise, the first picture in output order that has PicOrderCntVal greater than the PicOrderCntVal of the current picture plus the value of recovery\_poc\_cnt is referred to as the recovery point picture. The recovery point picture shall not precede the current GRA picture in decoding order. The value of recovery\_poc\_cnt shall be in the range of −MaxPicOrderCntLsb / 2 to MaxPicOrderCntLsb / 2 − 1, inclusive.

The variable RpPicOrderCntVal is derived as follows:

RpPicOrderCntVal = PicOrderCntVal + recovery\_poc\_cnt (7‑69)

**slice\_pic\_order\_cnt\_lsb** specifies the picture order count modulo MaxPicOrderCntLsb for the current picture. The length of the slice\_pic\_order\_cnt\_lsb syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits. The value of the slice\_pic\_order\_cnt\_lsb shall be in the range of 0 to MaxPicOrderCntLsb − 1, inclusive.

**no\_output\_of\_prior\_pics\_flag** affects the output of previously-decoded pictures in the decoded picture buffer after the decoding of an IDR picture that is not the first picture in the bitstream as specified in Annex C.

**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.

**ref\_pic\_list\_sps\_flag**[ i ] equal to 1 specifies that reference picture list i of the current slice is derived based on one of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i in the active SPS. ref\_pic\_list\_sps\_flag[ i ] equal to 0 specifies that reference picture list i of the current slice is derived based on the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure with listIdx equal to i that is directly included in the slice headers of the current picture. When num\_ref\_pic\_lists\_in\_sps[ i ] is equal to 0, the value of ref\_pic\_list\_sps\_flag[ i ] is inferred to be equal to 0. When rpl1\_idx\_present\_flag is equal to 0, the value of ref\_pic\_list\_sps\_flag[ 1 ] is inferred to be equal to ref\_pic\_list\_sps\_flag[ 0 ].

**ref\_pic\_list\_idx**[ i ] specifies the index, into the list of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures with listIdx equal to i included in the active SPS, of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure with listIdx equal to i that is used for derivation of reference picture list i of the current picture. The syntax element ref\_pic\_list\_idx[ i ] is represented by Ceil( Log2( num\_ref\_pic\_lists\_in\_sps[ i ] ) ) bits. When not present, the value of ref\_pic\_list\_idx[ i ] is inferred to be equal to 0. The value of ref\_pic\_list\_idx[ i ] shall be in the range of 0 to num\_ref\_pic\_lists\_in\_sps[ i ] − 1, inclusive. When ref\_pic\_list\_sps\_flag[ i ] is equal to 1 and num\_ref\_pic\_lists\_in\_sps[ i ] is equal to 1, the value of ref\_pic\_list\_idx[ i ] is inferred to be equal to 0. When ref\_pic\_list\_sps\_flag[ i ] is equal to 1 and rpl1\_idx\_present\_flag is equal to 0, the value of ref\_pic\_list\_idx[ 1 ] is inferred to be equal to ref\_pic\_list\_idx[ 0 ].

The variable RplsIdx[ i ] is derived as follows:

RplsIdx[ i ] = ref\_pic\_list\_sps\_flag[ i ] ? ref\_pic\_list\_idx[ i ] : num\_ref\_pic\_lists\_in\_sps[ i ] (7‑70)

**slice\_poc\_lsb\_lt**[ i ][ j ] specifies the value of the picture order count modulo MaxPicOrderCntLsb of the j-th LTRP entry in the i-th reference picture list. The length of the slice\_poc\_lsb\_lt[ i ][ j ] syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits.

The variable PocLsbLt[ i ][ j ] is derived as follows:

PocLsbLt[ i ][ j ] = ltrp\_in\_slice\_header\_flag[ i ][ RplsIdx[ i ] ] ? (7‑71)  
 slice\_poc\_lsb\_lt[ i ][ j ] : rpls\_poc\_lsb\_lt[ listIdx ][ RplsIdx[ i ] ][ j ]

**delta\_poc\_msb\_present\_flag**[ i ][ j ] equal to 1 specifies that delta\_poc\_msb\_cycle\_lt[ i ][ j ] is present. delta\_poc\_msb\_present\_flag[ i ][ j ]equal to 0 specifies that delta\_poc\_msb\_cycle\_lt[ i ][ j ] is not present.

Let prevTid0Pic be the previous picture in decoding order that has NuhLayerId the same as the current picture, has TemporalId equal to 0, and is not a RASL or RADL picture. Let setOfPrevPocVals be a set consisting of the following:

– the PicOrderCntVal of prevTid0Pic,

– the PicOrderCntVal of each picture that is referred to by entries in RefPicList[ 0 ] or RefPicList[ 1 ] of prevTid0Pic and has NuhLayerId the same as the current picture,

– the PicOrderCntVal of each picture that follows prevTid0Pic in decoding order, has NuhLayerId the same as the current picture, and precedes the current picture in decoding order.

When there is more than one value in setOfPrevPocVals for which the value modulo MaxPicOrderCntLsb is equal to PocLsbLt[ i ][ j ], the value of delta\_poc\_msb\_present\_flag[ i ][ j ] shall be equal to 1.

**delta\_poc\_msb\_cycle\_lt**[ i ][ j ] specifies the value of the variable FullPocLt[ i ][ j ] as follows:

if( j = = 0 )  
 DeltaPocMsbCycleLt[ i ][ j ] = delta\_poc\_msb\_cycle\_lt[ i ][ j ]  
else (7‑72)  
 DeltaPocMsbCycleLt[ i ][ j ] = delta\_poc\_msb\_cycle\_lt[ i ][ j ] + DeltaPocMsbCycleLt[ i ][ j − 1 ]  
FullPocLt[ i ][ j ] = PicOrderCntVal − DeltaPocMsbCycleLt[ i ][ j ] \* MaxPicOrderCntLsb −  
 ( PicOrderCntVal & ( MaxPicOrderCntLsb − 1 ) ) + PocLsbLt[ i ][ j ]

The value of delta\_poc\_msb\_cycle\_lt[ i ][ j ] shall be in the range of 0 to 2(32 − log2\_max\_pic\_order\_cnt\_lsb\_minus4 − 4 ), inclusive. When not present, the value of delta\_poc\_msb\_cycle\_lt[ i ][ j ] is inferred to be equal to 0.

**num\_ref\_idx\_active\_override\_flag** equal to 1 specifies that the syntax element num\_ref\_idx\_active\_minus1[ 0 ] is present for P and B slices and that the syntax element num\_ref\_idx\_active\_minus1[ 1 ] is present for B slices. num\_ref\_idx\_active\_override\_flag equal to 0 specifies that the syntax elements num\_ref\_idx\_active\_minus1[ 0 ] and num\_ref\_idx\_active\_minus1[ 1 ] are not present. When not present, the value of num\_ref\_idx\_active\_override\_flag is inferred to be equal to 1.

**num\_ref\_idx\_active\_minus1**[ i ] is used for the derivation of the variable NumRefIdxActive[ i ] as specified by Equation 7‑73. The value of num\_ref\_idx\_active\_minus1[ i ] shall be in the range of 0 to 14, inclusive.

For i equal to 0 or 1, when the current slice is a B slice, num\_ref\_idx\_active\_override\_flag is equal to 1, and num\_ref\_idx\_active\_minus1[ i ] is not present, num\_ref\_idx\_active\_minus1[ i ] is inferred to be equal to 0.

When the current slice is a P slice, num\_ref\_idx\_active\_override\_flag is equal to 1, and num\_ref\_idx\_active\_minus1[ 0 ] is not present, num\_ref\_idx\_active\_minus1[ 0 ] is inferred to be equal to 0.

The variable NumRefIdxActive[ i ] is derived as follows:

for( i = 0; i < 2; i++ ) {  
 if( slice\_type = = B | | ( slice\_type = = P && i = = 0 ) ) {  
 if( num\_ref\_idx\_active\_override\_flag )  
 NumRefIdxActive[ i ] = num\_ref\_idx\_active\_minus1[ i ] + 1 (7‑73)  
 else {  
 if( num\_ref\_entries[ i ][ RplsIdx[ i ] ] >= num\_ref\_idx\_default\_active\_minus1[ i ] + 1 )  
 NumRefIdxActive[ i ] = num\_ref\_idx\_default\_active\_minus1[ i ] + 1  
 else  
 NumRefIdxActive[ i ] = num\_ref\_entries[ i ][ RplsIdx[ i ] ]  
 }  
 } else // slice\_type = = I | | ( slice\_type = = P && i = = 1 )  
 NumRefIdxActive[ i ] = 0  
}

The value of NumRefIdxActive[ i ] − 1 specifies the maximum reference index for reference picture list i that may be used to decode the slice. When the value of NumRefIdxActive[ i ] is equal to 0, no reference index for reference picture list i may be used to decode the slice.

**partition\_constraints\_override\_flag** equal to 1 specifies that partition constraint parameters are present in the slice header. partition\_constraints\_override\_flag equal to 0 specifies that partition constraint parameters are not present in the slice header. When not present, the value of partition\_constraints\_override\_flag is inferred to be equal to 0.

**slice\_log2\_diff\_min\_qt\_min\_cb\_luma** specifies the difference between the base 2 logarithm of the minimum size in luma samples of a luma leaf block resulting from quadtree splitting of a CTU and the base 2 logarithm of the minimum coding block size in luma samples for luma CUs in the current slice. The value of slice\_log2\_diff\_min\_qt\_min\_cb\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred as follows:

* If slice\_type equal to 2 (I), the value of slice\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_luma
* Otherwise (slice\_type equal to 0 (B) or 1 (P)), the value of slice\_log2\_diff\_min\_qt\_min\_cb\_luma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_inter\_slice.

**slice\_max\_mtt\_hierarchy\_depth\_luma** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in the current slice. The value of slice\_max\_mtt\_hierarchy\_depth\_luma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice\_max\_mtt\_hierarchy\_depth\_luma is inferred as follows:

* If slice\_type equal to 2 (I), the value of slice\_max\_mtt\_hierarchy\_depth\_luma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_intra\_slice\_luma
* Otherwise (slice\_type equal to 0 (B) or 1 (P)), the value of slice\_max\_mtt\_hierarchy\_depth\_luma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_inter\_slice.

**slice\_log2\_diff\_max\_bt\_min\_qt\_luma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in the current slice. The value of slice\_log2\_diff\_max\_bt\_min\_qt\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of slice\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred as follows:

* If slice\_type equal to 2 (I), the value of slice\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_luma
* Otherwise (slice\_type equal to 0 (B) or 1 (P)), the value of slice\_log2\_diff\_max\_bt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_inter\_slice.

**slice\_log2\_diff\_max\_tt\_min\_qt\_luma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a luma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a luma leaf block resulting from quadtree splitting of a CTU in in the current slice. The value of slice\_log2\_diff\_max\_tt\_min\_qt\_luma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeY, inclusive. When not present, the value of slice\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred as follows:

* If slice\_type equal to 2 (I), the value of slice\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_luma
* Otherwise (slice\_type equal to 0 (B) or 1 (P)), the value of slice\_log2\_diff\_max\_tt\_min\_qt\_luma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_inter\_slice.

**slice\_log2\_diff\_min\_qt\_min\_cb\_chroma** specifies the difference between the base 2 logarithm of the minimum size in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA and the base 2 logarithm of the minimum coding block size in luma samples for chroma CUs with treeType equal to DUAL\_TREE\_CHROMA in the current slice. The value of slice\_log2\_diff\_min\_qt\_min\_cb\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the value of slice\_log2\_diff\_min\_qt\_min\_cb\_chroma is inferred to be equal to sps\_log2\_diff\_min\_qt\_min\_cb\_intra\_slice\_chroma.

**slice\_max\_mtt\_hierarchy\_depth\_chroma** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf with treeType equal to DUAL\_TREE\_CHROMA in the current slice. The value of slice\_max\_mtt\_hierarchy\_depth\_chroma shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive. When not present, the values of slice\_max\_mtt\_hierarchy\_depth\_chroma is inferred to be equal to sps\_max\_mtt\_hierarchy\_depth\_intra\_slices\_chroma.

**slice\_log2\_diff\_max\_bt\_min\_qt\_chroma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a binary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in the current slice. The value of slice\_log2\_diff\_max\_bt\_min\_qt\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of slice\_log2\_diff\_max\_bt\_min\_qt\_chroma is inferred to be equal to sps\_log2\_diff\_max\_bt\_min\_qt\_intra\_slice\_chroma

**slice\_log2\_diff\_max\_tt\_min\_qt\_chroma** specifies the difference between the base 2 logarithm of the maximum size (width or height) in luma samples of a chroma coding block that can be split using a ternary split and the minimum size (width or height) in luma samples of a chroma leaf block resulting from quadtree splitting of a chroma CTU with treeType equal to DUAL\_TREE\_CHROMA in the current slice. The value of slice\_log2\_diff\_max\_tt\_min\_qt\_chroma shall be in the range of 0 to CtbLog2SizeY − MinQtLog2SizeC, inclusive. When not present, the value of slice\_log2\_diff\_max\_tt\_min\_qt\_chroma is inferred to be equal to sps\_log2\_diff\_max\_tt\_min\_qt\_intra\_slice\_chroma

The variables MinQtLog2SizeY, MinQtLog2SizeC, MinQtSizeY, MinQtSizeC, MaxBtSizeY, MaxBtSizeC, MinBtSizeY, MaxTtSizeY, MaxTtSizeC, MinTtSizeY, MaxMttDepthY and MaxMttDepthC are derived as follows:

MinQtLog2SizeY = MinCbLog2SizeY + slice\_log2\_diff\_min\_qt\_min\_cb\_luma (7‑74)

MinQtLog2SizeC = MinCbLog2SizeY + slice\_log2\_diff\_min\_qt\_min\_cb\_chroma (7‑75)

MinQtSizeY = 1  <<  MinQtLog2SizeY (7‑76)

MinQtSizeC = 1  <<  MinQtLog2SizeC (7‑77)

MaxBtSizeY = 1  <<  ( MinQtLog2SizeY + slice\_log2\_diff\_max\_bt\_min\_qt\_luma ) (7‑78)

MaxBtSizeC = 1  <<  ( MinQtLog2SizeC + slice\_log2\_diff\_max\_bt\_min\_qt\_chroma ) (7‑79)

MinBtSizeY = 1  <<  MinCbLog2SizeY (7‑80)

MaxTtSizeY = 1  <<  ( MinQtLog2SizeY + slice\_log2\_diff\_max\_tt\_min\_qt\_luma ) (7‑81)

MaxTtSizeC = 1  <<  ( MinQtLog2SizeC + slice\_log2\_diff\_max\_tt\_min\_qt\_chroma ) (7‑82)

MinTtSizeY = 1  <<  MinCbLog2SizeY (7‑83)

MaxMttDepthY = slice\_max\_mtt\_hierarchy\_depth\_luma (7‑84)

MaxMttDepthC = slice\_max\_mtt\_hierarchy\_depth\_chroma (7‑85)

**slice\_temporal\_mvp\_enabled\_flag** specifies whether temporal motion vector predictors can be used for inter prediction. If slice\_temporal\_mvp\_enabled\_flag is equal to 0, the syntax elements of the current picture shall be constrained such that no temporal motion vector predictor is used in decoding of the current picture. Otherwise (slice\_temporal\_mvp\_enabled\_flag is equal to 1), temporal motion vector predictors may be used in decoding of the current picture. When not present, the value of slice\_temporal\_mvp\_enabled\_flag is inferred to be equal to 0.

**mvd\_l1\_zero\_flag** equal to 1 indicates that the mvd\_coding( x0, y0, 1 ) syntax structure is not parsed and MvdL1[ x0 ][ y0 ][ compIdx ] is set equal to 0 for compIdx = 0..1. mvd\_l1\_zero\_flag equal to 0 indicates that the mvd\_coding( x0, y0, 1 ) syntax structure is parsed.

**cabac\_init\_flag** specifies the method for determining the initialization table used in the initialization process for context variables. When cabac\_init\_flag is not present, it is inferred to be equal to 0.

**collocated\_from\_l0\_flag** equal to 1 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 0. collocated\_from\_l0\_flag equal to 0 specifies that the collocated picture used for temporal motion vector prediction is derived from reference picture list 1. When collocated\_from\_l0\_flag is not present, it is inferred to be equal to 1.

**six\_minus\_max\_num\_merge\_cand** specifies the maximum number of merging motion vector prediction (MVP) candidates supported in the slice subtracted from 6. The maximum number of merging MVP candidates, MaxNumMergeCand is derived as follows:

MaxNumMergeCand = 6 − six\_minus\_max\_num\_merge\_cand (7‑86)

The value of MaxNumMergeCand shall be in the range of 1 to 6, inclusive.

**five\_minus\_max\_num\_subblock\_merge\_cand** specifies the maximum number of subblock-based merging motion vector prediction (MVP) candidates supported in the slice subtracted from 5. When five\_minus\_max\_num\_subblock\_merge\_cand is not present, it is inferred to be equal to 5 − sps\_sbtmvp\_enabled\_flag. The maximum number of subblock-based merging MVP candidates, MaxNumSubblockMergeCand is derived as follows:

MaxNumSubblockMergeCand = 5 − five\_minus\_max\_num\_subblock\_merge\_cand (7‑87)

The value of MaxNumSubblockMergeCand shall be in the range of 0 to 5, inclusive.

**slice\_fpel\_mmvd\_enabled\_flag** equal to 1 specifies that merge mode with motion vector difference uses integer sample precision in the current slice. slice\_fpel\_mmvd\_enabled\_flag equal to 0 specifies that merge mode with motion vector difference can use fractional sample precision in the current slice. When not present, the value of slice\_fpel\_mmvd\_enabled\_flag is inferred to be 0.

**max\_num\_merge\_cand\_minus\_max\_num\_triangle\_cand** specifies the maximum number of triangular merge mode candidates supported in the slice subtracted from MaxNumMergeCand. The maximum number of triangular merge mode candidates, MaxNumTriangleMergeCand is derived as follows:

MaxNumTriangleMergeCand =   
 MaxNumMergeCand − max\_num\_merge\_cand\_minus\_max\_num\_triangle\_cand (7‑88)

When max\_num\_merge\_cand\_minus\_max\_num\_triangle\_cand is present, the value of MaxNumTriangleMergeCand shall be in the range of 2 to MaxNumMergeCand, inclusive. When max\_num\_merge\_cand\_minus\_max\_num\_triangle\_cand is not present, MaxNumTriangleMergeCand is set equal to 0. When MaxNumTriangleMergeCand is equal to 0, triangle merge mode is not allowed for the current slice.

**slice\_qp\_delta** specifies the initial value of QpY to be used for the coding blocks in the slice until modified by the value of CuQpDeltaVal in the coding unit layer. The initial value of the QpY quantization parameter for the slice, SliceQpY, is derived as follows:

SliceQpY = 26 + init\_qp\_minus26 + slice\_qp\_delta (7‑89)

The value of SliceQpY shall be in the range of −QpBdOffsetY to +63, inclusive.

**slice\_cb\_qp\_offset** specifies a difference to be added to the value of pps\_cb\_qp\_offset when determining the value of the Qp′Cb quantization parameter. The value of slice\_cb\_qp\_offset shall be in the range of −12 to +12, inclusive. When slice\_cb\_qp\_offset is not present, it is inferred to be equal to 0. The value of pps\_cb\_qp\_offset + slice\_cb\_qp\_offset shall be in the range of −12 to +12, inclusive.

**slice\_cr\_qp\_offset** specifies a difference to be added to the value of pps\_cr\_qp\_offset when determining the value of the Qp′Cr quantization parameter. The value of slice\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive. When slice\_cr\_qp\_offset is not present, it is inferred to be equal to 0. The value of pps\_cr\_qp\_offset + slice\_cr\_qp\_offset shall be in the range of −12 to +12, inclusive.

**slice\_joint\_cbcr\_qp\_offset** specifies a difference to be added to the value of pps\_joint\_cbcr\_qp\_offset when determining the value of the Qp′CbCr. The value of slice\_joint\_cbcr\_qp\_offset shall be in the range of −12 to +12, inclusive. When slice\_joint\_cbcr\_qp\_offset is not present, it is inferred to be equal to 0. The value of pps\_joint\_cbcr\_qp\_offset + slice\_joint\_cbcr\_qp\_offset shall be in the range of −12 to +12, inclusive.

**slice\_sao\_luma\_flag** equal to 1 specifies that SAO is enabled for the luma component in the current slice; slice\_sao\_luma\_flag equal to 0 specifies that SAO is disabled for the luma component in the current slice. When slice\_sao\_luma\_flag is not present, it is inferred to be equal to 0.

**slice\_sao\_chroma\_flag** equal to 1 specifies that SAO is enabled for the chroma component in the current slice; slice\_sao\_chroma\_flag equal to 0 specifies that SAO is disabled for the chroma component in the current slice. When slice\_sao\_chroma\_flag is not present, it is inferred to be equal to 0.

**slice\_alf\_enabled\_flag** equal to 1 specifies that adaptive loop filter is enabled and may be applied to Y, Cb, or Cr colour component in a slice. slice\_alf\_enabled\_flag equal to 0 specifies that adaptive loop filter is disabled for all colour components in a slice.

**slice\_num\_alf\_aps\_ids\_luma** specifies the number of ALF APSs that the slice refers to. The value of slice\_num\_alf\_aps\_ids\_luma shall be in the range of 0 to 6, inclusive. [Ed. (YK): The value range of this tb(v)-coded syntax element was not discussed/agreed at the 15th JVET meeting. The value 6 is just a placeholder put here during the editing of VVC WD5. It should be discussed such that a value that makes the most sense should be put here.]

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to 1 for intra slices and slices in an IRAP picture, and set equal to 6 otherwise.

**slice\_alf\_aps\_id\_luma**[ i ] specifies the adaptation\_parameter\_set\_id of the i-th ALF APS that the slice refers to. The TemporalId of the ALF APS NAL unit having adaptation\_parameter\_set\_id equal to slice\_alf\_aps\_id\_luma[ i ] shall be less than or equal to the TemporalId of the coded slice NAL unit.

When multiple ALF APSs with the same value of adaptation\_parameter\_set\_id are referred to by two or more slices of the same picture, the multiple ALF APSs with the same value of adaptation\_parameter\_set\_id shall have the same content.

For intra slices and slices in an IRAP picture, slice\_alf\_aps\_id\_luma[ i ] shall not refer to an ALF APS associated with other pictures rather than the picture containing the intra slices or the IRAP picture.

**slice\_alf\_chroma\_idc** equal to 0 specifies that the adaptive loop filter is not applied to Cb and Cr colour components. slice\_alf\_chroma\_idc equal to 1 indicates that the adaptive loop filter is applied to the Cb colour component. slice\_alf\_chroma\_idc equal to 2 indicates that the adaptive loop filter is applied to the Cr colour component. slice\_alf\_chroma\_idc equal to 3 indicates that the adaptive loop filter is applied to Cb and Cr colour components. When slice\_alf\_chroma\_idc is not present, it is inferred to be equal to 0.

The maximum value maxVal of the truncated unary binarization tu(v) is set equal to 3.

**slice\_alf\_aps\_id\_chroma** specifies the adaptation\_parameter\_set\_id that the chroma component of the slice refers to. When slice\_alf\_aps\_id\_chroma is not present, it is inferred to be equal to slice\_alf\_aps\_id\_luma[ 0 ]. The TemporalId of the ALF APS NAL unit having adaptation\_parameter\_set\_id equal to slice\_alf\_aps\_id\_chroma shall be less than or equal to the TemporalId of the coded slice NAL unit.

For intra slices and slices in an IRAP picture, slice\_alf\_aps\_id\_chroma shall not refer to an ALF APS associated with other pictures rather than the picture containing the intra slices or the IRAP picture.

**dep\_quant\_enabled\_flag** equal to 0 specifies that dependent quantization is disabled. dep\_quant\_enabled\_flag equal to 1 specifies that dependent quantization is enabled.

**sign\_data\_hiding\_enabled\_flag** equal to 0 specifies that sign bit hiding is disabled. sign\_data\_hiding\_enabled\_flag equal to 1 specifies that sign bit hiding is enabled. When sign\_data\_hiding\_enabled\_flag is not present, it is inferred to be equal to 0.

**deblocking\_filter\_override\_flag** equal to 1 specifies that deblocking parameters are present in the slice header. deblocking\_filter\_override\_flag equal to 0 specifies that deblocking parameters are not present in the slice header. When not present, the value of deblocking\_filter\_override\_flag is inferred to be equal to 0.

**slice\_deblocking\_filter\_disabled\_flag** equal to 1 specifies that the operation of the deblocking filter is not applied for the current slice. slice\_deblocking\_filter\_disabled\_flag equal to 0 specifies that the operation of the deblocking filter is applied for the current slice. When slice\_deblocking\_filter\_disabled\_flag is not present, it is inferred to be equal to pps\_deblocking\_filter\_disabled\_flag.

**slice\_beta\_offset\_div2** and **slice\_tc\_offset\_div2** specify the deblocking parameter offsets for β and tC (divided by 2) for the current slice. The values of slice\_beta\_offset\_div2 and slice\_tc\_offset\_div2 shall both be in the range of −6 to 6, inclusive. When not present, the values of slice\_beta\_offset\_div2 and slice\_tc\_offset\_div2 are inferred to be equal to pps\_beta\_offset\_div2 and pps\_tc\_offset\_div2, respectively.

**slice\_lmcs\_enabled\_flag** equal to 1 specifies that luma mappin with chroma scaling is enabled for the current slice. slice\_lmcs\_enabled\_flag equal to 0 specifies that luma mapping with chroma scaling is not enabled for the current slice. When slice\_lmcs\_enabled\_flag is not present, it is inferred to be equal to 0.

**slice\_lmcs\_aps\_id** specifies the adaptation\_parameter\_set\_id of the LMCS APS that the slice refers to. The TemporalId of the LMCS APS NAL unit having adaptation\_parameter\_set\_id equal to slice\_lmcs\_aps\_id shall be less than or equal to the TemporalId of the coded slice NAL unit.

When multiple LMCS APSs with the same value of adaptation\_parameter\_set\_id are referred to by two or more slices of the same picture, the multiple LMCS APSs with the same value of adaptation\_parameter\_set\_id shall have the same content.

**slice\_chroma\_residual\_scale\_flag** equal to 1 specifies that chroma residual scaling is enabled for the current slice. slice\_ chroma\_residual\_scale\_flag equal to 0 specifies that chroma residual scaling is not enabled for the current slice. When slice\_ chroma\_residual\_scale\_flag is not present, it is inferred to be equal to 0.

**num\_entry\_point\_offsets** is used to specify the variable NumEntryPoints, which specifies the number of entry points in the current slice as follows:

NumEntryPoints = entropy\_coding\_sync\_enabled\_flag ? num\_entry\_point\_offsets :  
 NumBricksInCurrSlice − 1 (7‑90)

[Ed. (YK): The value range of the ue(v)-coded num\_entry\_point\_offsets is missing.]

**offset\_len\_minus1** plus 1 specifies the length, in bits, of the entry\_point\_offset\_minus1[ i ] syntax elements. The value of offset\_len\_minus1 shall be in the range of 0 to 31, inclusive.

**entry\_point\_offset\_minus1**[ i ] plus 1 specifies the i-th entry point offset in bytes, and is represented by offset\_len\_minus1 plus 1 bits. The slice data that follow the slice header consists of NumEntryPoints + 1 subsets, with subset index values ranging from 0 to NumEntryPoints, inclusive. The first byte of the slice data is considered byte 0. When present, emulation prevention bytes that appear in the slice data portion of the coded slice NAL unit are counted as part of the slice data for purposes of subset identification. Subset 0 consists of bytes 0 to entry\_point\_offset\_minus1[ 0 ], inclusive, of the coded slice data, subset k, with k in the range of 1 to NumEntryPoints − 1, inclusive, consists of bytes firstByte[ k ] to lastByte[ k ], inclusive, of the coded slice data with firstByte[ k ] and lastByte[ k ] defined as:

(7‑91)

lastByte[ k ] = firstByte[ k ] + entry\_point\_offset\_minus1[ k ] (7‑92)

The last subset (with subset index equal to NumEntryPoints) consists of the remaining bytes of the coded slice data.

When entropy\_coding\_sync\_enabled\_flag is equal to 0, each subset shall consist of all coded bits of all CTUs in the slice that are within the same brick, and the number of subsets (i.e., the value of NumEntryPoints + 1) shall be equal to the number of bricks in the slice.

When entropy\_coding\_sync\_enabled\_flag is equal to 1, each subset k with k in the range of 0 to NumEntryPoints, inclusive, shall consist of all coded bits of all CTUs in a CTU row within a brick, and the number of subsets ( i.e., the value of NumEntryPoints + 1 ) shall be equal to the total number of brick-specific luma CTU rows in the slice.

#### 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. When delta\_chroma\_log2\_weight\_denom is not present, it is inferred to be equal to 0.

The variable ChromaLog2WeightDenom is derived to be equal to luma\_log2\_weight\_denom + delta\_chroma\_log2\_weight\_denom and the value shall be in the range of 0 to 7, inclusive.

**luma\_weight\_l0\_flag**[ i ] equal to 1 specifies that weighting factors for the luma component of list 0 prediction using RefPicList[ 0 ][ i ] are present. luma\_weight\_l0\_flag[ i ] equal to 0 specifies that these weighting factors are not present.

**chroma\_weight\_l0\_flag**[ i ] equal to 1 specifies that weighting factors for the chroma prediction values of list 0 prediction using RefPicList[ 0 ][ i ] are present. chroma\_weight\_l0\_flag[ i ] equal to 0 specifies that these weighting factors are not present. When chroma\_weight\_l0\_flag[ i ] is not present, it is inferred to be equal to 0.

**delta\_luma\_weight\_l0**[ i ] is the difference of the weighting factor applied to the luma prediction value for list 0 prediction using RefPicList[ 0 ][ i ].

The variable LumaWeightL0[ i ] is derived to be equal to ( 1  <<  luma\_log2\_weight\_denom ) + delta\_luma\_weight\_l0[ i ]. When luma\_weight\_l0\_flag[ i ] is equal to 1, the value of delta\_luma\_weight\_l0[ i ] shall be in the range of −128 to 127, inclusive. When luma\_weight\_l0\_flag[ i ]is equal to 0, LumaWeightL0[ i ] is inferred to be equal to 2luma\_log2\_weight\_denom.

**luma\_offset\_l0**[ i ] is the additive offset applied to the luma prediction value for list 0 prediction using RefPicList[ 0 ][ i ]. The value of luma\_offset\_l0[ i ] shall be in the range of −128 to 127, inclusive. When luma\_weight\_l0\_flag[ i ]is equal to 0, luma\_offset\_l0[ i ] is inferred to be equal to 0.

**delta\_chroma\_weight\_l0**[ i ][ j ] is the difference of the weighting factor applied to the chroma prediction values for list 0 prediction using RefPicList[ 0 ][ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaWeightL0[ i ][ j ] is derived to be equal to ( 1  <<  ChromaLog2WeightDenom ) + delta\_chroma\_weight\_l0[ i ][ j ]. When chroma\_weight\_l0\_flag[ i ] is equal to 1, the value of delta\_chroma\_weight\_l0[ i ][ j ] shall be in the range of −128 to 127, inclusive. When chroma\_weight\_l0\_flag[ i ] is equal to 0**,** ChromaWeightL0[ i ][ j ] is inferred to be equal to 2ChromaLog2WeightDenom.

**delta\_chroma\_offset\_l0**[ i ][ j ] is the difference of the additive offset applied to the chroma prediction values for list 0 prediction using RefPicList[ 0 ][ i ] with j equal to 0 for Cb and j equal to 1 for Cr.

The variable ChromaOffsetL0[ i ][ j ] is derived as follows:

ChromaOffsetL0[ i ][ j ] = Clip3( −128, 127,  
 ( 128 + delta\_chroma\_offset\_l0[ i ][ j ] − (7‑93)  
 ( ( 128 \* ChromaWeightL0[ i ][ j ] )  >>  ChromaLog2WeightDenom ) ) )

The value of delta\_chroma\_offset\_l0[ i ][ j ] shall be in the range of −4 \* 128 to 4 \* 127, inclusive. When chroma\_weight\_l0\_flag[ i ] is equal to 0**,** ChromaOffsetL0[ i ][ j ] is inferred to be equal to 0.

**luma\_weight\_l1\_flag**[ i ]**, chroma\_weight\_l1\_flag**[ i ]**, delta\_luma\_weight\_l1**[ i ], **luma\_offset\_l1**[ i ], **delta\_chroma\_weight\_l1**[ i ][ j ] and **delta\_chroma\_offset\_l1**[ i ][ j ] have the same semantics as luma\_weight\_l0\_flag[ i ], chroma\_weight\_l0\_flag[ i ], delta\_luma\_weight\_l0[ i ], luma\_offset\_l0[ i ], delta\_chroma\_weight\_l0[ i ][ j ] and delta\_chroma\_offset\_l0[ i ][ j ], respectively, with l0, L0, list 0 and List0 replaced by l1, L1, list 1 and List1, respectively.

The variable sumWeightL0Flags is derived to be equal to the sum of luma\_weight\_l0\_flag[ i ] + 2 \* chroma\_weight\_l0\_flag[ i ], for i = 0..NumRefIdxActive[ 0 ] − 1.

When slice\_type is equal to B, the variable sumWeightL1Flags is derived to be equal to the sum of luma\_weight\_l1\_flag[ i ] + 2 \* chroma\_weight\_l1\_flag[ i ], for i = 0..NumRefIdxActive[ 1 ] − 1.

It is a requirement of bitstream conformance that, when slice\_type is equal to P, sumWeightL0Flags shall be less than or equal to 24 and when slice\_type is equal to B, the sum of sumWeightL0Flags and sumWeightL1Flags shall be less than or equal to 24.

#### Adaptive loop filter data semantics

**alf\_luma\_filter\_signal\_flag** equal to 1 specifies that a luma filter set is signalled. alf\_luma\_filter\_signal\_flag equal to 0 specifies that a luma filter set is not signalled.

**alf\_chroma\_filter\_signal\_flag** equal to 1 specifies that a chroma filter is signalled. alf\_chroma\_filter\_signal\_flag equal to 0 specifies that a chroma filter is not signalled.

The variable NumAlfFilters specifying the number of different adaptive loop filters is set equal to 25.

**alf\_luma\_clip\_flag** equal to 0 specifies that linear adaptive loop filtering is applied on luma component. alf\_luma\_clip\_flag equal to 1 specifies that non-linear adaptive loop filtering may be applied on luma component.

**alf\_luma\_num\_filters\_signalled\_minus1** plus 1 specifies the number of adpative loop filter classes for which luma coefficients can be signalled. The value of alf\_luma\_num\_filters\_signalled\_minus1 shall be in the range of 0 to NumAlfFilters − 1, inclusive.

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to NumAlfFilters − 1.

**alf\_luma\_coeff\_delta\_idx**[ filtIdx ] specifies the indices of the signalled adaptive loop filter luma coefficient deltas for the filter class indicated by filtIdx ranging from 0 to NumAlfFilters − 1. When alf\_luma\_coeff\_delta\_idx[ filtIdx ] is not present, it is inferred to be equal to 0.

The maximum value maxVal of the truncated binary binarization tb(v) is set equal to alf\_luma\_num\_filters\_signalled\_minus1.

**alf\_luma\_use\_fixed\_filter\_flag** equal to 1 specifies that a fixed filter set is used to signal adaptive loop filter coefficients. alf\_luma\_use\_fixed\_filter\_flag equal to 0 specifies that a fixed filter set is not used to signal adaptive loop filter coefficients.

**alf\_luma\_fixed\_filter\_set\_idx** specifies a fixed filter set index. The value of alf\_luma\_fixed\_filter\_set\_idx shall be in a range of 0 to 15, inclusive. The maximum value maxVal of the truncated binary binarization tb(v) is set equal to 15.

**alf\_luma\_fixed\_filter\_pred\_present\_flag** equal to 1 specifies that alf\_luma\_fixed\_filter\_pred\_flag[ i ] is present. alf\_luma\_fixed\_filter\_pred\_present\_flag equal to 0 specifies that alf\_luma\_fixed\_filter\_pred\_flag[ i ] is not present.

**alf\_luma\_fixed\_filter\_pred\_flag**[ i ] equal to 1 specifies that a fixed filter is used to predict the i-th filter coefficients. alf\_luma\_fixed\_filter\_pred\_flag[ i ] equal to 0 specifies that a fixed filter is not used to predict the i-th filter coefficients. When alf\_luma\_fixed\_filter\_pred\_flag[ i ] is not present, it is inferred to be equal to 1.

**alf\_luma\_coeff\_delta\_flag** equal to 1 indicates that alf\_luma\_coeff\_delta\_prediction\_flag is not signalled. alf\_luma\_coeff\_delta\_flag equal to 0 indicates that alf\_luma\_coeff\_delta\_prediction\_flag may be signalled.

**alf\_luma\_coeff\_delta\_prediction\_flag** equal to 1 specifies that the signalled luma filter coefficient deltas are predicted from the deltas of the previous luma coefficients.alf\_luma\_coeff\_delta\_prediction\_flag equal to 0 specifies that the signalled luma filter coefficient deltas are not predicted from the deltas of the previous luma coefficients. When not present, alf\_luma\_coeff\_delta\_prediction\_flag is inferred to be equal to 0.

**alf\_luma\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for luma filter coefficient signalling. The value of alf\_luma\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_luma\_eg\_order\_increase\_flag**[ i ]equal to 1 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is incremented by 1. alf\_luma\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for luma filter coefficient signalling is not incremented by 1.

The order expGoOrderY[ i ] of the exp-Golomb code used to decode the values of alf\_luma\_coeff\_delta\_abs[ sfIdx ][ j ] is derived as follows:

expGoOrderY[ i ] = ( i = = 0 ? alf\_luma\_min\_eg\_order\_minus1 + 1 : expGoOrderY[ i − 1 ] ) + (7‑94)  
 alf\_luma\_eg\_order\_increase\_flag[ i ]

**alf\_luma\_coeff\_flag**[ sfIdx ]equal 1 specifies that the coefficients of the luma filter indicated by sfIdx are signalled. alf\_luma\_coeff\_flag[ sfIdx ] equal to 0 specifies that all filter coefficients of the luma filter indicated by sfIdx are set equal to 0. When not present, alf\_luma\_coeff\_flag[ sfIdx ] is set equal to 1.

**alf\_luma\_coeff\_delta\_abs**[ sfIdx ][ j ]specifies the absolute value of the j-th coefficient delta of the signalled luma filter indicated by sfIdx. When alf\_luma\_coeff\_delta\_abs[ sfIdx ][ j ] is not present, it is inferred to be equal 0.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

golombOrderIdxY[ ] = { 0, 0, 1, 0, 0, 1, 2, 1, 0, 0, 1, 2 } (7‑95)

k = expGoOrderY[ golombOrderIdxY[ j ] ] (7‑96)

**alf\_luma\_coeff\_delta\_sign**[ sfIdx ][ j ]specifies the sign of the j-th luma coefficient of the filter indicated by sfIdx as follows:

* If alf\_luma\_coeff\_delta\_sign[ sfIdx ][ j ] is equal to 0, the corresponding luma filter coefficient has a positive value.
* Otherwise (alf\_luma\_coeff\_delta\_sign[ sfIdx ][ j ] is equal to 1), the corresponding luma filter coefficient has a negative value.

When alf\_luma\_coeff\_delta\_sign[ sfIdx ][ j ] is not present, it is inferred to be equal to 0.

The variable filtCoeff[ sfIdx ][ j ] with sfIdx = 0..alf\_luma\_num\_filters\_signalled\_minus1, j = 0..11 is initialized as follows:

filtCoeff[ sfIdx ][ j ] = alf\_luma\_coeff\_delta\_abs[ sfIdx ][ j ] \* (7‑97)  
 ( 1 − 2 \* alf\_luma\_coeff\_delta\_sign[ sfIdx ][ j ] )

When alf\_luma\_coeff\_delta\_prediction\_flag is equal 1, filtCoeff[ sfIdx ][ j ] with sfIdx = 1..alf\_luma\_num\_filters\_signalled\_minus1 and j = 0..11 are modified as follows:

filtCoeff[ sfIdx ][ j ] += filtCoeff[ sfIdx − 1 ][ j ] (7‑98)

The luma filter coefficients AlfCoeffL[ adaptation\_parameter\_set\_id ] with elements AlfCoeffL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ], with filtIdx = 0..NumAlfFilters − 1 and j = 0..11 are derived as follows:

AlfCoeffL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ] = filtCoeff[ alf\_luma\_coeff\_delta\_idx[ filtIdx ] ][ j ] (7‑99)

When alf\_luma\_use\_fixed\_filter\_flag is equal to 1 and alf\_luma\_fixed\_filter\_pred\_flag[ filtIdx ] is equal to 1, the following applies for filtIdx = 0..NumAlfFilters − 1 and j = 0..11:

AlfCoeffL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ] +=  (7‑100)  
 AlfFixFiltCoeff[ AlfClassToFiltMap[ alf\_luma\_fixed\_filter\_set\_idx ][ filtIdx ] ][ j ]

The fixed filter coefficients AlfFixFiltCoeff[ i ][ j ] with i = 0..64, j = 0..11 and the class to filter mapping AlfClassToFiltMap[ m ][ n ] with m = 0..15 and n = 0..24 are derived as follows:

AlfFixFiltCoeff = (7‑101)

{

{ 0, 0, 2, −3, 1, −4, 1, 7, −1, 1, −1, 5}

{ 0, 0, 0, 0, 0, −1, 0, 1, 0, 0, −1, 2}

{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0}

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, 1}

{ 2, 2, −7, −3, 0, −5, 13, 22, 12, −3, −3, 17}

{−1, 0, 6, −8, 1, −5, 1, 23, 0, 2, −5, 10}

{ 0, 0, −1, −1, 0, −1, 2, 1, 0, 0, −1, 4}

{ 0, 0, 3, −11, 1, 0, −1, 35, 5, 2, −9, 9}

{ 0, 0, 8, −8, −2, −7, 4, 4, 2, 1, −1, 25}

{ 0, 0, 1, −1, 0, −3, 1, 3, −1, 1, −1, 3}

{ 0, 0, 3, −3, 0, −6, 5, −1, 2, 1, −4, 21}

{−7, 1, 5, 4, −3, 5, 11, 13, 12, −8, 11, 12}

{−5, −3, 6, −2, −3, 8, 14, 15, 2, −7, 11, 16}

{ 2, −1, −6, −5, −2, −2, 20, 14, −4, 0, −3, 25}

{ 3, 1, −8, −4, 0, −8, 22, 5, −3, 2, −10, 29}

{ 2, 1, −7, −1, 2, −11, 23, −5, 0, 2, −10, 29}

{−6, −3, 8, 9, −4, 8, 9, 7, 14, −2, 8, 9}

{ 2, 1, −4, −7, 0, −8, 17, 22, 1, −1, −4, 23}

{ 3, 0, −5, −7, 0, −7, 15, 18, −5, 0, −5, 27}

{ 2, 0, 0, −7, 1, −10, 13, 13, −4, 2, −7, 24}

{ 3, 3, −13, 4, −2, −5, 9, 21, 25, −2, −3, 12}

{−5, −2, 7, −3, −7, 9, 8, 9, 16, −2, 15, 12}

{ 0, −1, 0, −7, −5, 4, 11, 11, 8, −6, 12, 21}

{ 3, −2, −3, −8, −4, −1, 16, 15, −2, −3, 3, 26}

{ 2, 1, −5, −4, −1, −8, 16, 4, −2, 1, −7, 33}

{ 2, 1, −4, −2, 1, −10, 17, −2, 0, 2, −11, 33}

{ 1, −2, 7, −15, −16, 10, 8, 8, 20, 11, 14, 11}

{ 2, 2, 3, −13, −13, 4, 8, 12, 2, −3, 16, 24}

{ 1, 4, 0, −7, −8, −4, 9, 9, −2, −2, 8, 29}

{ 1, 1, 2, −4, −1, −6, 6, 3, −1, −1, −3, 30}

{−7, 3, 2, 10, −2, 3, 7, 11, 19, −7, 8, 10}

{ 0, −2, −5, −3, −2, 4, 20, 15, −1, −3, −1, 22}

{ 3, −1, −8, −4, −1, −4, 22, 8, −4, 2, −8, 28}

{ 0, 3, −14, 3, 0, 1, 19, 17, 8, −3, −7, 20}

{ 0, 2, −1, −8, 3, −6, 5, 21, 1, 1, −9, 13}

{−4, −2, 8, 20, −2, 2, 3, 5, 21, 4, 6, 1}

{ 2, −2, −3, −9, −4, 2, 14, 16, 3, −6, 8, 24}

{ 2, 1, 5, −16, −7, 2, 3, 11, 15, −3, 11, 22}

{ 1, 2, 3, −11, −2, −5, 4, 8, 9, −3, −2, 26}

{ 0, −1, 10, −9, −1, −8, 2, 3, 4, 0, 0, 29}

{ 1, 2, 0, −5, 1, −9, 9, 3, 0, 1, −7, 20}

{−2, 8, −6, −4, 3, −9, −8, 45, 14, 2, −13, 7}

{ 1, −1, 16, −19, −8, −4, −3, 2, 19, 0, 4, 30}

{ 1, 1, −3, 0, 2, −11, 15, −5, 1, 2, −9, 24}

{ 0, 1, −2, 0, 1, −4, 4, 0, 0, 1, −4, 7}

{ 0, 1, 2, −5, 1, −6, 4, 10, −2, 1, −4, 10}

{ 3, 0, −3, −6, −2, −6, 14, 8, −1, −1, −3, 31}

{ 0, 1, 0, −2, 1, −6, 5, 1, 0, 1, −5, 13}

{ 3, 1, 9, −19, −21, 9, 7, 6, 13, 5, 15, 21}

{ 2, 4, 3, −12, −13, 1, 7, 8, 3, 0, 12, 26}

{ 3, 1, −8, −2, 0, −6, 18, 2, −2, 3, −10, 23}

{ 1, 1, −4, −1, 1, −5, 8, 1, −1, 2, −5, 10}

{ 0, 1, −1, 0, 0, −2, 2, 0, 0, 1, −2, 3}

{ 1, 1, −2, −7, 1, −7, 14, 18, 0, 0, −7, 21}

{ 0, 1, 0, −2, 0, −7, 8, 1, −2, 0, −3, 24}

{ 0, 1, 1, −2, 2, −10, 10, 0, −2, 1, −7, 23}

{ 0, 2, 2, −11, 2, −4, −3, 39, 7, 1, −10, 9}

{ 1, 0, 13, −16, −5, −6, −1, 8, 6, 0, 6, 29}

{ 1, 3, 1, −6, −4, −7, 9, 6, −3, −2, 3, 33}

{ 4, 0, −17, −1, −1, 5, 26, 8, −2, 3, −15, 30}

{ 0, 1, −2, 0, 2, −8, 12, −6, 1, 1, −6, 16}

{ 0, 0, 0, −1, 1, −4, 4, 0, 0, 0, −3, 11}

{ 0, 1, 2, −8, 2, −6, 5, 15, 0, 2, −7, 9}

{ 1, −1, 12, −15, −7, −2, 3, 6, 6, −1, 7, 30}

},

AlfClassToFiltMap = (7‑102)

{

{ 8, 2, 2, 2, 3, 4, 53, 9, 9, 52, 4, 4, 5, 9, 2, 8, 10, 9, 1, 3, 39, 39, 10, 9, 52 }

{ 11, 12, 13, 14, 15, 30, 11, 17, 18, 19, 16, 20, 20, 4, 53, 21, 22, 23, 14, 25, 26, 26, 27, 28, 10 }

{ 16, 12, 31, 32, 14, 16, 30, 33, 53, 34, 35, 16, 20, 4, 7, 16, 21, 36, 18, 19, 21, 26, 37, 38, 39 }

{ 35, 11, 13, 14, 43, 35, 16, 4, 34, 62, 35, 35, 30, 56, 7, 35, 21, 38, 24, 40, 16, 21, 48, 57, 39 }

{ 11, 31, 32, 43, 44, 16, 4, 17, 34, 45, 30, 20, 20, 7, 5, 21, 22, 46, 40, 47, 26, 48, 63, 58, 10 }

{ 12, 13, 50, 51, 52, 11, 17, 53, 45, 9, 30, 4, 53, 19, 0, 22, 23, 25, 43, 44, 37, 27, 28, 10, 55 }

{ 30, 33, 62, 51, 44, 20, 41, 56, 34, 45, 20, 41, 41, 56, 5, 30, 56, 38, 40, 47, 11, 37, 42, 57, 8 }

{ 35, 11, 23, 32, 14, 35, 20, 4, 17, 18, 21, 20, 20, 20, 4, 16, 21, 36, 46, 25, 41, 26, 48, 49, 58 }

{ 12, 31, 59, 59, 3, 33, 33, 59, 59, 52, 4, 33, 17, 59, 55, 22, 36, 59, 59, 60, 22, 36, 59, 25, 55 }

{ 31, 25, 15, 60, 60, 22, 17, 19, 55, 55, 20, 20, 53, 19, 55, 22, 46, 25, 43, 60, 37, 28, 10, 55, 52 }

{ 12, 31, 32, 50, 51, 11, 33, 53, 19, 45, 16, 4, 4, 53, 5, 22, 36, 18, 25, 43, 26, 27, 27, 28, 10 }

{ 5, 2, 44, 52, 3, 4, 53, 45, 9, 3, 4, 56, 5, 0, 2, 5, 10, 47, 52, 3, 63, 39, 10, 9, 52 }

{ 12, 34, 44, 44, 3, 56, 56, 62, 45, 9, 56, 56, 7, 5, 0, 22, 38, 40, 47, 52, 48, 57, 39, 10, 9 }

{ 35, 11, 23, 14, 51, 35, 20, 41, 56, 62, 16, 20, 41, 56, 7, 16, 21, 38, 24, 40, 26, 26, 42, 57, 39 }

{ 33, 34, 51, 51, 52, 41, 41, 34, 62, 0, 41, 41, 56, 7, 5, 56, 38, 38, 40, 44, 37, 42, 57, 39, 10 }

{ 16, 31, 32, 15, 60, 30, 4, 17, 19, 25, 22, 20, 4, 53, 19, 21, 22, 46, 25, 55, 26, 48, 63, 58, 55 }

},

It is a requirement of bitstream conformance that the values of AlfCoeffL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ] with filtIdx = 0..NumAlfFilters − 1, j = 0..11 shall be in the range of −27 to 27 − 1, inclusive.

**alf\_luma\_clip\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for luma clipping index signalling. The value of alf\_luma\_clip\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_luma\_clip\_eg\_order\_increase\_flag**[ i ] equal to 1 specifies that the minimum order of the exp-Golomb code for luma clipping index signalling is incremented by 1. alf\_luma\_clip\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for luma clipping index signalling is not incremented by 1.

The order kClipY[ i ] of the exp-Golomb code used to decode the values of alf\_luma\_clip\_idx[ sfIdx ][ j ] is derived as follows:

kClipY[ i ] = ( i = = 0 ? alf\_luma\_clip\_min\_eg\_order\_minus1 + 1 : kClipY[ i − 1 ] ) + (7‑103)   
 alf\_luma\_clip\_eg\_order\_increase\_flag[ i ]

**alf\_luma\_clip\_idx**[ sfIdx ][ j ] specifies the clipping index of the clipping value to use before multiplying by the j-th coefficient of the signalled luma filter indicated by sfIdx. When alf\_luma\_clip\_idx[ sfIdx ][ j ] is not present, it is inferred to be equal 0 (no clipping). It is a requirement of bitstream conformance that the values of alf\_luma\_clip\_idx[ sfIdx ][ j ] with sfIdx = 0..alf\_luma\_num\_filters\_signalled\_minus1 and j = 0..11 shall be in the range of 0 to 3, inclusive.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

k = kClipY[ golombOrderIdxY[ j ] ] (7‑104)

The variable filterClips[ sfIdx ][ j ] with sfIdx = 0..alf\_luma\_num\_filters\_signalled\_minus1, j = 0..11 is initialized as follows:

filterClips[ sfIdx ][ j ] = Round( 2( BitDepthY \* ( 4 − alf\_luma\_clip\_idx[ sfIdx ][ j ] ) / 4 ) ) (7‑105)

The luma filter clipping values AlfClipL[ adaptation\_parameter\_set\_id ] with elements AlfClipL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ], with filtIdx = 0..NumAlfFilters − 1 and j = 0..11 are derived as follows:

AlfClipL[ adaptation\_parameter\_set\_id ][ filtIdx ][ j ] = filterClips[ alf\_luma\_coeff\_delta\_idx[ filtIdx ] ][ j ] (7‑106)

**alf\_chroma\_clip\_flag** equal to 0 specifies that linear adaptive loop filtering is applied on chroma components; alf\_chroma\_clip\_flag equal to 1 specifies that non-linear adaptive loop filtering is applied on chroma component. When not present, alf\_chroma\_clip\_flag is inferred to be equal to 0.

**alf\_chroma\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for chroma filter coefficient signalling. The value of alf\_chroma\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_chroma\_eg\_order\_increase\_flag**[ i ]equal to 1 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is incremented by 1. alf\_chroma\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for chroma filter coefficient signalling is not incremented by 1

The order expGoOrderC[ i ] of the exp-Golomb code used to decode the values of alf\_chroma\_coeff\_abs[ j ] is derived as follows:

expGoOrderC[ i ] = ( i = = 0 ? alf\_chroma\_min\_eg\_order\_minus1 + 1 : expGoOrderC[ i − 1 ] ) + (7‑107)  
 alf\_chroma\_eg\_order\_increase\_flag[ i ] (7‑108)

**alf\_chroma\_coeff\_abs**[ j ]specifies the absolute value of the j-th chroma filter coefficient. When alf\_chroma\_coeff\_abs[ j ] is not present, it is inferred to be equal 0. It is a requirement of bitstream conformance that the values of alf\_chroma\_coeff\_abs[ j ] shall be in the range of 0 to 27 − 1, inclusive.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

golombOrderIdxC[ ] = { 0, 0, 1, 0, 0, 1 } (7‑109)

k = expGoOrderC[ golombOrderIdxC[ j ] ] (7‑110)

**alf\_chroma\_coeff\_sign**[ j ]specifies the sign of the j-th chroma filter coefficient as follows:

* If alf\_chroma\_coeff\_sign[ j ] is equal to 0, the corresponding chroma filter coefficient has a positive value.
* Otherwise (alf\_chroma\_coeff\_sign[ j ] is equal to 1), the corresponding chroma filter coefficient has a negative value.

When alf\_chroma\_coeff\_sign[ j ] is not present, it is inferred to be equal to 0.

The chroma filter coefficients AlfCoeffC[ adaptation\_parameter\_set\_id ] with elements AlfCoeffC[ adaptation\_parameter\_set\_id ][ j ], with j = 0..5 are derived as follows:

AlfCoeffC[ adaptation\_parameter\_set\_id ][ j ] = alf\_chroma\_coeff\_abs[ j ] \* (7‑111)  
 ( 1 − 2 \* alf\_chroma\_coeff\_sign[ j ] )

It is a requirement of bitstream conformance that the values of AlfCoeffC[ adaptation\_parameter\_set\_id ][ j ] with j = 0..5 shall be in the range of −27 − 1 to 27 − 1, inclusive.

**alf\_chroma\_clip\_min\_eg\_order\_minus1** plus 1 specifies the minimum order of the exp-Golomb code for chroma clipping index signalling. The value of alf\_chroma\_clip\_min\_eg\_order\_minus1 shall be in the range of 0 to 6, inclusive.

**alf\_chroma\_clip\_eg\_order\_increase\_flag**[ i ] equal to 1 specifies that the minimum order of the exp-Golomb code for chroma clipping index signalling is incremented by 1. alf\_chroma\_clip\_eg\_order\_increase\_flag[ i ] equal to 0 specifies that the minimum order of the exp-Golomb code for chroma clipping index signalling is not incremented by 1.

The order expGoOrderC[ i ] of the exp-Golomb code used to decode the values of alf\_chroma\_clip\_idx[ j ] is derived as follows:

kClipC[ i ] = ( i = = 0 ? alf\_chroma\_clip\_min\_eg\_order\_minus1 + 1 : kClipC[ i − 1 ] ) + (7‑112)   
 alf\_chroma\_clip\_eg\_order\_increase\_flag[ i ] (7‑113)

**alf\_chroma\_clip\_idx**[ j ] specifies the clipping index of the clipping value to use before multiplying by the j-th coefficient of the chroma filter. When alf\_chroma\_clip\_idx[ j ] is not present, it is inferred to be equal 0 (no clipping). It is a requirement of bitstream conformance that the values of alf\_chroma\_clip\_idx[ j ] with j = 0..5 shall be in the range of 0 to 3, inclusive.

The order k of the exp-Golomb binarization uek(v) is derived as follows:

k = kClipC[ golombOrderIdxC[ j ] ] (7‑114)

The chroma filter clipping values AlfClipC[ adaptation\_parameter\_set\_id ] with elements AlfClipC[ adaptation\_parameter\_set\_id ][ j ], with j = 0..5 are derived as follows:

AlfClipC[ adaptation\_parameter\_set\_id ][ j ] = Round( 2( BitDepthC − 8 ) \* 2( 8 \* ( 3 − alf\_chroma\_clip\_idx[ j ] ) / 3 ) ) (7‑115)

#### Luma mapping with chroma scaling data semantics

**lmcs\_min\_bin\_idx** specifies the minimum bin index used in the luma mapping with chroma scaling construction process. The value of lmcs\_min\_bin\_idx shall be in the range of 0 to 15, inclusive.

**lmcs\_delta\_max\_bin\_idx** specifies the delta value between 15 and the maximum bin index LmcsMaxBinIdx used in the luma mapping with chroma scaling construction process. The value of lmcs\_delta\_max\_bin\_idx shall be in the range of 0 to 15, inclusive. The value of LmcsMaxBinIdx is set equal to 15 − lmcs\_delta\_max\_bin\_idx. The value of LmcsMaxBinIdx shall be greater than or equal to lmcs\_min\_bin\_idx.

**lmcs\_delta\_cw\_prec\_minus1** plus 1 specifies the number of bits used for the representation of the syntax lmcs\_delta\_abs\_cw[ i ]. The value of lmcs\_delta\_cw\_prec\_minus1 shall be in the range of 0 to BitDepthY − 2, inclusive.

**lmcs\_delta\_abs\_cw**[ i ] specifies the absolute delta codeword value for the ith bin.

**lmcs\_delta\_sign\_cw\_flag**[ i ] specifies the sign of the variable lmcsDeltaCW[ i ] as follows:

* If lmcs\_delta\_sign\_cw\_flag[ i ] is equal to 0, lmcsDeltaCW[ i ] is a positive value.
* Otherwise ( lmcs\_delta\_sign\_cw\_flag[ i ] is not equal to 0 ), lmcsDeltaCW[ i ] is a negative value.

When lmcs\_delta\_sign\_cw\_flag[ i ] is not present, it is inferred to be equal to 0.

The variable OrgCW is derived as follows:

OrgCW = (1 << BitDepthY ) / 16 (7‑116)

The variable lmcsDeltaCW[ i ], with i = lmcs\_min\_bin\_idx..LmcsMaxBinIdx, is derived as follows:

lmcsDeltaCW[ i ] = ( 1 − 2 \* lmcs\_delta\_sign\_cw\_flag[ i ] ) \* lmcs\_delta\_abs\_cw[ i ] (7‑117)

The variable lmcsCW[ i ] is derived as follows:

* For i = 0.. lmcs\_min\_bin\_idx − 1, lmcsCW[ i ] is set equal 0.
* For i = lmcs\_min\_bin\_idx..LmcsMaxBinIdx, the following applies:

lmcsCW[ i ] = OrgCW + lmcsDeltaCW[ i ] (7‑118)

The value of lmcsCW[ i ] shall be in the range of (OrgCW>>3) to (OrgCW<<3 − 1), inclusive.

* For i = LmcsMaxBinIdx + 1..15, lmcsCW[ i ] is set equal 0.

It is a requirement of bitstream conformance that the following condition is true:

<= (1 << BitDepthY ) − 1 (7‑119)

The variable InputPivot[ i ], with i = 0..16, is derived as follows:

InputPivot[ i ] = i \* OrgCW (7‑120)

The variable LmcsPivot[ i ] with i = 0..16, the variables ScaleCoeff[ i ] and InvScaleCoeff[ i ] with i = 0..15, are derived as follows:

LmcsPivot[ 0 ] = 0;  
for( i = 0; i <= 15; i++ ) {  
 LmcsPivot[ i + 1 ] = LmcsPivot[ i ] + lmcsCW[ i ]   
 ScaleCoeff[ i ] = ( lmcsCW[ i ] \* (1 << 11) + (1 << (Log2(OrgCW) − 1))) >> (Log2(OrgCW)) (7‑121)  
 if ( lmcsCW[ i ] = = 0 )  
 InvScaleCoeff[ i ] = 0  
 else  
 InvScaleCoeff[ i ] = OrgCW \* (1 << 11) / lmcsCW[ i ]  
}

The variable ChromaScaleCoeff[ i ], with i = 0…15, is derived as follows:

if ( lmcsCW[ i ] = = 0 )  
 ChromaScaleCoeff[ i ] = (1 << 11)  
else (7‑122)  
 ChromaScaleCoeff[ i ] = InvScaleCoeff[ i ]

### Reference picture list structure semantics

The ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure may be present in an SPS or in a slice header. Depending on whether the syntax structure is included in a slice header or an SPS, the following applies:

– If present in a slice header, the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure specifies reference picture list listIdx of the current picture (the picture containing the slice).

– Otherwise (present in an SPS), the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure specifies a candidate for reference picture list listIdx, and the term "the current picture" in the semantics specified in the remainder of this clause refers to each picture that 1) has one or more slices containing ref\_pic\_list\_idx[ listIdx ] equal to an index into the list of the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structures included in the SPS, and 2) is in a CVS that has the SPS as the active SPS.

**num\_ref\_entries**[ listIdx ][ rplsIdx ] specifies the number of entries in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure. The value of num\_ref\_entries[ listIdx ][ rplsIdx ] shall be in the range of 0 to sps\_max\_dec\_pic\_buffering\_minus1 + 14, inclusive.

**ltrp\_in\_slice\_header\_flag**[ listIdx ][ rplsIdx ] equal to 0 specifies that the POC LSBs of the LTRP entries in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure are present in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure. ltrp\_in\_slice\_header\_flag[ listIdx ][ rplsIdx ] equal to 1 specifies that the POC LSBs of the LTRP entries in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure are not present in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure.

**st\_ref\_pic\_flag**[ listIdx ][ rplsIdx ][ i ] equal to 1 specifies that the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure is an STRP entry. st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] equal to 0 specifies that the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure is an LTRP entry. When not present, the value of st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] is inferred to be equal to 1.

The variable NumLtrpEntries[ listIdx ][ rplsIdx ] is derived as follows:

for( i = 0, NumLtrpEntries[ listIdx ][ rplsIdx ] = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++ )  
 if( !st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) (7‑123)  
 NumLtrpEntries[ listIdx ][ rplsIdx ]++

**abs\_delta\_poc\_st**[ listIdx ][ rplsIdx ][ i ], when the i-th entry is the first STRP entry in ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure, specifies the absolute difference between the picture order count values of the current picture and the picture referred to by the i-th entry, or, when the i-th entry is an STRP entry but not the first STRP entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure, specifies the absolute difference between the picture order count values of the pictures referred to by the i-th entry and by the previous STRP entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure.

The value of abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] shall be in the range of 0 to 215 − 1, inclusive.

**strp\_entry\_sign\_flag**[ listIdx ][ rplsIdx ][ i ] equal to 1 specifies that i-th entry in the syntax structure ref\_pic\_list\_struct( listIdx, rplsIdx ) has a value greater than or equal to 0. strp\_entry\_sign\_flag[ listIdx ][ rplsIdx ][ i ] equal to 0 specifies that the i-th entry in the syntax structure ref\_pic\_list\_struct( listIdx, rplsIdx ) has a value less than 0. When not present, the value of strp\_entry\_sign\_flag[ listIdx ][ rplsIdx ][ i ] is inferred to be equal to 1.

The list DeltaPocSt[ listIdx ][ rplsIdx ] is derived as follows:

for( i = 0; i < num\_ref\_entries[ listIdx ][ rplsIdx ]; i++ ) {  
 if( st\_ref\_pic\_flag[ listIdx ][ rplsIdx ][ i ] ) { (7‑124)  
 DeltaPocSt[ listIdx ][ rplsIdx ][ i ] = ( strp\_entry\_sign\_flag[ listIdx ][ rplsIdx ][ i ] ) ?  
 abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ] : 0 − abs\_delta\_poc\_st[ listIdx ][ rplsIdx ][ i ]  
 }  
}

**rpls\_poc\_lsb\_lt**[ listIdx ][ rplsIdx ][ i ] specifies the value of the picture order count modulo MaxPicOrderCntLsb of the picture referred to by the i-th entry in the ref\_pic\_list\_struct( listIdx, rplsIdx ) syntax structure. The length of the rpls\_poc\_lsb\_lt[ listIdx ][ rplsIdx ][ i ] syntax element is log2\_max\_pic\_order\_cnt\_lsb\_minus4 + 4 bits.

### Slice data semantics

#### General slice data semantics

**end\_of\_brick\_one\_bit** shall be equal to 1.

#### Coding tree unit semantics

The CTU is the root node of the coding tree structure.

The array IsInSmr[ x ][ y ] specifying whether the sample at ( x, y ) is located inside a shared merging candidate list region, is initialized as follows for x = 0..CtbSizeY − 1 and y = 0..CtbSizeY − 1:

IsInSmr[ x ][ y ] = FALSE (7‑125)

**alf\_ctb\_flag**[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ]equal to 1 specifies that the adaptive loop filter is applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location ( xCtb, yCtb ). alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] equal to 0 specifies that the adaptive loop filter is not applied to the coding tree block of the colour component indicated by cIdx of the coding tree unit at luma location ( xCtb, yCtb ).

When alf\_ctb\_flag[ cIdx ][ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is not present, it is inferred to be equal to 0.

**alf\_ctb\_use\_first\_aps\_flag** equal to 1 sepcifies that the filter information in APS with adaptive\_parameter\_set\_id equal to slice\_alf\_aps\_id\_luma[ 0 ] is used. alf\_ctb\_use\_first\_aps\_flag equal to 0 specifies that the luma CTB does not use the filter information in APS with adaptive\_parameter\_set\_id equal to slice\_alf\_aps\_id\_luma[ 0 ]. When alf\_ctb\_use\_first\_aps\_flag is not present, it is inferred to be equal to 0.

**alf\_use\_aps\_flag** equal to 0 specifies that one of the fixed filter sets is applied to the luma CTB. alf\_use\_aps\_flag equal to 1 specifies that a filter set from an APS is applied to the luma CTB. When alf\_use\_aps\_flag is not present, it is inferred to be equal to 0.

**alf\_luma\_prev\_filter\_idx\_minus1** plus 1 specifies the previous filter that is applied to the luma CTB. The value of alf\_luma\_prev\_filter\_idx\_minus1 shall be in a range of 0 to slice\_num\_alf\_aps\_ids\_luma − 2, inclusive.

The variable AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] specifying the filter set index for the luma CTB at location ( xCtb, yCtb ) is derived as follows:

* If alf\_ctb\_use\_first\_aps\_flag is equal to 1, AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is set equal to 16.
* Otherwise, if alf\_use\_aps\_flag is equal to 0, AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is set equal to alf\_luma\_fixed\_filter\_idx.
* Otherwise, AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is set equal to 17 + alf\_luma\_prev\_filter\_idx\_minus1.

**alf\_luma\_fixed\_filter\_idx** specifies the fixed filter that is applied to the luma CTB. The value of alf\_luma\_fixed\_filter\_idx shall be in a range of 0 to 15, inclusive.

#### Sample adaptive offset semantics

**sao\_merge\_left\_flag** equal to 1 specifies that the syntax elements sao\_type\_idx\_luma, sao\_type\_idx\_chroma, sao\_band\_position, sao\_eo\_class\_luma, sao\_eo\_class\_chroma, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the left CTB. sao\_merge\_left\_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the left CTB. 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\_luma, sao\_type\_idx\_chroma, sao\_band\_position, sao\_eo\_class\_luma, sao\_eo\_class\_chroma, sao\_offset\_abs and sao\_offset\_sign are derived from the corresponding syntax elements of the above CTB. sao\_merge\_up\_flag equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above CTB. When sao\_merge\_up\_flag is not present, it is inferred to be equal to 0.

**sao\_type\_idx\_luma** specifies the offset type for the luma component. The array SaoTypeIdx[ cIdx ][ rx ][ ry ] specifies the offset type as specified in Table 7‑11 for the CTB at the location ( rx, ry ) for the colour component cIdx. The value of SaoTypeIdx[ 0 ][ rx ][ ry ] is derived as follows:

* If sao\_type\_idx\_luma is present, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to sao\_type\_idx\_luma.
* Otherwise (sao\_type\_idx\_luma is not present), SaoTypeIdx[ 0 ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to SaoTypeIdx[ 0 ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to SaoTypeIdx[ 0 ][ rx ][ ry − 1 ].
* Otherwise, SaoTypeIdx[ 0 ][ rx ][ ry ] is set equal to 0.

**sao\_type\_idx\_chroma** specifies the offset type for the chroma components. The values of SaoTypeIdx[ cIdx ][ rx ][ ry ] are derived as follows for cIdx equal to 1..2:

* If sao\_type\_idx\_chroma is present, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to sao\_type\_idx\_chroma.
* Otherwise (sao\_type\_idx\_chroma is not present), SaoTypeIdx[ cIdx ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to SaoTypeIdx[ cIdx ][ rx − 1 ][ ry ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to SaoTypeIdx[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, SaoTypeIdx[ cIdx ][ rx ][ ry ] is set equal to 0.

Table 7‑11 – Specification of the SAO type

|  |  |
| --- | --- |
| **SaoTypeIdx[ cIdx ][ rx ][ ry ]** | **SAO type (informative)** |
| 0 | Not applied |
| 1 | Band offset |
| 2 | Edge offset |

**sao\_offset\_abs**[ cIdx ][ rx ][ ry ][ i ] specifies the offset value of i-th category for the CTB at the location ( rx, 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 inferred to be 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 inferred to be equal to sao\_offset\_abs[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal to 0.

**sao\_offset\_sign**[ cIdx ][ rx ][ ry ][ i ] specifies the sign of the offset value of i-th category for the CTB at the location ( rx, ry ) for the colour component cIdx.

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 inferred to be 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 inferred to be equal to sao\_offset\_sign[ cIdx ][ rx ][ ry − 1 ][ i ].
* Otherwise, if SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2, the following applies:
* If i is equal to 0 or 1, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 0.
* Otherwise (i is equal to 2 or 3), sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 1.
* Otherwise, sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] is inferred to be equal 0.

The list SaoOffsetVal[ cIdx ][ rx ][ ry ][ i ] for i ranging from 0 to 4, inclusive, is derived as follows:

SaoOffsetVal[ cIdx ][ rx ][ ry ][ 0 ] = 0  
for( i = 0; i < 4; i++ )  
SaoOffsetVal[ cIdx ][ rx ][ ry ][ i + 1 ] = ( 1 − 2 \* sao\_offset\_sign[ cIdx ][ rx ][ ry ][ i ] ) \* (7‑126)  
 sao\_offset\_abs[ cIdx ][ rx ][ ry ][ i ]

**sao\_band\_position**[ cIdx ][ rx ][ ry ] specifies the displacement of the band offset of the sample range when SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 1.

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 inferred to be 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 inferred to be equal to sao\_band\_position[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, sao\_band\_position[ cIdx ][ rx ][ ry ] is inferred to be equal to 0.

**sao\_eo\_class**\_**luma** specifies the edge offset class for the luma component. The array SaoEoClass[ cIdx ][ rx ][ ry ] specifies the offset type as specified in Table 7‑12 for the CTB at the location ( rx, ry ) for the colour component cIdx. The value of SaoEoClass[ 0 ][ rx ][ ry ] is derived as follows:

* If sao\_eo\_class\_luma is present, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to sao\_eo\_class\_luma.
* Otherwise (sao\_eo\_class\_luma is not present), SaoEoClass[ 0 ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to SaoEoClass[ 0 ][ rx − 1 ][ ry  ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to SaoEoClass[ 0 ][ rx ][ ry − 1 ].
* Otherwise, SaoEoClass[ 0 ][ rx ][ ry ] is set equal to 0.

**sao\_eo\_class\_chroma** specifies the edge offset class for the chroma components. The values of SaoEoClass[ cIdx ][ rx ][ ry ] are derived as follows for cIdx equal to 1..2:

* If sao\_eo\_class\_chroma is present, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to sao\_eo\_class\_chroma.
* Otherwise (sao\_eo\_class\_chroma is not present), SaoEoClass[ cIdx ][ rx ][ ry ] is derived as follows:
* If sao\_merge\_left\_flag is equal to 1, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to SaoEoClass[ cIdx ][ rx − 1 ][ ry  ].
* Otherwise, if sao\_merge\_up\_flag is equal to 1, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to SaoEoClass[ cIdx ][ rx ][ ry − 1 ].
* Otherwise, SaoEoClass[ cIdx ][ rx ][ ry ] is set equal to 0.

Table 7‑12 – Specification of the SAO edge offset class

|  |  |
| --- | --- |
| **SaoEoClass[ cIdx ][ rx ][ ry ]** | **SAO edge offset class (informative)** |
| 0 | 1D 0-degree edge offset |
| 1 | 1D 90-degree edge offset |
| 2 | 1D 135-degree edge offset |
| 3 | 1D 45-degree edge offset |

#### Coding tree semantics

When all of the following conditions are true, IsInSmr[ x ][ y ] is set equal to TRUE for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* IsInSmr[ x0 ][ y0 ] is equal to FALSE
* cbWidth \* cbHeight / 4 is less than 32
* treeType is not equal to DUAL\_TREE\_CHROMA

When IsInSmr[ x0 ][ y0 ] is equal to TRUE. the arrays SmrX[ x ][ y ], SmrY[ x ][ y ], SmrW[ x ][ y ] and SmrH[ x ][ y ] are derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

SmrX[ x ][ y ] = x0 (7‑127)

SmrY[ x ][ y ] = y0 (7‑128)

SmrW[ x ][ y ] = cbWidth (7‑129)

SmrH[ x ][ y ] = cbHeight (7‑130)

The variables allowSplitQt, allowSplitBtVer, allowSplitBtHor, allowSplitTtVer, and allowSplitTtHor are derived as follows:

* The allowed quad split process as specified in clause 6.4.1 is invoked with the coding block size cbSize set equal to cbWidth, the current multi-type tree depth mttDepth, and treeType as inputs, and the output is assigned to allowSplitQt.
* The variables minQtSize, maxBtSize, maxTtSize and maxMttDepth are derived as follows:
* If treeType is equal to DUAL\_TREE\_CHROMA, minQtSize, maxBtSize, maxTtSize and maxMttDepth are set equal to MinQtSizeC, MaxBtSizeC, MaxTtSizeC and MaxMttDepthC + depthOffset, respectively.
* Otherwise, minQtSize, maxBtSize, maxTtSize and maxMttDepth are set equal to MinQtSizeY, MaxBtSizeY, MaxTtSizeY and MaxMttDepthY + depthOffset, respectively.
* The allowed binary split process as specified in clause 6.4.2 is invoked with the binary split mode SPLIT\_BT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, the minimum quadtree size minQtSize, the current partition index partIdx, and treeType as inputs, and the output is assigned to allowSplitBtVer.
* The allowed binary split process as specified in clause 6.4.2 is invoked with the binary split mode SPLIT\_BT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum binary tree size maxBtSize, the minimum quadtree size minQtSize, the current partition index partIdx, and treeType as inputs, and the output is assigned to allowSplitBtHor.
* The allowed ternary split process as specified in clause 6.4.3 is invoked with the ternary split mode SPLIT\_TT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum ternary tree size maxTtSize, and treeType as inputs, and the output is assigned to allowSplitTtVer.
* The allowed ternary split process as specified in clause 6.4.3 is invoked with the ternary split mode SPLIT\_TT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the maximum multi-type tree depth with offset maxMttDepth, the maximum ternary tree size maxTtSize, and treeType as inputs, and the output is assigned to allowSplitTtHor.

**split\_cu\_flag** equal to 0 specifies that a coding unit is not split. split\_cu\_flag equal to 1 specifies that a coding unit is split into four coding units using a quad split as indicated by the syntax element split\_qt\_flag, or into two coding units using a binary split or into three coding units using a ternary split as indicated by the syntax element mtt\_split\_cu\_binary\_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt\_split\_cu\_vertical\_flag.

When split\_cu\_flag is not present, the value of split\_cu\_flag is inferred as follows:

* If one or more of the following conditions are true, the value of split\_cu\_flag is inferred to be equal to 1:
* x0 + cbWidth is greater than pic\_width\_in\_luma\_samples.
* y0 + cbHeight is greater than pic\_height\_in\_luma\_samples.
* Otherwise, the value of split\_cu\_flag is inferred to be equal to 0.

**split\_qt\_flag** specifies whether a coding unit is split into coding units with half horizontal and vertical size.

When split\_qt\_flag is not present, the following applies:

* If allowSplitQt is equal to TRUE, the value of split\_qt\_flag is inferred to be equal to 1.
* Otherwise, the value of split\_qt\_flag is inferred to be equal to 0.

**mtt\_split\_cu\_vertical\_flag** equal to 0 specifies that a coding unit is split horizontally. mtt\_split\_cu\_vertical\_flag equal to 1 specifies that a coding unit is split vertically

When mtt\_split\_cu\_vertical\_flag is not present, it is inferred as follows:

* If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 0.
* Otherwise, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 1.

**mtt\_split\_cu\_binary\_flag** equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt\_split\_cu\_binary\_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.

When mtt\_split\_cu\_binary\_flag is not present, it is inferred as follows:

* If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to 0.
* Otherwise, if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred as to be equal to 1.
* Otherwise, if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to !mtt\_split\_cu\_vertical\_flag.
* Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to mtt\_split\_cu\_vertical\_flag.

The variable MttSplitMode[ x ][ y ][ mttDepth ] is derived from the value of mtt\_split\_cu\_vertical\_flag and from the value of mtt\_split\_cu\_binary\_flag as defined in Table 7‑13 for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.



Figure 7‑1 – Multi-type tree spliting modes indicated by MttSplitMode (informative)

MttSplitMode[ x0 ][ y0 ][ mttDepth ] represents horizontal and vertical binary and ternary splittings of a coding unit within the multi-type tree as illustrated in Figure 7‑1. 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.

Table 7‑13 – Specification of MttSplitMode[ x ][ y ][ mttDepth ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1

|  |  |  |
| --- | --- | --- |
| **MttSplitMode[ x0 ][ y0 ][ mttDepth ]** | **mtt\_split\_cu\_vertical\_flag** | **mtt\_split\_cu\_binary\_flag** |
| SPLIT\_TT\_HOR | 0 | 0 |
| SPLIT\_BT\_HOR | 0 | 1 |
| SPLIT\_TT\_VER | 1 | 0 |
| SPLIT\_BT\_VER | 1 | 1 |

When all of the following conditions are true, IsInSmr[ x ][ y ] is set equal to TRUE for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* IsInSmr[ x0 ][ y0 ] is equal to FALSE
* One of the following conditions is true:
* mtt\_split\_cu\_binary\_flag is equal to 1 and  cbWidth \* cbHeight / 2 is less than 32
* mtt\_split\_cu\_binary\_flag is equal to 0 and  cbWidth \* cbHeight / 4 is less than 32
* treeType is not equal to DUAL\_TREE\_CHROMA

When IsInSmr[ x0 ][ y0 ] is equal to TRUE. the arrays SmrX[ x ][ y ], SmrY[ x ][ y ], SmrW[ x ][ y ] and SmrH[ x ][ y ] are derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

SmrX[ x ][ y ] = x0 (7‑131)

SmrY[ x ][ y ] = y0 (7‑132)

SmrW[ x ][ y ] = cbWidth (7‑133)

SmrH[ x ][ y ] = cbHeight (7‑134)

#### Coding unit semantics

When all the following conditions are true, the history-based motion vector predictor list for the shared merging candidate list region is updated by setting NumHmvpSmrIbcCand equal to NumHmvpIbcCand, and setting HmvpSmrIbcCandList[ i ] equal to HmvpIbcCandList[ i ] for i = 0..NumHmvpIbcCand − 1:

* IsInSmr[ x0 ][ y0 ] is equal to TRUE.
* SmrX[ x0 ][ y0 ]  is equal to x0.
* SmrY[ x0 ][ y0 ]  is equal to y0.

The following assignments are made for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

CbPosX[ x ][ y ] = x0 (7‑135)

CbPosY[ x ][ y ] = y0 (7‑136)

CbWidth[ x ][ y ] = cbWidth (7‑137)

CbHeight[ x ][ y ] = cbHeight (7‑138)

**cu\_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 one or more of the following are parsed after cu\_skip\_flag[ x0 ][ y0 ]: the IBC mode flag pred\_mode\_ibc\_flag [ x0 ][ y0 ], and the merge\_data( ) syntax structure; when decoding an I slice, no more syntax elements except merge\_idx[ x0 ][ y0 ] are parsed after cu\_skip\_flag[ x0 ][ y0 ]. cu\_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 cu\_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.

When pred\_mode\_flag is not present, it is inferred as follows:

* If cbWidth is equal to 4 and cbHeight is equal to 4, pred\_mode\_flag is inferred to be equal to 1.
* Otherwise, pred\_mode\_flag is inferred to be equal to 1 when decoding an I slice, and equal to 0 when decoding a P or B slice, respectively.

The variable CuPredMode[ x ][ y ] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* If pred\_mode\_flag is equal to 0, CuPredMode[ x ][ y ] is set equal to MODE\_INTER.
* Otherwise (pred\_mode\_flag is equal to 1), CuPredMode[ x ][ y ] is set equal to MODE\_INTRA.

**pred\_mode\_ibc\_flag** equal to 1 specifies that the current coding unit is coded in IBC prediction mode. pred\_mode\_ibc\_flag equal to 0 specifies that the current coding unit is not coded in IBC prediction mode.

When pred\_mode\_ibc\_flag is not present, it is inferred as follows:

* If cu\_skip\_flag[ x0 ][ y0 ] is equal to 1, and cbWidth is equal to 4, and cbHeight is equal to 4, pred\_mode\_ibc\_flag is inferred to be equal 1.
* Otherwise, if both cbWidth and cbHeight are equal to 128, pred\_mode\_ibc\_flag is inferred to be equal to 0.
* Otherwise, pred\_mode\_ibc\_flag is infered to be equal to the value of sps\_ibc\_enabled\_flag when decoding an I slice, and 0 when decoding a P or B slice, respectively.

When pred\_mode\_ibc\_flag is equal to 1, the variable CuPredMode[ x ][ y ] is set to be equal to MODE\_IBC for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

**pcm\_flag**[ x0 ][ y0 ] equal to 1 specifies that the pcm\_sample( ) syntax structure is present and the transform\_tree( ) syntax structure is not present in the coding unit including the luma coding block at the location ( x0, y0 ). pcm\_flag[ x0 ][ y0 ] equal to 0 specifies that pcm\_sample( ) syntax structure is not present. When pcm\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The value of pcm\_flag[ x0 + i ][ y0 + j ] with i = 1..cbWidth − 1, j = 1..cbHeight − 1 is inferred to be equal to pcm\_flag[ x0 ][ y0 ].

**pcm\_alignment\_zero\_bit** is a bit equal to 0.

**intra\_bdpcm\_flag**[ x0 ][ y0 ] equal to 1 specifies that BDPCM is applied to the current luma coding block at the location ( x0, y0 ), i.e. the transform is skipped, the intra luma prediction mode is specified by intra\_bdpcm\_dir\_flag[ x0 ][ y0 ]. intra\_bdpcm\_dir\_flag[ x0 ][ y0 ] equal to 0 specifies that BDPCM is not applied to the current luma coding block at the location ( x0, y0 ).

When intra\_bdpcm\_flag[ x0 ][ y0 ] is not present it is inferred to be equal to 0.

The variable BdpcmFlag[ x ][ y ] is set equal to intra\_bdpcm\_flag[ x0 ][ y0 ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

**intra\_bdpcm\_dir\_flag**[ x0 ][ y0 ]equal to 0 specifies that the BDPCM prediction direction is horizontal. intra\_bdpcm\_dir\_flag[ x0 ][ y0 ] equal to 1 specifies that the BDPCM prediction direction is vertical.

The variable BdpcmDir[ x ][ y ] is set equal to intra\_bdpcm\_dir\_flag[ x0 ][ y0 ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

**intra\_mip\_flag**[ x0 ][ y0 ] equal to 1 specifies that the intra prediction type for luma samples is matrix-based intra prediction. intra\_mip\_flag[ x0 ][ y0 ] equal to 0 specifies that the intra prediction type for luma samples is not matrix-based intra prediction.

When intra\_mip\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The syntax elements **intra\_mip\_mpm\_flag**[ x0 ][ y0 ], **intra\_mip\_mpm\_idx**[ x0 ][ y0 ] and **intra\_mip\_mpm\_remainder** [ x0 ][ y0 ] specify the matrix-based 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 coding block relative to the top-left luma sample of the picture. When intra\_mip\_mpm\_flag[ x0 ][ y0 ] is equal to 1, the matrix-based intra prediction mode is inferred from a neighbouring intra-predicted coding unit according to clause 8.4.2.

When intra\_mip\_mpm\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 1.

**intra\_luma\_ref\_idx**[ x0 ][ y0 ] specifies the intra prediction reference line index IntraLumaRefLineIdx[ x ][ y ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1 as specified in Table 7‑14.

When intra\_luma\_ref\_idx[ x0 ][ y0 ] is not present it is inferred to be equal to 0.

Table 7‑14 – Specification of IntraLumaRefLineIdx[ x ][ y ] based on intra\_luma\_ref\_idx[ x0 ][ y0 ].

|  |  |
| --- | --- |
| intra\_luma\_ref\_idx[ x0 ][ y0 ] | IntraLumaRefLineIdx[ x ][ y ] x = x0..x0 + cbWidth − 1 y = y0..y0 + cbHeight − 1 |
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |

**intra\_subpartitions\_mode\_flag**[ x0 ][ y0 ] equal to 1 specifies that the current intra coding unit is partitioned into NumIntraSubPartitions[ x0 ][ y0 ] rectangular transform block subpartitions. intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] equal to 0 specifies that the current intra coding unit is not partitioned into rectangular transform block subpartitions.

When intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**intra\_subpartitions\_split\_flag**[ x0 ][ y0 ] specifies whether the intra subpartitions split type is horizontal or vertical. When intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If cbHeight is greater than MaxTbSizeY, intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is inferred to be equal to 0.
* Otherwise (cbWidth is greater than MaxTbSizeY), intra\_subpartitions\_split\_flag[ x0 ][ y0 ] is inferred to be equal to 1.

The variable IntraSubPartitionsSplitType specifies the type of split used for the current luma coding block as illustrated in Table 7‑15. IntraSubPartitionsSplitType is derived as follows:

* If intra\_subpartitions\_mode\_flag[ x0 ][ y0 ] is equal to 0, IntraSubPartitionsSplitType is set equal to 0.
* Otherwise, the IntraSubPartitionsSplitType is set equal to 1 + intra\_subpartitions\_split\_flag[ x0 ][ y0 ].

**Table 7‑15 – Name association to IntraSubPartitionsSplitType**

|  |  |
| --- | --- |
| IntraSubPartitionsSplitType | Name of IntraSubPartitionsSplitType |
| 0 | ISP\_NO\_SPLIT |
| 1 | ISP\_HOR\_SPLIT |
| 2 | ISP\_VER\_SPLIT |

The variable NumIntraSubPartitions specifies the number of transform block subpartitions into which an intra luma coding block is divided. NumIntraSubPartitions is derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT, NumIntraSubPartitions is set equal to 1.
* Otherwise, if one of the following conditions is true, NumIntraSubPartitions is set equal to 2:
  + cbWidth is equal to 4 and cbHeight is equal to 8,
  + cbWidth is equal to 8 and cbHeight is equal to 4.
* Otherwise, NumIntraSubPartitions is set equal to 4.

The syntax elements **intra\_luma\_mpm\_flag**[ x0 ][ y0 ], **intra\_luma\_not\_planar\_flag**[ x0 ][ y0 ], **intra\_luma\_mpm\_idx**[ x0 ][ y0 ] and **intra\_luma\_mpm\_remainder**[ 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 coding block relative to the top-left luma sample of the picture. When intra\_luma\_mpm\_flag[ x0 ][ y0 ] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted coding unit according to clause 8.4.3.

When intra\_luma\_mpm\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 1.

When intra\_luma\_not\_planar\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 1.

**intra\_chroma\_pred\_mode**[ x0 ][ y0 ] specifies the intra prediction mode for chroma samples. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture. When intra\_chroma\_pred\_mode is not present, it is inferred to be equal to 0.

**general\_merge\_flag**[ x0 ][ y0 ] specifies whether the inter prediction parameters for the current coding 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 coding block relative to the top-left luma sample of the picture.

When general\_merge\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If cu\_skip\_flag[ x0 ][ y0 ] is equal to 1, general\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise, general\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**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 coding 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.

**inter\_pred\_idc**[ x0 ][ y0 ] specifies whether list0, list1, or bi-prediction is used for the current coding unit according to Table 7‑16. 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.

Table 7‑16 – Name association to inter prediction mode

|  |  |  |  |
| --- | --- | --- | --- |
| **inter\_pred\_idc** | **Name of inter\_pred\_idc** | | |
| ( cbWidth + cbHeight )  >  12 | ( cbWidth + cbHeight )  = =  12 | ( cbWidth + cbHeight )  = =  8 |
| 0 | PRED\_L0 | PRED\_L0 | n.a. |
| 1 | PRED\_L1 | PRED\_L1 | n.a. |
| 2 | PRED\_BI | n.a. | n.a. |

When inter\_pred\_idc[ x0 ][ y0 ] is not present, it is inferred to be equal to PRED\_L0.

**sym\_mvd\_flag**[ x0 ][ y0 ] equal to 1 specifies that the syntax elements ref\_idx\_l0[ x0 ][ y0 ] and ref\_idx\_l1[ x0 ][ y0 ], and the mvd\_coding( x0, y0, refList ,cpIdx ) syntax structure for refList equal to 1 are not present. 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 sym\_mvd\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**ref\_idx\_l0**[ x0 ][ y0 ] specifies the list 0 reference picture index for the current coding unit. 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 ref\_idx\_l0[ x0 ][ y0 ] is not present it is inferred as follows:

* If sym\_mvd\_flag[ x0 ][ y0 ] is equal to 1, ref\_idx\_l0[ x0 ][ y0 ] is inferred to be equal to RefIdxSymL0.
* Otherwise (sym\_mvd\_flag[ x0 ][ y0 ] is equal to 0), ref\_idx\_l0[ x0 ][ y0 ] is inferred to be equal to 0.

**ref\_idx\_l1**[ x0 ][ y0 ] has the same semantics as ref\_idx\_l0, with l0, L0 and list 0 replaced by l1, L1 and list 1, respectively.

**inter\_affine\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, affine model based motion compensation is used to generate the prediction samples of the current coding unit. inter\_affine\_flag[ x0 ][ y0 ] equal to 0 specifies that the coding unit is not predicted by affine model based motion compensation. When inter\_affine\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**cu\_affine\_type\_flag**[ x0 ][ y0 ] equal to 1 specifies that for the current coding unit, when decoding a P or B slice, 6-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit. cu\_affine\_type\_flag[ x0 ][ y0 ] equal to 0 specifies that 4-parameter affine model based motion compensation is used to generate the prediction samples of the current coding unit.

MotionModelIdc[ x ][ y ] represents motion model of a coding unit as illustrated in Table 7‑17. The array indices x, y specify the luma sample location ( x, y ) relative to the top-left luma sample of the picture.

The variable MotionModelIdc[ x ][ y ] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* If general\_merge\_flag[ x0 ][ y0 ] is equal to 1, the following applies:

MotionModelIdc[ x ][ y ] = merge\_subblock\_flag[ x0 ][ y0 ] (7‑139)

* Otherwise (general\_merge\_flag[ x0 ][ y0 ] is equal to 0), the following applies:

MotionModelIdc[ x ][ y ] = inter\_affine\_flag[ x0 ][ y0 ] + cu\_affine\_type\_flag[ x0 ][ y0 ] (7‑140)

Table 7‑17 – Interpretation of MotionModelIdc[ x0 ][ y0 ]

|  |  |
| --- | --- |
| MotionModelIdc[ x ][ y ] | **Motion model for motion compensation** |
| 0 | Translational motion |
| 1 | 4-parameter affine motion |
| 2 | 6-parameter affine motion |

**amvr\_flag**[ x0 ][ y0 ] specifies the resolution of motion vector difference. 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. amvr\_flag[ x0 ][ y0 ] equal to 0 specifies that the resolution of the motion vector difference is 1/4 of a luma sample. amvr\_flag[ x0 ][ y0 ] equal to 1 specifies that the resolution of the motion vector difference is further specified by amvr\_precision\_flag[ x0 ][ y0 ].

When amvr\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If CuPredMode[ x0 ][ y0 ] is equal to MODE\_IBC, amvr\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise ( CuPredMode[ x0 ][ y0 ] is not equal to MODE\_IBC ), amvr\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**amvr\_precision\_flag**[ x0 ][ y0 ] equal to 0 specifies that the resolution of the motion vector difference is one integer luma sample if inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, and 1/16 of a luma sample otherwise. amvr\_precision\_flag[ x0 ][ y0 ] equal to 1 specifies that the resolution of the motion vector difference is four luma samples if inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, and one integer luma sample otherwise. 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 amvr\_precision\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

The motion vector differences are modified as follows:

* If inter\_affine\_flag[ x0 ][ y0 ] is equal to 0, the variable MvShift is derived and the variables MvdL0[ x0 ][ y0 ][ 0 ], MvdL0[ x0 ][ y0 ][ 1 ], MvdL1[ x0 ][ y0 ][ 0 ], MvdL1[ x0 ][ y0 ][ 1 ] are modified as follows:

MvShift = ( amvr\_flag[ x0 ][ y0 ] + amvr\_precision\_flag[ x0 ][ y0 ] ) << 1 (7‑141)

MvdL0[ x0 ][ y0 ][ 0 ] = MvdL0[ x0 ][ y0 ][ 0 ]  << ( MvShift + 2 ) (7‑142)

MvdL0[ x0 ][ y0 ][ 1 ] = MvdL0[ x0 ][ y0 ][ 1 ]  <<  ( MvShift + 2 ) (7‑143)

MvdL1[ x0 ][ y0 ][ 0 ] = MvdL1[ x0 ][ y0 ][ 0 ]<<( MvShift + 2 ) (7‑144)

MvdL1[ x0 ][ y0 ][ 1 ] = MvdL1[ x0 ][ y0 ][ 1 ]  << ( MvShift + 2 ) (7‑145)

* Otherwise (inter\_affine\_flag[ x0 ][ y0 ] is equal to 1), the variable MvShift is derived and the variables MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ], MvdCpL0[ x0 ][ y0 ][ 0 ][ 1 ], MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ], MvdCpL0[ x0 ][ y0 ][ 1 ][ 1 ], MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ] and MvdCpL0[ x0 ][ y0 ][ 2 ][ 1 ] are modified as follows:

MvShift = amvr\_precision\_flag[ x0 ][ y0 ]  ?   
  ( amvr\_precision\_flag[ x0 ][ y0 ]  <<  1 )  :  ( − (amvr\_flag[ x0 ][ y0 ]  <<  1) ) ) (7‑146)

MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 0 ][ 0 ]  << ( MvShift + 2 ) (7‑147)

MvdCpL1[ x0 ][ y0 ] [ 0 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 0 ][ 1 ]  <<  ( MvShift + 2 ) (7‑148)

MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 1 ][ 0 ]  << ( MvShift + 2 ) (7‑149)

MvdCpL1[ x0 ][ y0 ] [ 1 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 1 ][ 1 ]  <<  ( MvShift + 2 ) (7‑150)

MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ] = MvdCpL0[ x0 ][ y0 ][ 2 ][ 0 ]  << ( MvShift + 2 ) (7‑151)

MvdCpL1[ x0 ][ y0 ] [ 2 ][ 1 ] = MvdCpL1[ x0 ][ y0 ][ 2 ][ 1 ]  <<  ( MvShift + 2 ) (7‑152)

**bcw\_idx**[ x0 ][ y0 ] specifies the weight index of bi-prediction with CU weights. 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 bcw\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**cu\_cbf** equal to 1 specifies that the transform\_tree( ) syntax structure is present for the current coding unit. cu\_cbf equal to 0 specifies that the transform\_tree( ) syntax structure is not present for the current coding unit.

When cu\_cbf is not present, it is inferred as follows:

* If cu\_skip\_flag[ x0 ][ y0 ] is equal to 1 or BdpcmFlag[ x0 ][ y0 ] is equal to 1, cu\_cbf is inferred to be equal to 0.
* Otherwise, cu\_cbf is inferred to be equal to 1.

**cu\_sbt\_flag** equal to 1 specifies that for the current coding unit, subblock transform is used. cu\_sbt\_flag equal to 0 specifies that for the current coding unit, subblock transform is not used.

When cu\_sbt\_flag is not present, its value is inferred to be equal to 0.

NOTE – : When subblock transform is used, a coding unit is split into two transform units; one transform unit has residual data, the other does not have residual data.

**cu\_sbt\_quad\_flag** equal to 1 specifies that for the current coding unit, the subblock transform includes a transform unit of 1/4 size of the current coding unit. cu\_sbt\_quad\_flag equal to 0 specifies that for the current coding unit the subblock transform includes a transform unit of 1/2 size of the current coding unit.

When cu\_sbt\_quad\_flag is not present, its value is inferred to be equal to 0.

**cu\_sbt\_horizontal\_flag** equal to 1 specifies that the current coding unit is split horizontally into 2 transform units. cu\_sbt\_horizontal\_flag[ x0 ][ y0 ] equal to 0 specifies that the current coding unit is split vertically into 2 transform units.

When cu\_sbt\_horizontal\_flag is not present, its value is derived as follows:

* If cu\_sbt\_quad\_flag is equal to 1, cu\_sbt\_horizontal\_flag is set to be equal to allowSbtHorQ.
* Otherwise (cu\_sbt\_quad\_flag is equal to 0), cu\_sbt\_horizontal\_flag is set to be equal to allowSbtHorH.

**cu\_sbt\_pos\_flag** equal to 1 specifies that the tu\_cbf\_luma, tu\_cbf\_cb and tu\_cbf\_cr of the first transform unit in the current coding unit are not present in the bitstream. cu\_sbt\_pos\_flag equal to 0 specifies that the tu\_cbf\_luma, tu\_cbf\_cb and tu\_cbf\_cr of the second transform unit in the current coding unit are not present in the bitstream.

The variable SbtNumFourthsTb0 is derived as follows:

sbtMinNumFourths = cu\_sbt\_quad\_flag  ?  1  :  2 (7‑153)

SbtNumFourthsTb0 = cu\_sbt\_pos\_flag  ?  ( 4 − sbtMinNumFourths )  :  sbtMinNumFourths (7‑154)

**lfnst\_idx**[ x0 ][ y0 ] specifies whether and which one of the two low frequency non-separable transform kernels in a selected transform set is used. lfnst\_idx[ x0 ][ y0 ] equal to 0 specifies that the low frequency non-separable transform is not used. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left sample of the considered transform block relative to the top-left sample of the picture.

When lfnst\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

#### PCM sample semantics

**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.

#### Merge data semantics

**regular\_merge\_flag**[ x0 ][ y0 ] equal to 1 specifies that regular merge mode is used to generate the inter prediction parameters of the current coding unit. 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 regular\_merge\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If all the following conditions are true, regular\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 1:
* sps\_mmvd\_enabled\_flag is equal to 0.
* general\_merge\_flag[ x0 ][ y0 ] is equal to 1.
* cbWidth\*cbHeight is equal to 32.
* Otherwise, regular\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**mmvd\_merge\_flag**[ x0 ][ y0 ] equal to 1 specifies that merge mode with motion vector difference is used to generate the inter prediction parameters of the current coding unit. 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 mmvd\_merge\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If all the following conditions are true, mmvd\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 1:
* sps\_mmvd\_enabled\_flag is equal to 1.
* general\_merge\_flag[ x0 ][ y0 ] is equal to 1.
* cbWidth\*cbHeight is equal to 32.
* regular\_merge\_flag[ x0 ][ y0 ] is equal to 0.
* Otherwise, mmvd\_merge\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**mmvd\_cand\_flag**[ x0 ][ y0 ] specifies whether the first (0) or the second (1) candidate in the merging candidate list is used with the motion vector difference derived from mmvd\_distance\_idx[ x0 ][ y0 ] and mmvd\_direction\_idx[ x0 ][ y0 ]. 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 mmvd\_cand\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**mmvd\_distance\_idx**[ x0 ][ y0 ] specifies the index used to derive MmvdDistance[ x0 ][ y0 ] as specified in Table 7‑18. 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.

Table 7‑18 – Specification of MmvdDistance[ x0 ][ y0 ] based on mmvd\_distance\_idx[ x0 ][ y0 ].

|  |  |  |
| --- | --- | --- |
| mmvd\_distance\_idx[ x0 ][ y0 ] | MmvdDistance[ x0 ][ y0 ] | |
| slice\_fpel\_mmvd\_enabled\_flag = = 0 | slice\_fpel\_mmvd\_enabled\_flag = = 1 |
| 0 | 1 | 4 |
| 1 | 2 | 8 |
| 2 | 4 | 16 |
| 3 | 8 | 32 |
| 4 | 16 | 64 |
| 5 | 32 | 128 |
| 6 | 64 | 256 |
| 7 | 128 | 512 |

**mmvd\_direction\_idx**[ x0 ][ y0 ] specifies index used to derive MmvdSign[ x0 ][ y0 ] as specified in Table 7‑19. 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.

Table 7‑19 – Specification of MmvdSign[ x0 ][ y0 ] based on mmvd\_direction\_idx[ x0 ][ y0 ]

|  |  |  |
| --- | --- | --- |
| mmvd\_direction\_idx[ x0 ][ y0 ] | MmvdSign[ x0 ][ y0 ][0] | MmvdSign[ x0 ][ y0 ][1] |
| 0 | +1 | 0 |
| 1 | -1 | 0 |
| 2 | 0 | +1 |
| 3 | 0 | -1 |

Both components of of the merge plus MVD offset MmvdOffset[ x0 ][ y0 ] are derived as follows:

MmvdOffset[ x0 ][ y0 ][ 0 ] = ( MmvdDistance[ x0 ][ y0 ] << 2 ) \* MmvdSign[ x0 ][ y0 ][0] (7‑155)

MmvdOffset[ x0 ][ y0 ][ 1 ] = ( MmvdDistance[ x0 ][ y0 ] << 2 ) \* MmvdSign[ x0 ][ y0 ][1] (7‑156)

**merge\_subblock\_flag**[ x0 ][ y0 ] specifies whether the subblock-based inter prediction parameters for the current coding unit are inferred from neighbouring blocks. 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 merge\_subblock\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_subblock\_idx**[ x0 ][ y0 ] specifies the merging candidate index of the subblock-based merging candidate list where 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 merge\_subblock\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**ciip\_flag**[ x0 ][ y0 ] specifies whether the combined inter-picture merge and intra-picture prediction is applied for the current coding unit. 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 ciip\_flag[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

When ciip\_flag[ x0 ][ y0 ] is equal to 1, the variable IntraPredModeY[ x ][ y ] with x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is set to be equal to INTRA\_PLANAR.

The variable MergeTriangleFlag[ x0 ][ y0 ], which specifies whether triangular shape based motion compensation is used to generate the prediction samples of the current coding unit, when decoding a B slice. is derived as follows:

* If all the following conditions are true, MergeTriangleFlag[ x0 ][ y0 ] is set equal to 1:
* sps\_triangle\_enabled\_flag is equal to 1.
* slice\_type is equal to B.
* general\_merge\_flag[ x0 ][ y0 ] is equal to 1.
* MaxNumTriangleMergeCand is greater than or equal to 2.
* cbWidth \* cbHeight is greater than or equal to 64.
* regular\_merge\_flag[ x0 ][ y0 ] is equal to 0.
* mmvd\_merge\_flag[ x0 ][ y0 ] is equal to 0.
* merge\_subblock\_flag[ x0 ][ y0 ] is equal to 0.
* ciip\_flag[ x0 ][ y0 ] is equal to 0.
* Otherwise, MergeTriangleFlag[ x0 ][ y0 ] is set equal to 0.

**merge\_triangle\_split\_dir**[ x0 ][ y0 ] specifies the splitting direction of merge triangle mode. 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 merge\_triangle\_split\_dir[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

[Ed. (BB): Add semantics of what value corresponds to horizontal and which corresponds to vertical direction as well as consider adding the flag suffix.]

**merge\_triangle\_idx0**[ x0 ][ y0 ] specifies the first merging candidate index of the triangular shape based motion compensation candidate list where 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 merge\_triangle\_idx0[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**merge\_triangle\_idx1**[ x0 ][ y0 ] specifies the second merging candidate index of the triangular shape based motion compensation candidate list where 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 merge\_triangle\_idx1[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**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 coding block relative to the top-left luma sample of the picture.

When merge\_idx[ x0 ][ y0 ] is not present, it is inferred as follows:

* If mmvd\_merge\_flag[ x0 ][ y0 ] is equal to 1, merge\_idx[ x0 ][ y0 ] is inferred to be equal to mmvd\_cand\_flag[ x0 ][ y0 ].
* Otherwise (mmvd\_merge\_flag[ x0 ][ y0 ] is equal to 0), merge\_idx[ x0 ][ y0 ] is inferred to be equal to 0.

#### Motion vector difference semantics

**abs\_mvd\_greater0\_flag**[ compIdx ] specifies whether the absolute value of a motion vector component difference is greater than 0.

**abs\_mvd\_greater1\_flag**[ compIdx ] specifies whether the absolute value of a motion vector component difference is greater than 1.

When abs\_mvd\_greater1\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

**abs\_mvd\_minus2**[ compIdx ] plus 2 specifies the absolute value of a motion vector component difference.

When abs\_mvd\_minus2[ compIdx ] is not present, it is inferred to be equal to −1.

**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.

When mvd\_sign\_flag[ compIdx ] is not present, it is inferred to be equal to 0.

The motion vector difference lMvd[ compIdx ] for compIdx = 0..1 is derived as follows:

lMvd[ compIdx ] = abs\_mvd\_greater0\_flag[ compIdx ] \*  
 ( abs\_mvd\_minus2[ compIdx ] + 2 ) \* ( 1 − 2 \* mvd\_sign\_flag[ compIdx ] ) (7‑157)

The value of lMvd[ compIdx ] shall be in the range of −215 to 215 − 1, inclusive.

Depending in the value of MotionModelIdc[ x0 ][ y0 ], motion vector differences are derived as follows:

* If MotionModelIdc[ x0 ][ y0 ] is equal to 0, the variable MvdLX[ x0 ][ y0 ][ compIdx ], with X being 0 or 1, specifies the difference between a list X 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 coding 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.
* If refList is equal to 0, MvdL0[ x0 ][ y0 ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.
* Otherwise (refList is equal to 1), MvdL1[ x0 ][ y0 ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.
* Otherwise (MotionModelIdc[ x0 ][ y0 ] is not equal to 0), the variable MvdCpLX[ x0 ][ y0 ][ cpIdx ][ compIdx ], with X being 0 or 1, specifies the difference between a list X 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 coding block relative to the top-left luma sample of the picture, the array index cpIdx specifies the control point index. The horizontal motion vector component difference is assigned compIdx = 0 and the vertical motion vector component is assigned compIdx = 1.
* If refList is equal to 0, MvdCpL0[ x0 ][ y0 ][ cpIdx ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.
* Otherwise (refList is equal to 1), MvdCpL1[ x0 ][ y0 ][ cpIdx ][ compIdx ] is set equal to lMvd[ compIdx ] for compIdx = 0..1.

#### Transform tree semantics

[Ed. (BB): The transform scheme does not have any syntax for spliting a CU into TUs. However, if the height or width of a CU is larger than the current maximum transform length of 64 luma samples or the corresponding chroma sample length, the CU will be implicitly split to divide it into TUs.]

#### Transform unit semantics

The transform coefficient levels are represented by 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 Y, 1 for Cb, and 2 for Cr. The array indices xC and yC specify the transform coefficient location ( xC, yC ) within the current transform block. When the value of TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] is not specified in clause 7.3.7.11, it is inferred to be equal to 0.

**tu\_cbf\_cb**[ x0 ][ y0 ] equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location ( x0, y0 ) of the considered transform block.

When tu\_cbf\_cb[ x0 ][ y0 ] is not present in the current TU, its value is inferred to be equal to 0.

**tu\_cbf\_cr**[ x0 ][ y0 ] equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the top-left location ( x0, y0 ) of the considered transform block.

When tu\_cbf\_cr[ x0 ][ y0 ] is not present in the current TU, its value is inferred to be equal to 0.

**tu\_cbf\_luma**[ x0 ][ y0 ] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

When tu\_cbf\_luma[ x0 ][ y0 ] is not present in the current TU, its value is inferred as follows:

* If cu\_sbt\_flag is equal to 1 and one of the following conditions is true, tu\_cbf\_luma[ x0 ][ y0 ] is inferred to be equal to 0:
* subTuIndex is equal to 0 and cu\_sbt\_pos\_flag is equal to 1
* subTuIndex is equal to 1 and cu\_sbt\_pos\_flag is equal to 0
* Otherwise, tu\_cbf\_luma[ x0 ][ y0 ] is inferred to be equal to 1.

**cu\_qp\_delta\_abs** specifies the absolute value of the difference CuQpDeltaVal between the quantization parameter of the current coding unit and its prediction.

**cu\_qp\_delta\_sign\_flag** specifies the sign of CuQpDeltaVal as follows:

* If cu\_qp\_delta\_sign\_flag is equal to 0, the corresponding CuQpDeltaVal has a positive value.
* Otherwise (cu\_qp\_delta\_sign\_flag is equal to 1), the corresponding CuQpDeltaVal has a negative value.

When cu\_qp\_delta\_sign\_flag is not present, it is inferred to be equal to 0.

When cu\_qp\_delta\_abs is present, the variables IsCuQpDeltaCoded and CuQpDeltaVal are derived as follows:

IsCuQpDeltaCoded = 1 (7‑158)

CuQpDeltaVal = cu\_qp\_delta\_abs \* ( 1 − 2 \* cu\_qp\_delta\_sign\_flag ) (7‑159)

The value of CuQpDeltaVal shall be in the range of −( 32 + QpBdOffsetY / 2 ) to +( 31 + QpBdOffsetY / 2 ), inclusive.

**transform\_skip\_flag**[ x0 ][ y0 ] specifies whether a transform is applied to the luma 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. transform\_skip\_flag[ x0 ][ y0 ] equal to 1 specifies that no transform is applied to the luma transform block. transform\_skip\_flag[ x0 ][ y0 ] equal to 0 specifies that the decision whether transform is applied to the luma transform block or not depends on other syntax elements.

When transform\_skip\_flag[ x0 ][ y0 ] is not present, it is inferred as follows:

* If BdcpmFlag[ x0 ][ x0 ] is equal to 1, transform\_skip\_flag[ x0 ][ y0 ] is inferred to be equal to 1.
* Otherwise (BdcpmFlag[ x0 ][ x0 ] is equal to 0), transform\_skip\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

**tu\_mts\_idx**[ x0 ][ y0 ] specifies which transform kernels are applied to the residual samples along the horizontal and vertical direction of the associated luma transform block. 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.

When tu\_mts\_idx[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

**tu\_joint\_cbcr\_residual**[ x0 ][ y0 ] specifies whether indicated Cb residual is used to derive both Cb and Cr residuals. 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. tu\_joint\_cbcr\_residual[ x0 ][ y0 ] equal to 1 specifies that the indicated Cb residual is used to derive the Cr residual. tu\_joint\_cbcr\_residual[ x0 ][ y0 ] equal to 0 specifies that Cr residual may be present in the bitstream depending on other syntax elements.

When tu\_joint\_cbcr\_residual[ x0 ][ y0 ] is not present, it is inferred to be equal to 0.

#### Residual coding semantics

The array AbsLevel[ xC ][ yC ] represents an array of absolute values of transform coefficient levels for the current transform block and the array AbsLevelPass1[ xC ][ yC ] represents an array of partially reconstructed absolute values of transform coefficient levels for the current transform block. The array indices xC and yC specify the transform coefficient location ( xC, yC ) within the current transform block. When the value of AbsLevel[ xC ][ yC ] is not specified in clause 7.3.7.11, it is inferred to be equal to 0. When the value of AbsLevelPass1[ xC ][ yC ] is not specified in clause 7.3.7.11, it is inferred to be equal to 0.

The variables CoeffMin and CoeffMax specifying the minimum and maximum transform coefficient values are derived as follows:

CoeffMin = −( 1 << 15 ) (7‑160)

CoeffMax = ( 1 << 15 ) − 1 (7‑161)

The array QStateTransTable[ ][ ] is specified as follows:

QStateTransTable[ ][ ] = { { 0, 2 }, { 2, 0 }, { 1, 3 }, { 3, 1 } } (7‑162)

**last\_sig\_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\_sig\_coeff\_x\_prefix shall be in the range of 0 to ( log2ZoTbWidth  <<  1 ) − 1, inclusive.

When last\_sig\_coeff\_x\_prefix is not present, it is inferred to be 0.

**last\_sig\_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\_sig\_coeff\_y\_prefix shall be in the range of 0 to ( log2ZoTbHeight  <<  1 ) − 1, inclusive.

When last\_sig\_coeff\_y\_prefix is not present, it is inferred to be 0.

**last\_sig\_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\_sig\_coeff\_x\_suffix shall be in the range of 0 to ( 1  <<  ( ( last\_sig\_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\_sig\_coeff\_x\_suffix is not present, the following applies:

LastSignificantCoeffX = last\_sig\_coeff\_x\_prefix (7‑163)

* Otherwise (last\_sig\_coeff\_x\_suffix is present), the following applies:

LastSignificantCoeffX = ( 1  <<  ( (last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) ) \* (7‑164)  
 ( 2 + (last\_sig\_coeff\_x\_prefix & 1 ) ) + last\_sig\_coeff\_x\_suffix

**last\_sig\_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\_sig\_coeff\_y\_suffix shall be in the range of 0 to ( 1  <<  ( ( last\_sig\_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\_sig\_coeff\_y\_suffix is not present, the following applies:

LastSignificantCoeffY = last\_sig\_coeff\_y\_prefix (7‑165)

* Otherwise (last\_sig\_coeff\_y\_suffix is present), the following applies:

LastSignificantCoeffY = ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) ) \* (7‑166)  
 ( 2 + ( last\_sig\_coeff\_y\_prefix & 1 ) ) + last\_sig\_coeff\_y\_suffix

**coded\_sub\_block\_flag**[ xS ][ yS ] specifies the following for the subblock at location ( xS, yS ) within the current transform block, where a subblock is a (4x4) array of 16 transform coefficient levels:

* If coded\_sub\_block\_flag[ xS ][ yS ] is equal to 0, the 16 transform coefficient levels of the subblock at location ( xS, yS ) are inferred to be equal to 0.
* Otherwise (coded\_sub\_block\_flag[ xS ][ yS ] is equal to 1), the following applies:
* If ( xS, yS ) is equal to ( 0, 0 ) and ( LastSignificantCoeffX, LastSignificantCoeffY ) is not equal to ( 0, 0 ), at least one of the 16 sig\_coeff\_flag syntax elements is present for the subblock at location ( xS, yS ).
* Otherwise, at least one of the 16 transform coefficient levels of the subblock at location ( xS, yS ) has a non-zero value.

When coded\_sub\_block\_flag[ xS ][ yS ] is not present, it is inferred to be equal to 1.

**sig\_coeff\_flag**[ xC ][ yC ] specifies for the transform coefficient location ( xC, yC ) within the current transform block whether the corresponding transform coefficient level at the location ( xC, yC ) is non-zero as follows:

* If sig\_coeff\_flag[ xC ][ yC ] is equal to 0, the transform coefficient level at the location ( xC, yC ) is set equal to 0.
* Otherwise (sig\_coeff\_flag[ xC ][ yC ] is equal to 1), the transform coefficient level at the location ( xC, yC ) has a non‑zero value.

When sig\_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 all of the following conditions are true, sig\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 1:
* ( xC & ( (1 << log2SbW ) − 1 ), yC & ( (1 << log2SbH ) − 1 ) ) is equal to ( 0, 0 ).
* inferSbDcSigCoeffFlag is equal to 1.
* coded\_sub\_block\_flag[ xS ][ yS ] is equal to 1.
* Otherwise, sig\_coeff\_flag[ xC ][ yC ] is inferred to be equal to 0.

**abs\_level\_gtx\_flag**[ n ][ j ] specifies whether the absolute value of the transform coefficient level (at scanning position n) is greater than ( j << 1 ) + 1. When abs\_level\_gtx\_flag[ n ][ j ] is not present, it is inferred to be equal to 0.

**par\_level\_flag**[ n ] specifies the parity of the transform coefficient level at scanning position n. When par\_level\_flag[ n ] is not present, it is inferred to be equal to 0.

**abs\_remainder**[ n ] is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position n. When abs\_remainder[ n ] is not present, it is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of abs\_remainder[ n ] shall be constrained such that the corresponding value of TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] is in the range of CoeffMin to CoeffMax, inclusive.

**dec\_abs\_level**[ n ] is an intermediate value that is coded with Golomb-Rice code at the scanning position n. Given ZeroPos[ n ] that is derived in clause 9.5.3.2 during the parsing of dec\_abs\_level[ n ], the absolute value of a transform coefficient level at location ( xC, yC ) AbsLevel[ xC ][ yC ] is derived using as follows:

* If dec\_abs\_level[ n ] is equal to ZeroPos[ n ], AbsLevel[ xC ][ yC ] is set equal to 0.
* Otherwise if dec\_abs\_level[ n ] is less than ZeroPos[ n ], AbsLevel[ xC ][ yC ] is set equal to dec\_abs\_level[ n ] + 1;
* Otherwise (dec\_abs\_level[ n ] is greater than ZeroPos[ n ]), AbsLevel[ xC ][ yC ] is set equal to dec\_abs\_level[ n ].

The value of AbsLevelPass1[ xC ][ yC ] is derived as follows:

AbsLevelPass1[ xC ][ yC ] = Min( 4 + ( AbsLevel[ xC ][ yC ] & 1 ), AbsLevel[ xC ][ yC ] ) (7‑167)

It is a requirement of bitstream conformance that the value of dec\_abs\_level[ n ] shall be constrained such that the corresponding value of TransCoeffLevel[ x0 ][ y0 ][ cIdx ][ xC ][ yC ] is in the range of CoeffMin to CoeffMax, inclusive.

**coeff\_sign\_flag**[ n ] specifies the sign of a transform coefficient level for the scanning position n as follows:

* If coeff\_sign\_flag[ n ] is equal to 0, the corresponding transform coefficient level has a positive value.
* Otherwise (coeff\_sign\_flag[ n ] is equal to 1), the corresponding transform coefficient level has a negative value.

When coeff\_sign\_flag[ n ] is not present, it is inferred to be equal to 0.

# Decoding process

## General decoding process

* + 1. **General**

Input to this process is a bitstream. Output of this process is a list of decoded pictures.

The decoding process is specified such that all decoders that conform to a specified profile and level will produce numerically identical cropped decoded output pictures when invoking the decoding process associated with that profile for a bitstream conforming to that profile and level. Any decoding process that produces identical cropped decoded output pictures to those produced by the process described herein (with the correct output order or output timing, as specified) conforms to the decoding process requirements of this Specification.

For each IRAP picture in the bitstream, the following applies:

– If the picture is an IDR picture, the first picture in the bitstream in decoding order, or the first picture that follows an end of sequence NAL unit in decoding order, the variable NoIncorrectPicOutputFlag is set equal to 1.

– Otherwise, if some external means not specified in this Specification is available to set the variable HandleCraAsCvsStartFlag to a value for the picture, HandleCraAsCvsStartFlag is set equal to the value provided by the external means and NoIncorrectPicOutputFlag is set equal to HandleCraAsCvsStartFlag.

– Otherwise, HandleCraAsCvsStartFlag and NoIncorrectPicOutputFlag are both set equal to 0.

For each GRA picture in the bitstream, the following applies:

– If the current picture is the first picture in the bitstream in decoding order or the first picture that follows an end of sequence NAL unit in decoding order, the variable NoIncorrectPicOutputFlag is set equal to 1.

– Otherwise, NoIncorrectPicOutputFlag is set equal to 0.

NOTE – The above operations, for both IRAP pictures and GRA pictuures, are needed for identification of the CVSs in the bitstream.

For each CVS in the bitstream, the variable TargetLayerId, which identifies the target layer to be decoded, and the variable HighestTid, which identifies the highest temporal sub-layer to be decoded, are specified as follows:

– If some external means, not specified in this Specification, is available to set TargetLayerId, TargetLayerId is set by the external means.

– Otherwise, TargetLayerId is set equal to vps\_included\_layer\_id[ 0 ].

– If some external means, not specified in this Specification, is available to set HighestTid, HighestTid is set by the external means.

– Otherwise, HighestTid is set equal to sps\_max\_sub\_layers\_minus1.

For each CVS in the bitsstream, the sub-bitstream extraction process as specified in clause 10 is applied with the CVS, TargetLayerId, and HighestTid as inputs, and the output is assigned to a bitstream referred to as CvsToDecode. After that, the instances of CvsToDecode of all the CVSs are concatenated, in decoding order, and the result is assigned to the bitstream BitstreamToDecode.

Clause 8.1.2 is repeatedly invoked for each coded picture in BitstreamToDecode in decoding order.

* + 1. **Decoding process for a coded picture**

The decoding processes specified in this clause apply to each coded picture, referred to as the current picture and denoted by the variable CurrPic, in BitstreamToDecode.

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.

The decoding process for the current picture takes as inputs the syntax elements and upper-case variables from clause 7. When interpreting the semantics of each syntax element in each NAL unit, and in the remaining parts of clause 8, the term "the bitstream" (or part thereof, e.g., a CVS of the bitstream) refers to BitstreamToDecode (or part thereof).

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 CVS 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 one of the 3 sample arrays of the current picture, with the NAL units with colour\_plane\_id equal to 0, 1 and 2 being assigned to SL, SCb and SCr, respectively.

NOTE – The variable ChromaArrayType is derived as equal to 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 (when chroma\_format\_idc is equal to 0).

The decoding process operates as follows for the current picture CurrPic:

1. The decoding of NAL units is specified in clause 8.2.
2. The processes in clause 8.3 specify the following decoding processes using syntax elements in the slice header layer and above:

– Variables and functions relating to picture order count are derived as specified in clause 8.3.1. This needs to be invoked only for the first slice of a picture.

– At the beginning of the decoding process for each slice of a non-IDR picture, the decoding process for reference picture lists construction specified in clause 8.3.2 is invoked for derivation of reference picture list 0 (RefPicList[ 0 ]) and reference picture list 1 (RefPicList[ 1 ]).

– The decoding process for reference picture marking in clause 8.3.3 is invoked, wherein reference pictures may be marked as "unused for reference" or "used for long-term reference". This needs to be invoked only for the first slice of a picture.

– When the current picture is a CRA picture with NoIncorrectPicOutputFlag equal to 1 or GRA picture with NoIncorrectPicOutputFlag equal to 1, the decoding process for generating unavailable reference pictures specified in subclause 8.3.4 is invoked, which needs to be invoked only for the first slice of a picture.

– PictureOutputFlag is set as follows:

– If one of the following conditions is true, PictureOutputFlag is set equal to 0:

– the current picture is a RASL picture and NoIncorrectPicOutputFlag of the associated IRAP picture is equal to 1.

– gra\_enabled\_flag is equal to 1 and the current picture is GRA picture with NoIncorrectPicOutputFlag equal to 1.

– gra\_enabled\_flag is equal to 1, the current picture is associated with a GRA picture with NoIncorrectPicOutputFlag equal to 1, and PicOrderCntVal of the current picture is less than RpPicOrderCntVal of the associated GRA picture.

– Otherwise, PictureOutputFlag is set equal to 1.

1. [Ed. (YK): Add herein the invocation of the decoding processes for coding tree units, scaling, transform, in-loop filtering, etc.]
2. After all slices of the current picture have been decoded, the current decoded picture is marked as "used for short-term reference".

When gra\_enabled\_flag is equal to 1 and PicOrderCntVal of the current picture is greater than or equal to RpPicOrderCntVal of the previous GRA picture in decoding order for which there is no IRAP picure between the current picture and the previous GRA picture in decoding order, it is a requirement of bitstream conformance that the current and subsequent decoded pictures shall be an exact match to the pictures produced by starting the decoding process at the previous IRAP picture preceding the current picture in decoding order. [Ed. (YK): With the addition of the setting of the PictureOutputFlag above and the use in the picture output process that is part of the added HRD text, this constraint is no longer needed, as what is required is established by "Any decoding process that produces identical cropped decoded output pictures to those produced by the process described herein (with the correct output order or output timing, as specified) conforms to the decoding process requirements of this Specification."]

## NAL unit decoding process

Inputs to this process are NAL units of the current picture and their associated non-VCL NAL units.

Outputs of this process are the parsed 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 parses the RBSP syntax structure.

## Slice decoding process

### Decoding process for picture order count

Output of this process is PicOrderCntVal, the picture order count of the current picture.

Each coded picture is associated with a picture order count variable, denoted as PicOrderCntVal.

When the current picture is not a CLVSS picture, the variables prevPicOrderCntLsb and prevPicOrderCntMsb are derived as follows:

* Let prevTid0Pic be the previous picture in decoding order that has NuhLayerId equal to the NuhLayerId of the current picture and TemporalId equal to 0 and that is not a RASL or RADL picture.
* The variable prevPicOrderCntLsb is set equal to slice\_pic\_order\_cnt\_lsb of prevTid0Pic.
* The variable prevPicOrderCntMsb is set equal to PicOrderCntMsb of prevTid0Pic.

The variable PicOrderCntMsb of the current picture is derived as follows:

* If the current picture is a CLVSS picture, PicOrderCntMsb is set equal to 0.
* Otherwise, PicOrderCntMsb is derived as follows:

if( ( slice\_pic\_order\_cnt\_lsb < prevPicOrderCntLsb ) &&  
 ( ( prevPicOrderCntLsb − slice\_pic\_order\_cnt\_lsb ) >= ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb + MaxPicOrderCntLsb (8‑1)  
else if( (slice\_pic\_order\_cnt\_lsb > prevPicOrderCntLsb ) &&  
 ( ( slice\_pic\_order\_cnt\_lsb − prevPicOrderCntLsb ) > ( MaxPicOrderCntLsb / 2 ) ) )  
 PicOrderCntMsb = prevPicOrderCntMsb − MaxPicOrderCntLsb  
else  
 PicOrderCntMsb = prevPicOrderCntMsb

PicOrderCntVal is derived as follows:

PicOrderCntVal = PicOrderCntMsb + slice\_pic\_order\_cnt\_lsb (8‑2)

NOTE 1 – All CLVSS pictures will have PicOrderCntVal equal to slice\_pic\_order\_cnt\_lsb since for CLVSS pictures PicOrderCntMsb is set equal to 0.

The value of PicOrderCntVal shall be in the range of −231 to 231 − 1, inclusive.

In one CVS, the PicOrderCntVal values for any two coded pictures with the same value of NuhLayerId shall not be the same.

All pictures in any particular access unit shall have the same value of PicOrderCntVal.

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 are not in the range of −215 to 215 − 1, inclusive.

NOTE 2 – Let X be the current picture and Y and Z be two other pictures in the same CVS, 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.

### Decoding process for reference picture lists construction

This process is invoked at the beginning of the decoding process for each slice of a non-IDR picture.

Reference pictures are addressed through reference indices. A reference index is an index into a reference picture list. When decoding an I slice, no reference picture list is used in decoding of the slice data. When decoding a P slice, only reference picture list 0 (i.e., RefPicList[ 0 ]), is used in decoding of the slice data. When decoding a B slice, both reference picture list 0 and reference picture list 1 (i.e., RefPicList[ 1 ]) are used in decoding of the slice data.

At the beginning of the decoding process for each slice of a non-IDR picture, the reference picture lists RefPicList[ 0 ] and RefPicList[ 1 ] are derived. The reference picture lists are used in marking of reference pictures as specified in clause 8.3.3 or in decoding of the slice data.

NOTE 1 – For an I slice of a non-IDR picture that it is not the first slice of the picture, RefPicList[ 0 ] and RefPicList[ 1 ] may be derived for bitstream conformance checking purpose, but their derivation is not necessary for decoding of the current picture or pictures following the current picture in decoding order. For a P slice that it is not the first slice of a picture, RefPicList[ 1 ] may be derived for bitstream conformance checking purpose, but its derivation is not necessary for decoding of the current picture or pictures following the current picture in decoding order.

The reference picture lists RefPicList[ 0 ] and RefPicList[ 1 ] are constructed as follows:

for( i = 0; i < 2; i++ ) {  
 for( j = 0, k = 0, pocBase = PicOrderCntVal; j < num\_ref\_entries[ i ][ RplsIdx[ i ] ]; j++) {  
 if( st\_ref\_pic\_flag[ i ][ RplsIdx[ i ] ][ j ] ) {  
 RefPicPocList[ i ][ j ] = pocBase − DeltaPocSt[ i ][ RplsIdx[ i ] ][ j ]  
 if( there is a reference picture picA in the DPB with the same NuhLayerId as the current picture and  
 PicOrderCntVal equal to RefPicPocList[ i ][ j ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture" (8‑5)  
 pocBase = RefPicPocList[ i ][ j ]  
 } else {  
 if( !delta\_poc\_msb\_cycle\_lt[ i ][ k ] ) {  
 if( there is a reference picA in the DPB with the same NuhLayerId as the current picture and  
 PicOrderCntVal & ( MaxPicOrderCntLsb − 1 ) equal to PocLsbLt[ i ][ k ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture"  
 } else {  
 if( there is a reference picA in the DPB with the same NuhLayerId as the current picture and  
 PicOrderCntVal equal to FullPocLt[ i ][ k ] )  
 RefPicList[ i ][ j ] = picA  
 else  
 RefPicList[ i ][ j ] = "no reference picture"  
 }  
 k++  
 }  
 }  
}

For each i equal to 0 or 1, the first NumRefIdxActive[ i ] entries in RefPicList[ i ] are referred to as the active entries in RefPicList[ i ], and the other entries in RefPicList[ i ] are referred to as the inactive entries in RefPicList[ i ].

NOTE 2 – It is possible that a particular picture is referred to by both an entry in RefPicList[ 0 ] and an entry in RefPicList[ 1 ]. It is also possible that a particular picture is referred to by more than one entry in RefPicList[ 0 ] or by more than one entry in RefPicList[ 1 ].

NOTE 3 – The active entries in RefPicList[ 0 ] and the active entries in RefPicList[ 1 ] collectively refer to all reference pictures that may be used for inter prediction of the current picture and one or more pictures that follow the current picture in decoding order. The inactive entries in RefPicList[ 0 ] and the inactive entries in RefPicList[ 1 ] collectively refer to all reference pictures that are *not* used for inter prediction of the current picture but may be used in inter prediction for one or more pictures that follow the current picture in decoding order.

NOTE 4 – There may be one or more entries in RefPicList[ 0 ] or RefPicList[ 1 ] that are equal to "no reference picture" because the corresponding pictures are not present in the DPB. Each inactive entry in RefPicList[ 0 ] or RefPicList[ 0 ] that is equal to "no reference picture" should be ignored. An unintentional picture loss should be inferred for each active entry in RefPicList[ 0 ] or RefPicList[ 1 ] that is equal to "no reference picture".

It is a requirement of bitstream conformance that the following constraints apply:

* For each i equal to 0 or 1, num\_ref\_entries[ i ][ RplsIdx[ i ] ] shall not be less than NumRefIdxActive[ i ].
* The picture referred to by each active entry in RefPicList[ 0 ] or RefPicList[ 1 ] shall be present in the DPB and shall have TemporalId less than or equal to that of the current picture.
* The picture referred to by each entry in RefPicList[ 0 ] or RefPicList[ 1 ] shall not be the current picture.
* An STRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] of a slice of a picture and an LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] of the same slice or a different slice of the same picture shall not refer to the same picture.
* There shall be no LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ] for which the difference between the PicOrderCntVal of the current picture and the PicOrderCntVal of the picture referred to by the entry is greater than or equal to 224.
* Let setOfRefPics be the set of unique pictures referred to by all entries in RefPicList[ 0 ] and all entries in RefPicList[ 1 ]. The number of pictures in setOfRefPics shall be less than or equal to sps\_max\_dec\_pic\_buffering\_minus1 and setOfRefPics shall be the same for all slices of a picture.

### Decoding process for reference picture marking

This process is invoked once per picture, after decoding of a slice header and the decoding process for reference picture list construction for the slice as specified in clause 8.3.2, but prior to the decoding of the slice data. This process may result in one or more reference pictures in the DPB being marked as "unused for reference" or "used for long-term reference".

A decoded picture in the DPB can be marked as "unused for reference", "used for short-term reference" or "used for long-term reference", but only one among these three at any given moment during the operation of the decoding process. Assigning one of these markings to a picture implicitly removes another of these markings when applicable. When a 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).

STRPs are identified by their NuhLayerId and PicOrderCntVal values. LTRPs are identified by their NuhLayerId values and the Log2( MaxLtPicOrderCntLsb ) LSBs of their PicOrderCntVal values.

If the current picture is a CLVSS picture, all reference pictures currently in the DPB (if any) with the same NuhLayerId as the current picture are marked as "unused for reference".

Otherwise, the following applies:

* For each LTRP entry in RefPicList[ 0 ] or RefPicList[ 1 ], when the referred picture is an STRP with the same NuhLayerId as the current picture, the picture is marked as "used for long-term reference".
* Each reference picture with the same NuhLayerId as the current picture in the DPB that is not referred to by any entry in RefPicList[ 0 ] or RefPicList[ 1 ] is marked as "unused for reference".

### Decoding process for generating unavailable reference pictures

#### General decoding process for generating unavailable reference pictures

This process is invoked once per coded picture when the current picture is a CRA picture with NoIncorrectPicOutputFlag equal to 1 or a GRA picture with NoIncorrectPicOutputFlag equal to 1.

When this process is invoked, the following applies:

* For each RefPicList[ i ][ j ], with i in the range of 0 to 1, inclusive, and j in the range of 0 to num\_ref\_entries[ i ][ RplsIdx[ i ] ] − 1, inclusive, that is equal to "no reference picture", a picture is generated as specified in subclause 8.3.4.2 and the following applies:
* The value of NuhLayerId for the generated picture is set equal to NuhLayerId of the current picture.
* If st\_ref\_pic\_flag[ i ][ RplsIdx[ i ] ][ j ] is equal to 1, the value of PicOrderCntVal for the generated picture is set equal to RefPicPocList[ i ][ j ] and the generated picture is marked as "used for short-term reference".
* Otherwise (st\_ref\_pic\_flag[ i ][ RplsIdx[ i ] ][ j ] is equal to 0), the value of PicOrderCntVal for the generated picture is set equal to PicOrderCntVal − poc\_lsb\_lt[ i ][ RplsIdx[ i ] ][ j ] − ( delta\_poc\_msb\_present\_flag[ i ][ j ] ? ( DeltaPocMsbCycleLt[ i ][ j ] \* MaxPicOrderCntLsb ) : 0 ), and the generated picture is marked as "used for long-term reference".
* RefPicList[ i ][ j ] is set to be the generated reference picture.

#### Generation of one unavailable picture

When this process is invoked, an unavailable picture is generated as follows:

* The value of each element in the sample array SL for the picture is set equal to 1 << ( BitDepthY − 1 ).
* When ChromaArrayType is not equal to 0, the value of each element in the sample arrays SCb and SCr for the picture is set equal to 1 << ( BitDepthC − 1 ).
* The prediction mode CuPredMode[ x ][ y ] is set equal to MODE\_INTRA for x ranging from 0 to pic\_width\_in\_luma\_samples − 1, inclusive, and y ranging from 0 to pic\_height\_in\_luma\_samples − 1, inclusive.

It is a requirement of bitstream conformance that the output of the recovery point picture following a GRA picture with NoIncorrectPicOutputFlag equal to 1 and the pictures following that recovery point picture in output order and decoding order is independent of the values set for the elements of SL, SCb, SCr and CuPredMode[ x ][ y ]. [Ed. (YK): This constraint is not really needed, as once we have the constraint that requires all pictures starting from the recovery point to be mismatch-free, setting of whatever sample values for the unavailable reference pictures would not matter anyway. And this also applies to trailing pictures associated with a CRA picture with NoIncorrectPicOutputFlag equal to 1. Finally, this should be changed to be a NOTE.]

### Decoding process for symmetric motion vector difference reference indices

Output of this process are RefIdxSymL0 and RefIdxSymL1 specifying the list 0 and list 1 reference picture indices for symmetric motion vector differences, i.e., when sym\_mvd\_flag is equal to 1 for a coding unit.

The variable RefIdxSymLX with X being 0 and 1 is derived as follows:

* The variable currPic specifies the current picture.
* RefIdxSymL0 is set equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 0 ] − 1, the following applies:
* When all of the following conditions are true, RefIdxSymL0 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) > 0,
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) < DiffPicOrderCnt( currPic, RefPicList[ 0 ][ RefIdxSymL0 ] ) or RefIdxSymL0 is equal to −1.
* RefIdxSymL1 is set equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 1 ]  − 1, the following applies:
* When all of the following conditions are true, RefIdxSymL1 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) < 0,
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) > DiffPicOrderCnt( currPic, RefPicList[ 1 ][ RefIdxSymL1 ] ) or RefIdxSymL1 is equal to −1.
* When RefIdxSymL0 is equal to −1 or RefIdxSymL1 is equal to −1, the following applies:
* For each index i with i = 0..NumRefIdxActive[ 0 ]  − 1, the following applies:
* When all of the following conditions are true, RefIdxSymL0 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) < 0,
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ i ] ) > DiffPicOrderCnt( currPic, RefPicList[ 0 ][ RefIdxSymL0 ] ) or RefIdxSymL0 is equal to −1.
* For each index i with i = 0..NumRefIdxActive[ 1 ]  − 1, the following applies:
* When all of the following conditions are true, RefIdxSymL1 is set to i:
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) > 0,
* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ i ] ) < DiffPicOrderCnt( currPic, RefPicList[ 1 ][ RefIdxSymL1 ] ) or RefIdxSymL1 is equal to −1.

## Decoding process for coding units coded in intra prediction mode

### General decoding process for coding units coded in intra prediction mode

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

Output of this process is a modified reconstructed picture before in-loop filtering.

The derivation process for quantization parameters as specified in clause 8.7.1 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight, and the variable treeType as inputs.

When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the decoding process for luma samples is specified as follows:

* If pcm\_flag[ xCb ][ yCb ] is equal to 1, the reconstructed picture is modified as follows:

SL[ xCb + i ][ yCb + j ] =   
 pcm\_sample\_luma[ ( cbHeight \* j ) + i ]  <<  ( BitDepthY − PcmBitDepthY ), (8‑6)  
 with i = 0..cbWidth − 1, j = 0..cbHeight − 1

* Otherwise, the following applies:

1. The variable MipSizeId[ x ][ y ] for x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is derived as follows:

* If both cbWidth and cbHeight are equal to 4, MipSizeId[ x ][ y ] is set equal to 0.
* Otherwise, if both cbWidth and cbHeight are less than or equal to 8, MipSizeId[ x ][ y ] is set equal to 1.
* Otherwise, MipSizeId[ x ][ y ] is set equal to 2.

1. The luma intra prediction mode is derived as follows:

* If intra\_mip\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for matrix-based intra prediction mode as specified in clause 8.4.2 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.
* Otherwise, the derivation process for the luma intra prediction mode as specified in clause 8.4.3 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.

1. The variable predModeIntra is derived as follows:

* If BdpcmFlag[ xCb ][ yCb ] is equal to 1, predModeIntra is derived as follows:

predModeIntra = BdpcmDir[ xCb ][ yCb ] ? 50 : 18 (8‑7)

* Otherwise, predModeIntra is set equal to IntraPredModeY[ xCb ][ yCb ].

1. The general decoding process for intra blocks as specified in clause 8.4.5.1 is invoked with the sample location ( xTb0, yTb0 ) set equal to the luma location ( xCb, yCb ), the variable nTbW set equal to cbWidth, the variable nTbH set equal to cbHeight, predModeIntra, and the variable cIdx set equal to 0 as inputs, and the output is a modified reconstructed picture before in-loop filtering.

When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, and when ChromaArrayType is not equal to 0, the decoding process for chroma samples is specified as follows:

* If pcm\_flag[ xCb ][ yCb ] is equal to 1, the reconstructed picture is modified as follows:

SCb[ xCb / SubWidthC + i ][ yCb / SubHeightC + j ] =  
 pcm\_sample\_chroma[ ( cbHeight / SubWidthC \* j ) + i ]  <<  ( BitDepthC − PcmBitDepthC ),  
 with i = 0.. cbWidth / SubWidthC − 1 and j = 0.. cbHeight / SubHeightC − 1 (8‑8)

SCr[ xCb / SubWidthC + i ][ yCb / SubHeightC + j ] =  
 pcm\_sample\_chroma[ ( cbHeight / SubWidthC \* ( j + cbHeight / SubHeightC ) ) + i ]  <<  
 ( BitDepthC − PcmBitDepthC ),  
 with i = 0..cbWidth / SubWidthC − 1 and j = 0..cbHeight / SubHeightC − 1 (8‑9)

* Otherwise, the following applies:

1. The derivation process for the chroma intra prediction mode as specified in clause 8.4.4 is invoked with the luma location ( xCb, yCb ) , the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight as input.
2. The general decoding process for intra blocks as specified in clause 8.4.5.1 is invoked with the sample location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC , yCb / SubHeightC  ), the variable nTbW set equal to ( cbWidth / SubWidthC  ), the variable nTbH set equal to ( cbHeight / SubHeightC ), the variable predModeIntra set equal to IntraPredModeC[ xCb ][ yCb ], and the variable cIdx set equal to 1, and the output is a modified reconstructed picture before in-loop filtering.
3. The general decoding process for intra blocks as specified in clause 8.4.5.1 is invoked with the sample location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC  , yCb / SubHeightC ), the variable nTbW set equal to ( cbWidth / SubWidthC  ), the variable nTbH set equal to ( cbHeight / SubHeightC ), the variable predModeIntra set equal to IntraPredModeC[ xCb ][ yCb ], and the variable cIdx set equal to 2, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for MIP mode

Input to this process are:

* a luma location ( xCb , yCb ) specifying the top-left sample of the current luma coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the matrix-based intra prediction mode IntraPredModeY[ xCb ][ yCb ] is derived.

IntraPredModeY[ xCb ][ yCb ] is derived by the following ordered steps:

1. The neighbouring locations ( xNbA, yNbA ) and ( xNbB, yNbB ) are set equal to ( xCb − 1, yCb ) and ( xCb, yCb − 1 ), respectively.
2. For X being replaced by either A or B, the variables candMipModeX are derived as follows:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbX, yNbX ) as inputs, and the output is assigned to availableX.
* The candidate MIP mode candMipModeX is derived as follows:
* If one or more of the following conditions are true, candMipModeX is set equal to −1.
* The variable availableX is equal to FALSE.
* CuPredMode[ xNbX ][ yNbX ] is not equal to MODE\_INTRA and ciip\_flag[ xNbX ][ yNbX ] is not equal to 1.
* pcm\_flag[ xNbX ][ yNbX ] is equal to 1.
* X is equal to B and yCb − 1 is less than ( ( yCb  >>  CtbLog2SizeY )  <<  CtbLog2SizeY ).
* Otherwise, the following applies:
* If intra\_mip\_flag[ xNbX ][ yNbX ] is equal to 1, the following applies:.
* If MipSizeId[ xCb ][ yCb ] is equal to MipSizeId[ xNbX ][ yNbX ], candMipModeX is set equal to IntraPredModeY[ xNbX ][ yNbX ].
* Otherwise, candMipModeX is set equal to −1.
* Otherwise, candMipModeX is derived using IntraPredModeY[ xNbX ][ yNbX ] and MipSizeId[ xCb ][ yCb ] as specified in Table 8‑4.

1. The candMipModeList[ x ] with x = 0..2 is derived as follows, using mipMpmCand[ sizeId ] as specified in Table 8‑2 with sizeId set equal to MipSizeId[ xCb ][ yCb ]:

* If candMipModeA and candMipModeB are both equal to −1, the following applies:

candMipModeList[ 0 ] = mipMpmCand[ sizeId ][ 0 ] (8‑10)

candMipModeList[ 1 ] = mipMpmCand[ sizeId ][ 1 ] (8‑11)

candMipModeList[ 2 ] = mipMpmCand[ sizeId ][ 2 ] (8‑12)

* Otherwise, the following apllies:
* If candMipModeA is equal to candMipModeB or if either candMipModeA or candMipModeB is equal to −1, the following applies:

candMipModeList[ 0 ] = ( candMipModeA != −1 )  ?  candMipModeA  :  candMipModeB (8‑13)

* If candMipModeList[ 0 ] is equal to mipMpmCand[ sizeId ][ 0 ], the following applies:

candMipModeList[ 1 ] = mipMpmCand[ sizeId ][ 1 ] (8‑14)

candMipModeList[ 2 ] = mipMpmCand[ sizeId ][ 2 ] (8‑15)

* Otherwise, the following applies:

candMipModeList[ 1 ] = mipMpmCand[ sizeId ][ 0 ] (8‑16)

candMipModeList[ 2 ] = ( candMipModeList[ 0 ] != mipMpmCand[ sizeId ][ 1 ] )  ?    
 mipMpmCand[ sizeId ][ 1 ]  :  mipMpmCand[ sizeId ][ 2 ] (8‑17)

* Otherwise, the following applies:

candMipModeList[ 0 ] = candMipModeA (8‑18)

candMipModeList[ 1 ] = candMipModeB (8‑19)

* If candMipModeA and candMipModeB are both not equal to mipMpmCand[ sizeId ][ 0 ], the following applies:

candMipModeList[ 2 ] = mipMpmCand[ sizeId ][ 0 ] (8‑20)

* Otherwise, the following applies:
* If candMipModeA and candMipModeB are both not equal to mipMpmCand[ sizeId ][ 1 ], the following applies:

candMipModeList[ 2 ] = mipMpmCand[ sizeId ][ 1 ] (8‑21)

* Otherwise, the following applies:

candMipModeList[ 2 ] = mipMpmCand[ sizeId ][ 2 ] (8‑22)

1. IntraPredModeY[ xCb ][ yCb ] is derived by applying the following procedure:

* If intra\_mip\_mpm\_flag[ xCb ][ yCb ] is equal to 1, IntraPredModeY[ xCb ][ yCb ] is set equal to candMipModeList[ intra\_mip\_mpm\_idx[ xCb ][ yCb ] ].
* Otherwise, IntraPredModeY[ xCb ][ yCb ] is derived by applying the following ordered steps:

1. When candMipModeList[ i ] is greater than candMipModeList[ j ] for i = 0..1 and for each i, j = ( i + 1 )..2, both values are swapped as follows:

( candMipModeList[ i ], candMipModeList[ j ] ) = Swap( candMipModeList[ i ], candMipModeList[ j ] ) (8‑23)

1. IntraPredModeY[ xCb ][ yCb ] is derived by the following ordered steps:
   1. IntraPredModeY[ xCb ][ yCb ] is set equal to intra\_mip\_mpm\_remainder[ xCb ][ yCb ].
   2. For i ranging from 0 to 2, inclusive, when IntraPredModeY[ xCb ][ yCb ] is greater than or equal to candMipModeList[ i ], the value of IntraPredModeY[ xCb ][ yCb ] is incremented by one.

The variable IntraPredModeY[ x ][ y ] with x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is set to be equal to IntraPredModeY[ xCb ][ yCb ].

**Table 8‑1 – Specification of mapping between intra prediction and MIP modes**

|  |  |  |  |
| --- | --- | --- | --- |
| **IntraPredModeY[ xNbX ][ yNbX ]** | **MipSizeId** | | |
| **0** | **1** | **2** |
| **0** | 17 | 0 | 5 |
| **1** | 17 | 0 | 1 |
| **2, 3** | 17 | 10 | 3 |
| **4, 5** | 9 | 10 | 3 |
| **6,7** | 9 | 10 | 3 |
| **8, 9** | 9 | 10 | 3 |
| **10, 11** | 9 | 10 | 0 |
| **12, 13** | 17 | 4 | 0 |
| **14, 15** | 17 | 6 | 0 |
| **16, 17** | 17 | 7 | 4 |
| **18, 19** | 17 | 7 | 4 |
| **20, 21** | 17 | 7 | 4 |
| **22, 23** | 17 | 5 | 5 |
| **24, 25** | 17 | 5 | 1 |
| **26, 27** | 5 | 0 | 1 |
| **28, 29** | 5 | 0 | 1 |
| **30, 31** | 5 | 3 | 1 |
| **32, 33** | 5 | 3 | 1 |
| **34, 35** | 34 | 12 | 6 |
| **36, 37** | 22 | 12 | 6 |
| **38, 39** | 22 | 12 | 6 |
| **40, 41** | 22 | 12 | 6 |
| **42, 43** | 22 | 14 | 6 |
| **44, 45** | 34 | 14 | 10 |
| **46, 47** | 34 | 14 | 10 |
| **48, 49** | 34 | 16 | 9 |
| **50, 51** | 34 | 16 | 9 |
| **52, 53** | 34 | 16 | 9 |
| **54, 55** | 34 | 15 | 9 |
| **56, 57** | 34 | 13 | 9 |
| **58, 59** | 26 | 1 | 8 |
| **60, 61** | 26 | 1 | 8 |
| **62, 63** | 26 | 1 | 8 |
| **64, 65** | 26 | 1 | 8 |
| **66** | 26 | 1 | 8 |

**Table 8‑2 – Specification of MIP candidate modes mipMpmCand[ sizeId ][ x ]**

|  |  |  |  |
| --- | --- | --- | --- |
| **sizeId** | **candidate mode x** | | |
| **0** | **1** | **2** |
| **0** | 17 | 34 | 5 |
| **1** | 0 | 7 | 16 |
| **2** | 1 | 4 | 6 |

### Derivation process for luma intra prediction mode

Input to this process are:

* a luma location ( xCb , yCb ) specifying the top-left sample of the current luma coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the luma intra prediction mode IntraPredModeY[ xCb ][ yCb ] is derived.

Table 8‑3 specifies the value for the intra prediction mode IntraPredModeY[ xCb ][ yCb ] and the associated names.

Table 8‑3 – Specification of intra prediction mode and associated names

|  |  |
| --- | --- |
| **Intra prediction mode** | **Associated name** |
| 0 | INTRA\_PLANAR |
| 1 | INTRA\_DC |
| 2..66 | INTRA\_ANGULAR2..INTRA\_ANGULAR66 |
| 81..83 | INTRA\_LT\_CCLM, INTRA\_L\_CCLM, INTRA\_T\_CCLM |

NOTE – : The intra prediction modes INTRA\_LT\_CCLM, INTRA\_L\_CCLM and INTRA\_T\_CCLM are only applicable to chroma components.

IntraPredModeY[ xCb ][ yCb ] is derived as follows:

* If BdpcmFlag[ xCb ][ yCb ] is equal to 1 or intra\_luma\_not\_planar\_flag[ xCb ][ yCb ] is equal to 0, IntraPredModeY[ xCb ][ yCb ] is set equal to INTRA\_PLANAR.
* Otherwise (intra\_luma\_not\_planar\_flag[ xCb ][ yCb ] is equal to 1), the following ordered steps apply:

1. The neighbouring locations ( xNbA, yNbA ) and ( xNbB, yNbB ) are set equal to ( xCb − 1, yCb + cbHeight − 1 ) and ( xCb + cbWidth − 1, yCb − 1 ), respectively.
2. For X being replaced by either A or B, the variables candIntraPredModeX are derived as follows:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbX, yNbX ) as inputs, and the output is assigned to availableX.
* The candidate intra prediction mode candIntraPredModeX is derived as follows:
* If one or more of the following conditions are true, candIntraPredModeX is set equal to INTRA\_PLANAR.
* The variable availableX is equal to FALSE.
* CuPredMode[ xNbX ][ yNbX ] is not equal to MODE\_INTRA and ciip\_flag[ xNbX ][ yNbX ] is not equal to 1.
* pcm\_flag[ xNbX ][ yNbX ] is equal to 1.
* X is equal to B and yCb − 1 is less than ( ( yCb  >>  CtbLog2SizeY )  <<  CtbLog2SizeY ).
* Otherwise, candIntraPredModeX is derived as follows:
* If intra\_mip\_flag[ xNbX ][ yNbX ] is equal to 1, candIntraPredModeX is derived using IntraPredModeY[ xNbX ][ yNbX ] and MipSizeId[ xCb ][ yCb ] as specified in Table 8‑4.
* Otherwise, candIntraPredModeX is set equal to IntraPredModeY[ xNbX ][ yNbX ].

1. The candModeList[ x ] with x = 0..4 is derived as follows:

* If candIntraPredModeB is equal to candIntraPredModeA and candIntraPredModeA is greater than INTRA\_DC, candModeList[ x ] with x = 0..4 is derived as follows:

candModeList[ 0 ] = candIntraPredModeA (8‑24)

candModeList[ 1 ] = 2 + ( ( candIntraPredModeA + 61 ) % 64 ) (8‑25)

candModeList[ 2 ] = 2 + ( ( candIntraPredModeA − 1 ) % 64 ) (8‑26)

candModeList[ 3 ] = INTRA\_DC (8‑27)

candModeList[ 4 ] = 2 + ( ( candIntraPredModeA + 60 ) % 64 ) (8‑28)

* Otherwise if candIntraPredModeB is not equal to candIntraPredModeA and candIntraPredModeA or candIntraPredModeB is greater than INTRA\_DC, the following applies:
  + The variables minAB and maxAB are derived as follows:

minAB = Min( candIntraPredModeA, candIntraPredModeB ) (8‑29)

maxAB = Max( candIntraPredModeA, candIntraPredModeB ) (8‑30)

* + If candIntraPredModeA and candIntraPredModeB are both greater than INTRA\_DC, candModeList[ x ] with x = 0..4 is derived as follows:

candModeList[ 0 ] = candIntraPredModeA (8‑31)

candModeList[ 1 ] = candIntraPredModeB (8‑32)

candModeList[ 2 ] = INTRA\_DC (8‑33)

* + If maxAB − minAB is in the range of 2 to 62, inclusive, the following applies:

candModeList[ 3 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑34)

candModeList[ 4 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑35)

* + Otherwise, the following applies:

candModeList[ 3 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑36)

candModeList[ 4 ] = 2 + ( ( maxAB ) % 64 ) (8‑37)

* + Otherwise (candIntraPredModeA or candIntraPredModeB is greater than INTRA\_DC), candModeList[ x ] with x = 0..4 is derived as follows:

candModeList[ 0 ] = maxAB (8‑38)

candModeList[ 1 ] = INTRA\_DC (8‑39)

candModeList[ 2 ] = 2 + ( ( maxAB + 61 ) % 64 ) (8‑40)

candModeList[ 3 ] = 2 + ( ( maxAB − 1 ) % 64 ) (8‑41)

candModeList[ 4 ] = 2 + ( ( maxAB + 60 ) % 64 ) (8‑42)

* Otherwise, the following applies:

candModeList[ 0 ] = INTRA\_DC (8‑43)

candModeList[ 1 ] = INTRA\_ANGULAR50 (8‑44)

candModeList[ 2 ] = INTRA\_ANGULAR18 (8‑45)

candModeList[ 3 ] = INTRA\_ANGULAR46 (8‑46)

candModeList[ 4 ] = INTRA\_ANGULAR54 (8‑47)

1. IntraPredModeY[ xCb ][ yCb ] is derived by applying the following procedure:

* If intra\_luma\_mpm\_flag[ xCb ][ yCb ] is equal to 1, the IntraPredModeY[ xCb ][ yCb ] is set equal to candModeList[ intra\_luma\_mpm\_idx[ xCb ][ yCb ] ].
* Otherwise, IntraPredModeY[ xCb ][ yCb ] is derived by applying the following ordered steps:

1. When candModeList[ i ] is greater than candModeList[ j ] for i = 0..3 and for each i, j = ( i + 1 )..4, both values are swapped as follows:

( candModeList[ i ], candModeList[ j ] ) = Swap( candModeList[ i ], candModeList[ j ] ) (8‑48)

1. IntraPredModeY[ xCb ][ yCb ] is derived by the following ordered steps:
   1. IntraPredModeY[ xCb ][ yCb ] is set equal to intra\_luma\_mpm\_remainder[ xCb ][ yCb ].
   2. The value of IntraPredModeY[ xCb ][ yCb ] is incremented by one.
   3. For i equal to 0 to 4, inclusive, when IntraPredModeY[ xCb ][ yCb ] is greater than or equal to candModeList[ i ], the value of IntraPredModeY[ xCb ][ yCb ] is incremented by one.

The variable IntraPredModeY[ x ][ y ] with x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1 is set to be equal to IntraPredModeY[ xCb ][ yCb ].

**Table 8‑4 – Specification of mapping between MIP and intra prediction modes**

|  |  |  |  |
| --- | --- | --- | --- |
| **IntraPredModeY[ xNbX ][ yNbX ]** | **MipSizeId** | | |
| **0** | **1** | **2** |
| **0** | 0 | 0 | 1 |
| **1** | 18 | 1 | 1 |
| **2** | 18 | 0 | 1 |
| **3** | 0 | 1 | 1 |
| **4** | 18 | 0 | 18 |
| **5** | 0 | 22 | 0 |
| **6** | 12 | 18 | 1 |
| **7** | 0 | 18 | 0 |
| **8** | 18 | 1 | 1 |
| **9** | 2 | 0 | 50 |
| **10** | 18 | 1 | 0 |
| **11** | 12 | 0 |  |
| **12** | 18 | 1 |  |
| **13** | 18 | 0 |  |
| **14** | 1 | 44 |  |
| **15** | 18 | 0 |  |
| **16** | 18 | 50 |  |
| **17** | 0 | 1 |  |
| **18** | 0 | 0 |  |
| **19** | 50 |  |  |
| **20** | 0 |  |  |
| **21** | 50 |  |  |
| **22** | 0 |  |  |
| **23** | 56 |  |  |
| **24** | 0 |  |  |
| **25** | 50 |  |  |
| **26** | 66 |  |  |
| **27** | 50 |  |  |
| **28** | 56 |  |  |
| **29** | 50 |  |  |
| **30** | 50 |  |  |
| **31** | 1 |  |  |
| **32** | 50 |  |  |
| **33** | 50 |  |  |
| **34** | 50 |  |  |

### Derivation process for chroma intra prediction mode

Input to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current chroma coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

In this process, the chroma intra prediction mode IntraPredModeC[ xCb ][ yCb ] is derived.

The corresponding luma intra prediction mode lumaIntraPredMode is derived as follows:

* If intra\_mip\_flag[ xCb ][ yCb ] is equal to 1, lumaIntraPredMode is derived using IntraPredModeY[ xCb + cbWidth / 2 ][ yCb + cbHeight / 2 ] and sizeId set equal to MipSizeId[ xCb ][ yCb ] as specified in Table 8‑4.
* Otherwise, lumaIntraPredMode is set equal to IntraPredModeY[ xCb + cbWidth / 2 ][ yCb + cbHeight / 2 ].

The chroma intra prediction mode IntraPredModeC[ xCb ][ yCb ] is derived using intra\_chroma\_pred\_mode[ xCb ][ yCb ] and lumaIntraPredMode as specified in Table 8‑5 and Table 8‑6.

Table 8‑5 – Specification of IntraPredModeC[ xCb ][ yCb ] depending on intra\_chroma\_pred\_mode[ xCb ][ yCb ] and lumaIntraPredMode when sps\_cclm\_enabled\_flag is equal to 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | lumaIntraPredMode | | | | |
| 0 | 50 | 18 | 1 | X ( 0  <=  X  <=  66 ) |
| 0 | 66 | 0 | 0 | 0 | 0 |
| 1 | 50 | 66 | 50 | 50 | 50 |
| 2 | 18 | 18 | 66 | 18 | 18 |
| 3 | 1 | 1 | 1 | 66 | 1 |
| 4 | 0 | 50 | 18 | 1 | X |

Table 8‑6 – Specification of IntraPredModeC[ xCb ][ yCb ] depending on intra\_chroma\_pred\_mode[ xCb ][ yCb ] and lumaIntraPredMode when sps\_cclm\_enabled\_flag is equal to 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| intra\_chroma\_pred\_mode[ xCb ][ yCb ] | lumaIntraPredMode | | | | |
| 0 | 50 | 18 | 1 | X ( 0  <=  X  <=  66 ) |
| 0 | 66 | 0 | 0 | 0 | 0 |
| 1 | 50 | 66 | 50 | 50 | 50 |
| 2 | 18 | 18 | 66 | 18 | 18 |
| 3 | 1 | 1 | 1 | 66 | 1 |
| 4 | 81 | 81 | 81 | 81 | 81 |
| 5 | 82 | 82 | 82 | 82 | 82 |
| 6 | 83 | 83 | 83 | 83 | 83 |
| 7 | 0 | 50 | 18 | 1 | X |

When chroma\_format\_idc is equal to 2, the chroma intra prediction mode Y is derived using the chroma intra prediction mode X in Table 8‑5 and Table 8‑6 as specified in Table 8‑7, and the chroma intra prediction mode X is set equal to the chroma intra prediction mode Y afterwards.

Table 8‑7 – Specification of the 4:2:2 mapping process from chroma intra prediction mode X to mode Y when chroma\_format\_idc is equal to 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **mode X** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| **mode Y** | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 6 | 8 | 10 | 12 | 13 | 14 | 16 |
| **mode X** | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| **mode Y** | 18 | 20 | 22 | 23 | 24 | 26 | 28 | 30 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 |
| **mode X** | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 |
| **mode Y** | 42 | 43 | 44 | 44 | 44 | 45 | 46 | 46 | 46 | 47 | 48 | 48 | 48 | 49 | 50 | 51 | 52 | 52 |
| **mode X** | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 |  |  |  |  |  |
| **mode Y** | 52 | 53 | 54 | 54 | 54 | 55 | 56 | 56 | 56 | 57 | 58 | 59 | 60 |  |  |  |  |  |

### Decoding process for intra blocks

#### General decoding process for intra blocks

Inputs to this process are:

* a sample location ( xTb0, yTb0 ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable predModeIntra specifying the intra prediction mode,
* a variable cIdx specifying the colour component of the current block.

Output of this process is a modified reconstructed picture before in-loop filtering.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = ( cIdx  = =  0 ) ? MaxTbSizeY : MaxTbSizeY / Min( SubWidthC, SubHeightC )[Ed. (JC), should be max TB size for chroma same to Luma for 422 and 444 case, to be confirmed] (8‑49)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 \* SubWidthC, yTb0 \* SubHeightC ) (8‑50)

Depending on maxTbSize, the following applies:

* If IntraSubPartitionsSplitType is equal to NO\_ISP\_SPLIT and nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

newTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑51)

newTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑52)

1. The general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
2. If nTbW is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
3. If nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
4. If nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the general decoding process for intra blocks as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, the intra prediction mode predModeIntra, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

* Otherwise, the following ordered steps apply:
* The variables nW, nH, numPartsX and numPartsY are derived as follows:

nW = IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ? nTbW / NumIntraSubPartitions : nTbW (8‑53)

nH = IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ? nTbH / NumIntraSubPartitions : nTbH (8‑54)

numPartsX = IntraSubPartitionsSplitType = = ISP\_VER\_SPLIT ? NumIntraSubPartitions : 1 (8‑55)

numPartsY = IntraSubPartitionsSplitType = = ISP\_HOR\_SPLIT ? NumIntraSubPartitions : 1 (8‑56)

* For xPartIdx = 0..numPartsX − 1 and yPartIdx = 0..numPartsY − 1, the following applies:

1. The intra sample prediction process as specified in clause 8.4.5.2 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xTb0 + nW \* xPartIdx, yTb0 + nH \* yPartIdx ), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH set equal to nW and nH, the coding block width nCbW and height nCbH set equal to nTbW and nTbH, and the variable cIdx as inputs, and the output is an (nTbW)x(nTbH) array predSamples.
2. The scaling and transformation process as specified in clause 8.7.2 is invoked with the luma location ( xTbY, yTbY ) set equal to ( xTbY + nW \* xPartIdx, yTbY + nH \* yPartIdx ), the variable cIdx, the transform width nTbW and the transform height nTbH set equal to nW and nH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.
3. The picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the transform block location ( xTbComp, yTbComp ) set equal to ( xTb0 + nW \* xPartIdx, yTb0 + nH \* yPartIdx ), the transform block width nTbW, the transform block height nTbH set equal to nW and nH, the variable cIdx, the (nTbW)x(nTbH) array predSamples, and the (nTbW)x(nTbH) array resSamples as inputs, and the output is a modified reconstructed picture before in-loop filtering.

#### Intra sample prediction

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable predModeIntra specifying the intra prediction mode,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable nCbW specifying the coding block width,
* a variable nCbH specifying the coding block height,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The predicted samples predSamples[ x ][ y ] are derived as follows:

* If intra\_mip\_flag[ xTbComp ][ yTbComp ] is equal to 1 and cIdx is equal to 0, the matrix-based intra sample prediction process as specified in clause 8.4.5.2.1 is invoked with the location ( xTbCmp, yTbCmp ), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH as inputs, and the output is predSamples.
* Otherwise, the general intra sample prediction process as specified in clause 8.4.5.2.5 is invoked with the location ( xTbCmp, yTbCmp ), the intra prediction mode predModeIntra, the transform block width nTbW and height nTbH, the coding block width nCbW and height nCbH, and the variable cIdx as inputs, and the output is predSamples.

##### Matrix-based intra sample prediction

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable predModeIntra specifying the intra prediction mode,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

Variables numModes, boundarySize, predW, predH and predC are derived using MipSizeId[ xTbCmp ][ yTbCmp ] as specified in Table 8‑8.

Table 8‑8 – Specification of number of prediction modes numModes, boundary size boundarySize, and prediction sizes predW, predH and predC using MipSizeId

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **MipSizeId** | **numModes** | **boundarySize** | **predW** | **predH** | **predC** |
| **0** | 35 | 2 | 4 | 4 | 4 |
| **1** | 19 | 4 | 4 | 4 | 4 |
| **2** | 11 | 4 | Min( nTbW, 8 ) | Min( nTbH, 8 ) | 8 |

The flag isTransposed is derived as follows:

isTransposed = ( predModeIntra > ( numModes / 2 ) ) ? TRUE : FALSE (8‑57)

The flags needUpsBdryHor and needUpsBdryVer are derived as follows:

needUpsBdryHor = ( nTbW > predW ) ? TRUE : FALSE (8‑58)

needUpsBdryVer = ( nTbH > predH ) ? TRUE : FALSE (8‑59)

The variables upsBdryW and upsBdryH are derived as follows:

upsBdryW = ( nTbH > nTbW ) ? nTbW : predW (8‑60)

upsBdryH = ( nTbH > nTbW ) ? predH : nTbH (8‑61)

The variables mipW and mipH are derived as follows:

mipW = isTransposed ? predH : predW (8‑62)

mipH = isTransposed ? predW : predH (8‑63)

For the generation of the reference samples refT[ x ] with x = 0..nTbW − 1 and refL[ y ] with y = 0..nTbH − 1, the MIP reference sample derivation process as specified in clause 8.4.5.2.2 is invoked with the sample location ( xTbCmp, yTbCmp ), the transform block width nTbW, the transform block height nTbH as inputs, and top and left reference samples refT[ x ] with x = 0..nTbW − 1 and refL[ y ] with y = 0..nTbH − 1, respectively, as outputs.

For the generation of the boundary samples p[ x ] with x = 0..2 \* boundarySize − 1, the following applies:

* The MIP boundary downsampling process as specified in clause 8.4.5.2.3 is invoked for the top reference samples with the block size nTbW, the reference samples refT[ x ] with x = 0..nTbW − 1, the boundary size boundarySize, the upsampling boundary flag needUpsBdryHor, and the upsampling boundary size upsBdryW as inputs, and reduced boundary samples redT[ x ] with x = 0..boundarySize − 1 and upsampling boundary samples upsBdryT[ x ] with x = 0..upsBdryW − 1 as outputs.
* The MIP boundary downsampling process as specified in clause 8.4.5.2.3 is invoked for the left reference samples with the block size nTbH, the reference samples refL[ y ] with y = 0..nTbH − 1, the boundary size boundarySize, the upsampling boundary flag needUpsBdryVer, and the upsampling boundary size upsBdryH as inputs, and reduced boundary samples redL[ x ] with x = 0..boundarySize − 1 and upsampling boundary samples upsBdryL[ x ] with x = 0..upsBdryH − 1 as outputs.
* The reduced top and left boundary samples redT and redL are assigned to the boundary sample array p as follows:
* If isTransposed is equal to 1, p[ x ] is set equal to redL[ x ] with x = 0..boundarySize − 1 and p[ x + boundarySize ] is set equal to redT[ x ] with x = 0..boundarySize − 1.
* Otherwise, p[ x ] is set equal to redT[ x ] with x = 0..boundarySize − 1 and p[ x + boundarySize ] is set equal to redL[ x ] with x = 0..boundarySize − 1.

For the intra sample prediction process according to predModeIntra, the following ordered steps apply:

1. The matrix-based intra prediction samples predMip[ x ][ y ], with x = 0..mipW − 1, y = 0..mipH − 1 are derived as follows:

* The variable modeId is derived as follows:

modeId = predModeIntra − ( isTransposed ? numModes / 2 : 0 ) (8‑64)

* The weight matrix mWeight[ x ][ y ] with x = 0..2 \* boundarySize − 1, y = 0..predC \* predC − 1is derived using MipSizeId[ xTbCmp ][ yTbCmp ] and modeId as specified in Table 8-XX [Ed. (BB): add weight matrices once a non-10-bit weight solution is adopted].
* The bias vector vBias[ y ] with y = 0..predC \* predC − 1 is derived using MipSizeId[ xTbCmp ][ yTbCmp ] and modeId as specified in Table 8-XX [Ed. (BB): add bias vectors once a non-10-bit weight solution is adopted].
* The variable sW is derived using MipSizeId[ xTbCmp ][ yTbCmp ] and modeId as specified in Table 8‑9.
* The matrix-based intra prediction samples predMip[ x ][ y ], with x = 0..mipW − 1, y = 0..mipH − 1 are derived as follows:

oW = 1 << ( sW − 1 ) (8‑65)

sB = BitDepthY − 1 (8‑66)

incW = ( predC > mipW ) ? 2 : 1 (8‑67)

incH = ( predC > mipH ) ? 2 : 1 (8‑68)

predMip[ x ][ y ] = ( () +   
 ( vBias[ y \* incH \* predC + x \* incW ]  << sB ) + oW ) >> sW (8‑69)

1. When isTransposed is equal to TRUE, the predH x predW array predMip[ x ][ y ] with x = 0..predH − 1, y = 0..predW − 1 is transposed as follows:

predTemp[ y ][ x ] = predMip[ x ][ y ] (8‑70)

predMip =predTemp (8‑71)

1. The predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* If needUpsBdryVer is equal to TRUE or needUpsBdryHor is equal to TRUE, the MIP prediction upsampling process as specified in clause 8.4.5.2.4 is invoked with the input block width predW, the input block height predH, matrix-based intra prediction samples predMip[ x ][ y ] with x = 0..predW − 1, y = 0..predH − 1, the transform block width nTbW, the transform block height nTbH, the upsampling boundary width upsBdryW, the upsampling boundary height upsBdryH, the top upsampling boundary samples upsBdryT, and the left upsampling boundary samples upsBdryL as inputs, and the output is the predicted sample array predSamples.
* Otherwise, predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 is set equal to predMip[ x ][ y ].

1. The predicted samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are clipped as follows:

predSamples[ x ][ y ] = Clip1Y( predSamples[ x ][ y ] ) (8‑72)

Table 8‑9 – Specification of weight shift sW depending on MipSizeId and modeId

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **modeId** | | | | | | | | | | | | | | | | | |
| **MipSizeId** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| **0** | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| **1** | 8 | 8 | 8 | 9 | 8 | 8 | 8 | 8 | 9 | 8 |  | | | | | | | |
| **2** | 8 | 8 | 8 | 8 | 8 | 8 |  | | | | | | | | | | | |

##### MIP reference sample derivation process

Inputs to this process are:

* a sample location ( xTbY, yTbY ) specifying the top-left luma sample of the current transform block relative to the top‑left luma sample of the current picture,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height.

Outputs of this process are the top and left reference samples refT[ x ] with x = 0..nTbW − 1 and refL[ y ] with y = 0..nTbH − 1, respectively.

The neighbouring samples refT[ x ] with x = 0..nTbW − 1 and refL[ y ] with y = 0..nTbH − 1 are reconstructed samples prior to the in-loop filter process and derived as follows:

* The top and left neighbouring luma locations (xNbT, yNbT ) and (xNbL, yNbL ) are specified by:

( xNbT, yNbT ) = ( xTbY + x, yTbY − 1 ) (8‑73)

( xNbL, yNbL ) = ( xTbY − 1, yTbY + y ) (8‑74)

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the top neighbouring luma location ( xNbT, yNbT ) as inputs, and the output is assigned to availTop[ x ] with x = 0..nTbW − 1.
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the left neighbouring luma location ( xNbL, yNbL ) as inputs, and the output is assigned to availLeft[ y ] with y = 0..nTbH − 1.
* The top reference samples refT[ x ] with x = 0..nTbW − 1 are derived as follows:
* If all availTop[ x ] with x = 0..nTbW − 1 are equal to TRUE, the sample at the location ( xNbT, yNbT ) is assigned to refT[ x ] with x = 0..nTbW − 1.
* Otherwise, if availTop[ 0 ] is equal to FALSE, all refT[ x ] with x = 0..nTbW − 1 are set equal to 1 << ( BitDepthY − 1 ).
* Otherwise, reference samples refT[ x ] with x = 0..nTbW − 1 are derived by the following ordered steps:

1. The variable lastT is set equal to the position x of the first element in the sequence availTop[ x ] with x = 1..nTbW − 1 that is equal to FALSE.
2. For every x = 0..lastT − 1, the sample at the location ( xNbT, yNbT ) is assigned to refT[ x ].
3. For every x = lastT..nTbW − 1, refT[ x ] is set equal to refT[ lastT − 1 ].

* The left reference samples refL[ y ] with x = 0..nTbH − 1 are derived as follows:
* If all availLeft[ y ] with y = 0..nTbH − 1 are equal to TRUE, the sample at the location ( xNbL, yNbL ) is assigned to refL[ y ] with y = 0..nTbH − 1.
* Otherwise, if availLeft[ 0 ] is equal to FALSE, all refL[ y ] with y = 0..nTbH − 1 are set equal to 1 << ( BitDepthY − 1 ).
* Otherwise, reference samples refL[ y ] with y = 0..nTbH − 1 are derived by the following ordered steps:

1. The variable lastL is set equal to the position y of the first element in the sequence availLeft[ y ] with y = 1..nTbH − 1 that is equal to FALSE.
2. For every y = 0..lastL − 1, the sample at the location ( xNbL, yNbL ) is assigned to refL[ y ].
3. For every y = lastL..nTbH − 1, refL[ y ] is set equal to refL[ lastL − 1 ].

##### MIP boundary sample downsampling process

Inputs to this process are:

* a variable nTbS specifying the transform block size,
* reference samples refS[ x ] with x = 0..nTbS − 1,
* a variable boundarySize specifying the downsampled boundary size,
* a flag needUpsBdry specifying whether intermediate boundary samples are required for upsampling,
* a variable upsBdrySize specifying the boundary size for upsampling.

Outputs of this process are the reduced boundary samples redS[ x ] with x = 0..boundarySize − 1 and upsampling boundary samples upsBdryS[ x ] with x = 0..upsBdrySize − 1.

The upsampling boundary samples upsBdryS[ x ] with x = 0..upsBdrySize − 1 are derived as follows:

* If needUpsBdry is equal to TRUE and upsBdrySize is less than nTbS, the following applies:

uDwn = nTbS / upsBdrySize (8‑75)

upsBdryS[ x ] = ( + ( 1 << ( Log2( uDwn ) − 1 ) )) >> Log2( uDwn ) (8‑76)

* Otherwise (upsBdrySize is equal to nTbS), upsBdryS[ x ] is set equal to refS[ x ].

The reduced boundary samples redS[ x ] with x = 0..boundarySize − 1 are derived as follows:

* If boundarySize is less than upsBdrySize, the following applies:

bDwn = upsBdrySize / boundarySize (8‑77)

redS[ x ] = ( + ( 1 << ( Log2( bDwn ) − 1 ) )) >> Log2( bDwn ) (8‑78)

* Otherwise (boundarySize is equal to upsBdrySize), redS[ x ] is set equal to upsBdryS[ x ].

##### MIP prediction upsampling process

Inputs to this process are:

* a variable predW specifying the input block width,
* a variable predH specifying the input block height,
* matrix-based intra prediction samples predMip[ x ][ y ], with x = 0..predW − 1, y = 0..predH − 1,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable upsBdryW specifying the upsampling boundary width,
* a variable upsBdryH specifying the upsampling boundary height,
* top upsampling boundary samples upsBdryT[ x ] with x = 0..upsBdryW − 1,
* left upsampling boundary samples upsBdryL[ x ] with x = 0..upsBdryH − 1.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The sparse predicted samples predSamples[ m ][ n ] are derived from predMip[ x ][ y ], with x = 0..predW − 1, y = 0..predH − 1 as follows:

upHor = nTbW / predW (8‑79)

upVer = nTbH / predH (8‑80)

predSamples[ ( x + 1 ) \* upHor − 1 ][ ( y + 1 ) \* upVer − 1 ] = predMip[ x ][ y ] (8‑81)

The top boundary samples upsBdryT[ x ] with x = 0..upsBdryW − 1 are assigned to predSamples[ m ][ −1 ] as follows:

predSamples[ ( x + 1 ) \* ( nTbW / upsBdryW ) − 1 ][ −1 ] = upsBdryT[ x ] (8‑82)

The left boundary samples upsBdryL[ y ] with y = 0..upsBdryH − 1 are assigned to predSamples[ −1 ][ n ] as follows:

predSamples[ −1 ][ ( y + 1 ) \* ( nTbH / upsBdryH ) − 1 ] = upsBdryL[ y ] (8‑83)

The predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* If nTbH is greater than nTbW, the following ordered steps apply:
  + - 1. When upHor is greater than 1, horizontal upsampling for all sparse positions ( xHor, yHor ) = ( m \* upHor − 1, n \* upVer − 1 ) with m = 0..predW − 1, n = 1..predH is applied with dX = 1..upHor − 1 as follows:

sum = ( upHor − dX ) \* predSamples[ xHor ][ yHor ] + dX \* predSamples[ xHor + upHor ][ yHor ] (8‑84)

predSamples[ xHor + dX ][ yHor ] = ( sum + upHor /2 − ( sum < 0  ?  1  :  0 ) ) / upHor (8‑85)

* + - 1. Vertical upsampling for all sparse positions ( xVer, yVer ) = ( m, n \* upVer − 1 ) with m = 0..nTbW − 1, n = 0..predH − 1 is applied with dY = 1..upVer − 1 as follows:

sum = ( upVer − dY ) \* predSamples[ xVer ][ yVer ] + dY \* predSamples[ xVer ][ yVer + upVer ] (8‑86)

predSamples[ xVer ][ yVer + dY ] = ( sum + upVer /2 − ( sum < 0  ?  1  :  0 ) ) / upVer (8‑87)

* Otherwise, the following ordered steps apply:
  + - 1. When upVer is greater than 1, vertical upsampling for all sparse positions ( xVer, yVer ) = ( m \* upHor − 1, n \* upVer − 1 ) with m = 1..predW, n = 0..predH − 1 is applied with dY = 1..upVer − 1 as follows:

sum = ( upVer − dY ) \* predSamples[ xVer ][ yVer ] + dY \* predSamples[ xVer ][ yVer + upVer ] (8‑88)

predSamples[ xVer ][ yVer + dY ] = ( sum + upVer /2 − ( sum < 0  ?  1  :  0 ) ) / upVer (8‑89)

* + - 1. Horizontal upsampling for all sparse positions ( xHor, yHor ) = ( m \* upHor − 1, n ) with m = 0..predW − 1, n = 0..nTbH − 1 is applied with dX = 1..upHor − 1 as follows:.

sum = ( upHor − dX ) \* predSamples[ xHor ][ yHor ] + dX \* predSamples[ xHor + upHor ][ yHor ] (8‑90)

predSamples[ xHor + dX ][ yHor ] = ( sum + upHor /2 − ( sum < 0  ?  1  :  0 ) ) / upHor (8‑91)

##### General intra sample prediction

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable predModeIntra specifying the intra prediction mode,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable nCbW specifying the coding block width,
* a variable nCbH specifying the coding block height,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables refW and refH are derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

refW = nTbW \* 2 (8‑92)

refH = nTbH \* 2 (8‑93)

* Otherwise ( IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:

refW = nCbW \* 2 (8‑94)

refH = nCbH \* 2 (8‑95)

The variable refIdx specifying the intra prediction reference line index is derived as follows:

refIdx = ( cIdx  = =  0 )  ?  IntraLumaRefLineIdx[ xTbCmp ][ yTbCmp ]  :  0 (8‑96)

The wide angle intra prediction mode mapping process as specified in clause 8.4.5.2.6 is invoked with predModeIntra, nTbW, nTbH and cIdx as inputs, and the modified predModeIntra as output.

The variable refFilterFlag is derived as follows:

* If predModeIntra is equal to one of the following values: 0, −14, −12, −10, −6, 2, 34, 66, 72, 76, 78, 80, then refFilterFlag is set equal to 1.
* Otherwise, refFilterFlag is set equal to 0.

For the generation of the reference samples p[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, the following ordered steps apply:

1. The reference sample availability marking process as specified in clause 8.4.5.2.7 is invoked with the sample location ( xTbCmp, yTbCmp ), the intra prediction reference line index refIdx, the reference sample width refW, the reference sample height refH, the colour component index cIdx as inputs, and the reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = − refIdx..refW − 1, y = −1 − refIdx as output.
2. When at least one sample refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx is marked as "not available for intra prediction", the reference sample substitution process as specified in clause 8.4.5.2.8 is invoked with the intra prediction reference line index refIdx, the reference sample width refW, the reference sample height refH, the reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the modified reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx as output.
3. The reference sample filtering process as specified in clause 8.4.5.2.9 is invoked with the intra prediction reference line index refIdx, the transform block width nTbW and height nTbH, the reference sample width refW, the reference sample height refH, the reference filter flag refFilterFlag, the unfiltered samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, and the colour component index cIdx as inputs, and the reference samples p[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx as output.

The intra sample prediction process according to predModeIntra applies as follows:

* If predModeIntra is equal to INTRA\_PLANAR, the corresponding intra prediction mode process specified in clause 8.4.5.2.10 is invoked with the transform block width nTbW, and the transform block height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, if predModeIntra is equal to INTRA\_DC, the corresponding intra prediction mode process specified in clause 8.4.5.2.11 is invoked with the transform block width nTbW, the transform block height nTbH, the intra prediction reference line index refIdx, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, if predModeIntra is equal to INTRA\_LT\_CCLM, INTRA\_L\_CCLM or INTRA\_T\_CCLM, the corresponding intra prediction mode process specified in clause 8.4.5.2.13 is invoked with the intra prediction mode predModeIntra, the sample location ( xTbC, yTbC ) set equal to ( xTbCmp, yTbCmp ), the transform block width nTbW and height nTbH, and the reference sample array p as inputs, and the output is the predicted sample array predSamples.
* Otherwise, the corresponding intra prediction mode process specified in clause 8.4.5.2.12 is invoked with the intra prediction mode predModeIntra, the intra prediction reference line index refIdx, the transform block width nTbW, the transform block height nTbH, the reference sample width refW, the reference sample height refH, the coding block width nCbW and height nCbH, the reference filter flag refFilterFlag, the colour component index cIdx, and the reference sample array p as inputs, and the predicted sample array predSamples as outputs.

When all of the following conditions are true, the position-dependent prediction sample filtering process specified in clause 8.4.5.2.14 is invoked with the intra prediction mode predModeIntra, the transform block width nTbW, the transform block height nTbH, the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1, the reference sample width refW, the reference sample height refH, the reference samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1, and the colour component index cIdx as inputs, and the output is the modified predicted sample array predSamples:

* IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0
* refIdx is equal to 0 or cIdx is not equal to 0
* BdpcmFlag[ xTbCmp ][ xTbCmp ] is equal to 0
* One of the following conditions is true:
* predModeIntra is equal to INTRA\_PLANAR
* predModeIntra is equal to INTRA\_DC
* predModeIntra is equal to INTRA\_ANGULAR18
* predModeIntra is equal to INTRA\_ANGULAR50
* predModeIntra is less than or equal to INTRA\_ANGULAR10
* predModeIntra is greater than or equal to INTRA\_ANGULAR58

##### Wide angle intra prediction mode mapping process

Inputs to this process are:

* a variable predModeIntra specifying the intra prediction mode,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable cIdx specifying the colour component of the current block.

Output of this process is the modified intra prediction mode predModeIntra.

Output of this process is the modified intra prediction mode predModeIntra.

The variables nW and nH are derived as follows:

* If IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

nW = nTbW (8‑97)

nH = nTbH (8‑98)

* Otherwise ( IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:

nW = nCbW (8‑99)

nH = nCbH (8‑100)

The variable whRatio is set equal to Abs( Log2( nW / nH ) ).

For non-square blocks (nW is not equal to nH), the intra prediction mode predModeIntra is modified as follows:

* If all of the following conditions are true, predModeIntra is set equal to ( predModeIntra + 65 ).
* nW is greater than nH
* predModeIntra is greater than or equal to 2
* predModeIntra is less than ( whRatio > 1 )  ?  ( 8 + 2 \* whRatio )  :  8
* Otherwise, if all of the following conditions are true, predModeIntra is set equal to ( predModeIntra − 67 ).
* nH is greater than nW
* predModeIntra is less than or equal to 66
* predModeIntra is greater than ( whRatio > 1 )  ?  ( 60 − 2 \* whRatio )  :  60

##### Reference sample availability marking process

Inputs to this process are:

* a sample location ( xTbCmp, yTbCmp ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable refIdx specifying the intra prediction reference line index,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction.

The refW + refH + 1 + ( 2 \* refIdx ) neighbouring samples refUnfilt[ x ][ y ] that are constructed samples prior to the in-loop filter process, with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx, are derived as follows:

* The neighbouring location (xNbCmp, yNbCmp ) is specified by:

( xNbCmp, yNbCmp ) = ( xTbCmp + x, yTbCmp + y ) (8‑101)

* The current luma location ( xTbY, yTbY ) and the neighbouring luma location (xNbY, yNbY ) are derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTbCmp, yTbCmp ) :  (8‑102)  
 ( xTbCmp \* SubWidthC, yTbCmp \* SubHeightC )

( xNbY, yNbY ) = ( cIdx  = =  0 ) ? ( xNbCmp, yNbCmp ) :  (8‑103)  
 ( xNbCmp \* SubWidthC, yNbCmp \* SubHeightC )

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xTbY, yTbY ) and the neighbouring luma location ( xNbY, yNbY ) as inputs, and the output is assigned to availableN.
* Each sample refUnfilt[ x ][ y ] is derived as follows:
* If availableN is equal to FALSE, the sample refUnfilt[ x ][ y ] is marked as "not available for intra prediction".
* Otherwise, the sample refUnfilt[ x ][ y ] is marked as "available for intra prediction" and the sample at the location ( xNbCmp, yNbCmp ) is assigned to refUnfilt[ x ][ y ].

##### Reference sample substitution process

Inputs to this process are:

* a variable refIdx specifying the intra prediction reference line index,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified reference samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx for intra sample prediction.

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 values of the samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are modified as follows:

* If all samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are marked as "not available for intra prediction", all values of refUnfilt[ x ][ y ] are set equal to 1  <<  ( bitDepth − 1 ).
* Otherwise (at least one but not all samples refUnfilt[ x ][ y ] are marked as "not available for intra prediction"), the following ordered steps apply:

1. When refUnfilt[ −1 − refIdx ][ refH − 1 ] is marked as "not available for intra prediction", search sequentially starting from x = −1 − refIdx, y = refH − 1 to x = −1 − refIdx, y = −1 − refIdx, then from x = −refIdx, y = −1 − refIdx to x = refW − 1, y = −1 − refIdx, for a sample refUnfilt[ x ][ y ] that is marked as "available for intra prediction". Once a sample refUnfilt[ x ][ y ] marked as "available for intra prediction" is found, the search is terminated and the value of refUnfilt[ −1 − refIdx ][ refH − 1 ] is set equal to the value of refUnfilt[ x ][ y ].
2. For x = −1 − refIdx, y = refH − 2..−1 − refIdx, when refUnfilt[ x ][ y ] is marked as "not available for intra prediction", the value of refUnfilt[ x ][ y ] is set equal to the value of refUnfilt[ x ][ y + 1 ].
3. For x = −refIdx..refW − 1, y = −1, when refUnfilt[ x ][ y ] is marked as "not available for intra prediction", the value of refUnfilt[ x ][ y ] is set equal to the value of refUnfilt[ x − 1 ][ y ].

All samples refUnfilt[ x ][ y ] with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx are marked as "available for intra prediction".

##### Reference sample filtering process

Inputs to this process are:

* a variable refIdx specifying the intra prediction reference line index,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* a variable refFilterFlag specifying the value of reference filter flag,
* the (unfiltered) neighbouring samples refUnfilt[ x ][ y ], with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the reference samples p[ x ][ y ], with x = −1 − refIdx, y = −1 − refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1 − refIdx.

The variable filterFlag is derived as follows:

* If all of the following conditions are true, filterFlag is set equal to 1:
* refIdx is equal to 0
* nTbW \* nTbH is greater than 32
* cIdx is equal to 0
* IntraSubPartitionsSplitType is equal to ISP\_NO\_SPLIT
* refFilterFlag is equal to 1
* Otherwise, filterFlag is set equal to 0.

For the derivation of the reference samples p[ x ][ y ] the following applies:

* If filterFlag is equal to 1, the filtered sample values p[ x ][ y ] with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1 are derived as follows:

p[ −1 ][ −1 ] = ( refUnfilt[ −1 ][ 0 ] + 2 \* refUnfilt[ −1 ][ −1 ] + refUnfilt[ 0 ][ −1 ] + 2 )  >>  2 (8‑104)

p[ −1 ][ y ] = ( refUnfilt[ −1 ][ y + 1 ] + 2 \* refUnfilt[ −1 ][ y ] + refUnfilt[ −1 ][ y − 1 ] + 2 )  >>  2  
 for y = 0..refH − 2 (8‑105)

p[ −1 ][ refH − 1 ] = refUnfilt[ −1 ][ refH − 1 ] (8‑106)

p[ x ][ −1 ] = ( refUnfilt[ x − 1 ][ −1 ] + 2 \* refUnfilt[ x ][ −1 ] + refUnfilt[ x + 1 ][ −1 ] + 2 )  >>  2  
 for x = 0..refW − 2  (8‑107)

p[ refW − 1 ][ −1 ] = refUnfilt[ refW − 1 ][ −1 ] (8‑108)

* Otherwise, the reference samples values p[ x ][ y ] are set equal to the unfiltered sample values refUnfilt[ x ][ y ] with x = −1− refIdx, y = −1− refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1− refIdx.

##### Specification of INTRA\_PLANAR intra prediction mode

Inputs to this process are:

* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..nTbH and x = 0..nTbW, y = −1.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables nW and nH are derived as follows:

nW = Max( nTbW, 2 ) (8‑109)

nH = Max( nTbH, 2 ) (8‑110)

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1 and y = 0..nTbH − 1, are derived as follows:

predV[ x ][ y ] = ( ( nH − 1 − y ) \* p[ x ][ −1 ] + ( y + 1 ) \* p[ −1 ][ nTbH ] ) << Log2 ( nW )  (8‑111)

predH[ x ][ y ] = ( ( nW − 1 − x ) \* p[ −1 ][ y ] + ( x + 1 ) \* p[ nTbW ][ −1 ] ) << Log2 ( nH )  (8‑112)

predSamples[ x ][ y ] = ( predV[ x ][ y ] + predH[ x ][ y ] + nW \* nH ) >> (Log2 ( nW ) + Log2 ( nH ) + 1 ) (8‑113)

##### Specification of INTRA\_DC intra prediction mode

Inputs to this process are:

* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refIdx specifying the intra prediction reference line index,
* the neighbouring samples p[ x ][ y ], with x = −1 − refIdx, y = −1 − refIdx..nTbH − refIdx − 1 and x = − refIdx..nTbW − 1 − refIdx, y = −1 − refIdx.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1, are derived by the following ordered steps:

1. A variable dcVal is derived as follows:

* When nTbW is equal to nTbH:

dcVal = ( + (8‑114)   
 + nTbW )  >>  ( Log2( nTbW ) + 1 )

* When nTbW is greater than nTbH:

dcVal = ( + ( nTbW >> 1 ) )  >>  Log2( nTbW ) (8‑115)

* When nTbW is less than nTbH:

dcVal = ( + ( nTbH >> 1 ) )  >>  Log2( nTbH ) (8‑116)

1. The prediction samples predSamples[x][y] are derived as follows:

predSamples[ x ][ y ] = dcVal, with x = 0.. nTbW − 1, y = 0.. nTbH − 1 (8‑117)

##### Specification of INTRA\_ANGULAR2..INTRA\_ANGULAR66 intra prediction modes

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a variable refIdx specifying the intra prediction reference line index,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* a variable nCbW specifying the coding block width,
* a variable nCbH specifying the coding block height,
* a variable refFilterFlag specifying the value of reference filter flag,
* a variable cIdx specifying the colour component of the current block,
* the neighbouring samples p[ x ][ y ], with x = −1− refIdx, y = −1− refIdx..refH − 1 and x = −refIdx..refW − 1, y = −1− refIdx.

Outputs of this process are the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variable nTbS is set equal to ( Log2 ( nTbW ) + Log2 ( nTbH ) )  >>  1.

The variable filterFlag is derived as follows:

* If one or more of the following conditions is true, filterFlag is set equal to 0.
* refFilterFlag is equal to 1
* refIdx is not equal to 0
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and predModeIntra is greater than or equal to INTRA\_ANGULAR34 and nTbW is less than or equal to 8
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT and predModeIntra is less than INTRA\_ANGULAR34 and nTbH is less than or equal to 8.
* Otherwise, the following applies:
* The variable minDistVerHor is set equal to Min( Abs( predModeIntra − 50 ), Abs( predModeIntra − 18 ) ).
* The variable intraHorVerDistThres[ nTbS ] is specified in Table 8‑10.
* The variable filterFlag is derived as follows:
* If one or more of the following conditions is true, filterFlag is set equal to 1:
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT
* minDistVerHor is greater than intraHorVerDistThres[ nTbS ] and refFilterFlag is equal to 0
* Otherwise, filterFlag is set equal to 0.

Table 8‑10 – Specification of intraHorVerDistThres[ nTbS ] for various transform block sizes nTbS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **nTbS = 2** | **nTbS = 3** | **nTbS = 4** | **nTbS = 5** | **nTbS = 6** | **nTbS = 7** |
| **intraHorVerDistThres[ nTbS ]** | 16 | 14 | 2 | 0 | 0 | 0 |

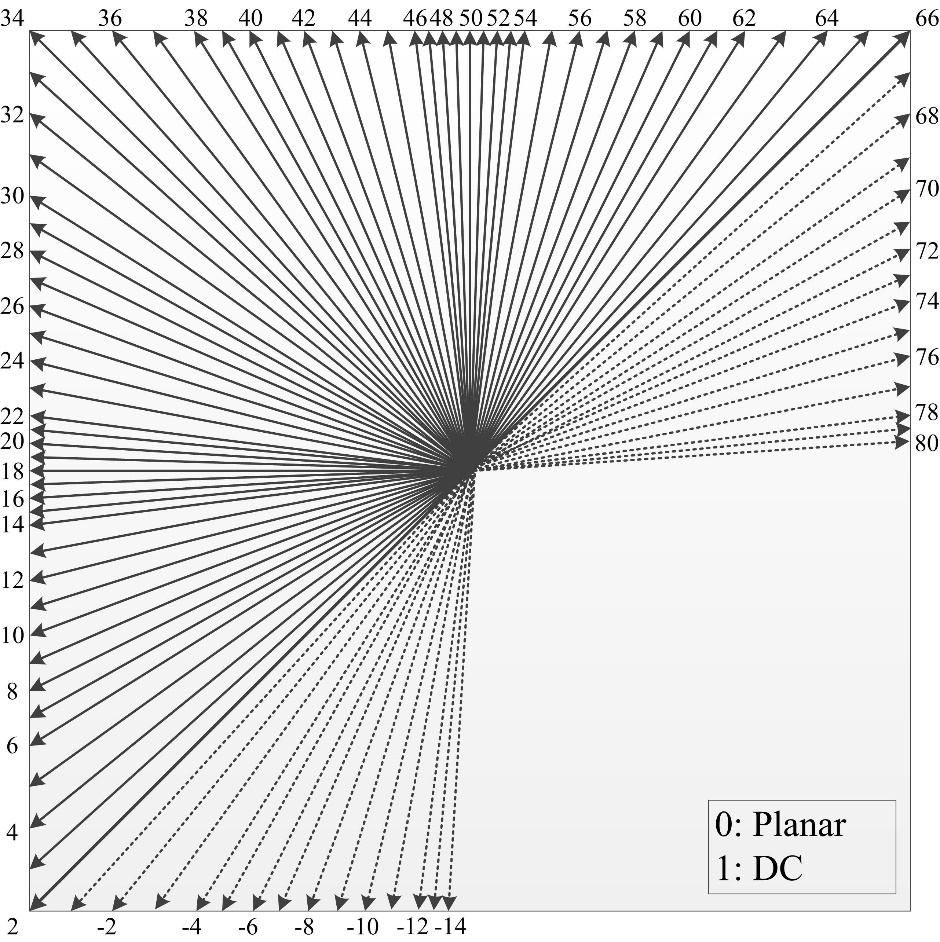


Figure 8‑1 – Intra prediction directions (informative)

Figure 8‑1 illustrates the 93 prediction directions, where the dashed directions are associated with the wide-angle modes that are only applied to non-square blocks.

Table 8‑11 specifies the mapping table between predModeIntra and the angle parameter intraPredAngle.

Table 8‑11 – Specification of intraPredAngle

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **predModeIntra** | **−14** | **−13** | **−12** | **−11** | **−10** | **−9** | **−8** | **−7** | **−6** | **−5** | **−4** | **−3** | **−2** | **−1** | **2** | **3** | **4** |
| **intraPredAngle** | 512 | 341 | 256 | 171 | 128 | 102 | 86 | 73 | 64 | 57 | 51 | 45 | 39 | 35 | 32 | 29 | 26 |
| **predModeIntra** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** |
| **intraPredAngle** | 23 | 20 | 18 | 16 | 14 | 12 | 10 | 8 | 6 | 4 | 3 | 2 | 1 | 0 | −1 | −2 | −3 |
| **predModeIntra** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** |
| **intraPredAngle** | −4 | −6 | −8 | −10 | −12 | −14 | −16 | −18 | −20 | −23 | −26 | −29 | −32 | −29 | −26 | −23 | −20 |
| **predModeIntra** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** |
| **intraPredAngle** | −18 | −16 | −14 | −12 | −10 | −8 | −6 | −4 | −3 | −2 | −1 | 0 | 1 | 2 | 3 | 4 | 6 |
| **predModeIntra** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** |
| **intraPredAngle** | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 23 | 26 | 29 | 32 | 35 | 39 | 45 | 51 | 57 | 64 |
| **predModeIntra** | **73** | **74** | **75** | **76** | **77** | **78** | **79** | **80** |  |  |  |  |  |  |  |  |  |
| **intraPredAngle** | 73 | 86 | 102 | 128 | 171 | 256 | 341 | 512 |  |  |  |  |  |  |  |  |  |

The inverse angle parameter invAngle is derived based on intraPredAngle as follows:

invAngle = Round (8‑118)

The interpolation filter coefficients fC[ phase ][ j ] and fG[ phase ][ j ] with phase = 0..31 and j = 0..3 are specified in Table 8‑12.

Table 8‑12 – Specification of interpolation filter coefficients fC and fG

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fractional sample position p** | **fC interpolation filter coefficients** | | | | **fG interpolation filter coefficients** | | | |
| **fC[ p ][ 0 ]** | **fC[ p ][ 1 ]** | **fC[ p ][ 2 ]** | **fC[ p ][ 3 ]** | **fG[ p ][ 0 ]** | **fG[ p ][ 1 ]** | **fG[ p ][ 2 ]** | **fG[ p ][ 3 ]** |
| 0 | 0 | 64 | 0 | 0 | 16 | 32 | 16 | 0 |
| 1 | −1 | 63 | 2 | 0 | 15 | 29 | 17 | 3 |
| 2 | −2 | 62 | 4 | 0 | 15 | 29 | 17 | 3 |
| 3 | −2 | 60 | 7 | −1 | 14 | 29 | 18 | 3 |
| 4 | −2 | 58 | 10 | −2 | 13 | 29 | 18 | 4 |
| 5 | −3 | 57 | 12 | −2 | 13 | 28 | 19 | 4 |
| 6 | −4 | 56 | 14 | −2 | 13 | 28 | 19 | 4 |
| 7 | −4 | 55 | 15 | −2 | 12 | 28 | 20 | 4 |
| 8 | −4 | 54 | 16 | −2 | 11 | 28 | 20 | 5 |
| 9 | −5 | 53 | 18 | −2 | 11 | 27 | 21 | 5 |
| 10 | −6 | 52 | 20 | −2 | 10 | 27 | 22 | 5 |
| 11 | −6 | 49 | 24 | −3 | 9 | 27 | 22 | 6 |
| 12 | −6 | 46 | 28 | −4 | 9 | 26 | 23 | 6 |
| 13 | −5 | 44 | 29 | −4 | 9 | 26 | 23 | 6 |
| 14 | −4 | 42 | 30 | −4 | 8 | 25 | 24 | 7 |
| 15 | −4 | 39 | 33 | −4 | 8 | 25 | 24 | 7 |
| 16 | −4 | 36 | 36 | −4 | 8 | 24 | 24 | 8 |
| 17 | −4 | 33 | 39 | −4 | 7 | 24 | 25 | 8 |
| 18 | −4 | 30 | 42 | −4 | 7 | 24 | 25 | 8 |
| 19 | −4 | 29 | 44 | −5 | 6 | 23 | 26 | 9 |
| 20 | −4 | 28 | 46 | −6 | 6 | 23 | 26 | 9 |
| 21 | −3 | 24 | 49 | −6 | 6 | 22 | 27 | 9 |
| 22 | −2 | 20 | 52 | −6 | 5 | 22 | 27 | 10 |
| 23 | −2 | 18 | 53 | −5 | 5 | 21 | 27 | 11 |
| 24 | −2 | 16 | 54 | −4 | 5 | 20 | 28 | 11 |
| 25 | −2 | 15 | 55 | −4 | 4 | 20 | 28 | 12 |
| 26 | −2 | 14 | 56 | −4 | 4 | 19 | 28 | 13 |
| 27 | −2 | 12 | 57 | −3 | 4 | 19 | 28 | 13 |
| 28 | −2 | 10 | 58 | −2 | 4 | 18 | 29 | 13 |
| 29 | −1 | 7 | 60 | −2 | 3 | 18 | 29 | 14 |
| 30 | 0 | 4 | 62 | −2 | 3 | 17 | 29 | 15 |
| 31 | 0 | 2 | 63 | −1 | 3 | 17 | 29 | 15 |

The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* If predModeIntra is greater than or equal to 34, the following ordered steps apply:

1. The reference sample array ref[ x ] is specified as follows:

* The following applies:

ref[ x ] = p[ −1 − refIdx + x ][ −1 − refIdx ], with x = 0..nTbW + refIdx (8‑119)

* If intraPredAngle is less than 0, the main reference sample array is extended as follows:
* When ( nTbH \* intraPredAngle )  >>  5 is less than −1,

ref[ x ] = p[ −1 − refIdx ][ −1 − refIdx + ( ( x \* invAngle + 128 )  >>  8 ) ],  
 with x = −1..( nTbH \* intraPredAngle )  >>  5 (8‑120)

ref[ ( ( nTbH \* intraPredAngle )  >>  5 ) − 1 ] = ref[ ( nTbH \* intraPredAngle )  >>  5 ] (8‑121)

ref[ nTbW + 1 + refIdx ] = ref[ nTbW + refIdx ] (8‑122)

* Otherwise,

ref[ x ] = p[ −1 − refIdx + x ][ −1 − refIdx ], with x = nTbW + 1 + refIdx..refW + refIdx (8‑123)

ref[ −1 ] = ref[ 0 ] (8‑124)

* The additional samples ref[ refW + refIdx +x ] with x = 1..( Max( 1, nTbW / nTbH ) \* refIdx + 1) are derived as follows:

ref[ refW + refIdx +x ] = p[ −1 + refW ][ −1 − refIdx ] (8‑125)

1. The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* The index variable iIdx and the multiplication factor iFact are derived as follows:

iIdx = ( ( y + 1 + refIdx ) \* intraPredAngle )  >>  5 + refIdx (8‑126)

iFact = ( ( y + 1 + refIdx ) \* intraPredAngle ) & 31 (8‑127)

* If cIdx is equal to 0, the following applies:
* The interpolation filter coefficients fT[ j ] with j = 0..3 are derived as follows:

fT[ j ] = filterFlag  ?  fG[ iFact ][ j ]  :  fC[ iFact ][ j ] (8‑128)

* The value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = Clip1Y( ( (  ) + 32 )  >>  6 ) (8‑129)

* Otherwise (cIdx is not equal to 0), 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 as follows:

predSamples[ x ][ y ] =   
 ( ( 32 − iFact ) \* ref[ x + iIdx + 1 ] + iFact \* ref[ x + iIdx + 2 ] + 16 )  >>  5 (8‑130)

* Otherwise, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = ref[ x + iIdx + 1 ] (8‑131)

* Otherwise (predModeIntra is less than 34), the following ordered steps apply:

1. The reference sample array ref[ x ] is specified as follows:

* The following applies:

ref[ x ] = p[ −1 − refIdx ][ −1 − refIdx + x ], with x = 0..nTbH + refIdx (8‑132)

* If intraPredAngle is less than 0, the main reference sample array is extended as follows:
* When ( nTbW \* intraPredAngle )  >>  5 is less than −1,

ref[ x ] = p[ −1 − refIdx + ( ( x \* invAngle + 128 )  >>  8 ) ][ −1 − refIdx ],  
 with x = −1..( nTbW \* intraPredAngle )  >>  5 (8‑133)

ref[ ( ( nTbW \* intraPredAngle )  >>  5 ) − 1 ] = ref[ ( nTbW \* intraPredAngle )  >>  5 ] (8‑134)

ref[ nTbG + 1 + refIdx ] = ref[ nTbH + refIdx ] (8‑135)

* Otherwise,

ref[ x ] = p[ −1 − refIdx ][ −1 − refIdx + x ], with x = nTbH + 1 + refIdx..refH + refIdx (8‑136)

ref[ −1 ] = ref[ 0 ] (8‑137)

* The additional samples ref[ refH + refIdx +x ] with x = 1..( Max( 1, nTbW / nTbH ) \* refIdx + 1) are derived as follows:

ref[ refH + refIdx +x ] = p[ −1 + refH ][ −1 − refIdx ] (8‑138)

1. The values of the prediction samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* The index variable iIdx and the multiplication factor iFact are derived as follows:

iIdx = ( ( x + 1 + refIdx ) \* intraPredAngle )  >>  5 (8‑139)

iFact = ( ( x + 1 + refIdx ) \* intraPredAngle ) & 31 (8‑140)

* If cIdx is equal to 0, the following applies:
* The interpolation filter coefficients fT[ j ] with j = 0..3 are derived as follows:

fT[ j ] = filterFlag  ?  fG[ iFact ][ j ]  :  fC[ iFact ][ j ] (8‑141)

* The value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = Clip1Y( ( (  ) + 32 )  >>  6 ) (8‑142)

* Otherwise (cIdx is not equal to 0), 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 as follows:

predSamples[ x ][ y ] =   
 ( ( 32 − iFact ) \* ref[ y + iIdx + 1 ] + iFact \* ref[ y + iIdx + 2 ] + 16 )  >>  5 (8‑143)

* Otherwise, the value of the prediction samples predSamples[ x ][ y ] is derived as follows:

predSamples[ x ][ y ] = ref[ y + iIdx + 1 ] (8‑144)

##### Specification of INTRA\_LT\_CCLM, INTRA\_L\_CCLM and INTRA\_T\_CCLM intra prediction mode

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a sample location ( xTbC, yTbC ) of the top-left sample of the current transform block relative to the top-left sample of the current picture,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* chroma neighbouring samples p[ x ][ y ], with x = −1, y = 0..2 \* nTbH − 1 and x = 0.. 2 \* nTbW − 1, y = − 1.

Output of this process are predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1.

The current luma location ( xTbY, yTbY ) is derived as follows:

( xTbY, yTbY )  =  ( xTbC << ( SubWidthC − 1 ), yTbC << ( SubHeightC − 1 ) ) (8‑145)

The variables availL, availT and availTL are derived as follows:

* The availability of left neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xTbC, yTbC ) and the neighbouring chroma location ( xTbC − 1, yTbC ) as inputs, and the output is assigned to availL.
* The availability of top neighbouring samples derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xTbC, yTbC ) and the neighbouring chroma location ( xTbC, yTbC − 1 ) as inputs, and the output is assigned to availT.
* The variable availTL is derived as follows:

availTL  =  availL  &&  availT (8‑146)

* The number of available top-right neighbouring chroma samples numTopRight is derived as follows:
* The variable numTopRight is set equal to 0 and availTR is set equal to TRUE.
* When predModeIntra is equal to INTRA\_T\_CCLM, the following applies for x = nTbW..2 \* nTbW − 1 until availTR is equal to FALSE or x is equal to 2 \* nTbW − 1:
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xTbC , yTbC ) and the neighbouring chroma location ( xTbC + x, yTbC − 1 ) as inputs, and the output is assigned to availableTR
* When availableTR is equal to TRUE, numTopRight is incremented by one.
* The number of available left-below neighbouring chroma samples numLeftBelow is derived as follows:
* The variable numLeftBelow is set equal to 0 and availLB is set equal to TRUE.
* When predModeIntra is equal to INTRA\_L\_CCLM, the following applies for y = nTbH..2 \* nTbH − 1 until availLB is equal to FALSE or y is equal to 2 \* nTbH − 1:
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xTbC , yTbC ) and the neighbouring chroma location ( xTbC − 1, yTbC + y ) as inputs, and the output is assigned to availableLB
* When availableLB is equal to TRUE, numLeftBelow is incremented by one.

The number of available neighbouring chroma samples on the top and top-right numSampT and the number of available neighbouring chroma samples on the left and left-below numSampL are derived as follows:

* If predModeIntra is equal to INTRA\_LT\_CCLM, the following applies:

numSampT = availT  ?  nTbW  :  0 (8‑147)

numSampL = availL  ?  nTbH  :  0 (8‑148)

* Otherwise, the following applies:

numSampT = ( availT  &&  predModeIntra  = =  INTRA\_T\_CCLM )  ?  ( nTbW + Min( numTopRight, nTbH ) )  :  0 (8‑149)

numSampL = ( availL  &&  predModeIntra = =  INTRA\_L\_CCLM )  ?  ( nTbH + Min( numLeftBelow, nTbW ) )  :  0 (8‑150)

The variable bCTUboundary is derived as follows:

bCTUboundary = ( yTbC & ( 1 << ( CtbLog2SizeY − 1 ) − 1 )  = =  0 )  ?  TRUE  :  FALSE. (8‑151)

The variable cntN and array pickPosN with N being replaced by L and T, are derived as follows:

* The variable numIs4N is derived as follows:

numIs4N = ( ( availT  &&  availL  &&  predModeIntra  = =  INTRA\_LT\_CCLM )  ?  0  :  1) (8‑152)

* The variable startPosN is set equal to numSampN  >>  ( 2 + numIs4N ).
* The variable pickStepN is set equal to Max( 1, numSampN  >>  ( 1 + numIs4N ) ).
* If availN is equal to TRUE and predModeIntra is equal to INTRA\_LT\_CCLM or INTRA\_N\_CCLM, the following assignments are made:
  + - cntN is set equal to Min( numSampN, ( 1 + numIs4N ) << 1 ).
    - pickPosN[ pos ] is set equal to (startPosN + pos \* pickStepN), with pos = 0.. cntN − 1.
* Otherwise, cntN is set equal to 0.

The prediction samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

* If both numSampL and numSampT are equal to 0, the following applies:

predSamples[ x ][ y ] = 1 << ( BitDepthC − 1 ) (8‑153)

* Otherwise, the following ordered steps apply:
  1. The collocated luma samples pY[ x ][ y ] with x = 0..nTbW \* SubWidthC − 1, y= 0..nTbH \* SubHeightC − 1 are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY + x, yTbY + y ).
  2. The neighbouring luma samples pY[ x ][ y ] are derived as follows:
     + When numSampL is greater than 0, the neighbouring left luma samples pY[ x ][ y ] with x = −1..−3, y = 0..SubHeightC \* numSampL − 1, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY + x , yTbY +y ).
     + When numSampT is greater than 0, the neighbouring top luma samples pY[ x ][ y ] with x = 0..SubWidthC \* numSampT − 1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY+ x, yTbY + y ).
     + When availTL is equal to TRUE, the neighbouring top-left luma samples pY[ x ][ y ] with x = −1, y = −1, −2, are set equal to the reconstructed luma samples prior to the deblocking filter process at the locations ( xTbY+ x, yTbY + y ).
  3. The down-sampled collocated luma samples pDsY[ x ][ y ] with x = 0..nTbW − 1,  y = 0..nTbH − 1 are derived as follows:
     + If both SubWidthC and SubHeightC are equal to 1, the following applies:
       - pDsY[ x ][ y ] with x = 1..nTbW − 1, y = 1..nTbH − 1 is derived as follows:

pDstY[ x ][ y ] = pY[ x ][ y ] (8‑154)

* + - Otherwise, the following applies:
      * The one-dimensional filter coefficients array F1 and F2, and the 2-dimensional filter coefficients arrays F3 and F4 are specified as follows.

F1[ i ] with i = 0..2 (8‑155)

F2[ 0 ] = 1, F2[ 1 ] = 2, F2[ 2 ] = 1 (8‑156)

F3[ i ][ j ] = F4[ i ][ j ] = 0, with i = 0..2, j = 0..2 (8‑157)

* + - * + If both SubWidthC and SubHeightC are equal to 2, the following applies:

F1[ 0 ] = 1, F1[ 1 ] = 1 (8‑158)

F3[ 0 ][ 1 ] = 1, F3[ 1 ][ 1 ] = 4, F2[ 2 ][ 1 ] = 1, F3[ 1 ][ 0 ] = 1, F3[ 1 ][ 2 ] = 1 (8‑159)

F4[ 0 ][ 1 ] = 1, F4[ 1 ][ 1 ] = 2, F4[ 2 ][ 1 ] = 1 (8‑160)

F4[ 0 ][ 2 ] = 1, F4[ 1 ][ 2 ] = 2, F4[ 2 ][ 2 ] = 1 (8‑161)

* + - * + Otherwise, the following applies:

F1[ 0 ] = 2, F1[ 1 ] = 0 (8‑162)

F3[ 1 ][ 1 ] = 8 (8‑163)

F4[ 0 ][ 1 ] = 2, F4[ 1 ][ 1 ] = 4, F4[ 2 ][ 1 ] = 2, (8‑164)

* + - * If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
        + pDsY[ x ][ y ] with x = 1..nTbW − 1, y = 1..nTbH − 1 is derived as follows:

pDsY[ x ][ y ] = ( F3[ 1 ][ 0 ] \* pY[ SubWidthC \* x ][ SubHeightC \* y − 1 ] +  
 F3[ 0 ][ 1 ] \* pY[ SubWidthC \* x − 1 ][ SubHeightC \* y ] +  
 F3[ 1 ][ 1 ] \* pY[ SubWidthC \* x ][ SubHeightC \* y ] + (8‑165)  
 F3[ 2 ][ 1 ] \* pY[ SubWidthC \* x + 1 ][ SubHeightC \* y ] +  
 F3[ 1 ][ 2 ] \* pY[ SubWidthC \* x ][ SubHeightC \* y + 1 ] + 4 ) >> 3

* + - * + If availL is equal to TRUE, pDsY[ 0 ][ y ] with y = 1..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( F3[ 1 ][ 0 ] \* pY[ 0 ][ SubHeightC \* y − 1 ] +  
 F3[ 0 ][ 1 ] \* pY[ − 1 ][ SubHeightC \* y ] +  
 F3[ 1 ][ 1 ] \* pY[ 0 ][ SubHeightC \* y ] + (8‑166)  
 F3[ 2 ][ 1 ] \* pY[ 1 ][ SubHeightC \* y ] +  
 F3[ 1 ][ 2 ] \* pY[ 0 ][ SubHeightC \* y + 1 ] + 4 ) >> 3

* + - * + Otherwise (availL is equal to FALSE), pDsY[ 0 ][ y ] with y = 1..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( F2[ 0 ] \* pY[ 0 ][ SubHeightC \* y − 1 ] +  
 F2[ 1 ] \* pY[ 0 ][ SubHeightC \* y ] + (8‑167)  
 F2[ 2 ] \* pY[ 0 ][ SubHeightC \* y + 1 ] + 2 ) >> 2

* + - * + If availT is equal to TRUE, pDsY[ x ][ 0 ] with x = 1..nTbW − 1 is derived as follows:

pDsY[ x ][ 0 ] = ( F3[ 1 ][ 0 ] \* pY[ SubWidthC \* x ][ − 1 ] +  
 F3[ 0 ][ 1 ] \* pY[ SubWidthC \* x − 1 ][ 0 ] +  
 F3[ 1 ][ 1 ] \* pY[ SubWidthC \* x ][ 0 ] + (8‑168)  
 F3[ 2 ][ 1 ] \* pY[ SubWidthC \* x + 1 ][ 0 ] +  
 F3[ 1 ][ 2 ] \* pY[ SubWidthC \* x ][ 1 ] + 4 ) >> 3

* + - * + Otherwise (availT is equal to FALSE), pDsY[ x ][ 0 ] with x = 1..nTbW − 1 is derived:

pDsY[ x ][ 0 ] = ( F2[ 0 ] \* pY[ SubWidthC \* x − 1 ][ 0 ] +  
 F2[ 1 ] \* pY[ SubWidthC \* x ][ 0 ] + (8‑169)  
 F2[ 2 ] \* pY[ SubWidthC \* x + 1 ][ 0 ] + 2 ) >> 2

* + - * + If availL is equal to TRUE and availT is equal to TRUE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( F3[ 1 ][ 0 ] \* pY[ 0 ][ − 1 ] +  
 F3[ 0 ][ 1 ] \* pY[ − 1 ][ 0 ] +  
 F3[ 1 ][ 1 ] \* pY[ 0 ][ 0 ] + (8‑170)  
 F3[ 2 ][ 1 ] \* pY[ 1 ][ 0 ] +  
 F3[ 1 ][ 2 ] \* pY[ 0 ][ 1 ] + 4 ) >> 3

* + - * + Otherwise if availL is equal to TRUE and availT is equal to FALSE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( F2[ 0 ] \* pY[ − 1 ][ 0 ] + F2[ 1 ] \* pY[ 0 ][ 0 ] +  
 F2[ 2 ] \* pY[ 1 ][ 0 ] + 2 ) >> 2 (8‑171)

* + - * + Otherwise if availL is equal to FALSE and availT is equal to TRUE, pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = ( F2[ 0 ] \* pY[ 0 ][ − 1 ] + F2[ 1 ] \* pY[ 0 ][ 0 ] +  
 F2[ 2 ] \* pY[ 0 ][ 1 ] + 2 ) >> 2 (8‑172)

* + - * + Otherwise (availL is equal to FALSE and availT is equal to FALSE), pDsY[ 0 ][ 0 ] is derived as follows:

pDsY[ 0 ][ 0 ] = pY[ 0 ][ 0 ] (8‑173)

* + - * Otherwise (sps\_cclm\_colocated\_chroma\_flag is equal to 0), the following applies:
        + pDsY[ x ][ y ] with x = 1..nTbW − 1, y = 0..nTbH − 1 is derived as follows:

pDsY[ x ][ y ] = ( F4[ 0 ][ 1 ] \* pY[ SubWidthC \* x − 1 ][ SubHeightC \* y ] +  
 F4[ 0 ][ 2 ] \* pY[ SubWidthC \* x − 1 ][ SubHeightC \* y + 1 ] +  
 F4[ 1 ][ 1 ] \* pY[ SubWidthC \* x ][ SubHeightC \* y ] + (8‑174)  
 F4[ 1 ][ 2 ] \* pY[ SubWidthC \* x ][ SubHeightC \* y + 1] +  
 F4[ 2 ][ 1 ] \* pY[ SubWidthC \* x + 1 ][ SubHeightC \* y ] +  
 F4[ 2 ][ 2 ] \* pY[ SubWidthC \* x + 1][ SubHeightC \* y + 1 ] + 4 ) >> 3

* + - * + If availL is equal to TRUE, pDsY[ 0 ][ y ] with y = 0..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( F4[ 0 ][ 1 ] \* pY[ − 1 ][ SubHeightC \* y ] +  
 F4[ 0 ][ 2 ] \* pY[ − 1 ][ SubHeightC \* y + 1 ] +  
 F4[ 1 ][ 1 ] \* pY[ 0 ][ SubHeightC \* y ] + (8‑175)  
 F4[ 1 ][ 2 ] \* pY[ 0 ][ SubHeightC \* y + 1] +  
 F4[ 2 ][ 1 ] \* pY[ 1 ][ SubHeightC \* y ] +  
 F4[ 2 ][ 2 ] \* pY[ 1 ][ SubHeightC \* y + 1 ] + 4 ) >> 3

* + - * + Otherwise (availL is equal to FALSE), pDsY[ 0 ][ y ] with y = 0..nTbH − 1 is derived as follows:

pDsY[ 0 ][ y ] = ( F1[ 0 ] \* pY[ 0 ][ SubHeightC \* y ]  +  
 F1[ 1 ] \* pY[ 0 ][ SubHeightC \* y + 1] + 1 ) >> 1 (8‑176)

* 1. When numSampL is greater than 0, the selected neighbouring left chroma samples pSelC[ idx ] are set equal to p[ −1 ][ pickPosL[ idx ] ] with idx = 0..cntL − 1, and the selected down-sampled neighbouring left luma samples pSelDsY[ idx ] with idx = 0..cntL − 1 are derived as follows:
     + The variable y is set equal to pickPosL[ idx ].
     + If both SubWidthC and SubHeightC are equal to 1, the following applies:

pSelDsY[ idx ] = pY[ − 1][ y ] (8‑177)

* + - Otherwise the following applies:
      * If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
        + If y is greater than 0 or availTL is equal to TRUE, pSelDsY[ idx ]  is derived as follows:

pSelDsY[ idx ] = ( F3[ 1 ][ 0 ] \* pY[ − SubWidthC ][ SubHeightC \* y − 1 ] +  
 F3[ 0 ][ 1 ] \* pY[ −1 − SubWidthC ][ SubHeightC \* y ] +  
 F3[ 1 ][ 1 ] \* pY[ −SubWidthC ][ SubHeightC \* y ]  + (8‑178)  
 F3[ 2 ][ 1 ] \* pY[ 1 − SubWidthC ][ SubHeightC \* y ] +  
 F3[ 1 ][ 2 ] \* pY[ −SubWidthC ][ SubHeightC \* y + 1 ] + 4 ) >> 3

* + - * + Otherwise (y is equal to 0), pSelDsY[ idx ]  is derived as follows:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ −1 − SubWidthC ][ 0 ] +  
 F2[ 1 ] \* pY[ −SubWidthC ][ 0 ] + (8‑179)  
 F2[ 2 ] \* pY[ 1 − SubWidthC ][ 0] + 2 ) >> 2

* + - * Otherwise (sps\_cclm\_colocated\_chroma\_flag is equal to 0), the following applies:

pSelDsY[ idx ] = ( F4[ 0 ][ 1 ] \* pY[ −1 − SubWidthC ][ SubHeightC \* y ] +  
 F4[ 0 ][ 2 ] \* pY[ −1 − SubWidthC ][ SubHeightC \* y + 1 ] +  
 F4[ 1 ][ 1 ] \* pY[ −SubWidthC ][ SubHeightC \* y ] + (8‑180)  
 F4[ 1 ][ 2 ] \* pY[ −SubWidthC ][ SubHeightC \* y + 1] +  
 F4[ 2 ][ 1 ] \* pY[ 1 − SubWidthC ][ SubHeightC \* y ] +  
 F4[ 2 ][ 2 ] \* pY[ 1 − SubWidthC][ SubHeightC \* y + 1 ] + 4 ) >> 3

* 1. When numSampT is greater than 0, the selected neighbouring top chroma samples pSelC[ idx ] are set equal to p[ pickPosT[ idx − cntL ] ][ -1 ] with idx = cntL..cntL + cntT − 1, and the down-sampled neighbouring top luma samples pSelDsY[ idx ] with idx = 0..cntL + cntT − 1 are specified as follows:
     + The variable x is set equal to pickPosT[ idx − cntL ].
     + If both SubWidthC and SubHeightC are equal to 1, the following applies:

pSelDsY[ idx ] = pY[ x ][ − 1] (8‑181)

* + - Otherwise, the following applies:
      * If sps\_cclm\_colocated\_chroma\_flag is equal to 1, the following applies:
        + If x is greater than 0, the following applies:

If bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F3[ 1 ][ 0 ] \* pY[ SubWidthC \* x ][ − 1 − SubHeightC ] +  
 F3[ 0 ][ 1 ] \* pY[ SubWidthC \* x − 1 ][ −SubHeightC ] +  
 F3[ 1 ][ 1 ] \* pY[ SubWidthC \* x ][ −SubHeightC] + (8‑182)  
 F3[ 2 ][ 1 ] \* pY[ SubWidthC \* x + 1 ][ −SubHeightC] +  
 F3[ 1 ][ 2 ] \* pY[ SubWidthC \* x ][ 1 − SubHeightC ] + 4 ) >> 3

Otherwise (bCTUboundary is equal to TRUE), the following applies:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ SubWidthC \* x − 1 ][ −1 ] +  
 F2[ 1 ] \* pY[ SubWidthC \* x ][ −1 ]  + (8‑183)  
 F2[ 2 ] \* pY[ SubWidthC \* x + 1 ][ −1 ] + 2 ) >> 2

* + - * + Otherwise (x is equal to 0), the following applies:

If availTL is equal to TRUE and bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F3[ 1 ][ 0 ] \* pY[ −1 ][ − 1 − SubHeightC ] +  
 F3[ 0 ][ 1 ] \* pY[ −1 ][ −SubHeightC ] +  
 F3[ 1 ][ 1 ] \* pY[ 0 ][ −SubHeightC ] + (8‑184)  
 F3[ 2 ][ 1 ] \* pY[ 1 ][ −SubHeightC ] +  
 F3[ 1 ][ 2 ] \* pY[ −1 ][ 1 − SubHeightC ] + 4 ) >> 3

Otherwise if availTL is equal to TRUE and bCTUboundary is equal to TRUE, the following applies:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ −1 ][ −1 ] + F2[ 1 ] \* pY[ 0 ][ −1 ]  +  
 F2[ 2 ] \* pY[ 1 ][ −1 ] + 2 ) >> 2 (8‑185)

Otherwise if availTL is equal to FALSE and bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ 0 ][ −1 ] + F2[ 1 ] \* pY[ 0 ][ −2 ] +  
 F2[ 2 ] \* pY[ 0 ][ −1 ] + 2 ) >> 2 (8‑186)

Otherwise (availTL is equal to FALSE and bCTUboundary is equal to TRUE), the following applies:

pSelDsY[ idx ] = pY[ 0 ][ −1 ] (8‑187)

* + - * Otherwise (sps\_cclm\_colocated\_chroma\_flag is equal to 0), the following applies:
        + If x is greater than 0, the following applies:

If bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F4[ 0 ][ 1 ] \* pY[ SubWidthC x − 1 ][ −2 ] +  
 F4[ 0 ][ 2 ] \* pY[ SubWidthC \* x − 1 ][ −1 ] +  
 F4[ 1 ][ 1 ] \* pY[ SubWidthC \* x ][ −2 ] + (8‑188)  
 F4[ 1 ][ 2 ] \* pY[ SubWidthC \* x ][ −1] +  
 F4[ 2 ][ 1 ] \* pY[ SubWidthC \* x + 1 ][ −2 ] +  
 F4[ 2 ][ 2 ] \* pY[ SubWidthC \* x + 1 ][ −1 ] + 4 ) >> 3

Otherwise (bCTUboundary is equal to TRUE), the following applies:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ SubWidthC \* x − 1 ][ −1 ] +  
 F2[ 1 ] \* pY[ SubWidthC \* x ][ −1 ] + (8‑189)  
 F2[ 2 ] \* pY[ SubWidthC \* x + 1][ −1 ] + 2 ) >> 2

* + - * + Otherwise (x is equal to 0), the following applies:

If availTL is equal to TRUE and bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F4[ 0 ][ 1 ] \* pY[ −1 ][ −2 ] + F4[ 0 ][ 2 ] \* pY[ −1 ][ −1 ] +  
 F4[ 1 ][ 1 ] \* pY[ 0 ][ −2 ] + F4[ 1 ][ 2 ] \* pY[ 0 ][ −1 ] + (8‑190) F4[ 2 ][ 1 ] \* pY[ 1 ][ −2 ] + F4[ 2 ][ 2 ] \* pY[ 1 ][ −1 ] + 4 ) >> 3

Otherwise if availTL is equal to TRUE and bCTUboundary is equal to TRUE, the following applies:

pSelDsY[ idx ] = ( F2[ 0 ] \* pY[ −1 ][ −1 ] + F2[ 1 ] \* pY[ 0 ][ −1 ] +  
 F2[ 2 ] \* pY[ 1 ][ − 1 ] + 2 ) >> 2 (8‑191)

Otherwise if availTL is equal to FALSE and bCTUboundary is equal to FALSE, the following applies:

pSelDsY[ idx ] = ( F1[ 1 ] \* pY[ 0 ][ −2 ] + F1[ 0 ] \* pY[ 0 ][ −1 ] + 1 ) >> 1 (8‑192)

Otherwise (availTL is equal to FALSE and bCTUboundary is equal to TRUE), the following applies:

pSelDsY[ idx ] = pY[ 0 ][ −1 ] (8‑193)

* 1. When cntT+ cntL is not equal to 0, the variables minY, maxY, minC and maxC are derived as follows:
     + When cntT + cntL is equal to 2, pSelComp[ 3 ] is set equal to pSelComp[ 0 ], pSelComp[ 2 ] is set equal to pSelComp[ 1 ], pSelComp[ 0 ] is set equal to pSelComp[ 1 ], and pSelComp[ 1 ] is set equal to pSelComp[ 3 ], with Comp being replaced by DsY and C.
     + The arrays minGrpIdx and maxGrpIdx are derived as follows:

minGrpIdx[ 0 ] = 0 (8‑194)

minGrpIdx[ 1 ] = 2 (8‑195)

maxGrpIdx[ 0 ] = 1 (8‑196)

maxGrpIdx[ 1 ] = 3 (8‑197)

* + - When pSelDsY[ minGrpIdx[ 0 ] ] is greater than pSelDsY[ minGrpIdx[ 1 ] ], minGrpIdx[ 0 ] and minGrpIdx[ 1 ] are swapped as follows:

( minGrpIdx[ 0 ], minGrpIdx[ 1 ] ) = Swap( minGrpIdx[ 0 ], minGrpIdx[ 1 ] ) (8‑198)

* + - When pSelDsY[ maxGrpIdx[ 0 ] ] is greater than pSelDsY[ maxGrpIdx[ 1 ] ], maxGrpIdx[ 0 ] and maxGrpIdx[ 1 ] are swapped as follows:

( maxGrpIdx[ 0 ], maxGrpIdx[ 1 ] ) = Swap( maxGrpIdx[ 0 ], maxGrpIdx[ 1 ] ) (8‑199)

* + - When pSelDsY[ minGrpIdx[ 0 ] ] is greater than pSelDsY[ maxGrpIdx[ 1 ] ], arrays minGrpIdx and maxGrpIdx are swapped as follows:

( minGrpIdx, maxGrpIdx ) = Swap( minGrpIdx, maxGrpIdx ) (8‑200)

* + - When pSelDsY[ minGrpIdx[ 1 ] ] is greater than pSelDsY[ maxGrpIdx[ 0 ] ], minGrpIdx[ 1 ] and maxGrpIdx[ 0 ] are swapped as follows:

( minGrpIdx[ 1 ], maxGrpIdx[ 0 ] ) = Swap( minGrpIdx[ 1 ], maxGrpIdx[ 0 ] ) (8‑201)

* + - The variables maxY, maxC, minY and minC are derived as follows:

maxY = ( pSelDsY[ maxGrpIdx[ 0 ] ] + pSelDsY[ maxGrpIdx[ 1 ] ] + 1 )  >>  1 (8‑202)

maxC = ( pSelC[ maxGrpIdx[ 0 ] ] + pSelC[ maxGrpIdx[ 1 ] ] + 1 )  >>  1 (8‑203)

minY = ( pSelDsY[ minGrpIdx[ 0 ] ] + pSelDsY[ minGrpIdx[ 1 ] ] + 1 )  >>  1 (8‑204)

minC = ( pSelC[ minGrpIdx[ 0 ] ] + pSelC[ minGrpIdx[ 1 ] ] + 1 )  >>  1 (8‑205)

* 1. The variables a, b, and k are derived as follows:
     + If numSampL is equal to 0, and numSampT is equal to 0, the following applies:

k = 0 (8‑206)

a = 0 (8‑207)

b = 1 << ( BitDepthC − 1) (8‑208)

* + - Otherwise, the following applies:

diff = maxY − minY (8‑209)

* + - If diff is not equal to 0, the following applies:

diffC = maxC − minC (8‑210)

x = Floor( Log2( diff ) ) (8‑211)

normDiff = ( ( diff << 4 ) >> x ) & 15 (8‑212)

x += ( normDiff  !=  0 ) ? 1 : 0 (8‑213)

y = Floor( Log2( Abs ( diffC ) ) ) + 1 (8‑214)

a = ( diffC \* ( divSigTable[ normDiff ] | 8 ) + 2y − 1 ) >> y (8‑215)

k = ( ( 3 + x − y ) < 1 ) ? 1 : 3 + x − y (8‑216)

a = ( ( 3 + x − y ) < 1 ) ? Sign( a ) \* 15 : a (8‑217)

b = minC − ( ( a \* minY ) >> k ) (8‑218)

where divSigTable[ ] is specified as follows:

divSigTable[ ] = { 0, 7, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1, 1, 1, 0 } (8‑219)

* + - Otherwise (diff is equal to 0), the following applies:

k = 0 (8‑220)

a = 0 (8‑221)

b = minC (8‑222)

* 1. The prediction samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0.. nTbH − 1 are derived as follows:

predSamples[ x ][ y ] = Clip1C( ( ( pDsY[ x ][ y ] \* a ) >> k ) + b ) (8‑223)

##### Position-dependent intra prediction sample filtering process

Inputs to this process are:

* the intra prediction mode predModeIntra,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable refW specifying the reference samples width,
* a variable refH specifying the reference samples height,
* the predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y = 0..nTbH − 1,
* the neighbouring samples p[ x ][ y ], with x = −1, y = −1..refH − 1 and x = 0..refW − 1, y = −1,
* a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified predicted samples predSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1.

Depending on the value of cIdx, the function clip1Cmp is set as follows:

* If cIdx is equal to 0, clip1Cmp is set equal to Clip1Y.
* Otherwise, clip1Cmp is set equal to Clip1C.

The variable nScale is set to ( ( Log2( nTbW ) + Log2( nTbH ) − 2 )  >>  2 ).

The reference sample arrays mainRef[ x ] and sideRef[ y ], with x = 0..refW − 1 and y = 0..refH − 1 are derived as follows:

mainRef[ x ] = p[ x ][ −1 ] (8‑224)  
sideRef[ y ] = p[ −1 ][ y ]

The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

* If predModeIntra is equal to INTRA\_PLANAR or INTRA\_DC, the following applies:

refL[ x ][ y ] = p[ −1 ][ y ] (8‑225)

refT[ x ][ y ] = p[ x ][ −1 ] (8‑226)

wT[ y ] = 32  >>  ( ( y  <<  1 )  >>  nScale ) (8‑227)

wL[ x ] = 32  >>  ( ( x  <<  1 )  >>  nScale ) (8‑228)

wTL[ x ][ y ] = ( predModeIntra  = =  INTRA\_DC )  ?  ( ( wL[ x ]>>  4 ) + ( wT[ y ]>>  4 ) )  :  0 (8‑229)

* Otherwise, if predModeIntra is equal to INTRA\_ANGULAR18 or INTRA\_ANGULAR50, the following applies:

refL[ x ][ y ] = p[ −1 ][ y ] (8‑230)

refT[ x ][ y ] = p[ x ][ −1 ] (8‑231)

wT[ y ] = ( predModeIntra  = = INTRA\_ANGULAR18 ) ? 32  >>  ( ( y  <<  1 )  >>  nScale ) : 0 (8‑232)

wL[ x ] = ( predModeIntra  = = INTRA\_ANGULAR50 ) ? 32  >>  ( ( x  <<  1 )  >>  nScale ) : 0 (8‑233)

wTL[ x ][ y ] = ( predModeIntra  = = INTRA\_ANGULAR18 ) ? wT[ y ] : wL[ x ] (8‑234)

* Otherwise, if predModeIntra is equal to INTRA\_ANGULAR2 or INTRA\_ANGULAR66, the following applies:

refL[ x ][ y ] = p[ −1 ][ x + y + 1 ] (8‑235)

refT[ x ][ y ] = p[ x + y + 1 ][ −1 ] (8‑236)

wT[ y ] = ( 32  >>  1 )  >>  ( ( y  <<  1 )  >>  nScale ) (8‑237)

wL[ x ] = ( 32  >>  1 )  >>  ( ( x  <<  1 )  >>  nScale ) (8‑238)

wTL[ x ][ y ] = 0 (8‑239)

* Otherwise, if predModeIntra is less than or equal to INTRA\_ANGULAR10, the following ordered steps apply:

1. The variables dXPos[ y ], dXFrac[ y ], dXInt[ y ] and dX[ x ][ y ] are derived as follows using invAngle as specified in clause 8.4.5.2.12 depending on intraPredMode:

dXPos[ y ] = ( ( y + 1 ) \* invAngle + 2 )  >>  2  
dXFrac[ y ] = dXPos[ y ] & 63 (8‑240)  
dXInt[ y ] = dXPos [ y ]  >>  6  
dX[ x ][ y ] = x + dXInt[ y ]

1. The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are derived as follows:

refL[ x ][ y ] = 0 (8‑241)

refT[ x ][ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  mainRef[ dX[ x ][ y ] + ( dXFrac[ y ]  >>  5 ) ]  :  0 (8‑242)

wT[ y ] = ( dX[ x ][ y ] < refW − 1 )  ?  32  >>  ( ( y  <<  1 )  >>  nScale )  :  0 (8‑243)

wL[ x ] = 0 (8‑244)

wTL[ x ][ y ] = 0 (8‑245)

* Otherwise, if predModeIntra is greater than or equal to INTRA\_ANGULAR58, the following ordered steps apply:

1. The variables dYPos[ x ], dYFrac[ x ], dYInt[ x ] and dY[ x ][ y ] are derived as follows using invAngle as specified in clause 8.4.5.2.12 depending on intraPredMode:

dYPos[ x ] = ( ( x + 1 ) \* invAngle + 2 )  >>  2  
dYFrac[ x ] = dYPos[ x ] & 63 (8‑246)  
dYInt[ x ] = dYPos[ x ]  >>  6  
dY[ x ][ y ] = y + dYInt[ x ]

1. The variables refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are derived as follows:

refL[ x ][ y ] = ( dY[ x ][ y ] < refH − 1 )  ?  sideRef[ dY[ x ][ y ] + ( dYFrac[ x ]  >>  5 ) ]  :  0 (8‑247)

refT[ x ][ y ] = 0 (8‑248)

wT[ y ] = 0 (8‑249)

wL[ x ] = ( dY[ x ][ y ] < refH − 1 )  ?  32  >>  ( ( x  <<  1 )  >>  nScale )  :  0 (8‑250)

wTL[ x ][ y ] = 0 (8‑251)

* Otherwise, refL[ x ][ y ], refT[ x ][ y ], wT[ y ], wL[ x ] and wTL[ x ][ y ] are all set equal to 0.

The values of the modified predicted samples predSamples[ x ][ y ], with x = 0..nTbW − 1, y =0..nTbH − 1 are derived as follows:

predSamples[ x ][ y ] = clip1Cmp( (  refL[ x ][ y ] \* wL[ x ] + refT[ x ][ y ] \* wT[ y ] −   
 p[ −1 ][ −1 ] \* wTL[ x ][ y ] + (8‑252) ( 64 − wL[ x ] − wT[ y ] + wTL[ x ][ y ] ) \* predSamples[ x ][ y ] + 32  )  
 >> 6 )

## Decoding process for coding units coded in inter prediction mode

### General decoding process for coding units coded in inter prediction mode

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

Output of this process is a modified reconstructed picture before in-loop filtering.

The derivation process for quantization parameters as specified in clause 8.7.1 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight, and the variable treeType as inputs.

The decoding process for coding units coded in inter prediction mode consists of the following ordered steps:

1. The variable dmvrFlag is set equal to 0.
2. The motion vector components and reference indices of the current coding unit are derived as follows:

* If MergeTriangleFlag[ xCb ][ yCb ], inter\_affine\_flag[ xCb ][ yCb ] and merge\_subblock\_flag[ xCb ][ yCb ] are all equal to 0, the following applies:
* The derivation process for motion vector components and reference indices as specified in clause 8.5.2.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the luma motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], the reference indices refIdxL0 and refIdxL1 and the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], and the bi-prediction weight index bcwIdx as outputs.
* When all of the following conditions are true, dmvrFlag is set equal to 1:
* sps\_dmvr\_enabled\_flag is equal to 1
* general\_merge\_flag[ xCb ][ yCb ] is equal to 1
* both predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] are equal to 1
* mmvd\_merge\_flag[ xCb ][ yCb ] is equal to 0
* DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ]) is equal to DiffPicOrderCnt( RefPicList[ 1 ][ refIdxL1 ], currPic )
* BcwIdx[ xCb ][ yCb ] is equal to 0
* Both luma\_weight\_l0\_flag[ refIdxL0 ] and luma\_weight\_l1\_flag[ refIdxL1 ] are equal to 0
* cbWidth is greater than or equal to 8
* cbHeight is greater than or equal to 8
* cbHeight\*cbWidth is greater than or equal to 128
* If dmvrFlag is equal to 1, the following applies:
* For X being 0 and 1, 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 clause 8.5.6.2 with X and refIdxLX as inputs.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the subblock width sbWidth and the subblock height sbHeight are derived as follows:

numSbX = ( cbWidth > 16 )  ?  ( cbWidth >> 4 ) : 1 (8‑253)

numSbY = ( cbHeight > 16 )  ?  ( cbHeight >> 4 ) : 1 (8‑254)

sbWidth = ( cbWidth > 16 )  ?  16 : cbWidth (8‑255)

sbHeight = ( cbHeight > 16 )  ?  16 : cbHeight (8‑256)

* For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:
* The luma motion vectors mvLX[ xSbIdx ][ ySbIdx ] and the prediction list utilization flags predFlagLX[ xSbIdx ][ ySbIdx ] with X equal to 0 and 1, and the luma location ( xSb[xSbIdx][ySbIdx], ySb[xSbIdx][ySbIdx] ) specifying the top-left sample of the coding subblock relative to the top‑left luma sample of the current picture are derived as follows:

mvLX[ xSbIdx ][ ySbIdx ] = mvLX[ 0 ][ 0 ] (8‑257)

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑258)

xSb[ xSbIdx ][ ySbIdx ] =  xCb + xSbIdx \* sbWidth (8‑259)

ySb[ xSbIdx ][ ySbIdx ] =  yCb + ySbIdx \* sbHeight (8‑260)

* The decoder side motion vector refimenent process specified in clause  8.5.3.1 is invoked with xSb[ xSbIdx ][ ySbIdx ], ySb[ xSbIdx ][ ySbIdx ], sbWidth, sbHeight, the motion vectors mvLX[ xSbIdx ][ ySbIdx ] and the reference picture array refPicLXL as inputs and delta motion vectors dMvLX[ xSbIdx ][ ySbIdx ] as outputs with X equal to 0 and 1.
* When ChromaArrayType is not equal to 0, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and mvCLX[ xSbIdx ][ ySbIdx ] as outputs with X equal to 0 and 1.
* Otherwise (dmvrFlag is equal to 0), the following applies:
* When ChromaArrayType is not equal to 0, and treeType is equal to SINGLE\_TREE, and predFlagLX[ 0 ][0 ], with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ 0 ][ 0 ] and refIdxLX as inputs, and mvCLX[ 0 ][ 0 ] as output.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are both set equal to 1.
* Otherwise, if MergeTriangleFlag[ xCb ][ yCb ] is equal to 1, inter\_affine\_flag[ xCb ][ yCb ] and merge\_subblock\_flag[ xCb ][ yCb ] are both equal to 0, the derivation process for triangle motion vector components and reference indices as specified in clause 8.5.4.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the luma motion vectors mvA and mvB, the chroma motion vectors mvCA and mvCB, the reference indices refIdxA and refIdxB and the prediction list flags predListFlagA and predListFlagB as outputs.
* Otherwise (inter\_affine\_flag[ xCb ][ yCb ] or merge\_subblock\_flag[ xCb ][ yCb ] is equal to 1), the derivation process for subblock motion vector components and reference indices as specified in clause 8.5.5.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight as inputs, and the reference indices refIdxL0 and refIdxL1, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the prediction list utilization flags predFlagLX[ xSbIdx ][ ySbIdx ], the luma motion vector array mvLX[ xSbIdx ][ ySbIdx ], and the chroma motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..(cbWidth >> 2) − 1, and ySbIdx = 0..( cbHeight >> 2 ) − 1, and with X being 0 or 1, and the bi-prediction weight index bcwIdx as outputs.

1. The arrays of luma and chroma motion vectors after decoder side motion vector refinement, refMvLX[ xSbIdx ][ ySbIdx ] and refMvCLX[ xSbIdx ][ ySbIdx ], with X being 0 and 1, are derived as follows for xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1:

* If dmvrFlag is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with refMvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and refMvCLX[ xSbIdx ][ ySbIdx ] as output and the input refMvLX[ xSbIdx ][ ySbIdx ] is derived as follows;

refMvLX[ xSbIdx ][ ySbIdx ] = mvLX[ xSbIdx ][ ySbIdx ] + dMvLX[ xSbIdx ][ ySbIdx ] (8‑261)

refMvLX[ xSbIdx ][ ySbIdx ][ 0 ] = Clip3( −217, 217 − 1, refMvLX[ xSbIdx ][ ySbIdx ][ 0 ] ) (8‑262)

refMvLX[ xSbIdx ][ ySbIdx ][ 1 ] = Clip3( −217, 217− 1, refMvLX[ xSbIdx ][ ySbIdx ][ 1 ] ) (8‑263)

* Otherwise (dmvrFlag is equal to 0), the following applies:

refMvLX[ xSbIdx ][ ySbIdx ] = mvLX[ xSbIdx ][ ySbIdx ] (8‑264)

refMvCLX [ xSbIdx ][ ySbIdx ] = mvCLX[ xSbIdx ][ ySbIdx ] (8‑265)

NOTE – The array refMvLX is stored in MvDmvrLX and used in the derivation process for collocated motion vectors in clause 8.5.2.12. The array of non-refine luma motion vectors MvLX is used in the spatial motion vector prediction and deblocking boundary strength derivation processes.

1. The prediction samples of the current coding unit are derived as follows:

* If MergeTriangleFlag[ xCb ][ yCb ] is equal to 0, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process for inter blocks as specified in clause 8.5.6.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ], and the refined luma motion vectors refMvL0[ xSbIdx ][ ySbIdx ] and refMvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the bi-prediction weight index bcwIdx, and the variable cIdx set equal to 0 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples as outputs.
      * When ChromaArrayType is not equal to 0. the decoding process for inter blocks as specified in clause 8.5.6.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], and the refined chroma motion vectors refMvCL0[ xSbIdx ][ ySbIdx ] and refMvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the bi-prediction weight index bcwIdx, and the variable cIdx set equal to 1 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCb of prediction chroma samples for the chroma components Cb as outputs.
      * When ChromaArrayType is not equal to 0. the decoding process for inter blocks as specified in clause 8.5.6.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], and the refined chroma motion vectors refMvCL0[ xSbIdx ][ ySbIdx ] and refMvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the bi-prediction weight index bcwIdx, and the variable cIdx set equal to 2 as inputs, and the inter prediction samples (predSamples) that are an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCr of prediction chroma samples for the chroma components Cr as outputs.
* Otherwise (MergeTriangleFlag[ xCb ][ yCb ] is equal to 1), the decoding process for triangular inter blocks as specified in clause 8.5.7.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the luma motion vectors mvA and mvB, the chroma motion vectors mvCA and mvCB, the reference indices refIdxA and refIdxB, and the prediction list flags predListFlagA and predListFlagB as inputs, and the inter prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples and two (cbWidth / SubWidthC)x(cbHeight / SubHeightC) arrays predSamplesCb and predSamplesCr of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

1. The variables NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ] are set equal to numSbX and numSbY, respectively.
2. The residual samples of the current coding unit are derived as follows:

* The decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the luma location ( xCb, yCb ), the width nTbW set equal to the luma coding block width cbWidth, the height nTbH set equal to the luma coding block height cbHeight and the variable cIdx set equal to 0 as inputs, and the array resSamplesL as output.
* When ChromaArrayType is not equal to 0. the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC, yCb / SubHeightC ), the width nTbW set equal to the chroma coding block width cbWidth / SubWidthC, the height nTbH set equal to the chroma coding block height cbHeight / SubHeightC and the variable cIdx set equal to 1 as inputs, and the array resSamplesCb as output.
* When ChromaArrayType is not equal to 0. the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC, yCb / SubHeightC ), the width nTbW set equal to the chroma coding block width cbWidth / SubWidthC, the height nTbH set equal to the chroma coding block height cbHeight / SubHeightC and the variable cIdx set equal to 2 as inputs, and the array resSamplesCr as output.

1. The reconstructed samples of the current coding unit are derived as follows:

* The picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb, yCb ), the block width bWidth set equal to cbWidth, the block height bHeight set equal to cbHeight, the variable cIdx set equal to 0, the (cbWidth)x(cbHeight) array predSamples set equal to predSamplesL and the (cbWidth)x(cbHeight) array resSamples set equal to resSamplesL as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* When ChromaArrayType is not equal to 0. the picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ), the block width bWidth set equal to cbWidth / SubWidthC, the block height bHeight set equal to cbHeight / SubHeightC, the variable cIdx set equal to 1, the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamples set equal to predSamplesCb and the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array resSamples set equal to resSamplesCb as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* When ChromaArrayType is not equal to 0. the picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ), the block width bWidth set equal to cbWidth / SubWidthC, the block height bHeight set equal to cbHeight / SubHeightC, the variable cIdx set equal to 2, the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamples set equal to predSamplesCr and the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array resSamples set equal to resSamplesCr as inputs, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for motion vector components and reference indices

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vectors in 1/16 fractional-sample accuracy mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ],
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ],
* the bi-prediction weight index bcwIdx.

Let the variable LX be RefPicList[ X ], with X being 0 or 1, of the current picture.

For the derivation of the variables mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], refIdxL0 and refIdxL1, as well as predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], the following applies:

* If general\_merge\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for luma motion vectors for merge mode as specified in clause 8.5.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the luma motion vectors mvL0[ 0 ][ 0 ], mvL1[ 0 ][ 0 ], the reference indices refIdxL0, refIdxL1, the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], the bi-prediction weight index bcwIdx and the merging candidate list mergeCandList.
* Otherwise, the following applies:
* For X being replaced by either 0 or 1 in the variables predFlagLX[ 0 ][0 ], mvLX[ 0 ][0 ] and refIdxLX, in PRED\_LX, and in the syntax elements ref\_idx\_lX and MvdLX, the following ordered steps apply:

1. The variables refIdxLX and predFlagLX[ 0 ][0 ] are derived as follows:

* If inter\_pred\_idc[ xCb ][ yCb ] is equal to PRED\_LX or PRED\_BI,

refIdxLX = ref\_idx\_lX[ xCb ][ yCb ] (8‑266)

predFlagLX[ 0 ][0 ] = 1 (8‑267)

* Otherwise, the variables refIdxLX and predFlagLX[ 0 ][0 ] are specified by:

refIdxLX = −1 (8‑268)

predFlagLX[ 0 ][0 ] = 0 (8‑269)

1. The variable mvdLX is derived as follows:

mvdLX[ 0 ] = MvdLX[ xCb ][ yCb ][ 0 ] (8‑270)

mvdLX[ 1 ] = MvdLX[ xCb ][ yCb ][ 1 ] (8‑271)

1. When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for luma motion vector prediction in clause 8.5.2.8 is invoked with the luma coding block location ( xCb, yCb ), the coding block width cbWidth, the coding block height cbHeight and the variable refIdxLX as inputs, and the output being mvpLX.
2. When predFlagLX[ 0 ][ 0 ] is equal to 1, the luma motion vector mvLX[ 0 ][ 0 ] is derived as follows:

uLX[ 0 ] = ( mvpLX[ 0 ] + mvdLX[ 0 ] + 218 ) % 218 (8‑272)

mvLX[ 0 ][ 0 ][ 0 ] = ( uLX[ 0 ] >= 217 ) ? ( uLX[ 0 ] − 218 ) : uLX[ 0 ] (8‑273)

uLX[ 1 ] = ( mvpLX[ 1 ] + mvdLX[ 1 ] + 218 ) % 218 (8‑274)

mvLX[ 0 ][ 0 ][ 1 ] = ( uLX[ 1 ] >= 217 ) ? ( uLX[ 1 ] − 218 ) : uLX[ 1 ] (8‑275)

NOTE 1– The resulting values of mvLX[ 0 ][ 0 ][ 0 ] and mvLX[ 0 ][ 0 ][ 1 ] as specified above will always be in the range of −217 to 217 − 1, inclusive.

* The bi-prediction weight index bcwIdx is set equal to bcw\_idx[ xCb ][ yCb ].

When all of the following conditions are true, refIdxL1 is set equal to −1, predFlagL1 is set equal to 0, and bcwIdx is set equal to 0:

* predFlagL0[ 0 ][ 0 ] is equal to 1.
* predFlagL1[ 0 ][ 0 ] is equal to 1.
* The value of ( cbWidth + cbHeight ) is equal to 12.

The updating process for the history-based motion vector predictor list as specified in clause 8.5.2.16 is invoked with luma motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], reference indices refIdxL0 and refIdxL1, prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], and bi-prediction weight index bcwIdx.

#### Derivation process for luma motion vectors for merge mode

This process is only invoked when general\_merge\_flag[ xCb ][ yCb ] is equal to 1, where ( xCb, yCb ) specify the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vectors in 1/16 fractional-sample accuracy mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ],
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ],
* the bi-prediction weight index bcwIdx.
* the merging candidate list mergeCandList.

The bi-prediction weight index bcwIdx is set equal to 0.

The motion vectors mvL0[ 0 ][ 0 ] and mvL1[ 0 ][ 0 ], the reference indices refIdxL0 and refIdxL1 and the prediction utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] are derived by the following ordered steps:

1. The derivation process for spatial merging candidates from neighbouring coding units as specified in clause 8.5.2.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, and the luma coding block height cbHeight as inputs, and the output being the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2, the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1 and refIdxLXB2, the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1 and predFlagLXB2, and the motion vectors mvLXA0, mvLXA1, mvLXB0, mvLXB1 and mvLXB2, with X being 0 or 1, and the bi-prediction weight indices bcwIdxA0,bcwIdxA1, bcwIdxB0,bcwIdxB1, bcwIdxB2.
2. The reference indices, refIdxLXCol, with X being 0 or 1, and the bi-prediction weight index bcwIdxCol for the temporal merging candidate Col are set equal to 0.
3. The derivation process for temporal luma motion vector prediction as specified in in clause 8.5.2.11 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL0Col as inputs, and the output being the availability flag availableFlagL0Col and the temporal motion vector mvL0Col. The variables availableFlagCol, predFlagL0Col and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col (8‑276)

predFlagL0Col = availableFlagL0Col (8‑277)

predFlagL1Col = 0 (8‑278)

1. When slice\_type is equal to B, the derivation process for temporal luma motion vector prediction as specified in clause 8.5.2.11 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and the variable refIdxL1Col as inputs, and the output being the availability flag availableFlagL1Col and the temporal motion vector mvL1Col. The variables availableFlagCol and predFlagL1Col are derived as follows:

availableFlagCol = availableFlagL0Col  | |  availableFlagL1Col (8‑279)

predFlagL1Col = availableFlagL1Col (8‑280)

1. The merging candidate list, mergeCandList, is constructed as follows:

i = 0  
if( availableFlagA1 )  
 mergeCandList[ i++ ] = A1  
if( availableFlagB1 )  
 mergeCandList[ i++ ] = B1if( availableFlagB0 )  
 mergeCandList[ i++ ] = B0 (8‑281)if( availableFlagA0 )  
 mergeCandList[ i++ ] = A0if( availableFlagB2 )  
 mergeCandList[ i++ ] = B2if( availableFlagCol )  
 mergeCandList[ i++ ] = Col

1. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the mergeCandList.
2. When numCurrMergeCand is less than (MaxNumMergeCand − 1) and NumHmvpCand is greater than 0, the following applies:

* The derivation process of history-based merging candidates as specified in 8.5.2.6 is invoked with mergeCandList and numCurrMergeCand as inputs, and modified mergeCandList and numCurrMergeCand as outputs.
* numOrigMergeCand is set equal to numCurrMergeCand.

1. When numCurrMergeCand is less than MaxNumMergeCand and greater than 1, the following applies:

* The derivation process for pairwise average merging candidate specified in clause 8.5.2.4 is invoked with mergeCandList, the reference indices refIdxL0N and refIdxL1N, the prediction list utilization flags predFlagL0N and predFlagL1N, the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList, and numCurrMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0avgCand and refIdxL1avgCand, the prediction list utilization flags predFlagL0avgCand and predFlagL1avgCand and the motion vectors mvL0avgCand and mvL1avgCand of candidate avgCand being added into mergeCandList. The bi-prediction weight index bcwIdx of candidate avgCand being added into mergeCandList is set equal to 0.
* numOrigMergeCand is set equal to numCurrMergeCand.

1. The derivation process for zero motion vector merging candidates specified in clause 8.5.2.5 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 in mergeCandList and numCurrMergeCand as inputs, and the output is assigned to mergeCandList, numCurrMergeCand, the reference indices refIdxL0zeroCandm and refIdxL1zeroCandm, the prediction list utilization flags predFlagL0zeroCandm and predFlagL1zeroCandm and the motion vectors mvL0zeroCandm and mvL1zeroCandm of every new candidate zeroCandm being added into mergeCandList. The bi-prediction weight index bcwIdx of every new candidate zeroCandm being added into mergeCandList is set equal to 0. The number of candidates being added, numZeroMergeCand, is set equal to ( numCurrMergeCand − numOrigMergeCand ). When numZeroMergeCand is greater than 0, m ranges from 0 to numZeroMergeCand − 1, inclusive.
2. The following assignments are made with N being the candidate at position merge\_idx[ xCb ][ yCb ] in the merging candidate list mergeCandList ( N = mergeCandList[ merge\_idx[ xCb ][ yCb ] ] ) and X being replaced by 0 or 1:

refIdxLX = refIdxLXN (8‑282)

predFlagLX[ 0 ][ 0 ] = predFlagLXN (8‑283)

mvLX[ 0 ][ 0 ][ 0 ] = mvLXN[ 0 ] (8‑284)

mvLX[ 0 ][ 0 ][ 1 ] = mvLXN[ 1 ] (8‑285)

bcwIdx = bcwIdxN (8‑286)

1. When mmvd\_merge\_flag[ xCb ][ yCb ] is equal to 1, the following applies:

* The derivation process for merge motion vector difference as specified in 8.5.2.7 is invoked with the luma location ( xCb, yCb ), the reference indices refIdxL0, refIdxL1 and the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ] as inputs, and the motion vector differences mMvdL0 and mMvdL1 as outputs.
* The motion vector difference mMvdLX is added to the merge motion vectors mvLX for X being 0 and 1 as follows:

mvLX[ 0 ][ 0 ][ 0 ] += mMvdLX[ 0 ] (8‑287)

mvLX[ 0 ][ 0 ][ 1 ] += mMvdLX[ 1 ] (8‑288)

mvLX[ 0 ][ 0 ][ 0 ] = Clip3( −217, 217 − 1, mvLX[ 0 ][ 0 ][ 0 ] ) (8‑289)

mvLX[ 0 ][ 0 ] [ 1 ] = Clip3( −217, 217− 1, mvLX[ 0 ][ 0 ][ 1 ] ) (8‑290)

#### Derivation process for spatial merging candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are as follows, with X being 0 or 1:

* the availability flags availableFlagA0, availableFlagA1, availableFlagB0, availableFlagB1 and availableFlagB2 of the neighbouring coding units,
* the reference indices refIdxLXA0, refIdxLXA1, refIdxLXB0, refIdxLXB1 and refIdxLXB2 of the neighbouring coding units,
* the prediction list utilization flags predFlagLXA0, predFlagLXA1, predFlagLXB0, predFlagLXB1 and predFlagLXB2 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvLXA0, mvLXA1, mvLXB0, mvLXB1 and mvLXB2 of the neighbouring coding units,
* the bi-prediction weight indices bcwIdxA0,bcwIdxA1, bcwIdxB0,bcwIdxB1, and bcwIdxB2.

For the derivation of availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 the following applies:

* The luma location ( xNbA1, yNbA1 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbHeight − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
* The variables availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 are derived as follows:
* If availableA1 is equal to FALSE, availableFlagA1 is set equal to 0, both components of mvLXA1 are set equal to 0, refIdxLXA1 is set equal to −1 and predFlagLXA1 is set equal to 0, with X being 0 or 1, and bcwIdxA1 is set equal to 0.
* Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvLXA1 = MvLX[ xNbA1 ][ yNbA1 ] (8‑291)

refIdxLXA1 = RefIdxLX[ xNbA1 ][ yNbA1 ] (8‑292)

predFlagLXA1 = PredFlagLX[ xNbA1 ][ yNbA1 ] (8‑293)

bcwIdxA1 = BcwIdx[ xNbA1 ][ yNbA1 ] (8‑294)

For the derivation of availableFlagB1, refIdxLXB1, predFlagLXB1 and mvLXB1 the following applies:

* The luma location ( xNbB1, yNbB1 ) inside the neighbouring luma coding block is set equal to ( xCb + cbWidth − 1, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB1, yNbB1 ) as inputs, and the output is assigned to the block availability flag availableB1.
* The variables availableFlagB1, refIdxLXB1, predFlagLXB1 and mvLXB1 are derived as follows:
* If one or more of the following conditions are true, availableFlagB1 is set equal to 0, both components of mvLXB1 are set equal to 0, refIdxLXB1 is set equal to −1 and predFlagLXB1 is set equal to 0, with X being 0 or 1, and bcwIdxB1 is set equal to 0:
  + availableB1 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB1, yNbB1 ) have the same motion vectors and the same reference indices.
* Otherwise, availableFlagB1 is set equal to 1 and the following assignments are made:

mvLXB1 = MvLX[ xNbB1 ][ yNbB1 ] (8‑295)

refIdxLXB1 = RefIdxLX[ xNbB1 ][ yNbB1 ] (8‑296)

predFlagLXB1 = PredFlagLX[ xNbB1 ][ yNbB1 ] (8‑297)

bcwIdxB1 = BcwIdx[ xNbB1 ][ yNbB1 ] (8‑298)

For the derivation of availableFlagB0, refIdxLXB0, predFlagLXB0 and mvLXB0 the following applies:

* The luma location ( xNbB0, yNbB0 ) inside the neighbouring luma coding block is set equal to ( xCb + cbWidth, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB0, yNbB0 ) as inputs, and the output is assigned to the block availability flag availableB0.
* The variables availableFlagB0, refIdxLXB0, predFlagLXB0 and mvLXB0 are derived as follows:
* If one or more of the following conditions are true, availableFlagB0 is set equal to 0, both components of mvLXB0 are set equal to 0, refIdxLXB0 is set equal to −1 and predFlagLXB0 is set equal to 0, with X being 0 or 1, and bcwIdxB0 is set equal to 0:
  + availableB0 is equal to FALSE.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB0, yNbB0 ) have the same motion vectors and the same reference indices.
  + availableA1 is equal to TRUE, the luma locations ( xNbA1, yNbA1 ) and ( xNbB0, yNbB0 ) have the same motion vectors and the same reference indices and MergeTriangleFlag[ xCb ][ yCb ] is equal to 1.
* Otherwise, availableFlagB0 is set equal to 1 and the following assignments are made:

mvLXB0 = MvLX[ xNbB0 ][ yNbB0 ] (8‑299)

refIdxLXB0 = RefIdxLX[ xNbB0 ][ yNbB0 ] (8‑300)

predFlagLXB0 = PredFlagLX[ xNbB0 ][ yNbB0 ] (8‑301)

bcwIdxB0 = BcwIdx[ xNbB0 ][ yNbB0 ] (8‑302)

For the derivation of availableFlagA0, refIdxLXA0, predFlagLXA0 and mvLXA0 the following applies:

* The luma location ( xNbA0, yNbA0 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbWidth ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA0, yNbA0 ) as inputs, and the output is assigned to the block availability flag availableA0.
* The variables availableFlagA0, refIdxLXA0, predFlagLXA0 and mvLXA0 are derived as follows:
* If one or more of the following conditions are true, availableFlagA0 is set equal to 0, both components of mvLXA0 are set equal to 0, refIdxLXA0 is set equal to −1 and predFlagLXA0 is set equal to 0, with X being 0 or 1, and bcwIdxA0 is set equal to 0:
  + availableA0 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices.
  + availableB1 is equal to TRUE, the luma locations ( xNbB1, yNbB1 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices and MergeTriangleFlag[ xCb ][ yCb ] is equal to 1.
  + availableB0 is equal to TRUE, the luma locations ( xNbB0, yNbB0 ) and ( xNbA0, yNbA0 ) have the same motion vectors and the same reference indices and MergeTriangleFlag[ xCb ][ yCb ] is equal to 1.
* Otherwise, availableFlagA0 is set equal to 1 and the following assignments are made:

mvLXA0 = MvLX[ xNbA0 ][ yNbA0 ] (8‑303)

refIdxLXA0 = RefIdxLX[ xNbA0 ][ yNbA0 ] (8‑304)

predFlagLXA0 = PredFlagLX[ xNbA0 ][ yNbA0 ] (8‑305)

bcwIdxA0 = BcwIdx[ xNbA0 ][ yNbA0 ] (8‑306)

For the derivation of availableFlagB2, refIdxLXB2, predFlagLXB2 and mvLXB2 the following applies:

* The luma location ( xNbB2, yNbB2 ) inside the neighbouring luma coding block is set equal to ( xCb − 1, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB2, yNbB2 ) as inputs, and the output is assigned to the block availability flag availableB2.
* The variables availableFlagB2, refIdxLXB2, predFlagLXB2 and mvLXB2 are derived as follows:
* If one or more of the following conditions are true, availableFlagB2 is set equal to 0, both components of mvLXB2 are set equal to 0, refIdxLXB2 is set equal to −1 and predFlagLXB2 is set equal to 0, with X being 0 or 1, and bcwIdxB2 is set equal to 0:
  + availableB2 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices.
  + availableB1 is equal to TRUE and the luma locations ( xNbB1, yNbB1 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices.
  + availableB0 is equal to TRUE, the luma locations ( xNbB0, yNbB0 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices and MergeTriangleFlag[ xCb ][ yCb ] is equal to 1.
  + availableA0 is equal to TRUE, the luma locations ( xNbA0, yNbA0 ) and ( xNbB2, yNbB2 ) have the same motion vectors and the same reference indices and MergeTriangleFlag[ xCb ][ yCb ] is equal to 1.
  + availableFlagA0 + availableFlagA1 + availableFlagB0 + availableFlagB1 is equal to 4 and MergeTriangleFlag[ xCb ][ yCb ] is equal to 0.
* Otherwise, availableFlagB2 is set equal to 1 and the following assignments are made:

mvLXB2 = MvLX[ xNbB2 ][ yNbB2 ] (8‑307)

refIdxLXB2 = RefIdxLX[ xNbB2 ][ yNbB2 ] (8‑308)

predFlagLXB2 = PredFlagLX[ xNbB2 ][ yNbB2 ] (8‑309)

bcwIdxB2 = BcwIdx[ xNbB2 ][ yNbB2 ] (8‑310)

#### Derivation process for pairwise average merging candidate

Inputs to this process are:

* a merging candidate list mergeCandList,
* the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
* the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
* the motion vectors in 1/16 fractional-sample accuracy mvL0N and mvL1N of every candidate N in mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0avgCand and refIdxL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0avgCand and predFlagL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process,
* the motion vectors in 1/16 fractional-sample accuracy mvL0avgCand and mvL1avgCand of candidate avgCand added into mergeCandList during the invocation of this process.

The variable numRefLists is derived as follows:

numRefLists  =  ( slice\_type = = B )  ?  2 : 1 (8‑311)

The following assignments are made, with p0Cand being the candidate at position 0 and p1Cand being the candidate at position 1 in the merging candidate list mergeCandList:

p0Cand = mergeCandList[ 0 ] (8‑312)

p1Cand = mergeCandList[ 1 ] (8‑313)

The candidate avgCand is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to avgCand, and the reference indices, the prediction list utilization flags and the motion vectors of avgCand are derived as follows and numCurrMergeCand is incremented by 1:

* For each reference picture list LX with X ranging from 0 to ( numRefLists − 1 ), the following applies:
  + If predFlagLXp0Cand is equal to 1 and predFlagLXp1Cand is equal to 1, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], and mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp0Cand (8‑314)

predFlagLXavgCand = 1 (8‑315)

* + - The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLXp0Cand[ 0 ] + mvLXp1Cand[ 0 ], rightShift set equal to1, and leftShift set equal to 0 as inputs and the rounded mvLXavgCand[ 0 ] as output.
    - The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLXp0Cand[ 1 ] + mvLXp1Cand[ 1 ], rightShift set equal to1, and leftShift set equal to 0 as inputs and the rounded mvLXavgCand[ 1 ] as output.
  + Otherwise, if predFlagLXp0Cand is equal to 1 and predFlagLXp1Cand is equal to 0, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp0Cand (8‑316)

predFlagLXavgCand = 1 (8‑317)

mvLXavgCand[ 0 ] = mvLXp0Cand[ 0 ] (8‑318)

mvLXavgCand[ 1 ] = mvLXp0Cand[ 1 ] (8‑319)

* + Otherwise, if predFlagLXp0Cand is equal to 0 and predFlagLXp1Cand is equal to 1, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = refIdxLXp1Cand (8‑320)

predFlagLXavgCand = 1 (8‑321)

mvLXavgCand[ 0 ] = mvLXp1Cand[ 0 ] (8‑322)

mvLXavgCand[ 1 ] = mvLXp1Cand[ 1 ] (8‑323)

* + Otherwise, if predFlagLXp0Cand is equal to 0 and predFlagLXp1Cand is equal to 0, the variables refIdxLXavgCand, predFlagLXavgCand, mvLXavgCand[ 0 ], mvLXavgCand[ 1 ] are derived as follows:

refIdxLXavgCand = −1 (8‑324)

predFlagLXavgCand = 0 (8‑325)

mvLXavgCand[ 0 ] = 0 (8‑326)

mvLXavgCand[ 1 ] = 0 (8‑327)

* When numRefLists is equal to 1, the following applies:

refIdxL1avgCand = −1 (8‑328)

predFlagL1avgCand = 0 (8‑329)

#### Derivation process for zero motion vector merging candidates

Inputs to this process are:

* a merging candidate list mergeCandList,
* the reference indices refIdxL0N and refIdxL1N of every candidate N in mergeCandList,
* the prediction list utilization flags predFlagL0N and predFlagL1N of every candidate N in mergeCandList,
* the motion vectors mvL0N and mvL1N of every candidate N in mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList.

Outputs of this process are:

* the merging candidate list mergeCandList,
* the number of elements numCurrMergeCand within mergeCandList,
* the reference indices refIdxL0zeroCandm and refIdxL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process,
* the prediction list utilization flags predFlagL0zeroCandm and predFlagL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process,
* the motion vectors mvL0zeroCandm and mvL1zeroCandm of every new candidate zeroCandm added into mergeCandList during the invocation of this process.

The variable numRefIdx is derived as follows:

* If slice\_type is equal to P, numRefIdx is set equal to NumRefIdxActive[ 0 ].
* Otherwise (slice\_type is equal to B), numRefIdx is set equal to Min( NumRefIdxActive[ 0 ], NumRefIdxActive[ 1 ] ).

When numCurrMergeCand is less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable zeroIdx is set equal to 0 and the following ordered steps are repeated until numCurrMergeCand is equal to MaxNumMergeCand:

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 ( numCurrMergeCand − numInputMergeCand ) is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to zeroCandm, and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑330)

refIdxL1zeroCandm = −1 (8‑331)

predFlagL0zeroCandm = 1 (8‑332)

predFlagL1zeroCandm = 0 (8‑333)

mvL0zeroCandm[ 0 ] = 0 (8‑334)

mvL0zeroCandm[ 1 ] = 0 (8‑335)

mvL1zeroCandm[ 0 ] = 0 (8‑336)

mvL1zeroCandm[ 1 ] = 0 (8‑337)

numCurrMergeCand = numCurrMergeCand + 1 (8‑338)

* + Otherwise (slice\_type is equal to B), the candidate zeroCandm with m equal to ( numCurrMergeCand − numInputMergeCand ) is added at the end of mergeCandList, i.e., mergeCandList[ numCurrMergeCand ] is set equal to zeroCandm, and the reference indices, the prediction list utilization flags and the motion vectors of zeroCandm are derived as follows and numCurrMergeCand is incremented by 1:

refIdxL0zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑339)

refIdxL1zeroCandm = ( zeroIdx < numRefIdx ) ? zeroIdx : 0 (8‑340)

predFlagL0zeroCandm = 1 (8‑341)

predFlagL1zeroCandm = 1 (8‑342)

mvL0zeroCandm[ 0 ] = 0 (8‑343)

mvL0zeroCandm[ 1 ] = 0 (8‑344)

mvL1zeroCandm[ 0 ] = 0 (8‑345)

mvL1zeroCandm[ 1 ] = 0 (8‑346)

numCurrMergeCand = numCurrMergeCand + 1 (8‑347)

1. The variable zeroIdx is incremented by 1.

#### Derivation process for history-based merging candidates

Inputs to this process are:

* a merge candidate list mergeCandList,
* the number of available merging candidates in the list numCurrMergeCand.

Outputs to this process are:

* the modified merging candidate list mergeCandList,
* the modified number of merging candidates in the list numCurrMergeCand.

The variables isPrunedA1 and isPrunedB1 are both set equal to FALSE.

For each candidate in HmvpCandList[ hMvpIdx ] with index hMvpIdx = 1..NumHmvpCand, the following ordered steps are repeated until numCurrMergeCand is equal to 5:

1. The variable sameMotion is derived as follows:
   * If all of the following conditions are true for any merging candidate N with N being A1 or B1, sameMotion and isPrunedN are both set equal to TRUE:
   * hMvpIdx is less than or equal to 2.
   * The candidate HmvpCandList[ NumHmvpCand − hMvpIdx] is equal to the merging candidate N.
   * isPrunedN is equal to FALSE.
   * Otherwise, sameMotion is set equal to FALSE.
2. When sameMotion is equal to FALSE, the candidate HmvpCandList[ NumHmvpCand − hMvpIdx] is added to the merging candidate list as follows:

mergeCandList[ numCurrMergeCand++ ] = HmvpCandList[ NumHmvpCand − hMvpIdx ] (8‑348)

#### Derivation process for merge motion vector difference

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* reference indices refIdxL0 and refIdxL1,
* prediction list utilization flags predFlagL0 and predFlagL1.

Outputs of this process are the luma merge motion vector differences in 1/16 fractional-sample accuracy mMvdL0 and mMvdL1.

The variable currPic specifies the current picture.

The luma merge motion vector differences mMvdL0 and mMvdL1 are derived as follows:

* If both predFlagL0 and predFlagL1 are equal to 1, the following applies:

currPocDiffL0  =  DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ] ) (8‑349)

currPocDiffL1  =  DiffPicOrderCnt( currPic, RefPicList[ 1 ][ refIdxL1 ] ) (8‑350)

* If currPocDiffL0 is equal to currPocDiffL1, the following applies:

mMvdL0[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑351)

mMvdL0[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑352)

mMvdL1[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑353)

mMvdL1[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑354)

* Otherwise, if Abs( currPocDiffL0 ) is greater than or equal to Abs( currPocDiffL1 ), the following applies:

mMvdL0[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑355)

mMvdL0[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑356)

* + - If RefPicList[ 0 ][ refIdxL0 ] is not a long-term reference picture and RefPicList[ 1 ][ refIdxL1 ] is not a long-term reference picture, the following applies:

td = Clip3( −128, 127, currPocDiffL0 ) (8‑357)

tb = Clip3( −128, 127, currPocDiffL1 ) (8‑358)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑359)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑360)

mMvdL1[ 0 ] = Clip3( −215, 215 − 1, (distScaleFactor \* mMvdL0[ 0 ] + (8‑361)  
 128 − ( distScaleFactor \* mMvdL0[ 0 ] >= 0 ) ) >> 8 )

mMvdL1[ 1 ] = Clip3( −215, 215 − 1, (distScaleFactor \* mMvdL0[ 1 ] + (8‑362)  
 128 − ( distScaleFactor \* mMvdL0[ 1 ] >= 0 ) ) >> 8 )

* + - Otherwise, the following applies:

mMvdL1[ 0 ] = Sign( currPocDiffL0 )  = =  Sign( currPocDiffL1 )  ?   
 mMvdL0[ 0 ]  :  −mMvdL0[ 0 ] (8‑363)

mMvdL1[ 1 ] = Sign( currPocDiffL0 )  = =  Sign( currPocDiffL1)  ?   
 mMvdL0[ 1 ]  :  −mMvdL0[ 1 ] (8‑364)

* Otherwise (Abs( currPocDiffL0 ) is less than Abs( currPocDiffL1 )), the following applies:

mMvdL1[ 0 ]  =  MmvdOffset[ xCb ][ yCb ][ 0 ] (8‑365)

mMvdL1[ 1 ]  =  MmvdOffset[ xCb ][ yCb ][ 1 ] (8‑366)

* + - If RefPicList[ 0 ][ refIdxL0 ] is not a long-term reference picture and RefPicList[ 1 ][ refIdxL1 ] is not a long-term reference picture, the following applies:

td = Clip3( −128, 127, currPocDiffL1 ) (8‑367)

tb = Clip3( −128, 127, currPocDiffL0 ) (8‑368)

tx = ( 16384 + ( Abs( td ) >> 1 ) ) / td (8‑369)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 ) >> 6 ) (8‑370)

mMvdL0[ 0 ] = Clip3( −215, 215 − 1, (distScaleFactor \* mMvdL1[ 0 ] + (8‑371)  
  128 − (distScaleFactor \* mMvdL1[ 0 ] >= 0) ) >> 8 )

mMvdL0[ 1 ] = Clip3( −215, 215 − 1, , (distScaleFactor \* mMvdL1[ 1 ] + (8‑372)  
  128 − (distScaleFactor \* mMvdL1[ 1 ] >= 0) ) >> 8 ) )

* + - Otherwise, the following applies:

mMvdL10[ 0 ] = Sign( currPocDiffL0 )  = =  Sign( currPocDiffL1 )  ?   
 mMvdL1[ 0 ]  :  −mMvdL1[ 0 ] (8‑373)

mMvdL0[ 1 ] = Sign( currPocDiffL0 )  = =  Sign( currPocDiffL1)  ?   
 mMvdL1[ 1 ]  :  −mMvdL1[ 1 ] (8‑374)

* Otherwise ( predFlagL0 or predFlagL1 are equal to 1 ), the following applies for X being 0 and 1:

mMvdLX[ 0 ] = ( predFlagLX = = 1 )  ?  MmvdOffset[ xCb ][ yCb ][ 0 ]  :  0 (8‑375)

mMvdLX[ 1 ] = ( predFlagLX = = 1 )  ?  MmvdOffset[ xCb ][ yCb ][ 1 ]  :  0 (8‑376)

#### Derivation process for luma motion vector prediction

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is the prediction mvpLX in 1/16 fractional-sample accuracy of the motion vector mvLX, with X being 0 or 1.

The motion vector predictor mvpLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidate list as specified in clause 8.5.2.9 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the motion vector predictor candidate list, mvpListLX with X being 0 or 1, as output.
2. The motion vector predictor mvpLX with X being 0 or 1 is derived as follows:

mvpLX = mvpListLX[ mvp\_lX\_flag[ xCb ][ yCb ] ] (8‑377)

#### Derivation process for motion vector predictor candidate list

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding unit partition refIdxLX, with X being 0 or 1.

Output of this process is motion vector predictor candidate list mvpListLX in 1/16 fractional-sample accuracy with X being 0 or 1.

The motion vector predictor candidate list mvpListLX with X being 0 or 1 is derived in the following ordered steps:

1. The derivation process for spatial motion vector predictor candidates from neighbouring coding unit partitions as specified in clause 8.5.2.10 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and the availability flags availableFlagLXN and the motion vectors mvLXN, with N being replaced by A or B, as output.
2. The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLXN, with N being replaced by A or B, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXN, with N being replaced by A or B, as output.
3. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0.
4. Otherwise, the following applies:

* The derivation process for temporal luma motion vector prediction as specified in clause 8.5.2.11 is with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX, with X being 0 or 1 as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLXCol, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvLXCol as output.

1. The motion vector predictor candidate list, mvpListLX, is constructed as follows:

numCurrMvpCand = 0  
if( availableFlagLXA ) {  
 mvpListLX[ numCurrMvpCand++ ] = mvLXA  
 if( availableFlagLXB && ( mvLXA != mvLXB ) )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXB (8‑378)  
} else if( availableFlagLXB )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXB  
if( numCurrMvpCand < 2 && availableFlagLXCol )  
 mvpListLX[ numCurrMvpCand++ ] = mvLXCol

1. When numCurrMvpCand is less than 2 and NumHmvpCand is greater than 0, the following applies for i= 1..Min( 4, NumHmvpCand ) until numCurrMvpCand is equal to 2:

* For each reference picture list LY with Y = X..( 1 − X ), the following applies until numCurrMvpCand is equal to 2:
* When the reference picture corresponding to the reference index of the history-based motion vector predictor candidate HmvpCandList[ i − 1 ] in the reference picture list LY is the same as the reference picture corresponding to reference index refIdxLX in the reference picture list LX, the following applies:
* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to the LY motion vector of the candidate HmvpCandList[ i − 1 ], rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded LY motion vector of the candidate HmvpCandList[ i − 1 ] as output is assigned to mvpListLX[ numCurrMvpCand++ ].

1. When numCurrMvpCand is less than 2, the following applies for until numCurrMvpCand is equal to 2:

mvpListLX[ numCurrMvpCand ][ 0 ] = 0 (8‑379)

mvpListLX[ numCurrMvpCand ][ 1 ] = 0 (8‑380)

numCurrMvpCand++ (8‑381)

#### Derivation process for spatial motion vector predictor candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the reference index of the current coding 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 in 1/16 fractional-sample accuracy of the neighbouring coding units,
* the availability flags availableFlagLXN of the neighbouring coding units.

Figure 8‑2 provides an overview of spatial motion vector neighbours.

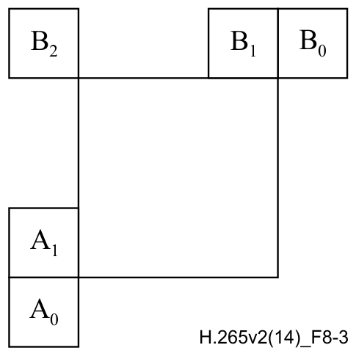


Figure 8‑2 – Spatial motion vector neighbours (informative)

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ) and the variable currPic specifies the current picture.

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. The sample location ( xNbA0, yNbA0 ) is set equal to ( xCb − 1, yCb + cbHeight ) and the sample location ( xNbA1, yNbA1 ) is set equal to ( xNbA0, yNbA0 − 1 ).
2. The availability flag availableFlagLXA is set equal to 0 and both components of mvLXA are set equal to 0.
3. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA0, yNbA0 ) as inputs, and the output is assigned to the block availability flag availableA0.
4. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
5. When availableA0 or availableA1 is equal to TRUE, the variable isScaledFlagLX is set equal to 1.
6. The following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* When availableAk is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
* If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLX[ xNbAk ][ yNbAk ] (8‑382)

* Otherwise, when PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = MvLY[ xNbAk ][ yNbAk ] (8‑383)

1. When availableFlagLXA is equal to 0, the following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ) or until availableFlagLXA is equal to 1:

* When availableAk is equal to TRUE and availableFlagLXA is equal to 0, the following applies:
* If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 , availableFlagLXA is set equal to 1 and the following assignments are made:

mvLXA = MvLX[ xNbAk ][ yNbAk ] (8‑384)

refIdxA = RefIdxLX[ xNbAk ][ yNbAk ] (8‑385)

refPicListA = RefPicList[ X ] (8‑386)

* Otherwise, if PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1, availableFlagLXA is set equal to 1 and the following assignments are made:

mvLXA = MvLY[ xNbAk ][ yNbAk ] (8‑387)

refIdxA = RefIdxLY[ xNbAk ][ yNbAk ] (8‑388)

refPicListA = RefPicList[ Y ] (8‑389)

* When availableFlagLXA is equal to 1, DiffPicOrderCnt( refPicListA[ refIdxA ], RefPicList[ X ][ refIdxLX ] ) is not equal to 0, mvLXA is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑390)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑391)

mvLXA = Clip3( −131072, 131071, ( distScaleFactor \* mvLXA +   
 128 − ( distScaleFactor \* mvLXA >= 0 ) )  >>  8 ) (8‑392)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListA[ refIdxA ] ) ) (8‑393)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) ) (8‑394)

The motion vector mvLXB and the availability flag availableFlagLXB are derived in the following ordered steps:

1. The sample locations ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ) and ( xNbB2, yNbB2 ) are set equal to ( xCb + cbWidth, yCb − 1 ), ( xCb + cbWidth − 1, yCb − 1 ) and ( xCb − 1, yCb − 1 ), respectively.
2. The availability flag availableFlagLXB is set equal to 0 and the both components of mvLXB are set equal to 0.
3. The following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
* If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1, and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[ xNbBk ][ yNbBk ] (8‑395)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑396)

* Otherwise, when PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[ xNbBk ][ yNbBk ] (8‑397)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑398)

1. When isScaledFlagLX is equal to 0 and availableFlagLXB is equal to 1, availableFlagLXA is set equal to 1 and the following applies:

mvLXA = mvLXB (8‑399)

1. When isScaledFlagLX is equal to 0, availableFlagLXB is set equal to 0 and the following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ) or until availableFlagLXB is equal to 1:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and availableFlagLXB is equal to 0, the following applies:
* If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLX[ xNbBk ][ yNbBk ] (8‑400)

refIdxB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑401)

refPicListB = RefPicList[ X ] (8‑402)

* Otherwise, when PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1, availableFlagLXB is set equal to 1 and the following assignments are made:

mvLXB = MvLY[ xNbBk ][ yNbBk ] (8‑403)

refIdxB = RefIdxLY[ xNbBk ][ yNbBk ] (8‑404)

refPicListB = RefPicList[ Y ] (8‑405)

* When availableFlagLXB is equal to 1, DiffPicOrderCnt( refPicListB[ refIdxB ], RefPicList[ X ][ refIdxLX ] ) is not equal to 0, mvLXB is derived as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑406)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑407)

mvLXB = Clip3( −131072, 131071,  (distScaleFactor \* mvLXB +   
 128 − ( distScaleFactor \* mvLXB >= 0 ) )  >>  8 ) (8‑408)

where td and tb are derived as follows:

td = Clip3( −128, 127, DiffPicOrderCnt( currPic, refPicListB[ refIdxB ] ) ) (8‑409)

tb = Clip3( −128, 127, DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) ) (8‑410)

#### Derivation process for temporal luma motion vector prediction

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

* the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
* the availability flag availableFlagLXCol.

The variable currCb specifies the current luma coding block at luma location ( xCb, yCb ).

The variables mvLXCol and availableFlagLXCol are derived as follows:

* If slice\_temporal\_mvp\_enabled\_flag is equal to 0 or ( cbWidth \* cbHeight ) is less than or equal to 32, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* Otherwise (slice\_temporal\_mvp\_enabled\_flag is equal to 1), the following ordered steps apply:

1. The bottom right collocated motion vector is derived as follows:

xColBr = xCb + cbWidth (8‑411)

yColBr = yCb + cbHeight (8‑412)

* If yCb  >>  CtbLog2SizeY is equal to yColBr  >>  CtbLog2SizeY, yColBr is less than pic\_height\_in\_luma\_samples and xColBr is less than pic\_width\_in\_luma\_samples, the following applies:
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColBr  >>  3 )  <<  3, ( yColBr  >>  3 )  <<  3 ) inside the collocated picture specified by ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.
* Otherwise, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.

1. When availableFlagLXCol is equal to 0, the central collocated motion vector is derived as follows:

xColCtr = xCb + ( cbWidth  >>  1 ) (8‑413)

yColCtr = yCb + ( cbHeight  >>  1 ) (8‑414)

* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColCtr  >>  3 )  <<  3, ( yColCtr  >>  3 )  <<  3 ) inside the collocated picture specified by ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLX and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

#### Derivation process for collocated motion vectors

Inputs to this process are:

* a variable currCb specifying the current coding block,
* a variable colCb specifying the collocated coding block inside the collocated picture specified by ColPic,
* a luma location ( xColCb, yColCb ) specifying the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic,
* a reference index refIdxLX, with X being 0 or 1,
* a flag indicating a subblock temporal merging candidate sbFlag.

Outputs of this process are:

* the motion vector prediction mvLXCol in 1/16 fractional-sample accuracy,
* the availability flag availableFlagLXCol.

The variable currPic specifies the current picture.

The arrays predFlagL0Col[ x ][ y ], mvL0Col[ x ][ y ] and refIdxL0Col[ x ][ y ] are set equal to PredFlagL0[ x ][ y ], MvDmvrL0[ x ][ y ] and RefIdxL0[ x ][ y ], respectively, of the collocated picture specified by ColPic, and the arrays predFlagL1Col[ x ][ y ], mvL1Col[ x ][ y ] and refIdxL1Col[ x ][ y ] are set equal to PredFlagL1[ x ][ y ], MvDmvrL1[ x ][ y ] and RefIdxL1[ x ][ y ], respectively, of the collocated picture specified by ColPic.

[Ed. (BB): Define ColPic NoBackwardPredFlag.]

The variables mvLXCol and availableFlagLXCol are derived as follows:

* If colCb is coded in an intra or IBC prediction mode, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* Otherwise, the motion vector mvCol, the reference index refIdxCol and the reference list identifier listCol are derived as follows:
  + If sbFlag is equal to 0, availableFlagLXCol is set to 1 and the following applies:
  + If predFlagL0Col[ xColCb ][ yColCb ] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL1Col[ xColCb ][ yColCb ], refIdxL1Col[ xColCb ][ yColCb ] and L1, respectively.
  + Otherwise, if predFlagL0Col[ xColCb ][ yColCb ] is equal to 1 and predFlagL1Col[ xColCb ][ yColCb ] is equal to 0, mvCol, refIdxCol and listCol are set equal to mvL0Col[ xColCb ][ yColCb ], refIdxL0Col[ xColCb ][ yColCb ] and L0, respectively.
  + Otherwise (predFlagL0Col[ xColCb ][ yColCb ] is equal to 1 and predFlagL1Col[ xColCb ][ yColCb ] is equal to 1), the following assignments are made:
    - * If NoBackwardPredFlag is equal to 1, mvCol, refIdxCol and listCol are set equal to mvLXCol[ xColCb ][ yColCb ], refIdxLXCol[ xColCb ][ yColCb ] and LX, respectively.
      * Otherwise, mvCol, refIdxCol and listCol are set equal to mvLNCol[ xColCb ][ yColCb ], refIdxLNCol[ xColCb ][ yColCb ] and LN, respectively, with N being the value of collocated\_from\_l0\_flag.
  + Otherwise (sbFlag is equal to 1), the following applies:
  + If PredFlagLXCol[ xColCb ][ yColCb ] is equal to 1, mvCol, refIdxCol, and listCol are set equal to mvLXCol[ xColCb ][ yColCb ], refIdxLXCol[ xColCb ][ yColCb ], and LX, respectively, availableFlagLXCol is set to 1.
  + Otherwise (PredFlagLXCol[ xColCb ][ yColCb ] is equal to 0), the following applies:
    - * If DiffPicOrderCnt( aPic, currPic ) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice and PredFlagLYCol[ xColCb ][ yColCb ] is equal to 1, mvCol, refIdxCol, and listCol are set to mvLYCol[ xColCb ][ yColCb ], refIdxLYCol[ xColCb ][ yColCb ] and LY, respectively, with Y being equal to !X where X being the value of X this process is invoked for. availableFlagLXCol is set to 1.
      * Both the components of mvLXCol are set to 0 and availableFlagLXCol is set equal to 0.
  + When availableFlagLXCol is equal to TRUE, mvLXCol and availableFlagLXCol are derived as follows:
  + If LongTermRefPic( currPic, currCb, refIdxLX, LX ) is not equal to LongTermRefPic( ColPic, colCb, refIdxCol, listCol ), both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
  + Otherwise, the variable availableFlagLXCol is set equal to 1, refPicList[ listCol ][ refIdxCol ] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing coding block colCb in the collocated picture specified by ColPic, and the following applies:

colPocDiff = DiffPicOrderCnt( ColPic, refPicList[ listCol ][ refIdxCol ] ) (8‑415)

currPocDiff = DiffPicOrderCnt( currPic, RefPicList[ X ][ refIdxLX ] ) (8‑416)

* + - * The temporal motion buffer compression process for collocated motion vectors as specified in clause 8.5.2.15 is invoked with mvCol as input, and the modified mvCol as output.
      * If RefPicList[ X ][ refIdxLX ] is a long-term reference picture, or colPocDiff is equal to currPocDiff, mvLXCol is derived as follows:

mvLXCol = mvCol (8‑417)

* + - * Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

tx = ( 16384 + ( Abs( td )  >>  1 ) ) / td (8‑418)

distScaleFactor = Clip3( −4096, 4095, ( tb \* tx + 32 )  >>  6 ) (8‑419)

mvLXCol =  Clip3( −131072, 131071, (distScaleFactor \* mvCol +  
 128 − ( distScaleFactor \* mvCol  >= 0 ) )  >>  8 ) ) (8‑420)

where td and tb are derived as follows:

td = Clip3( −128, 127, colPocDiff ) (8‑421)

tb = Clip3( −128, 127, currPocDiff ) (8‑422)

#### Derivation process for chroma motion vectors

Input to this process are:

* a luma motion vector in 1/16 fractional-sample accuracy mvLX,
* the reference index refIdxLX.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvCLX.

A chroma motion vector is derived from the corresponding luma motion vector.

The chroma motion vector mvCLX, is derived as follows:

mvCLX[ 0 ] = mvLX[ 0 ] \* 2 / SubWidthC (8‑423)

mvCLX[ 1 ] = mvLX[ 1 ] \* 2 / SubHeightC (8‑424)

#### Rounding process for motion vectors

Inputs to this process are

* the motion vector mvX,
* the right shift parameter rightShift for rounding,
* the left shift parameter leftShift for resolution increase.

Output of this process is the rounded motion vector mvX.

For the rounding of mvX, the following applies:

offset = ( rightShift = = 0 )  ?  0  :  ( 1  <<  ( rightShift − 1 ) ) (8‑425)

mvX[ 0 ] = ( ( mvX[ 0 ] + offset − ( mvX[ 0 ] >= 0 ) ) >> rightShift ) << leftShift (8‑426)

mvX[ 1 ] = ( ( mvX[ 0 ] + offset − ( mvX[ 1 ] >= 0 ) ) >> rightShift ) << leftShift (8‑427)

#### Temporal motion buffer compression process for collocated motion vectors

Input to this process is a motion vector mv.

Outputs of this process is the rounded motion vector mv.

For each motion vector component compIdx being 0 and 1, mv[ compIdx ] is modified as follows:

s = mv[ compIdx ] >> 17 (8‑428)

f = Floor( Log2( ( mv[ compIdx ] ^ s ) | 31 ) ) − 4 (8‑429)

mask = ( −1 << f ) >> 1 (8‑430)

round = ( 1 << f ) >> 2 (8‑431)

mv[ compIdx ] = ( mv[ compIdx ] + round ) & mask (8‑432)

NOTE – This process enables storage of collocated motion vectors using a bit reduced representation. Each signed 18-bit motion vector component can be represented in a mantissa plus exponent format with a 6-bit signed mantissa and a 4-bit exponent.

#### Updating process for the history-based motion vector predictor candidate list

Inputs to this process are:

* luma motion vectors in 1/16 fractional-sample accuracy mvL0 and mvL1,
* reference indices refIdxL0 and refIdxL1,
* prediction list utilization flags predFlagL0 and predFlagL1,
* bi-prediction weight index bcwIdx.

The MVP candidate hMvpCand consists of the luma motion vectors mvL0 and mvL1, the reference indices refIdxL0 and refIdxL1, the prediction list utilization flags predFlagL0 and predFlagL1, and the bi-prediction weight index bcwIdx.

The candidate list HmvpCandList is modified using the candidate hMvpCand by the following ordered steps:

1. The variable identicalCandExist is set equal to FALSE and the variable removeIdx is set equal to 0.
2. When NumHmvpCand is greater than 0, for each index hMvpIdx with hMvpIdx = 0..NumHmvpCand − 1, the following steps apply until identicalCandExist is equal to TRUE:
   * When hMvpCand is equal to HmvpCandList[ hMvpIdx ], identicalCandExist is set equal to TRUE and removeIdx is set equal to hMvpIdx.
3. The candidate list HmvpCandList is updated as follows:
   * If identicalCandExist is equal to TRUE or NumHmvpCand is equal to 5, the following applies:
   * For each index i with i = ( removeIdx + 1 )..( NumHmvpCand − 1 ), HmvpCandList[ i − 1] is set equal to HmvpCandList[ i ].
   * HmvpCandList[ NumHmvpCand − 1 ] is set equal to mvCand.
   * Otherwise (identicalCandExist is equal to FALSE and NumHmvpCand is less than 5), the following applies:
   * HmvpCandList[ NumHmvpCand++ ] is set equal to mvCand.

### Decoder side motion vector refinement process

#### General

Inputs to this process are:

* a luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture,
* a variable sbWidth specifying the width of the current coding subblock in luma samples,
* a variable sbHeight specifying the height of the current coding subblock in luma samples,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0 and mvL1,
* the selected luma reference picture sample arrays refPicL0L and refPicL1L.

Outputs of this process are:

* delta luma motion vectors dMvL0 and dMvL1.

The variable subPelFlag is set equal to 0, the variable srRange is set equal to 2 and the integer sample offset ( intOffX, intOffY ) is set equal to ( 0, 0 ).

Both components of the delta luma motion vectors dMvL0 and dMvL1 are set equal to zero and modified as follows:

* For each X being 0 or 1, the ( sbWidth + 2 \* srRange ) x ( sbHeight + 2 \* srRange ) array predSamplesLXL of prediction luma sample values is derived by invoking the fractional sample bilinear interpolation process specified in 8.5.3.2.1 with the luma location ( xSb, ySb ), the prediction block width set equal to ( sbWidth + 2 \* srRange ), the prediction block height set equal to ( sbHeight + 2 \* srRange ), the reference picture sample array refPicLXL, the motion vector mvLX and the refinement search range srRange as inputs.
* The variable minSad is derived by invoking the sum of absolute differences calculation process specified in clause 8.5.3.3 with the width sbW and height sbH of the current coding subblock set equal to sbWidth and sbHeight, the prediction sample arrays pL0 and pL1 set equal to predSamplesL0L and predSamplesL1L, and the offset ( dX, dY ) set equal to ( 0, 0 ) as inputs, and minSad as output.
* When minSad is greater than or equal to sbHeight \* sbWidth, the following applies:
* The 2-D array sadArray[ dX + 2 ][ dY + 2 ] with dX = −2..2 and dY = −2..2 is derived by invoking the sum of absolute differences calculation process specified in clause 8.5.3.3 with the width sbW and height sbH of the current coding subblock set equal to sbWidth and sbHeight, the prediction sample arrays pL0 and pL1 set equal to predSamplesL0L and predSamplesL1L, and the offset ( dX, dY ) as inputs, and sadArray[ dX + 2 ][ d Y + 2 ] as output.
* The integer sample offset ( intOffX, intOffY ) is modified by invoking the array entry selection process specified in clause 8.5.3.4 with the 2-D array sadArray[ dX + 2 ][ dY + 2 ] with dX = −2..2 and dY = −2..2, the best integer sample offset ( intOffX, intOffY ), and minSad as input, and the modified best integer sample offset ( intOffX, intOffY ) as output.
* When intOffX is not equal to −2 or 2, and intOffY is not equal to −2 or 2, subPelFlag is set equal to 1.
* The delta luma motion vector dMvL0 is modified as follows:

dMvL0[ 0 ] += 16 \* intOffX (8‑433)

dMvL0[ 1 ] += 16 \* intOffY (8‑434)

* When subPelFlag is equal to 1, the parametric motion vector refinement process specified in clause 8.5.3.5 is invoked with the 3x3 2-D array sadArray[ dX + 2 ][ dY + 2 ] with dX = intOffX − 1, intOffX, intOffX + 1 and dY = intOffY − 1, intOffY, intOffY + 1, and the delta motion vector dMvL0 as inputs and the modified dMvL0 as output.
* The delta motion vector dMvL1 is derived as follows:

dMvL1[ 0 ] = −dMvL0[ 0 ] (8‑435)

dMvL1[ 1 ] = −dMvL0[ 1 ] (8‑436)

#### Fractional sample bilinear interpolation process

##### General

Inputs to this process are:

* a luma location ( xSb, ySb ) specifying the top-left sample of the current subblock relative to the top‑left luma sample of the current picture,
* a variable pbWidth specifying the width of the current prediction block in luma samples,
* a variable pbHeight specifying the height of the current prediction block in luma samples,
* a luma motion vector mvLX given in 1/16-luma-sample units,
* the selected reference picture sample array refPicLXL,
* the refinement search range srRange.

Output of this process is:

* a ( pbWidth ) x ( pbHeight ) array predSamplesLXL of luma prediction sample values.

Let ( xIntL, yIntL ) be a luma location given in full-sample units and ( xFracL, yFracL ) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample array refPicLXL.

For each luma sample location ( xL = 0..pbWidth − 1, yL = 0..pbHeight − 1 ) inside the luma prediction sample array predSamplesLXL, the corresponding luma prediction sample value predSamplesLXL[ xL ][ yL ] is derived as follows:

* The variables xIntL, yIntL, xFracL and yFracL are derived as follows:

xIntL = xSb + ( mvLX[ 0 ]  >>  4 ) + xL − srRange (8‑437)

yIntL = ySb + ( mvLX[ 1 ]  >>  4 ) + yL − srRange (8‑438)

xFracL = mvLX[ 0 ] & 15 (8‑439)

yFracL = mvLX[ 1 ] & 15 (8‑440)

* The luma prediction sample value predSamplesLXL[ xL ][ yL ] is derived by invoking the luma sample bilinear interpolation process specified in clause 8.5.3.2.2 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLXL as inputs.

##### Luma sample bilinear 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

The variables shift1, shift2, shift3, shift4, offset1, offset2 and offset3 are derived as follows:

shift1 = BitDepthY − 6 (8‑441)

offset1 = 1 << ( shift1 − 1 ) (8‑442)

shift2 = 4 (8‑443)

offset2 = 1 << ( shift2 − 1 ) (8‑444)

shift3 = 10 − BitDepthY (8‑445)

shift4 = BitDepthY − 10 (8‑446)

offset4 = 1 << ( shift4 − 1 ) (8‑447)

The variable picW is set equal to pic\_width\_in\_luma\_samples and the variable picH is set equal to pic\_height\_in\_luma\_samples.

The luma interpolation filter coefficients fbL[ p ] for each 1/16 fractional sample position p equal to xFracL or  yFracL are specified in Table 8‑13.

The luma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..1:

xInti = Clip3( 0, picW − 1, sps\_ref\_wraparound\_enabled\_flag ?  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, ( xIntL + i ) ) : (8‑448)  
 xIntL + i )

yInti = Clip3( 0, picH − 1, yIntL + i ) (8‑449)

The predicted luma sample value predSampleLXL is derived as follows:

– If both xFracLand yFracL are equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = BitDepthY <= 10  ?  (refPicLXL[ xInt0 ][ yInt0 ] << shift3 ) :   
 ( ( refPicLXL[ xInt0 ][ yInt0 ] + offset4 ) >> shift4 ) (8‑450)

– Otherwise, if xFracL is not equal to 0 and yFracL is equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = (  + offset1 )  >>  shift1 (8‑451)

– Otherwise, if xFracL is equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = (  + offset1 )  >>  shift1 (8‑452)

– Otherwise, if xFracL is not equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

* The sample array temp[ n ] with n = 0..1, is derived as follows:

temp[ n ] = (  + offset1 )  >>  shift1 (8‑453)

* The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL = (  + offset2 )  >>  shift2 (8‑454)

Table 8‑13 – Specification of the luma bilinear interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p.

|  |  |  |
| --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | |
| **fbL[ p ][ 0 ]** | **fbL[ p ][ 1 ]** |
| 1 | 15 | 1 |
| 2 | 14 | 2 |
| 3 | 13 | 3 |
| 4 | 12 | 4 |
| 5 | 11 | 5 |
| 6 | 10 | 6 |
| 7 | 9 | 7 |
| 8 | 8 | 8 |
| 9 | 7 | 9 |
| 10 | 6 | 10 |
| 11 | 5 | 11 |
| 12 | 4 | 12 |
| 13 | 3 | 13 |
| 14 | 2 | 14 |
| 15 | 1 | 15 |

#### Sum of absolute differences calculation process

Inputs to this process are:

* two variables nSbW and nSbH specifying the width and the height of the current subblock,
* two ( nSbW + 4 ) x ( nSbH + 4 ) arrays pL0 and pL1 containing the predicted samples for L0 and L1 respectively,
* an integer sample offset ( dX, dY ) in L0.

Output of this process is:

* the variable sad specifying the sum of absolute differences at the integer sample at the offset ( dX, dY ) in L0

(8‑455)

#### Array entry selection process

Inputs to this process are:

* a 2-D array of sum of absolute differences values sadArray[ dX + 2 ][ d Y + 2 ] with dX = −2..2 and dY = −2..2,
* an integer sample offset ( intOffX, intOffY ),
* a variable minSad.

Output of this process is the modified integer sample ( intOffX, intOffY ).

The following steps are applied to modify the integer sample offset ( intOffX, intOffY ):

for ( dY = −2; dY <= 2; dY++) {  
 for ( dX = −2; dX <= 2; dX++) {  
 if ( sadArray[ dX + 2 ][ dY + 2 ] < minSad) {  
 minSad = sadArray[ dX + 2 ][ dY + 2 ] (8‑456)  
 intOffX = dX  
 intOffY = dY  
 }  
 }  
}

#### Parametric motion vector refinement process

Inputs to this process are:

* a 3x3 2-D array sadArray[ dX + 1 ][ dY + 1 ] with dX = −1..1 and dY = −1..1,
* a delta luma motion vector dMvL0.

Output of this process is the modified delta luma motion vector dMvL0.

The variable dMvX is derived by invoking the derivation process for delta motion vector component offset specified in clause 8.5.3.5.1 with the SAD values sadMinus, sadCenter and sadPlus set equal to sadArray[ 0 ][ 1 ], sadArray[ 1 ][ 1 ], and sadArray[ 2 ][ 1 ] as inputs, and dMvX set equal to the output dMVc.

The variable dMvY is derived by invoking the derivation process for delta motion vector component offset specified in clause 8.5.3.5.1 with the SAD values sadMinus, sadCenter and sadPlus set equal to sadArray[ 1 ][ 0 ], sadArray[ 1 ][ 1 ], and sadArray[ 1 ][ 2 ] as inputs, and dMvY set equal to the output dMVc.

The delta luma motion vector dMvL0 is modified as follows:

dMvL0[ 0 ] += dMvX (8‑457)

dMvL0[ 1 ] += dMvY (8‑458)

NOTE – dMvC with C being X or Y is constrained to be between −8 and 8 since sadMinus, sadCenter, and sadPlus are all positive, and sadCenter is the smallest value among the three. This allows the division to be performed with up to 4 quotient bits and can be implemented using compares, shifts, and subtractions.

##### Derivation process for delta motion vector component offset

Inputs to this process are 3 SAD values sadMinus, sadCenter, and sadPlus.

Output of this process is the delta motion vector component correction offset dMvC.

The offset dMVc is derived using the following pseudo code:

denom = ( ( sadMinus + sadPlus ) − ( sadCenter << 1 ) ) << 3  
if ( denom = = 0 )   
 dMvC = 0  
else {  
 if (sadMinus = = sadCenter )   
 dMvC = −8  
 else if (sadPlus = = sadCenter )   
 dMvC = 8  
 else {  
 num = ( sadMinus − sadPlus ) << 4  
 signNum = 0  
 if ( num < 0 ) {  
 num = −num  
 signNum = 1  
 }  
 quotient = 0 (8‑459)  
 counter = 3  
 while ( counter > 0 ) {  
 counter = counter − 1  
 quotient = quotient << 1  
 if ( num >= denom ) {  
 num = num − denom  
 quotient = quotient + 1  
 }  
 denom = (denom >> 1)  
 }  
 if (signNum = = 1)  
 dMvC = −quotient  
 else  
 dMvC = quotient  
 }  
}

NOTE 1– The above process is equivalent to an integer division of num = ( sadMinus − sadPlus ) << 3 by denom = ( sadMinus + sadPlus − ( sadCenter << 1 ). Given the fact that sadMinus, sadCenter, and sadPlus are all positive, and sadCenter is the smallest value among the three, the value is limited to be in the range of −8 to 8, inclusive.

### Derivation process for triangle motion vector components and reference indices

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vectors in 1/16 fractional-sample accuracy mvA and mvB,
* the chroma motion vectors in 1/32 fractional-sample accuracy mvCA and mvCB,
* the reference indices refIdxA and refIdxB,
* the prediction list flags predListFlagA and predListFlagB.

The derivation process for luma motion vectors for triangle merge mode as specified in clause 8.5.4.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight as inputs, and the output being the luma motion vectors mvA, mvB, the reference indices refIdxA, refIdxB and the prediction list flags predListFlagA and predListFlagB.

The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvA and refIdxA as input, and the output being mvCA.

The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvB and refIdxB as input, and the output being mvCB.

#### Derivation process for luma motion vectors for merge triangle mode

This process is only invoked when MergeTriangleFlag[ xCb ][ yCb ] is equal to 1, where ( xCb, yCb ) specify the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vectors in 1/16 fractional-sample accuracy mvA and mvB,
* the reference indices refIdxA and refIdxB,
* the prediction list flags predListFlagA and predListFlagB.

The motion vectors mvA and mvB, the reference indices refIdxA and refIdxB and the prediction list flags predListFlagA and predListFlagB are derived by the following ordered steps:

1. The derivation process for luma motion vectors for merge mode as specified in clause 8.5.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the luma motion vectors mvL0[ 0 ][ 0 ], mvL1[ 0 ][ 0 ], the reference indices refIdxL0, refIdxL1, the prediction list utilization flags predFlagL0[ 0 ][ 0 ] and predFlagL1[ 0 ][ 0 ], the bi-prediction weight index bcwIdx and the merging candidate list mergeCandList.
2. The variables m and n, being the merge index for triangle partition 0 and 1 resepctively, are derived using merge\_triangle\_idx0[ xCb ][ yCb ] and merge\_triangle\_idx1[ xCb ][ yCb ] as follows:

m = merge\_triangle\_idx0[ xCb ][ yCb ] (8‑460)

n = merge\_triangle\_idx1[ xCb ][ yCb ] + ( merge\_triangle\_idx1[ xCb ][ yCb ] >= m ) ? 1 : 0 (8‑461)

1. Let refIdxL0M and refIdxL1M, predFlagL0M and predFlagL1M, and mvL0M and mvL1M be the reference indices, the prediction list utilization flags and the motion vectors of the merging candidate M at position m in the merging candidate list mergeCandList ( M = mergeCandList[ m ] ).
2. The variable X is set equal to ( m & 0x01 ).
3. When predFlagLXM is equal to 0, X is set equal to ( 1 − X ).
4. The following applies:

mvA[ 0 ] = mvLXM[ 0 ] (8‑462)

mvA[ 1 ] = mvLXM[ 1 ] (8‑463)

refIdxA = refIdxLXM (8‑464)

predListFlagA = X (8‑465)

1. Let refIdxL0N and refIdxL1N, predFlagL0N and predFlagL1N, and mvL0N and mvL1N be the reference indices, the prediction list utilization flags and the motion vectors of the merging candidate N at position m in the merging candidate list mergeCandList ( N = mergeCandList[ n ] ).
2. The variable X is set equal to ( n & 0x01 ).
3. When predFlagLXN is equal to 0, X is set equal to ( 1 − X ).
4. The following applies:

mvB[ 0 ] = mvLXN[ 0 ] (8‑466)

mvB[ 1 ] = mvLXN[ 1 ] (8‑467)

refIdxB = refIdxLXN (8‑468)

predListFlagB = X (8‑469)

### Derivation process for subblock motion vector components and reference indices

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the reference indices refIdxL0 and refIdxL1,
* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbX − 1,
* the luma subblock motion vector arrays in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the chroma subblock motion vector arrays in 1/32 fractional-sample accuracy mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the bi-prediction weight index bcwIdx.

For the derivation of the variables mvL0[ xSbIdx ][ ySbIdx ], mvL1[ xSbIdx ][ ySbIdx ], mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, numSbX, numSbY, predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the following applies:

* If merge\_subblock\_flag[ xCb ][ yCb ] is equal to 1, the derivation process for motion vectors and reference indices in subblock merge mode as specified in 8.5.5.2 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxL0, refIdxL1, the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ], the luma subblock motion vector arrays mvL0[ xSbIdx ][ ySbIdx ] and mvL0[ xSbIdx ][ ySbIdx ], and the chroma subblock motion vector arrays mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ], with xSbIdx = 0.. numSbX − 1, ySbIdx = 0 .. numSbY − 1, and the bi-prediction weight index bcwIdx as outputs.
* Otherwise (merge\_subblock\_flag[ xCb ][ yCb ] is equal to 0), for X being replaced by either 0 or 1 in the variables predFlagLX, cpMvLX, MvdCpLX, and refIdxLX, in PRED\_LX, and in the syntax element ref\_idx\_lX, the following ordered steps apply:
* For the derivation of the number of control point motion vectors numCpMv, the control point motion vectors cpMvLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, refIdxLX, predFlagLX[ 0 ][ 0 ], the following applies:

1. The number of control point motion vectors numCpMv is set equal to MotionModelIdc[ xCb ][ yCb ] + 1.
2. The variables refIdxLX and predFlagLX are derived as follows:

* If inter\_pred\_idc[ xCb ][ yCb ] is equal to PRED\_LX or PRED\_BI,

refIdxLX = ref\_idx\_lX[ xCb ][ yCb ] (8‑470)

predFlagLX[ 0 ][ 0 ] = 1 (8‑471)

* Otherwise, the variables refIdxLX and predFlagLX are specified by:

refIdxLX = −1 (8‑472)

predFlagLX[ 0 ][ 0 ] = 0 (8‑473)

1. The variable mvdCpLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1, is derived as follows:

mvdCpLX[ cpIdx ][ 0 ] = MvdCpLX[ xCb ][ yCb ][ cpIdx ][ 0 ] (8‑474)

mvdCpLX[ cpIdx ][ 1 ] = MvdCpLX[ xCb ][ yCb ][ cpIdx ][ 1 ] (8‑475)

1. When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for luma affine control point motion vector predictors as specified in clause 8.5.5.7 is invoked with the luma coding block location ( xCb, yCb ), and the variables cbWidth, cbHeight, refIdxLX, and the number of control point motion vectors numCpMv as inputs, and the output being mvpCpLX[ cpIdx ] with cpIdx ranging from 0 to numCpMv − 1.
2. When predFlagLX[ 0 ][ 0 ] is equal to 1, the luma motion vectors cpMvLX[ cpIdx ] with cpIdx ranging from 0 to NumCpMv − 1, are derived as follows:

uLX[ cpIdx ][ 0 ] = ( mvpCpLX[ cpIdx ][ 0 ] + mvdCpLX[ cpIdx ][ 0 ] + 218 ) % 218 (8‑476)

cpMvLX[ cpIdx ][ 0 ] = (uLX[ cpIdx ][ 0 ] >= 217 ) ? (uLX[ cpIdx ][ 0 ] − 218 ) :   
 uLX[ cpIdx ][ 0 ] (8‑477)

uLX[ cpIdx ][ 1 ] = ( mvpCpLX[ cpIdx ][ 1 ] + mvdCpLX[ cpIdx ][ 1 ] + 218 ) % 218 (8‑478)

cpMvLX[ cpIdx ][ 1 ] = (uLX[ cpIdx ][ 1 ] >= 217 ) ? (uLX[ cpIdx ][ 1 ] − 218 ) :   
 uLX[ cpIdx ][ 1 ] (8‑479)

* The variables numSbX and numSbY are derived as follows:

numSbX = ( cbWidth >> 2 ) (8‑480)

numSbY = ( cbHeight >> 2 ) (8‑481)

* For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑482)

* When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.5.5.9 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma prediction block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvLX[ cpIdx ] with cpIdx being 0..2, the reference index refIdxLX and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY as inputs, the luma motion vector array mvLX[ xSbIdx ][ ySbIdx ] and the chroma motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 as outputs.
* The bi-prediction weight index bcwIdx is set equal to bcw\_idx[ xCb ][ yCb ].

#### Derivation process for motion vectors and reference indices in subblock merge mode

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the luma coding block.

Outputs of this process are:

* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flag arrays predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ],
* the luma subblock motion vector arrays in 1/16 fractional-sample accuracy mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the chroma subblock motion vector arrays in 1/32 fractional-sample accuracy mvCL0[ xSbIdx ][ ySbIdx ] and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1,
* the bi-prediction weight index bcwIdx.

The variables numSbX, numSbY and the subblock merging candidate list, subblockMergeCandList are derived by the following ordered steps:

1. When sps\_sbtmvp\_enabled\_flag is equal to 1, the following applies:

* For the derivation of availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 the following applies:
  + - The luma location ( xNbA1, yNbA1 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbHeight − 1 ).
    - The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
    - The variables availableFlagA1, refIdxLXA1, predFlagLXA1 and mvLXA1 are derived as follows:
    - If availableA1 is equal to FALSE, availableFlagA1 is set equal to 0, both components of mvLXA1 are set equal to 0, refIdxLXA1 is set equal to −1 and predFlagLXA1 is set equal to 0, with X being 0 or 1, and bcwIdxA1 is set equal to 0.
    - Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvLXA1 = MvLX[ xNbA1 ][ yNbA1 ] (8‑483)

refIdxLXA1 = RefIdxLX[ xNbA1 ][ yNbA1 ] (8‑484)

predFlagLXA1 = PredFlagLX[ xNbA1 ][ yNbA1 ] (8‑485)

* The derivation process for subblock-based temporal merging candidates as specified in clause 8.5.5.3 is invoked with the luma location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight , the availability flag availableFlagA1, the reference index refIdxLXA1, the prediction list utilization flag predFlagLXA1, and the motion vector mvLXA1 as inputs and the output being the availability flag availableFlagSbCol, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the reference indices refIdxLXSbCol, the luma motion vectors mvLXSbCol[ xSbIdx ][ ySbIdx ] and the prediction list utilization flags predFlagLXSbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.

1. When sps\_affine\_enabled\_flag is equal to 1, the sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), ( xNbB2, yNbB2 ), ( xNbB3, yNbB3 ), and the variables numSbX and numSbY are derived as follows:

( xA0, yA0 ) = ( xCb − 1, yCb + cbHeight ) (8‑486)

( xA1, yA1 ) = ( xCb − 1, yCb + cbHeight − 1 ) (8‑487)

( xA2, yA2 ) = ( xCb − 1, yCb ) (8‑488)

( xB0, yB0 ) = ( xCb + cbWidth , yCb − 1 ) (8‑489)

( xB1, yB1 ) = ( xCb + cbWidth − 1, yCb − 1 ) (8‑490)

( xB2, yB2 ) = ( xCb − 1, yCb − 1 ) (8‑491)

( xB3, yB3 ) = ( xCb, yCb − 1 ) (8‑492)

numSbX = cbWidth >> 2 (8‑493)

numSbY = cbHeight >> 2 (8‑494)

1. When sps\_affine\_enabled\_flag is equal to 1, the variable availableFlagA is set equal to FALSE and the following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbAk, yNbAk ) as inputs, and the output is assigned to the block availability flag availableAk.
* When availableAk is equal to TRUE and MotionModelIdc[ xNbAk ][ yNbAk ] is greater than 0 and availableFlagA is equal to FALSE, the following applies:
  + - The variable availableFlagA is set equal to TRUE, motionModelIdcA is set equal to MotionModelIdc[ xNbAk ][ yNbAk ], ( xNb, yNb ) is set equal to ( CbPosX[ xNbAk ][ yNbAk ], CbPosY[ xNbAk ][ yNbAk ] ), nbW is set equal to CbWidth[ xNbAk ][ yNbAk ], nbH is set equal to CbHeight[ xNbAk ][ yNbAk ], numCpMv is set equal to MotionModelIdc[ xNbAk ][ yNbAk ] + 1, and bcwIdxA is set equal to BcwIdx[ xNbAk ][ yNbAk ].
    - For X being replaced by either 0 or 1, the following applies:
      * When PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvLXA[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

predFlagLXA = PredFlagLX[ xNbAk ][ yNbAk ] (8‑495)

refIdxLXA = RefIdxLX[ xNbAk ][ yNbAk ] (8‑496)

1. When sps\_affine\_enabled\_flag is equal to 1, the variable availableFlagB is set equal to FALSE and the following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and MotionModelIdc[ xNbBk ][ yNbBk ] is greater than 0 and availableFlagB is equal to FALSE, the following applies:
  + - The variable availableFlagB is set equal to TRUE, motionModelIdcB is set equal to MotionModelIdc[ xNbBk ][ yNbBk ], ( xNb, yNb ) is set equal to ( CbPosX[ xNbAB ][ yNbBk ], CbPosY[ xNbBk ][ yNbBk ] ), nbW is set equal to CbWidth[ xNbBk ][ yNbBk ], nbH is set equal to CbHeight[ xNbBk ][ yNbBk ], numCpMv is set equal to MotionModelIdc[ xNbBk ][ yNbBk ] + 1, and bcwIdxB is set equal to BcwIdx[ xNbBk ][ yNbBk ].
    - For X being replaced by either 0 or 1, the following applies:
      * When PredFlagLX[ xNbBk ][ yNbBk ] is equal to TRUE, the derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvLXB[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

predFlagLXB = PredFlagLX[ xNbBk ][ yNbBk ] (8‑497)

refIdxLXB = RefIdxLX[ xNbBk ][ yNbBk ] (8‑498)

1. When sps\_affine\_enabled\_flag is equal to 1, the derivation process for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.6 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the availability flags availableA0, availableA1, availableA2, availableB0, availableB1, availableB2, availableB3 as inputs, and the availability flags availableFlagConstK, the reference indices refIdxLXConstK, prediction list utilization flags predFlagLXConstK, motion model indices motionModelIdcConstK, bi-prediction weight indices bcwIdxConstK and cpMvpLXConstK[ cpIdx ] with X being 0 or 1, K = 1..6, cpIdx = 0..2 as outputs.
2. The initial subblock merging candidate list, subblockMergeCandList, is constructed as follows:

i = 0  
if( availableFlagSbCol )  
 subblockMergeCandList[ i++ ] = SbCol  
if( availableFlagA && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = A  
if( availableFlagB && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Bif( availableFlagConst1 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const1 (8‑499)if( availableFlagConst2 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const2if( availableFlagConst3 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const3  
if( availableFlagConst4 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const4if( availableFlagConst5 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const5  
if( availableFlagConst6 && i < MaxNumSubblockMergeCand )  
 subblockMergeCandList[ i++ ] = Const6

1. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the subblockMergeCandList.
2. When numCurrMergeCand is less than MaxNumSubblockMergeCand, the following is repeated until numCurrMrgeCand is equal to MaxNumSubblockMergeCand, with mvZero[0] and mvZero[1] both being equal to 0:

* The reference indices, the prediction list utilization flags and the motion vectors of zeroCandm with m equal to ( numCurrMergeCand − numOrigMergeCand ) are derived as follows:

refIdxL0ZeroCandm = 0 (8‑500)

predFlagL0ZeroCandm = 1 (8‑501)

cpMvL0ZeroCandm[ 0 ] = mvZero (8‑502)

cpMvL0ZeroCandm[ 1 ] = mvZero (8‑503)

cpMvL0ZeroCandm[ 2 ] = mvZero (8‑504)

refIdxL1ZeroCandm = ( slice\_type = = B ) ? 0 : −1 (8‑505)

predFlagL1ZeroCandm = ( slice\_type = = B ) ? 1 : 0 (8‑506)

cpMvL1ZeroCandm[ 0 ] = mvZero (8‑507)

cpMvL1ZeroCandm[ 1 ] = mvZero (8‑508)

cpMvL1ZeroCandm[ 2 ] = mvZero (8‑509)

motionModelIdcZeroCandm = 1 (8‑510)

bcwIdxZeroCandm = 0 (8‑511)

* The candidate zeroCandm with m equal to ( numCurrMergeCand − numOrigMergeCand ) is added at the end of subblockMergeCandList and numCurrMergeCand is incremented by 1 as follows:

subblockMergeCandList[ numCurrMergeCand++ ] = zeroCandm (8‑512)

The variables refIdxL0, refIdxL1, predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], mvL0[ xSbIdx ][ ySbIdx ], mvL1[ xSbIdx ][ ySbIdx ], mvCL0[ xSbIdx ][ ySbIdx ], and mvCL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1 are derived as follows:

* If subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] is equal to SbCol, the bi-prediction weight index bcwIdx is set equal to 0 and the following applies with X being 0 or 1:

refIdxLX = refIdxLXSbCol (8‑513)

* + - * For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLXSbCol[ xSbIdx ][ ySbIdx ] (8‑514)

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 0 ] (8‑515)

mvLX[ xSbIdx ][  ySbIdx ][ 1 ] = mvLXSbCol[ xSbIdx ][ ySbIdx ][ 1 ] (8‑516)

* + - * When predFlagLX[ xSbIdx ][ ySbIdx ], is equal to 1, the derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvLX[ xSbIdx ][ ySbIdx ] and refIdxLX as inputs, and the output being mvCLX[ xSbIdx ][ ySbIdx ].
      * The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[ x ][ y ] = 0  (8‑517)

* Otherwise (subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] is not equal to SbCol), the following applies with X being 0 or 1:
  + - * The following assignments are made with N being the candidate at position merge\_subblock\_idx[ xCb ][ yCb ] in the subblock merging candidate list subblockMergeCandList ( N = subblockMergeCandList[ merge\_subblock\_idx[ xCb ][ yCb ] ] ):

refIdxLX = refIdxLXN (8‑518)

predFlagLX[ 0][ 0 ] = predFlagLXN (8‑519)

cpMvLX[ 0 ] = cpMvLXN[ 0 ] (8‑520)

cpMvLX[ 1 ] = cpMvLXN[ 1 ] (8‑521)

cpMvLX[ 2 ] = cpMvLXN[ 2 ] (8‑522)

numCpMv = motionModelIdxN + 1 (8‑523)

bcwIdx = bcwIdxN (8‑524)

* + - * For xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1, the following applies:

predFlagLX[ xSbIdx ][ ySbIdx ] = predFlagLX[ 0 ][ 0 ] (8‑525)

* + - * When predFlagLX[ 0 ][ 0 ] is equal to 1, the derivation process for motion vector arrays from affine control point motion vectors as specified in subclause 8.5.5.9 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma prediction block height cbHeight, the number of control point motion vectors numCpMv, the control point motion vectors cpMvLX[ cpIdx ] with cpIdx being 0..2, and the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY as inputs, the luma subblock motion vector array mvLX[ xSbIdx ][ ySbIdx ] and the chroma subblock motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 as outputs.
      * The following assignment is made for x = xCb ..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

MotionModelIdc[ x ][ y ] = numCpMv − 1 (8‑526)

#### Derivation process for subblock-based temporal merging candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.
* the availability flag availableFlagA1 of the neighbouring coding unit,
* the reference index refIdxLXA1 of the neighbouring coding unit with X being 0 or 1,
* the prediction list utilization flag predFlagLXA1 of the neighbouring coding unit with X being 0 or 1,
* the motion vector in 1/16 fractional-sample accuracy mvLXA1 of the neighbouring coding unit with X being 0 or 1.

Outputs of this process are:

* the availability flag availableFlagSbCol,
* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY,
* the reference indices refIdxL0SbCol and refIdxL1SbCol,
* the luma motion vectors in 1/16 fractional-sample accuracy mvL0SbCol[ xSbIdx ][ ySbIdx ] and mvL1SbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1,
* the prediction list utilization flags predFlagL0SbCol[ xSbIdx ][ ySbIdx ] and predFlagL1SbCol[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1.

The availability flag availableFlagSbCol is derived as follows.

* If one or more of the following conditions is true, availableFlagSbCol is set equal to 0.
* slice\_temporal\_mvp\_enabled\_flag is equal to 0.
* sps\_sbtmvp\_enabled\_flag is equal to 0.
* cbWidth is less than 8.
* cbHeight is less than 8.
* Otherwise, the following ordered steps apply:

1. The location ( xCtb, yCtb ) of the top-left sample of the luma coding tree block that contains the current coding block and the location ( xCtr, yCtr ) of the below-right center sample of the current luma coding block are derived as follows:

xCtb = ( xCb  >>  CtuLog2Size )  <<  CtuLog2Size (8‑527)

yCtb = ( yCb  >>  CtuLog2Size )  <<  CtuLog2Size (8‑528)

xCtr = xCb + ( cbWidth / 2 ) (8‑529)

yCtr = yCb + ( cbHeight / 2 ) (8‑530)

1. The luma location ( xColCtrCb, yColCtrCb ) is set equal to the top-left sample of the collocated luma coding block covering the location given by ( xCtr, yCtr ) inside ColPic relative to the top-left luma sample of the collocated picture specified by ColPic.
2. The derivation process for subblock-based temporal merging base motion data as specified in clause 8.5.5.4 is invoked with the location ( xCtb, yCtb ), the location ( xColCtrCb, yColCtrCb ), the availability flag availableFlagA1, and the prediction list utilization flag predFlagLXA1, and the reference index refIdxLXA1, and the motion vector mvLXA1, with X being 0 and 1 as inputs and the motion vectors ctrMvLX, and the prediction list utilization flags ctrPredFlagLX of the collocated block, with X being 0 and 1, and the temporal motion vector tempMv as outputs.
3. The variable availableFlagSbCol is derived as follows:

* If both ctrPredFlagL0 and ctrPredFlagL1 are equal to 0, availableFlagSbCol is set equal to 0.
* Otherwise, availableFlagSbCol is set equal to 1.

When availableFlagSbCol is equal to 1, the following applies:

* The variables numSbX, numSbY, sbWidth, sbHeight and refIdxLXSbCol are derived as follows:

numSbX  =  cbWidth >> 3 (8‑531)

numSbY  =  cbHeight >> 3 (8‑532)

sbWidth  =  cbWidth / numSbX (8‑533)

sbHeight  =  cbHeight / numSbY (8‑534)

refIdxLXSbCol  =  0 (8‑535)

* For xSbIdx = 0..numSbX − 1 and ySbIdx = 0 .. numSbY − 1, the motion vectors mvLXSbCol[ xSbIdx ][ ySbIdx ] and prediction list utilization flags predFlagLXSbCol[ xSbIdx ][ ySbIdx ] are derived as follows:
* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

xSb  =  xCb + xSbIdx \* sbWidth (8‑536)

ySb  =  yCb + ySbIdx \* sbHeight (8‑537)

* The location ( xColSb, yColSb ) of the collocated subblock inside ColPic is derived as follows.

xColSb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) + 3 ), (8‑538)  
 xSb + ( tempMv[0]  >>  4 ) )

yColSb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) − 1 ), (8‑539)  
 ySb + ( tempMv[1]  >>  4 ) )

* The variable currCb specifies the luma coding block covering the current coding subblock inside the current picture.
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColSb >> 3 ) << 3, ( yColSb >> 3 ) << 3 ) inside the ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxL0 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL0SbCol[ xSbIdx ][ ySbIdx ] and availableFlagL0SbCol.
* The derivation process for collocated motion vectors as specified in clause 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxL1 set equal to 0 and sbFlag set equal to 1 as inputs and the output being assigned to the motion vector of the subblock mvL1SbCol[ xSbIdx ][ ySbIdx ] and availableFlagL1SbCol.
* When availableFlagL0SbCol and availableFlagL1SbCol are both equal to 0, the following applies for X being 0 and 1:

mvLXSbCol[ xSbIdx ][ ySbIdx ] = ctrMvLX (8‑540)

predFlagLXSbCol[ xSbIdx ][ ySbIdx ] = ctrPredFlagLX (8‑541)

#### Derivation process for subblock-based temporal merging base motion data

Inputs to this process are:

* the location ( xCtb, yCtb ) of the top-left sample of the luma coding tree block that contains the current coding block,
* the location ( xColCtrCb, yColCtrCb ) of the top-left sample of the collocated luma coding block that covers the below-right center sample.
* the availability flag availableFlagA1 of the neighbouring coding unit,
* the reference index refIdxLXA1 of the neighbouring coding unit,
* the prediction list utilization flag predFlagLXA1 of the neighbouring coding unit,
* the motion vector in 1/16 fractional-sample accuracy mvLXA1 of the neighbouring coding unit.

Outputs of this process are:

* the motion vectors ctrMvL0 and ctrMvL1,
* the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1,
* the temporal motion vector tempMv.

The variable tempMv is set as follows:

tempMv[ 0 ] = 0 (8‑542)

tempMv[ 1 ] = 0 (8‑543)

The variable currPic specifies the current picture.

When availableFlagA1 is equal to TRUE, the following applies:

* If all of the following conditions are true, tempMv is set equal to mvL1A1:
  + predFlagL1A1 is equal to 1,
  + DiffPicOrderCnt(ColPic, RefPicList[ 1 ][refIdxL1A1]) is equal to 0,
  + DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice,
  + slice\_type is equal to B,
  + collocated\_from\_l0\_flag is equal to 0.
* Otherwise if all of the following conditions are true, tempMv is set equal to mvL0A1:
  + predFlagL0A1 is equal to 1,
  + DiffPicOrderCnt(ColPic, RefPicList[ 0 ][refIdxL0A1]) is equal to 0.

The location ( xColCb, yColCb ) of the collocated block inside ColPic is derived as follows.

xColCb = Clip3( xCtb,   
 Min( CurPicWidthInSamplesY − 1, xCtb + ( 1  <<  CtbLog2SizeY ) + 3 ), (8‑544)  
 xColCtrCb + ( tempMv[0]  >>  4 ) )

yColCb = Clip3( yCtb,   
 Min( CurPicHeightInSamplesY − 1, yCtb + ( 1  <<  CtbLog2SizeY ) − 1 ), (8‑545)  
 yColCtrCb + ( tempMv[1]  >>  4 ) )

The array colPredMode is set equal to the prediction mode array CuPredMode of the collocated picture specified by ColPic.

The motion vectors ctrMvL0 and ctrMvL1, and the prediction list utilization flags ctrPredFlagL0 and ctrPredFlagL1 are derived as follows:

* If colPredMode[xColCb][yColCb] is equal to MODE\_INTER, the following applies:
* The variable currCb specifies the luma coding block covering ( xCtrCb ,yCtrCb ) inside the current picture.
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColCb >> 3 ) << 3, ( yColCb >> 3 ) << 3 ) inside the ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
  + The derivation process for collocated motion vectors specified in clause 8.5.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL0 set equal to 0, and sbFlag set equal to 1 as inputs and the output being assigned to ctrMvL0 and ctrPredFlagL0.
  + The derivation process for collocated motion vectors specified in clause 8.5.2.12 is invoked with currCb, colCb, (xColCb, yColCb), refIdxL1 set equal to 0, and sbFlag set equal to 1 as inputs and the output being assigned to ctrMvL1 and ctrPredFlagL1.
* Otherwise, the following applies:

ctrPredFlagL0 = 0 (8‑546)

ctrPredFlagL1 = 0 (8‑547)

#### Derivation process for luma affine control point motion vectors from a neighbouring block

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* a luma location ( xNb, yNb ) specifying the top-left sample of the neighbouring luma coding block relative to the top-left luma sample of the current picture,
* two variables nNbW and nNbH specifying the width and the height of the neighbouring luma coding block,
* the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1.

The variable isCTUboundary is derived as follows:

* If all the following conditions are true, isCTUboundary is set equal to TRUE:
* ( ( yNb + nNbH ) % CtbSizeY ) is equal to 0
* yNb + nNbH is equal to yCb
* Otherwise, isCTUboundary is set equal to FALSE.

The variables log2NbW and log2NbH are derived as follows:

log2NbW = Log2( nNbW ) (8‑548)

log2NbH = Log2( nNbH ) (8‑549)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

* If isCTUboundary is equal to TRUE, the following applies:

mvScaleHor = MvLX[ xNb ][ yNb + nNbH − 1 ][ 0 ] << 7 (8‑550)

mvScaleVer = MvLX[ xNb ][ yNb + nNbH − 1 ][ 1 ] << 7 (8‑551)

dHorX = ( MvLX[ xNb + nNbW − 1 ][ yNb + nNbH − 1 ][ 0 ] − MvLX[ xNb ][ yNb + nNbH − 1 ][ 0 ] )    
 << ( 7 − log2NbW ) (8‑552)

dVerX = ( MvLX[ xNb + nNbW − 1 ][ yNb + nNbH − 1 ][ 1 ] − MvLX[ xNb ][ yNb + nNbH − 1 ][ 1 ] )    
 << ( 7 − log2NbW ) (8‑553)

* Otherwise (isCTUboundary is equal to FALSE), the following applies:

mvScaleHor = CpMvLX[ xNb ][ yNb ][ 0 ][ 0 ] << 7 (8‑554)

mvScaleVer = CpMvLX[ xNb ][ yNb ][ 0 ][ 1 ] << 7 (8‑555)

dHorX = ( CpMvLX[ xNb + nNbW − 1 ][ yNb ][ 1 ][ 0 ] − CpMvLX[ xNb ][ yNb ][ 0 ][ 0 ] )   
 << ( 7 − log2NbW ) (8‑556)

dVerX = ( CpMvLX[ xNb + nNbW − 1 ][ yNb ][ 1 ][ 1 ] − CpMvLX[ xNb ][ yNb ][ 0 ][ 1 ] )   
 << ( 7 − log2NbW ) (8‑557)

The variables dHorY and dVerY are derived as follows:

* If isCTUboundary is equal to FALSE and MotionModelIdc[ xNb ][ yNb ] is equal to 2, the following applies:

dHorY = ( CpMvLX[ xNb ][ yNb + nNbH − 1 ][ 2 ][ 0 ] − CpMvLX[ xNb ][ yNb ][ 2 ][ 0 ] )   
 << ( 7 − log2NbH ) (8‑558)

dVerY = ( CpMvLX[ xNb ][ yNb + nNbH − 1 ][ 2 ][ 1 ] − CpMvLX[ xNb ][ yNb ][ 2 ][ 1 ] )   
 << ( 7 − log2NbH ) (8‑559)

* Otherwise (isCTUboundary is equal to TRUE or MotionModelIdc[ xNb ][ yNb ] is equal to 1), the following applies,

dHorY = − dVerX (8‑560)

dVerY = dHorX (8‑561)

The luma affine control point motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 and X being 0 or 1 are derived as follows:

* When isCTUboundary is equal to TRUE, yNb is set equal to yCb.
* The first two control point motion vectors cpMvLX[ 0 ] and cpMvLX[ 1 ] are derived as follows:

cpMvLX[ 0 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb − xNb ) + dHorY \* ( yCb − yNb ) ) (8‑562)

cpMvLX[ 0 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb − yNb ) ) 8‑563)

cpMvLX[ 1 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb + cbWidth − xNb ) + dHorY \* ( yCb − yNb ) ) (8‑564)

cpMvLX[ 1 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb + cbWidth − xNb ) + dVerY \* ( yCb − yNb ) ) (8‑565)

* If numCpMv is equal to 3, the third control point vector cpMvLX[ 2 ] is derived as follows:

cpMvLX[ 2 ][ 0 ] = ( mvScaleHor + dHorX \* ( xCb − xNb ) + dHorY \* ( yCb + cbHeight − yNb ) ) (8‑566)

cpMvLX[ 2 ][ 1 ] = ( mvScaleVer + dVerX \* ( xCb − xNb ) + dVerY \* ( yCb + cbHeight − yNb ) ) (8‑567)

* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ cpIdx ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLX[ cpIdx ] as output, with X being 0 or 1 and cpIdx = 0 .. numCpMv − 1.
* The motion vectors cpMvLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 are clipped as follows:

cpMvLX[ cpIdx ][ 0 ] = Clip3( −217, 217 − 1, cpMvLX[ cpIdx ][ 0 ] ) (8‑568)

cpMvLX[ cpIdx ][ 1 ] = Clip3( −217, 217 − 1, cpMvLX[ cpIdx ][ 1 ] ) (8‑569)

#### Derivation process for constructed affine control point motion vector merging candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* the availability flags availableA0, availableA1, availableA2, availableB0, availableB1, availableB2, availableB3,
* the sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), ( xNbB2, yNbB2 ) and ( xNbB3, yNbB3 ).

Output of this process are:

* the availability flag of the constructed affine control point motion vector merging candidiates availableFlagConstK, with K = 1..6,
* the reference indices refIdxLXConstK, with K = 1..6, X being 0 or 1,
* the prediction list utilization flags predFlagLXConstK, with K = 1..6, X being 0 or 1,
* the affine motion model indices motionModelIdcConstK, with K = 1..6,
* the bi-prediction weight indices bcwIdxConstK, with K = 1..6,
* the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2, K = 1..6 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLXCorner[ 0 ], reference index refIdxLXCorner[ 0 ], prediction list utilization flag predFlagLXCorner[ 0 ], bi-prediction weight index bcwIdxCorner[ 0 ] and the availability flag availableFlagCorner[ 0 ] with X being 0 and 1 are derived as follows:

* The availability flag availableFlagCorner[ 0 ] is set equal to FALSE.
* The following applies for ( xNbTL, yNbTL ) with TL being replaced by B2, B3, and A2:
  + - * When availableTL is equal to TRUE and availableFlagCorner[ 0 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 0 ] = RefIdxLX[ xNbTL ][ yNbTL ] (8‑570)

predFlagLXCorner[ 0 ] = PredFlagLX[ xNbTL ][ yNbTL ] (8‑571)

cpMvLXCorner[ 0 ] = MvLX[ xNbTL ][ yNbTL ] (8‑572)

bcwIdxCorner[ 0 ] = BcwIdx[ xNbTL ][ yNbTL ] (8‑573)

availableFlagCorner[ 0 ] = TRUE (8‑574)

The second (top-right) control point motion vector cpMvLXCorner[ 1 ], reference index refIdxLXCorner[ 1 ], prediction list utilization flag predFlagLXCorner[ 1 ], bi-prediction weight index bcwIdxCorner[ 1 ] and the availability flag availableFlagCorner[ 1 ] with X being 0 and 1 are derived as follows

* The availability flag availableFlagCorner[ 1 ] is set equal to FALSE.
* The following applies for ( xNbTR, yNbTR ) with TR being replaced by B1 and B0:
  + - * When availableTR is equal to TRUE and availableFlagCorner[ 1 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 1 ] = RefIdxLX[ xNbTR ][ yNbTR ] (8‑575)

predFlagLXCorner[ 1 ] = PredFlagLX[ xNbTR ][ yNbTR ] (8‑576)

cpMvLXCorner[ 1 ] = MvLX[ xNbTR ][ yNbTR ] (8‑577)

bcwIdxCorner[ 1 ] = BcwIdx[ xNbTR ][ yNbTR ] (8‑578)

availableFlagCorner[ 1 ] = TRUE (8‑579)

The third (bottom-left) control point motion vector cpMvLXCorner[ 2 ], reference index refIdxLXCorner[ 2 ], prediction list utilization flag predFlagLXCorner[ 2 ], bi-prediction weight index bcwIdxCorner[ 2 ] and the availability flag availableFlagCorner[ 2 ] with X being 0 and 1 are derived as follows:

* The availability flag availableFlagCorner[ 2 ] is set equal to FALSE.
* The following applies for ( xNbBL, yNbBL ) with BL being replaced by A1 and A0:
  + - * When availableBL is equal to TRUE and availableFlagCorner[ 2 ] is equal to FALSE, the following applies with X being 0 and 1:

refIdxLXCorner[ 2 ] = RefIdxLX[ xNbBL ][ yNbBL ] (8‑580)

predFlagLXCorner[ 2 ] = PredFlagLX[ xNbBL ][ yNbBL ] (8‑581)

cpMvLXCorner[ 2 ] = MvLX[ xNbBL ][ yNbBL ] (8‑582)

bcwIdxCorner[ 2 ] = BcwIdx[ xNbBL ][ yNbBL ] (8‑583)

availableFlagCorner[ 2 ] = TRUE (8‑584)

The fourth (collocated bottom-right) control point motion vector cpMvLXCorner[ 3 ], reference index refIdxLXCorner[ 3 ], prediction list utilization flag predFlagLXCorner[ 3 ], bi-prediction weight index bcwIdxCorner[ 3 ] and the availability flag availableFlagCorner[ 3 ] with X being 0 and 1 are derived as follows:

* The reference indices for the temporal merging candidate, refIdxLXCorner[ 3 ], with X being 0 or 1, are set equal to 0.
* The variables mvLXCol and availableFlagLXCol, with X being 0 or 1, are derived as follows:
* If slice\_temporal\_mvp\_enabled\_flag is equal to 0, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* Otherwise (slice\_temporal\_mvp\_enabled\_flag is equal to 1), the following applies:

xColBr = xCb + cbWidth (8‑585)

yColBr = yCb + cbHeight (8‑586)

* If yCb  >>  CtbLog2SizeY is equal to yColBr  >>  CtbLog2SizeY, yColBr is less than pic\_height\_in\_luma\_samples and xColBr is less than pic\_width\_in\_luma\_samples, the following applies:
* The variable colCb specifies the luma coding block covering the modified location given by ( ( xColBr  >>  3 )  <<  3, ( yColBr  >>  3 )  <<  3 ) inside the collocated picture specified by ColPic.
* The luma location ( xColCb, yColCb ) is set equal to the top-left sample of the collocated luma coding block specified by colCb relative to the top-left luma sample of the collocated picture specified by ColPic.
* The derivation process for collocated motion vectors as specified in clause 8.5.2.12 is invoked with currCb, colCb, ( xColCb, yColCb ), refIdxLXCorner[ 3 ] and sbFlag set equal to 0 as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.
* Otherwise, both components of mvLXCol are set equal to 0 and availableFlagLXCol is set equal to 0.
* The variables availableFlagCorner[ 3 ], predFlagL0Corner[ 3 ], cpMvL0Corner[ 3 ] and predFlagL1Corner[ 3 ] are derived as follows:

availableFlagCorner[ 3 ] = availableFlagL0Col (8‑587)

predFlagL0Corner[ 3 ] = availableFlagL0Col (8‑588)

cpMvL0Corner[ 3 ] = mvL0Col (8‑589)

predFlagL1Corner[ 3 ] = 0 (8‑590)

* When slice\_type is equal to B, the variables availableFlagCorner[ 3 ], predFlagL1Corner[ 3 ] and cpMvL1Corner[ 3 ] are derived as follows:

availableFlagCorner[ 3 ] = availableFlagL0Col  | |  availableFlagL1Col (8‑591)

predFlagL1Corner[ 3 ] = availableFlagL1Col (8‑592)

cpMvL1Corner[ 3 ] = mvL1Col (8‑593)

bcwIdxCorner[ 3 ] = 0 (8‑594)

When sps\_affine\_type\_flag is equal to 1, the first four constructed affine control point motion vector merging candidates ConstK with K = 1..4 including the availability flags availableFlagConstK, the reference indices refIdxLXConstK, the prediction list utilization flags predFlagLXConstK, the affine motion model indices motionModelIdcConstK, and the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2 and X being 0 or 1 are derived as follows:

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst1 = 1 (8‑595)

refIdxLXConst1 = refIdxLXCorner[ 0 ] (8‑596)

cpMvLXConst1[ 0 ] = cpMvLXCorner[ 0 ] (8‑597)

cpMvLXConst1[ 1 ] = cpMvLXCorner[ 1 ] (8‑598)

cpMvLXConst1[ 2 ] = cpMvLXCorner[ 2 ] (8‑599)

* The bi-prediction weight index bcwIdxConst1 is derived as follows:
* If availableFlagL0 is equal 1 and availableFlagL1 is equal to 1, the derivation process for bi-prediction weight index for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.10 is invoked with the bi-prediction weight indices bcwIdxCorner[ 0 ], bcwIdxCorner[ 1 ] and bcwIdxCorner[ 2 ] as inputs, and the output is assigned to the bi-prediction weight index bcwIdxConst1.
* Otherwise, the bi-prediction weight index bcwIdxConst1 is set equal to 0.
* The variables availableFlagConst1 and motionModelIdcConst1 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst1 is set equal to TRUE and motionModelIdcConst1 is set equal to 2.
      * Otherwise, availableFlagConst1 is set equal to FALSE and motionModelIdcConst1 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst2 = 1 (8‑600)

refIdxLXConst2 = refIdxLXCorner[ 0 ] (8‑601)

cpMvLXConst2[ 0 ] = cpMvLXCorner[ 0 ] (8‑602)

cpMvLXConst2[ 1 ] = cpMvLXCorner[ 1 ] (8‑603)

cpMvLXConst2[ 2 ] = cpMvLXCorner[ 3 ] + cpMvLXCorner[ 0 ] − cpMvLXCorner[ 1 ] (8‑604)

cpMvLXConst2[ 2 ][ 0 ] = Clip3( −217, 217 − 1, cpMvLXConst2[ 2 ][ 0 ] ) (8‑605)

cpMvLXConst2[ 2 ][ 1 ] = Clip3( −217, 217− 1, cpMvLXConst2[ 2 ][ 1 ] ) (8‑606)

* The bi-prediction weight index bcwIdxConst2 is derived as follows:
* If availableFlagL0 is equal to 1 and availableFlagL1 is equal to 1, the derivation process for bi-prediction weight index for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.10 is invoked with the bi-prediction weight indices bcwIdxCorner[ 0 ], bcwIdxCorner[ 1 ] and bcwIdxCorner[ 3 ] as inputs, and the output is assigned to the bi-prediction weight index bcwIdxConst2.
* Otherwise, the bi-prediction weight index bcwIdxConst2 is set equal to 0.
* The variables availableFlagConst2 and motionModelIdcConst2 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst2 is set equal to TRUE and motionModelIdcConst2 is set equal to 2.
      * Otherwise, availableFlagConst2 is set equal to FALSE and motionModelIdcConst2 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst3 = 1 (8‑607)

refIdxLXConst3 = refIdxLXCorner[ 0 ] (8‑608)

cpMvLXConst3[ 0 ] = cpMvLXCorner[ 0 ] (8‑609)

cpMvLXConst3[ 1 ] = cpMvLXCorner[ 3 ] + cpMvLXCorner[ 0 ] − cpMvLXCorner[ 2 ] (8‑610)

cpMvLXConst3[ 1 ][ 0 ] = Clip3( −217, 217 − 1, cpMvLXConst3[ 1 ][ 0 ] ) (8‑611)

cpMvLXConst3[ 1 ][ 1 ] = Clip3( −217, 217 − 1, cpMvLXConst3[ 1 ][ 1 ] ) (8‑612)

cpMvLXConst3[ 2 ] = cpMvLXCorner[ 2 ] (8‑613)

* The bi-prediction weight index bcwIdxConst3 is derived as follows:
* If availableFlagL0 is equal to 1 and availableFlagL1 is equal to 1, the derivation process for bi-prediction weight index for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.10 is invoked with the bi-prediction weight indices bcwIdxCorner[ 0 ], bcwIdxCorner[ 2 ] and bcwIdxCorner[ 3 ] as inputs, and the output is assigned to the bi-prediction weight index bcwIdxConst3.
* Otherwise, the bi-prediction weight index bcwIdxConst3 is set equal to 0.
* The variables availableFlagConst3 and motionModelIdcConst3 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst3 is set equal to TRUE and motionModelIdcConst3 is set equal to 2.
      * Otherwise, availableFlagConst3 is set equal to FALSE and motionModelIdcConst3 is set equal to 0.

1. When availableFlagCorner[ 1 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE and availableFlagCorner[ 3 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 1 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* predFlagLXCorner[ 3 ] is equal to 1
* refIdxLXCorner[ 1 ] is equal to refIdxLXCorner[ 2 ]
* refIdxLXCorner[ 1 ] is equal to refIdxLXCorner[ 3 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst4 = 1 (8‑614)

refIdxLXConst4 = refIdxLXCorner[ 1 ] (8‑615)

cpMvLXConst4[ 0 ] = cpMvLXCorner[ 1 ] + cpMvLXCorner[ 2 ] − cpMvLXCorner[ 3 ] (8‑616)

cpMvLXConst4[ 0 ][ 0 ] = Clip3( −217, 217 − 1, cpMvLXConst4[ 0 ][ 0 ] ) (8‑617)

cpMvLXConst4[ 0 ][ 1 ] = Clip3( −217, 217 − 1, cpMvLXConst4[ 0 ][ 1 ] ) (8‑618)

cpMvLXConst4[ 1 ] = cpMvLXCorner[ 1 ] (8‑619)

cpMvLXConst4[ 2 ] = cpMvLXCorner[ 2 ] (8‑620)

* The bi-prediction weight index bcwIdxConst4 is derived as follows:
* If availableFlagL0 is equal to 1 and availableFlagL1 is equal to 1, the derivation process for bi-prediction weight index for constructed affine control point motion vector merging candidates as specified in clause 8.5.5.10 is invoked with the bi-prediction weight indices bcwIdxCorner[ 1 ], bcwIdxCorner[ 2 ] and bcwIdxCorner[ 3 ] as inputs, and the output is assigned to the bi-prediction weight index bcwIdxConst4.
* Otherwise, the bi-prediction weight index bcwIdxConst4 is set equal to 0.
* The variables availableFlagConst4 and motionModelIdcConst4 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst4 is set equal to TRUE and motionModelIdcConst4 is set equal to 2.
      * Otherwise, availableFlagConst4 is set equal to FALSE and motionModelIdcConst4 is set equal to 0.

The last two constructed affine control point motion vector merging candidates ConstK with K = 5..6 including the availability flags availableFlagConstK, the reference indices refIdxLXConstK, the prediction list utilization flags predFlagLXConstK, the affine motion model indices motionModelIdcConstK, and the constructed affine control point motion vectors cpMvLXConstK[ cpIdx ] with cpIdx = 0..2 and X being 0 or 1 are derived as follows:

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 1 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 1 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 1 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following assignments are made:

predFlagLXConst5 = 1 (8‑621)

refIdxLXConst5 = refIdxLXCorner[ 0 ] (8‑622)

cpMvLXConst5[ 0 ] = cpMvLXCorner[ 0 ] (8‑623)

cpMvLXConst5[ 1 ] = cpMvLXCorner[ 1 ] (8‑624)

* The bi-prediction weight index bcwIdxConst5 is derived as follows:
* If availableFlagL0 is equal to 1, and availableFlagL1 is equal to 1, and bcwIdxCorner[ 0 ] is equal to bcwIdxCorner[ 1 ], bcwIdxConst5 is set equal to bcwIdxCorner[ 0 ]
* Otherwise, the bi-prediction weight index bcwIdxConst5 is set equal to 0.
* The variables availableFlagConst5 and motionModelIdcConst5 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst5 is set equal to TRUE and motionModelIdcConst5 is set equal to 1.
      * Otherwise, availableFlagConst5 is set equal to FALSE and motionModelIdcConst5 is set equal to 0.

1. When availableFlagCorner[ 0 ] is equal to TRUE and availableFlagCorner[ 2 ] is equal to TRUE, the following applies:

* For X being replaced by 0 or 1, the following applies:
  + - * The variable availableFlagLX is derived as follows:
      * If all of following conditions are TRUE, availableFlagLX is set equal to TRUE:
* predFlagLXCorner[ 0 ] is equal to 1
* predFlagLXCorner[ 2 ] is equal to 1
* refIdxLXCorner[ 0 ] is equal to refIdxLXCorner[ 2 ]
  + - * Otherwise, availableFlagLX is set equal to FALSE.
      * When availableFlagLX is equal to TRUE, the following applies:
      * The second control point motion vector cpMvLXCorner[ 1 ] is derived as follows:

cpMvLXCorner[ 1 ][ 0 ] = ( cpMvLXCorner[ 0 ][ 0 ] << 7 ) +    
 ( ( cpMvLXCorner[ 2 ][ 1 ] − cpMvLXCorner[ 0 ][ 1 ] ) (8‑625)  
  << ( 7 + Log2( cbHeight / cbWidth ) ) )

cpMvLXCorner[ 1 ][ 1 ] = ( cpMvLXCorner[ 0 ][ 1 ] << 7 ) +    
 ( ( cpMvLXCorner[ 2 ][ 0 ] − cpMvLXCorner[ 0 ][ 0 ] ) (8‑626)  
  << ( 7 + Log2( cbHeight / cbWidth ) ) )

* + - * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLXCorner[ 1 ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded cpMvLXCorner[ 1 ] as output.
      * The following assignments are made:

predFlagLXConst6 = 1 (8‑627)

refIdxLXConst6 = refIdxLXCorner[ 0 ] (8‑628)

cpMvLXConst6[ 0 ] = cpMvLXCorner[ 0 ] (8‑629)

cpMvLXConst6[ 1 ] = cpMvLXCorner[ 1 ] (8‑630)

cpMvLXConst6[ 0 ][ 0 ] = Clip3( −217, 217 − 1, cpMvLXConst6[ 0 ][ 0 ] ) (8‑631)

cpMvLXConst6[ 0 ][ 1 ] = Clip3( −217, 217 − 1, cpMvLXConst6[ 0 ][ 1 ] ) (8‑632)

cpMvLXConst6[ 1 ][ 0 ] = Clip3( −217, 217 − 1, cpMvLXConst6[ 1 ][ 0 ] ) (8‑633)

cpMvLXConst6[ 1 ][ 1 ] = Clip3( −217, 217 − 1, cpMvLXConst6[ 1 ][ 1 ] ) (8‑634)

* The bi-prediction weight index bcwIdxConst6 is derived as follows:
* If availableFlagL0 is equal to 1 and availableFlagL1 is equal to 1 and bcwIdxCorner[ 0 ] is equal to bcwIdxCorner[ 2 ], bcwIdxConst6 is set equal to bcwIdxCorner[ 0 ]
* Otherwise, the bi-prediction weight index bcwIdxConst6 is set equal to 0.
* The variables availableFlagConst6 and motionModelIdcConst6 are derived as follows:
  + - * If availableFlagL0 or availableFlagL1 is equal to 1, availableFlagConst6 is set equal to TRUE and motionModelIdcConst6 is set equal to 1.
      * Otherwise, availableFlagConst6 is set equal to FALSE and motionModelIdcConst6 is set equal to 0.

#### Derivation process for luma affine control point motion vector predictors

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* the reference index of the current coding unit refIdxLX, with X being 0 or 1,
* the number of control point motion vectors numCpMv.

Output of this process are the luma affine control point motion vector predictors mvpCpLX[ cpIdx ] with X being 0 or 1, and cpIdx = 0 .. numCpMv − 1.

For the derivation of the control point motion vectors predictor candidate list, cpMvpListLX with X being 0 or 1, the following ordered steps apply:

1. The number of control point motion vector predictor candidates in the list numCpMvpCandLX is set equal to 0.
2. The variables availableFlagA and availableFlagB are both set equal to FALSE.
3. The sample locations ( xNbA0, yNbA0 ), ( xNbA1, yNbA1 ), ( xNbA2, yNbA2 ), ( xNbB0, yNbB0 ), ( xNbB1, yNbB1 ), and ( xNbB2, yNbB2 ) are derived as follows:

( xA0, yA0 ) = ( xCb − 1, yCb + cbHeight ) (8‑635)

( xA1, yA1 ) = ( xCb − 1, yCb + cbHeight − 1 ) (8‑636)

( xB0, yB0 ) = ( xCb + cbWidth , yCb − 1 ) (8‑637)

( xB1, yB1 ) = ( xCb + cbWidth − 1, yCb − 1 ) (8‑638)

( xB2, yB2 ) = ( xCb − 1, yCb − 1 ) (8‑639)

1. The following applies for ( xNbAk, yNbAk ) from ( xNbA0, yNbA0 ) to ( xNbA1, yNbA1 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbAk, yNbAk ) as inputs, and the output is assigned to the block availability flag availableAk.
* When availableAk is equal to TRUE and MotionModelIdc[ xNbAk ][ yNbAk ] is greater than 0 and availableFlagA is equal to FALSE, the following applies:
  + - The variable ( xNb, yNb ) is set equal to ( CbPosX[ xNbAk ][ yNbAk ], CbPosY[ xNbAk ][ yNbAk ] ), nbW is set equal to CbWidth[ xNbAk ][ yNbAk ],and nbH is set equal to CbHeight[ xNbAk ][ yNbAk ].
    - If PredFlagLX[ xNbAk ][ yNbAk ] is equal to 1 and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagA is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLX[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑640)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑641)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑642)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑643)

* + - Otherwise if PredFlagLY[ xNbAk ][ yNbAk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbAk ][ yNbAk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagA is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLY[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLY[ 0 ] (8‑644)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLY[ 1 ] (8‑645)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLY[ 2 ] (8‑646)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑647)

1. The following applies for ( xNbBk, yNbBk ) from ( xNbB0, yNbB0 ) to ( xNbB2, yNbB2 ):

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbBk, yNbBk ) as inputs, and the output is assigned to the block availability flag availableBk.
* When availableBk is equal to TRUE and MotionModelIdc[ xNbBk ][ yNbBk ] is greater than 0 and availableFlagB is equal to FALSE, the following applies:
  + - The variable ( xNb, yNb ) is set equal to ( CbPosX[ xNbBk ][ yNbBk ], CbPosY[ xNbBk ][ yNbBk ] ), nbW is set equal to CbWidth[ xNbBk ][ yNbBk ],and nbH is set equal to CbHeight[ xNbBk ][ yNbBk ].
    - If PredFlagLX[ xNbBk ][ yNbBk ] is equal to 1 and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagB is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLX[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLX[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑648)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑649)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑650)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑651)

* + - Otherwise if PredFlagLY[ xNbBk ][ yNbBk ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, the following applies:
      * The variable availableFlagB is set equal to TRUE
      * The derivation process for luma affine control point motion vectors from a neighbouring block as specified in clause 8.5.5.5 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width and height (cbWidth, cbHeight), the neighbouring luma coding block location ( xNb, yNb ), the neighbouring luma coding block width and height (nbW, nbH), and the number of control point motion vectors numCpMv as input, the control point motion vector predictor candidates cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvpLY[ cpIdx ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvpLY[ cpIdx ] with cpIdx = 0 .. numCpMv − 1 as output.
      * The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLY[ 0 ] (8‑652)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLY[ 1 ] (8‑653)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLY[ 2 ] (8‑654)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑655)

1. When numCpMvpCandLX is less than 2, the following applies

* The derivation process for constructed affine control point motion vector prediction candidate as specified in clause 8.5.5.8 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, and the reference index of the current coding unit refIdxLX as inputs, and the availability flag availableConsFlagLX, the availability flags availableFlagLX[ cpIdx ] and cpMvpLX[ cpIdx ] with cpIdx = 0..numCpMv − 1 as outputs.
* When availableConsFlagLX is equal to 1, and numCpMvpCandLX is equal to 0, the following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ 0 ] (8‑656)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ 1 ] (8‑657)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ 2 ] (8‑658)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑659)

1. The following applies for cpIdx = 0..numCpMv − 1:

* When numCpMvpCandLX is less than 2 and availableFlagLX[ cpIdx ] is equal to 1, the following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = cpMvpLX[ cpIdx ] (8‑660)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = cpMvpLX[ cpIdx ] (8‑661)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = cpMvpLX[ cpIdx ] (8‑662)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑663)

1. When numCpMvpCandLX is less than 2, the following applies:

* The derivation process for temporal luma motion vector prediction as specified in clause 8.5.2.11 is with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight and refIdxLX as inputs, and with the output being the availability flag availableFlagLXCol and the temporal motion vector predictor mvLXCol.
* When availableFlagLXCol is equal to 1, the following applies:
* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLXCol, rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded mvLXCol as output.
* The following assignments are made:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvLXCol (8‑664)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvLXCol (8‑665)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvLXCol (8‑666)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑667)

1. When numCpMvpCandLX is less than 2, the following is repeated until numCpMvpCandLX is equal to 2, with mvZero[0] and mvZero[1] both being equal to 0:

cpMvpListLX[ numCpMvpCandLX ][ 0 ] = mvZero (8‑668)

cpMvpListLX[ numCpMvpCandLX ][ 1 ] = mvZero (8‑669)

cpMvpListLX[ numCpMvpCandLX ][ 2 ] = mvZero (8‑670)

numCpMvpCandLX = numCpMvpCandLX + 1 (8‑671)

The affine control point motion vector predictor cpMvpLX with X being 0 or 1 is derived as follows:

cpMvpLX = cpMvpListLX[ mvp\_lX\_flag[ xCb ][ yCb ] ] (8‑672)

#### Derivation process for constructed affine control point motion vector prediction candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the current luma coding block,
* the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,

Output of this process are:

* the availability flag of the constructed affine control point motion vector prediction candidiates availableConsFlagLX with X being 0 or 1,
* the availability flags availableFlagLX[ cpIdx ] with cpIdx = 0..2 and X being 0 or 1,
* the constructed affine control point motion vector prediction candidiates cpMvLX[ cpIdx ] with cpIdx = 0..numCpMv − 1 and X being 0 or 1.

The first (top-left) control point motion vector cpMvLX[ 0 ] and the availability flag availableFlagLX[ 0 ] are derived in the following ordered steps:

1. The sample locations ( xNbB2, yNbB2 ), ( xNbB3, yNbB3 ) and ( xNbA2, yNbA2 ) are set equal to ( xCb − 1, yCb − 1 ), ( xCb , yCb − 1 ) and ( xCb − 1, yCb ), respectively.
2. The availability flag availableFlagLX[ 0 ] is set equal to 0 and both components of cpMvLX[ 0 ] are set equal to 0.
3. The following applies for ( xNbTL, yNbTL ) withTL being replaced by B2, B3, and A2:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbTL, yNbTL ) as inputs, and the output is assigned to the coding block availability flag availableTL.
* When availableTL is equal to TRUE and availableFlagLX[ 0 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbTL ][ yNbTL ] is equal to 1, and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbTL ][ yNbTL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLX[ xNbTL ][ yNbTL ] (8‑673)

* Otherwise, when PredFlagLY[ xNbTL ][ yNbTL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbTL ][ yNbTL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 0 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 0 ] = MvLY[ xNbTL ][ yNbTL ] (8‑674)

* When availableFlagLX[ 0 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 0 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvLX[ 0 ] as output.

The second (top-right) control point motion vector cpMvLX[ 1 ] and the availability flag availableFlagLX[ 1 ] are derived in the following ordered steps:

1. The sample locations ( xNbB1, yNbB1 ) and ( xNbB0, yNbB0 ) are set equal to ( xCb + cbWidth − 1, yCb − 1 ) and ( xCb + cbWidth, yCb − 1 ), respectively.
2. The availability flag availableFlagLX[ 1 ] is set equal to 0 and both components of cpMvLX[ 1 ] are set equal to 0.
3. The following applies for ( xNbTR, yNbTR ) withTR being replaced by B1 and B0:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbTR, yNbTR ) as inputs, and the output is assigned to the coding block availability flag availableTR.
* When availableTR is equal to TRUE and availableFlagLX[ 1 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbTR ][ yNbTR ] is equal to 1, and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbTR ][ yNbTR ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLX[ xNbTR ][ yNbTR ] (8‑675)

* Otherwise, when PredFlagLY[ xNbTR ][ yNbTR ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbTR ][ yNbTR ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 1 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 1 ] = MvLY[ xNbTR ][ yNbTR ] (8‑676)

* When availableFlagLX[ 1 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 1 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvLX[ 1 ] as output.

The third (bottom-left) control point motion vector cpMvLX[ 2 ] and the availability flag availableFlagLX[ 2 ] are derived in the following ordered steps:

1. The sample locations ( xNbA1, yNbA1 ) and ( xNbA0, yNbA0 ) are set equal to ( xCb − 1, yCb + cbHeight − 1 ) and ( xCb − 1, yCb + cbHeight ), respectively.
2. The availability flag availableFlagLX[ 2 ] is set equal to 0 and both components of cpMvLX[ 2 ] are set equal to 0.
3. The following applies for ( xNbBL, yNbBL ) with BL being replaced by A1 and A0:

* The availability derivation process for a coding block as specified in clause  6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the luma location ( xNbY, yNbY ) set equal to ( xNbBL, yNbBL ) as inputs, and the output is assigned to the coding block availability flag availableBL.
* When availableBL is equal to TRUE and availableFlagLX[ 2 ] is equal to 0, the following applies:
* If PredFlagLX[ xNbBL ][ yNbBL ] is equal to 1, and DiffPicOrderCnt( RefPicList[ X ][ RefIdxLX[ xNbBL ][ yNbBL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLX[ xNbBL ][ yNbBL ] (8‑677)

* Otherwise, when PredFlagLY[ xNbBL ][ yNbBL ] (with Y = !X) is equal to 1 and DiffPicOrderCnt( RefPicList[ Y ][ RefIdxLY[ xNbBL ][ yNbBL ] ], RefPicList[ X ][ refIdxLX ] ) is equal to 0, availableFlagLX[ 2 ] is set equal to 1 and the following assignments are made:

cpMvLX[ 2 ] = MvLY[ xNbBL ][ yNbBL ] (8‑678)

* When availableFlagLX[ 2 ] is equal to 1, the rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to cpMvLX[ 2 ], rightShift set equal to ( MvShift + 2 ), and leftShift set equal to ( MvShift + 2 ) as inputs and the rounded cpMvLX[ 2 ] as output.

The variable availableConsFlagLX is derived as follows:

* + If availableFlagLX[ 0 ] is equal to 1 and availableFlagLX[ 1 ] is equal to 1 and availableFlagLX[ 2 ] is equal to 1, availableConsFlagLX is set equal to 1
  + Otherwise, if availableFlagLX[ 0 ] is equal to 1, and availableFlagLX[ 1 ] is equal to 1, and MotionModelIdc[ xCb ][ yCb ] is equal to 1, availableConsFlagLX is set equal to 1.
  + Otherwise, availableConsFlagLX is set equal to 0.

#### Derivation process for motion vector arrays from affine control point motion vectors

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* two variables cbWidth and cbHeight specifying the width and the height of the luma coding block,
* the number of control point motion vectors numCpMv,
* the control point motion vectors cpMvLX[ cpIdx ], with cpIdx = 0..numCpMv − 1 and X being 0 or 1,
* the reference index refIdxLX and X being 0 or 1,
* the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY.

Outputs of this process are:

* the luma subblock motion vector array mvLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1,
* the chroma subblock motion vector array mvCLX[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, ySbIdx = 0 .. numSbY − 1 and X being 0 or 1.

The following assignments are made for x = xCb..xCb + cbWidth − 1 and y = yCb..yCb + cbHeight − 1:

CpMvLX[ x ][ y ][ 0 ] = cpMvLX[ 0 ] (8‑679)

CpMvLX[ x ][ y ][ 1 ] = cpMvLX[ 1 ] (8‑680)

CpMvLX[ x ][ y ][ 2 ] = cpMvLX[ 2 ] (8‑681)

The variables log2CbW and log2CbH are derived as follows:

log2CbW = Log2( cbWidth ) (8‑682)

log2CbH = Log2( cbHeight ) (8‑683)

The variables mvScaleHor, mvScaleVer, dHorX and dVerX are derived as follows:

mvScaleHor = cpMvLX[ 0 ][ 0 ] << 7 (8‑684)

mvScaleVer = cpMvLX[ 0 ][ 1 ] << 7 (8‑685)

dHorX = ( cpMvLX[ 1 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbW ) (8‑686)

dVerX = ( cpMvLX[ 1 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbW ) (8‑687)

The variables dHorY and dVerY are derived as follows:

* If numCpMv is equal to 3, the following applies:

dHorY = ( cpMvLX[ 2 ][ 0 ] − cpMvLX[ 0 ][ 0 ] ) << ( 7 − log2CbH ) (8‑688)

dVerY = ( cpMvLX[ 2 ][ 1 ] − cpMvLX[ 0 ][ 1 ] ) << ( 7 − log2CbH ) (8‑689)

* Otherwise ( numCpMv is equal to 2), the following applies:

dHorY = − dVerX (8‑690)

dVerY = dHorX (8‑691)

The variable fallbackModeTriggered is set equal to 1 and modified as follows:

* The variables bxWX4, bxHX4, bxWXh, bxHXh, bxWXvand bxHXv are derived as follows:

maxW4 = Max( 0, Max( 4 \* ( 2048 + dHorX ),   
 Max( 4\*dHorY, 4 \* ( 2048 + dHorX ) + 4 \* dHorY ) ) ) (8‑692)

minW4 = Min( 0, Min( 4 \* ( 2048 + dHorX ),   
 Min( 4\*dHorY, 4 \* ( 2048 + dHorX ) + 4 \* dHorY ) ) ) (8‑693)

maxH4 = Max( 0, Max( 4 \* dVerX,   
 Max( 4\* ( 2048 + dVerY ), 4 \* dVerX + 4 \* ( 2048 + dVerY ) ) ) ) (8‑694)

minH4 = Min( 0, Min( 4 \* dVerX,   
 Min( 4\* ( 2048 + dVerY ), 4 \* dVerX + 4 \* ( 2048 + dVerY ) ) ) ) (8‑695)

bxWX4 = ( ( maxW4 − minW4 ) >> 11 ) + 9 (8‑696)

bxHX4 = ( ( maxH4 − minH4 ) >> 11 ) + 9 (8‑697)

bxWXh = ( (Max( 0, 4 \* ( 2048 + dHorX ) ) − Min( 0, 4 \* ( 2048 + dHorX ) ) ) >> 11 ) + 9 (8‑698)

bxHXh = ( ( Max( 0, 4 \* dVerX ) − Min( 0, 4 \* dVerX ) ) >> 11 ) + 9 (8‑699)

bxWXv = ( ( Max( 0, 4 \* dHorY ) − Min( 0, 4 \* dHorY ) ) >> 11 ) + 9 (8‑700)

bxHXv =  ( ( Max( 0, 4 \* ( 2048 + dVerY ) ) − Min( 0, 4 \* ( 2048 + dVerY ) ) ) >> 11 ) + 9 (8‑701)

* If inter\_pred\_idc[ xCb ][ yCb ] is equal to PRED\_BI and bxWX4 \* bxHX4 is less than or equal to 225, fallbackModeTriggered is set equal to 0.
* Otherwise, if both bxWXh \* bxHXh is less than or equal to 165 and bxWXv \* bxHXv is less than or equal to 165, fallbackModeTriggered is set equal to 0.

For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:

* The variables xPosCb and yPosCb are derived as follows
* If fallbackModeTriggered is equal to 1, the following applies:

xPosCb = ( cbWidth >> 1 ) (8‑702)

yPosCb = ( cbHeight >> 1 ) (8‑703)

* Otherwise (fallbackModeTriggered is equal to 0), the following applies:

xPosCb = 2 + ( xSbIdx << 2 ) (8‑704)

yPosCb = 2 + ( ySbIdx << 2 ) (8‑705)

* The luma motion vector mvLX[ xSbIdx ][ ySbIdx ] is derived as follows :

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = ( mvScaleHor + dHorX \* xPosCb + dHorY \* yPosCb ) (8‑706)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = ( mvScaleVer + dVerX \* xPosCb + dVerY \* yPosCb ) (8‑707)

* The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvLX[ xSbIdx ][ ySbIdx ], rightShift set equal to 7, and leftShift set equal to 0 as inputs and the rounded mvLX[ xSbIdx ][ ySbIdx ] as output.
* The motion vectors mvLX[ xSbIdx ][ ySbIdx ] are clipped as follows:

mvLX[ xSbIdx ][ ySbIdx ][ 0 ] = Clip3( −217, 217 − 1, mvLX[ xSbIdx ][ ySbIdx ][ 0 ] ) (8‑708)

mvLX[ xSbIdx ][ ySbIdx ][ 1 ] = Clip3( −217, 217 − 1, mvLX[ xSbIdx ][ ySbIdx ][ 1 ] ) (8‑709)

For xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:

* The average luma motion vector mvAvgLX is derived as follows:
  + If both SubWidthC and SubHeightC are equal to 1, the following applies:

mvAvgLX = mvLX[ xSbIdx ][ ySbIdx ] (8‑710)

* + Otherwise, the following applies:

mvAvgLX = mvLX[ ( xSbIdx >> ( SubWidthC − 1 ) << ( SubWidthC − 1 ) ) ]   
 [ (ySbIdx >> ( SubHeightC − 1 ) << ( SubHeightC − 1 ) ) ] +    
 mvLX[ ( xSbIdx >> (SubWidthC − 1 ) << ( SubWidthC − 1) ) + ( SubWidthC − 1 ) ] (8‑711)  
 [ ( ySbIdx >> ( SubHeightC − 1 ) << ( SubHeightC − 1 ) ) + ( SubHeightC − 1 ) ]

mvAvgLX[ 0 ] = ( mvAvgLX[ 0 ] + 1 − ( mvAvgLX[ 0 ] >= 0 ) ) >> 1 (8‑712)

mvAvgLX[ 1 ] = ( mvAvgLX[ 1 ] + 1 − ( mvAvgLX[ 1 ] >= 0 ) ) >> 1 (8‑713)

* The derivation process for chroma motion vectors in clause 8.5.2.13 is invoked with mvAvgLX and refIdxLX as inputs, and the chroma motion vector mvCLX[ xSbIdx ][ ySbIdx ] as output.

[Ed. (BB): This way four 2x2 chroma subblocks (4x4 chroma block) share the same motion vector which is derived from the average of two 4x4 luma subblock motion vectors. In the decoding process motion compensation is still performed on 2x2 chroma blocks which is however a motion compensation on a chroma 4x4 block because all chroma MVs inside a 4x4 chroma block are the same. I would prefer an editorial change that makes it more clear that affine chroma MC is performed on 4x4 chroma blocks.]

#### Derivation process for bi-prediction weight index for constructed affine control potin motion vector merging candidates

Inputs to this process are:

* three bi-prediction weight indices bcwIdxCorner0, bcwIdxCorner1 and bcwIdxCorner2.

Output of this process is:

* the bi-prediction weight index bcwIdxConst.

The bi-prediction weight index bcwIdxConst is derived as follows:

* For X being replaced by either 0, 1 or 2, bcwIdxGroupX is set equal to bcwIdxGLut[ bcwIdxCornerX ] with bcwIdxGLut[ k ] = { 0, 0, 0, 1, 2 }.
* If bcwIdxCorner0 is equal to bcwIdxCorner1 and bcwIdxGroup0 is equal to bcwIdxGroup2, bcwIdxConst is set equal to bcwIdxCorner0.
* Otherwise, if bcwIdxCorner0 is equal to bcwIdxCorner2 and bcwIdxGroup0 is equal to bcwIdxGroup1, bcwIdxConst is set equal to bcwIdxCorner0.
* Otherwise, if bcwIdxCorner1 is equal to bcwIdxCorner2 and bcwIdxGroup1 is equal to bcwIdxGroup0, bcwIdxConst is set equal to bcwIdxCorner2.
* Otherwise, bcwIdxConst is set equal to 0.

### Decoding process for inter blocks

#### General

This process is invoked when decoding a coding unit coded in inter prediction mode.

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,
* the motion vectors mvL0[ xSbIdx ][ ySbIdx ] and mvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* the refined motion vectors refMvL0[ xSbIdx ][ ySbIdx ] and refMvL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* the reference indices refIdxL0 and refIdxL1,
* the prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* the bi-prediction weight index bcwIdx,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an array predSamples of prediction samples.

Let predSamplesL0L, predSamplesL1L and predSamplesIntraL be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesL0Cb, predSamplesL1Cb, predSamplesL0Cr and predSamplesL1Cr, predSamplesIntraCb, and predSamplesIntraCr be (cbWidth / SubWidthC)x(cbHeight / SubHeightC) arrays of predicted chroma sample values.

* + The variable currPic specifies the current picture and the variable bdofFlag is derived as follows:
    - If all of the following conditions are true, bdofFlag is set equal to TRUE.
      * sps\_bdof\_enabled\_flag is equal to 1.
      * predFlagL0[ xSbIdx ][ ySbIdx ] and predFlagL1[ xSbIdx ][ ySbIdx ] are both equal to 1.
      * DiffPicOrderCnt( currPic, RefPicList[ 0 ][ refIdxL0 ] ) \* DiffPicOrderCnt( currPic, RefPicList[ 1 ][ refIdxL1 ] ) is less than 0.
      * MotionModelIdc[ xCb ][ yCb ] is equal to 0.
      * merge\_subblock\_flag[ xCb ][ yCb ] is equal to 0.
      * sym\_mvd\_flag[ xCb ][ yCb ] is equal to 0.
      * BcwIdx[ xCb ][ yCb ] is equal to 0.
      * luma\_weight\_l0\_flag[ refIdxL0 ] and luma\_weight\_l1\_flag[ refIdxL1 ] are both equal to 0.
      * cbHeight is greater than or equal to 8
      * cIdx is equal to 0.
    - Otherwise, bdofFlag is set equal to FALSE.
* If numSbY is equal to 1 and numSbX is equal to 1 the following applies:
  + When bdofFlag is equal to TRUE, the variables numSbY, numSbX are modified as follows:

numSbX = ( cbWidth > 16 )  ?  ( cbWidth >> 4 )  :  1 (8‑714)

numSbY = ( cbHeight > 16 )  ?  ( cbHeight >> 4 )  :  1 (8‑715)

* + For X = 0..1, xSbIdx = 0..numSbX − 1 and ySbIdx = 0..numSbY − 1, the following applies:
    - predFlagLX[ xSbIdx ][ ySbIdx ] is set equal to predFlagLX[ 0 ][ 0 ].
    - refMvLX[ xSbIdx ][ ySbIdx ] is set equal to refMvLX[ 0 ][ 0 ].
    - mvLX[ xSbIdx ][ ySbIdx ] is set equal to mvLX[ 0 ][ 0 ].

The width and the height of the current coding sublock sbWidth, sbHeight in luma samples are derived as follows:

sbWidth  =  cbWidth / numSbX (8‑716)

sbHeight  =  cbHeight / numSbY (8‑717)

For each coding subblock at subblock index ( xSbIdx, ySbIdx ) with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1, the following applies:

* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

( xSb, ySb )  =  ( xCb + xSbIdx \* sbWidth, yCb + ySbIdx \* sbHeight ) (8‑718)

* For X being each of 0 and 1, when predFlagLX[ xSbIdx ][ ySbIdx ] 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 clause 8.5.6.2 with X and refIdxLX as inputs.
  + The motion vector offset mvOffset is set equal to refMvLX[ xSbIdx ][ xSbIdx ] − mvLX[ xSbIdx ][ ySbIdx ].
  + When one or more of the following conditions is true, mvOffset[ 0 ] is set equal to 0:
    - xSb is not equal to xCb and mvOffset[ 0 ] is less than 0
    - ( xSb + sbWidth ) is not equal to ( xCb + cbWidth) and mvOffset[ 0 ] is greater than 0
  + When one or more of the following conditions is true, mvOffset[ 1 ] is set equal to 0:
    - ySb is not equal to yCb and mvOffset[ 1 ] is less than 0
    - ( ySb + sbHeight ) is not equal to ( yCb + cbHeight ) and mvOffset[ 1 ] is greater than 0
  + If cIdx is equal to 0, the following applies:
    - The array predSamplesLXL is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xSb, ySb ), the coding subblock width sbWidth, the coding subblock height sbHeight in luma samples, the luma motion vector offset mvOffset, the refined luma motion vector refMvLX[ xSbIdx ][ xSbIdx ], the reference array refPicLXL, bdofFlag, and cIdx as inputs.
  + Otherwise if cIdx is equal to 1, the following applies:
    - The array predSamplesLXCb is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xSb, ySb ), the coding subblock width sbWidth / SubWidthC, the coding subblock height sbHeight / SubHeightC, the chroma motion vector offset mvOffset, the refined chroma motion vector refMvLX[ xSbIdx ][ xSbIdx ], the reference array refPicLXCb, bdofFlag, and cIdx as inputs.
  + Otherwise (cIdx is equal to 2), the following applies:
    - The array predSamplesLXCr is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xSb, ySb ), the coding subblock width sbWidth / SubWidthC, the coding subblock height sbHeight / SubHeightC, the chroma motion vector offset mvOffset, the refined chroma motion vector refMvLX[ xSbIdx ][ xSbIdx ], the reference array refPicLXCr, bdofFlag, and cIdx as inputs.
* If bdofFlag is equal to TRUE, the following applies:
* The variable shift is set equal to Max( 2, 14 − BitDepthY ).
* The variables sbDiffThres, bdofBlkDiffThres, and sbSumDiff are derived as follows:

sbDiffThres = ( 1 << ( BitDepthY − 8 + shift ) ) \* sbWidth\*sbHeight (8‑719)

bdofBlkDiffThres = 1 << ( BitDepthY − 3 + shift ) (8‑720)

sbSumDiff = 0 (8‑721)

* For xIdx = 0..(sbWidth >> 2 ) − 1 and yIdx = 0..( sbHeight >> 2 ) − 1, the variables bdofBlkSumDiff and the bidirectional optical flow utilization flag bdofUtilizationFlag[ xIdx ][ yIdx ] are derived as follows:

bdofBlkSumDiff = ( predSamplesL0L[ ( xIdx << 2 ) + 1 + i ][ ( yIdx << 2 ) + 1 + j ] −  (8‑722)  
 predSamplesL1L[ ( xIdx << 2 ) + 1 + i ][ ( yIdx << 2 ) + 1 + j] )

bdofUtilizationFlag[ xIdx ][ yIdx ] =  bdofBlkSumDiff >= bdofBlkDiffThres (8‑723)

sbSumDiff += bdofBlkSumDiff (8‑724)

* The variable sbBdofFlag is derived as follows:
* If sbSumDiff is less than sbDiffThres, sbBdofFlag is set equal to FALSE.
* Otherwise, sbBdofFlag set equal to TRUE.
* The array predSamples of prediction samples is derived as follows:
* If cIdx is equal to 0, the prediction samples inside the current luma coding subblock, predSamples[ xL + xSb ][ yL + ySb ] with xL = 0..sbWidth − 1 and yL = 0..sbHeight − 1, are derived as follows:
* If sbBdofFlag is equal to TRUE, the bidirectional optical flow sample prediction process as specified in clause 8.5.6.4 is invoked with nCbW set equal to the luma coding subblock width sbWidth, nCbH set equal to the luma coding subblock height sbHeight and the sample arrays predSamplesL0L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1 and bdofUtilizationFlag[ xIdx ][ yIdx ] with xIdx = 0..( sbWidth >> 2 ) − 1, yIdx = 0..( sbHeight >> 2 ) – 1 as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise (sbBdofFlag is equal to FALSE), the weighted sample prediction process as specified in clause 8.5.6.5 is invoked with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays predSamplesL0L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise, if cIdx is equal to 1, the prediction samples inside the current chroma component Cb coding subblock, predSamples[ xC + xSb / SubWidthC ][ yC + ySb / SubHeightC ] with xC = 0..sbWidth / SubWidthC − 1 and yC = 0..sbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.6.5 with nCbW set equal to sbWidth / SubWidthC, nCbH set equal to sbHeight / SubHeightC, the sample arrays predSamplesL0Cb and predSamplesL1Cb, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the prediction samples inside the current chroma component Cr coding subblock, predSamples[ xC + xSb / SubWidthC ][ yC + ySb / SubHeightC ] with xC = 0..sbWidth / SubWidthC − 1 and yC = 0..sbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.6.5 with nCbW set equal to sbWidth / SubWidthC, nCbH set equal to sbHeight / SubHeightC, the sample arrays predSamplesL0Cr and predSamplesL1Cr, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs.
* When cIdx is equal to 0, the following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

MvL0[ xSb + x ][ ySb + y ] = mvL0[ xSbIdx ][ ySbIdx ] (8‑725)

MvL1[ xSb + x ][ ySb + y ] = mvL1[ xSbIdx ][ ySbIdx ] (8‑726)

MvDmvrL0[ xSb + x ][ ySb + y ] = refMvL0[ xSbIdx ][ ySbIdx ] (8‑727)

MvDmvrL1[ xSb + x ][ ySb + y ] = refMvL1[ xSbIdx ][ ySbIdx ] (8‑728)

RefIdxL0[ xSb + x ][ ySb + y ] = refIdxL0 (8‑729)

RefIdxL1[ xSb + x ][ ySb + y ] = refIdxL1 (8‑730)

PredFlagL0[ xSb + x ][ ySb + y ] = predFlagL0[ xSbIdx ][ ySbIdx ] (8‑731)

PredFlagL1[ xSb + x ][ ySb + y ] = predFlagL1[ xSbIdx ][ ySbIdx ] (8‑732)

BcwIdx[ xSb + x ][ ySb + y ] = bcwIdx (8‑733)

When ciip\_flag[ xCb ][ yCb ] is equal to 1, the array predSamples of prediction samples is modified as follows:

* If cIdx is equal to 0, the following applies:
* The general intra sample prediction process as specified in clause 8.4.5.2.5 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb, yCb ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth and cbHeight, the coding block width nCbW and height nCbH set equal to cbWidth and cbHeight, and the variable cIdx as inputs, and the output is assigned to the (cbWidth)x(cbHeight) array predSamplesIntraL.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.6.6 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb, yCb ), the coding block width cbWidth, the coding block height cbHeight, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamples and predSamplesIntraL, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth)x(cbHeight) array predSamples.
* Otherwise, if cIdx is equal to 1, the following applies:
* The general intra sample prediction process as specified in clause 8.4.5.2.5 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb / SubWidthC , yCb / SubHeightC ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth / SubWidthC  and cbHeight / SubHeightC, the coding block width nCbW and height nCbH set equal to cbWidth / SubWidthC  and cbHeight / SubHeightC, and the variable cIdx as inputs, and the output is assigned to the (cbWidth / SubWidthC )x(cbHeight / SubHeightC) array predSamplesIntraCb.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.6.6 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb, yCb ), the coding block width cbWidth / SubWidthC , the coding block height cbHeight / SubHeightC, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamplesCb and predSamplesIntraCb, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth / SubWidthC )x(cbHeight / SubHeightC) array predSamples.
* Otherwise (cIdx is equal to 2), the following applies:
* The general intra sample prediction process as specified in clause 8.4.5.2.5 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb / SubWidthC , yCb / SubHeightC ), the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], the transform block width nTbW and height nTbH set equal to cbWidth / SubWidthC  and cbHeight / SubHeightC, the coding block width nCbW and height nCbH set equal to cbWidth / SubWidthC  and cbHeight / SubHeightC, and the variable cIdx as inputs, and the output is assigned to the (cbWidth / SubWidthC )x(cbHeight / SubHeightC) array predSamplesIntraCr.
* The weighted sample prediction process for combined merge and intra prediction as specified in clause 8.5.6.6 is invoked with the location ( xTbCmp, yTbCmp ) set equal to ( xCb, yCb ), the coding block width cbWidth / SubWidthC , the coding block height cbHeight / SubHeightC, the sample arrays predSamplesInter and predSamplesIntra set equal to predSamplesCr and predSamplesIntraCr, respectively, the intra prediction mode predModeIntra set equal to IntraPredModeY[ xCb ][ yCb ], and the colour component index cIdx as inputs, and the output is assigend to the (cbWidth / SubWidthC )x(cbHeight / SubHeightC) array predSamples.

#### Reference picture selection process

Inputs to this process are:

* a value X representing a reference list being equal to either 0 or 1,
* a reference index refIdxLX.

Output of this process is a reference picture consisting of a two-dimensional array of luma samples refPicLXL and two two-dimensional arrays of chroma samples refPicLXCb and refPicLXCr.

The output reference picture RefPicList[ X ][ refIdxLX ], where X is the value of X that this process is invoked for, consists of a pic\_width\_in\_luma\_samples by pic\_height\_in\_luma\_samples array of luma samples refPicLXL and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples refPicLXCb and refPicLXCr.

The reference picture sample arrays refPicLXL, refPicLXCb and refPicLXCr correspond to decoded sample arrays SL, SCb and SCr derived in clause 8.8 for a previously-decoded picture.

#### Fractional sample interpolation process

##### General

Inputs to this process are:

* a luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture,
* a variable sbWidth specifying the width of the current coding subblock,
* a variable sbHeight specifying the height of the current coding subblock,
* a motion vector offset mvOffset,
* a refined motion vector refMvLX,
* the selected reference picture sample array refPicLX,
* the bidirectional optical flow flag bdofFlag,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an (sbWidth +bdofOffset)x(sbHeight +bdofOffset) array predSamplesLX of prediction sample values.

The bidirectional optical flow boundary offset bdofOffset is derived as follows:

bdofOffset = bdofFlag ? 2 : 0 (8‑734)

The motion vector mvLX is set equal to ( refMvLX − mvOffset ).

* If cIdx is equal to 0, the following applies:
  + Let ( xIntL, yIntL ) be a luma location given in full-sample units and ( xFracL, yFracL ) be an offset given in 1/16-sample units. These variables are used only in this clause for specifying fractional-sample locations inside the reference sample arrays refPicLX.
  + The top-left coordinate of the bounding block for reference sample padding ( xSbIntL, ySbIntL ) is set equal to ( xSb + ( mvLX[ 0 ]  >>  4 ), ySb + ( mvLX[ 1 ]  >>  4 ) ).
  + For each luma sample location ( xL = 0..sbWidth − 1 +bdofOffset, yL = 0..sbHeight − 1 +bdofOffset ) inside the prediction luma sample array predSamplesLX, the corresponding prediction luma sample value predSamplesLX[ xL ][ yL ] is derived as follows:
* The variables xIntL, yIntL, xFracL and yFracL are derived as follows:

xIntL = xSb + ( refMvLX[ 0 ]  >>  4 ) + xL (8‑735)

yIntL = ySb + ( refMvLX[ 1 ]  >>  4 ) + yL (8‑736)

xFracL = refMvLX[ 0 ] & 15 (8‑737)

yFracL = refMvLX[ 1 ] & 15 (8‑738)

* + If bdofFlag is equal to TRUE and one or more of the following conditions are true, the prediction luma sample value predSamplesLX[ xL ][ yL ] is derived by invoking the luma integer sample fetching process as specified in clause 8.5.6.3.3 with ( xIntL, yIntL ), ( xFracL, yFracL ) and refPicLX as inputs:
    - xL is equal to 0.
    - xL is equal to sbWidth + 1.
    - yL is equal to 0.
    - yL is equal to sbHeight + 1.
  + Otherwise, the prediction luma sample value predSamplesLX[ xL ][ yL ] is derived by invoking the luma sample 8-tap interpolation filtering process as specified in clause 8.5.6.3.2 with ( xIntL, yIntL ), ( xFracL, yFracL ), ( xSbIntL, ySbIntL ), refPicLX, sbWidth, sbHeight and ( xSb, ySb ) as inputs.
* Otherwise (cIdx is not equal to 0), the following applies:
  + Let ( xIntC, yIntC ) be a chroma location given in full-sample units and ( xFracC, yFracC ) be an offset given in 1/32 sample units. These variables are used only in this clause for specifying general fractional-sample locations inside the reference sample arrays refPicLX.
  + The top-left coordinate of the bounding block for reference sample padding ( xSbIntC, ySbIntC ) is set equal to ( (xSb / SubWidthC ) + ( mvLX[ 0 ]  >>  5), ( ySb / SubHeightC ) + ( mvLX[ 1 ]  >>  5 ) ).
  + For each chroma sample location ( xC = 0..sbWidth − 1, yC = 0.. sbHeight − 1 ) inside the prediction chroma sample arrays predSamplesLX, the corresponding prediction chroma sample value predSamplesLX[ xC ][ yC ] is derived as follows:
* The variables xIntC, yIntC, xFracC and yFracC are derived as follows:

xIntC = ( xSb / SubWidthC ) + ( refMvLX[ 0 ]  >>  5 ) + xC (8‑739)

yIntC = ( ySb / SubHeightC ) + ( refMvLX[ 1 ]  >>  5 ) + yC (8‑740)

xFracC = refMvLX[ 0 ] & 31 (8‑741)

yFracC = refMvLX[ 1 ] & 31 (8‑742)

* + The prediction sample value predSamplesLX[ xC ][ yC ] is derived by invoking the process specified in clause 8.5.6.3.4 with ( xIntC, yIntC ), ( xFracC, yFracC ), ( xSbIntC, ySbIntC ), sbWidth, sbHeight and refPicLX as inputs.

##### Luma sample interpolation filtering process

Inputs to this process are:

* a luma location in full-sample units ( xIntL, yIntL ),
* a luma location in fractional-sample units ( xFracL, yFracL ),
* a luma location in full-sample units ( xSbIntL, ySbIntL ) specifying the top-left sample of the bounding block for reference sample padding relative to the top‑left luma sample of the reference picture,
* the luma reference sample array refPicLXL,
* a variable sbWidth specifying the width of the current subblock,
* a variable sbHeight specifying the height of the current subblock,
* a luma location ( xSb, ySb ) specifying the top-left sample of the current subblock relative to the top‑left luma sample of the current picture,

Output of this process is a predicted luma sample value predSampleLXL

The variables shift1, shift2 and shift3 are derived as follows:

* The variable shift1 is set equal to Min( 4, BitDepthY − 8 ), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max( 2, 14 − BitDepthY ).
* The variable picW is set equal to pic\_width\_in\_luma\_samples and the variable picH is set equal to pic\_height\_in\_luma\_samples.

The luma interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p equal to xFracL or  yFracL are derived as follows:

* If MotionModelIdc[ xSb ][ ySb ] is greater than 0, and sbWidth and sbHeight are both equal to 4, the luma interpolation filter coefficients fL[ p ] are specified in Table 8‑15.
* Otherwise, the luma interpolation filter coefficients fL[ p ] are specified in Table 8‑14.

The luma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..7:

xInti = Clip3( 0, picW − 1, sps\_ref\_wraparound\_enabled\_flag ?  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, xIntL + i − 3 ) : (8‑743)   
 xIntL + i − 3 )

yInti = Clip3( 0, picH − 1, yIntL + i − 3 ) (8‑744)

The luma locations in full-sample units are further modified as follows for i = 0..7:

xInti = Clip3( xSbIntL − 3, xSbIntL + sbWidth + 4, xInti ) (8‑745)

yInti = Clip3( ySbIntL − 3, ySbIntL + sbHeight + 4, yInti ) (8‑746)

The predicted luma sample value predSampleLXL is derived as follows:

* If both xFracLand yFracL are equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL = refPicLXL[ xInt3 ][ yInt3 ] << shift3 (8‑747)

* Otherwise, if xFracL is not equal to 0 and yFracL is equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL =   >>  shift1 (8‑748)

* Otherwise, if xFracL is equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:

predSampleLXL =   >>  shift1 (8‑749)

* Otherwise, if xFracL is not equal to 0 and yFracL is not equal to 0, the value of predSampleLXL is derived as follows:
* The sample array temp[ n ] with n = 0..7, is derived as follows:

temp[ n ] =   >>  shift1 (8‑750)

* The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL =   >>  shift2 (8‑751)

Table 8‑14 – Specification of the luma interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | | | | | | | |
| **fL[ p ][ 0 ]** | **fL[ p ][ 1 ]** | **fL[ p ][ 2 ]** | **fL[ p ][ 3 ]** | **fL[ p ][ 4 ]** | **fL[ p ][ 5 ]** | **fL[ p ][ 6 ]** | **fL[ p ][ 7 ]** |
| 1 | 0 | 1 | −3 | 63 | 4 | −2 | 1 | 0 |
| 2 | −1 | 2 | −5 | 62 | 8 | −3 | 1 | 0 |
| 3 | −1 | 3 | −8 | 60 | 13 | −4 | 1 | 0 |
| 4 | −1 | 4 | −10 | 58 | 17 | −5 | 1 | 0 |
| 5 | −1 | 4 | −11 | 52 | 26 | −8 | 3 | −1 |
| 6 | −1 | 3 | −9 | 47 | 31 | −10 | 4 | −1 |
| 7 | −1 | 4 | −11 | 45 | 34 | −10 | 4 | −1 |
| 8 | −1 | 4 | −11 | 40 | 40 | −11 | 4 | −1 |
| 9 | −1 | 4 | −10 | 34 | 45 | −11 | 4 | −1 |
| 10 | −1 | 4 | −10 | 31 | 47 | −9 | 3 | −1 |
| 11 | −1 | 3 | −8 | 26 | 52 | −11 | 4 | −1 |
| 12 | 0 | 1 | −5 | 17 | 58 | −10 | 4 | −1 |
| 13 | 0 | 1 | −4 | 13 | 60 | −8 | 3 | −1 |
| 14 | 0 | 1 | −3 | 8 | 62 | −5 | 2 | −1 |
| 15 | 0 | 1 | −2 | 4 | 63 | −3 | 1 | 0 |

Table 8‑15 – Specification of the luma interpolation filter coefficients fL[ p ] for each 1/16 fractional sample position p for affine motion mode.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | | | | | | | |
| **fL[ p ][ 0 ]** | **fL[ p ][ 1 ]** | **fL[ p ][ 2 ]** | **fL[ p ][ 3 ]** | **fL[ p ][ 4 ]** | **fL[ p ][ 5 ]** | **fL[ p ][ 6 ]** | **fL[ p ][ 7 ]** |
| 1 | 0 | 1 | −3 | 63 | 4 | −2 | 1 | 0 |
| 2 | 0 | 1 | −5 | 62 | 8 | −3 | 1 | 0 |
| 3 | 0 | 2 | −8 | 60 | 13 | −4 | 1 | 0 |
| 4 | 0 | 3 | −10 | 58 | 17 | −5 | 1 | 0 |
| 5 | 0 | 3 | −11 | 52 | 26 | −8 | 2 | 0 |
| 6 | 0 | 2 | −9 | 47 | 31 | −10 | 3 | 0 |
| 7 | 0 | 3 | −11 | 45 | 34 | −10 | 3 | 0 |
| 8 | 0 | 3 | −11 | 40 | 40 | −11 | 3 | 0 |
| 9 | 0 | 3 | −10 | 34 | 45 | −11 | 3 | 0 |
| 10 | 0 | 3 | −10 | 31 | 47 | −9 | 2 | 0 |
| 11 | 0 | 2 | −8 | 26 | 52 | −11 | 3 | 0 |
| 12 | 0 | 1 | −5 | 17 | 58 | −10 | 3 | 0 |
| 13 | 0 | 1 | −4 | 13 | 60 | −8 | 2 | 0 |
| 14 | 0 | 1 | −3 | 8 | 62 | −5 | 1 | 0 |
| 15 | 0 | 1 | −2 | 4 | 63 | −3 | 1 | 0 |

##### Luma integer sample fetching process

Inputs to this process are:

* a luma location in full-sample units ( xIntL, yIntL ),
* the luma reference sample array refPicLXL,

Output of this process is a predicted luma sample value predSampleLXL

The variable shift is set equal to Max( 2, 14 − BitDepthY ).

The variable picW is set equal to pic\_width\_in\_luma\_samples and the variable picH is set equal to pic\_height\_in\_luma\_samples.

The luma locations in full-sample units ( xInt, yInt ) are derived as follows:

xInt = Clip3( 0, picW − 1, sps\_ref\_wraparound\_enabled\_flag ? (8‑752)  
 ClipH( ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY, picW, xIntL ) : xIntL )

yInt = Clip3( 0, picH − 1, yIntL ) (8‑753)

The predicted luma sample value predSampleLXL is derived as follows:

predSampleLXL = refPicLXL[ xInt ][ yInt ] << shift3 (8‑754)

##### Chroma sample interpolation process

Inputs to this process are:

– a chroma location in full-sample units ( xIntC, yIntC ),

– a chroma location in 1/32 fractional-sample units ( xFracC, yFracC ),

– a chroma location in full-sample units ( xSbIntC, ySbIntC ) specifying the top-left sample of the bounding block for reference sample padding relative to the top‑left chroma sample of the reference picture,

* a variable sbWidth specifying the width of the current subblock,
* a variable sbHeight specifying the height of the current subblock,

– the chroma reference sample array refPicLXC.

Output of this process is a predicted chroma sample value predSampleLXC

The variables shift1, shift2 and shift3 are derived as follows:

– The variable shift1 is set equal to Min( 4, BitDepthC − 8 ), the variable shift2 is set equal to 6 and the variable shift3 is set equal to Max( 2, 14 − BitDepthC ).

– The variable picWC is set equal to pic\_width\_in\_luma\_samples / SubWidthC and the variable picHC is set equal to pic\_height\_in\_luma\_samples / SubHeightC.

The chroma interpolation filter coefficients fC[ p ] for each 1/32 fractional sample position p equal to xFracC or  yFracC are specified in Table 8‑16.

The variable xOffset is set equal to ( sps\_ref\_wraparound\_offset\_minus1 + 1 ) \* MinCbSizeY ) / SubWidthC.

The chroma locations in full-sample units ( xInti, yInti ) are derived as follows for i = 0..3:

xInti = Clip3( 0, picWC − 1, sps\_ref\_wraparound\_enabled\_flag ? ClipH( xOffset, picWC, xIntC + i − 1 ) : (8‑755)  
 xIntC + i − 1 )

yInti = Clip3( 0, picHC − 1, yIntC + i − 1 ) (8‑756)

The chroma locations in full-sample units ( xInti, yInti ) are further modified as follows for i = 0..3:

xInti = Clip3( xSbIntC − 1, xSbIntC + sbWidth + 2, xInti ) (8‑757)

yInti = Clip3( ySbIntC − 1, ySbIntC + sbHeight + 2, yInti ) (8‑758)

The predicted chroma sample value predSampleLXC is derived as follows:

– If both xFracC and yFracC are equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC = refPicLXC[ xInt1 ][ yInt1 ] << shift3 (8‑759)

– Otherwise if xFracC is not equal to 0 and yFracC is equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC =   >>  shift1 (8‑760)

– Otherwise if xFracC is equal to 0 and yFracC is not equal to 0, the value of predSampleLXC is derived as follows:

predSampleLXC =   >>  shift1 (8‑761)

– Otherwise if xFracC is not equal to 0 and yFracC is not equal to 0, the value of predSampleLXC is derived as follows:

* The sample array temp[ n ] with n = 0..3, is derived as follows:

temp[ n ] =   >>  shift1 (8‑762)

* The predicted chroma sample value predSampleLXC is derived as follows:

predSampleLXC =( fC[ ][ 0 ] \* temp[ 0 ] +  
  fC[  ][ 1 ] \* temp[ 1 ] +  
  fC[  ][ 2 ] \* temp[ 2 ] + (8‑763)  
  fC[  ][ 3 ] \* temp[ 3 ] ) >> shift2

Table 8‑16 – Specification of the chroma interpolation filter coefficients fC[ p ] for each 1/32 fractional sample position p.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Fractional sample position p** | **interpolation filter coefficients** | | | |
| **fC[ p ][ 0 ]** | **fC[ p ][ 1 ]** | **fC[ p ][ 2 ]** | **fC[ p ][ 3 ]** |
| 1 | −1 | 63 | 2 | 0 |
| 2 | −2 | 62 | 4 | 0 |
| 3 | -2 | 60 | 7 | −1 |
| 4 | −2 | 58 | 10 | −2 |
| 5 | −3 | 57 | 12 | −2 |
| 6 | −4 | 56 | 14 | −2 |
| 7 | −4 | 55 | 15 | −2 |
| 8 | −4 | 54 | 16 | −2 |
| 9 | −5 | 53 | 18 | −2 |
| 10 | −6 | 52 | 20 | −2 |
| 11 | −6 | 49 | 24 | −3 |
| 12 | −6 | 46 | 28 | −4 |
| 13 | −5 | 44 | 29 | −4 |
| 14 | −4 | 42 | 30 | −4 |
| 15 | −4 | 39 | 33 | −4 |
| 16 | −4 | 36 | 36 | −4 |
| 17 | −4 | 33 | 39 | −4 |
| 18 | −4 | 30 | 42 | −4 |
| 19 | −4 | 29 | 44 | −5 |
| 20 | −4 | 28 | 46 | −6 |
| 21 | −3 | 24 | 49 | −6 |
| 22 | −2 | 20 | 52 | −6 |
| 23 | −2 | 18 | 53 | −5 |
| 24 | −2 | 16 | 54 | −4 |
| 25 | −2 | 15 | 55 | −4 |
| 26 | −2 | 14 | 56 | −4 |
| 27 | −2 | 12 | 57 | −3 |
| 28 | −2 | 10 | 58 | −2 |
| 29 | −1 | 7 | 60 | −2 |
| 30 | 0 | 4 | 62 | −2 |
| 31 | 0 | 2 | 63 | −1 |

#### Bidirectional optical flow prediction process

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW + 2)x(nCbH + 2) luma prediction sample arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags predFlagL0 and predFlagL1,
* the reference indices refIdxL0 and refIdxL1,
* the bidirectional optical flow utilization flags bdofUtilizationFlag[ xIdx ][ yIdx ] with xIdx = 0..( nCbW >> 2 ) − 1, yIdx = 0..( nCbH >> 2 ) − 1.

Output of this process is the (nCbW)x(nCbH) array pbSamples of luma prediction sample values.

Variables bitDepth, shift1, shift2, shift3, shift4, offset4, and mvRefineThres are derived as follows:

* The variable bitDepth is set equal to BitDepthY.
* The variable shift1 is set to equal to Max( 6, bitDepth − 6 ).
* The variable shift2 is set to equal to Max( 4, bitDepth − 8 ).
* The variable shift3 is set to equal to Max( 1, bitDepth − 11 ).
* The variable shift4 is set equal to Max( 3, 15 − bitDepth ) and the variable offset4 is set equal to 1  <<  ( shift4 − 1 ).
* The variable mvRefineThres is set equal to 1 << Max( 5, bitDepth − 7 ).

For xIdx = 0..( nCbW >> 2 ) − 1 and yIdx = 0..( nCbH >> 2 ) − 1, the following applies:

* The variable xSb is set equal to ( xIdx << 2) + 1 and ySb is set equal to ( yIdx << 2 ) + 1.
* If bdofUtilizationFlag[ xIdx ][ yIdx ] is equal to FALSE, for x = xSb − 1..xSb + 2, y = ySb − 1.. ySb + 2, the prediction sample values of the current subblock are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 2bitDepth ) − 1, ( predSamplesL0[ x + 1 ][ y + 1 ] + offset4 + (8‑764)  
 predSamplesL1[ x + 1 ][ y + 1 ] )  >>  shift4 )

* Otherwise (bdofUtilizationFlag[ xIdx ][ yIdx ] is equal to TRUE), the prediction sample values of the current subblock are derived as follows:
  + For x =xSb − 1..xSb + 4, y = ySb − 1..ySb + 4, the following ordered steps apply:

1. The locations ( hx, vy ) for each of the corresponding sample locations ( x, y ) inside the prediction sample arrays are derived as follows:

hx = Clip3( 1, nCbW, x ) (8‑765)

vy = Clip3( 1, nCbH, y ) (8‑766)

1. The variables gradientHL0[ x ][ y ], gradientVL0[ x ][ y ], gradientHL1[ x ][ y ] and gradientVL1[ x ][ y ] are derived as follows:

gradientHL0[ x ][ y ]  =  (predSamplesL0[ hx + 1 ][vy] − predSampleL0[ hx − 1 ][ vy] ) >> shift1 (8‑767)

gradientVL0[ x ][ y ]  =   (predSampleL0[ hx ][ vy + 1 ] − predSampleL0[ hx][vy − 1 ] ) >> shift1 (8‑768)

gradientHL1[ x ][ y ]  =  (predSamplesL1[ hx + 1 ][vy] − predSampleL1[ hx − 1 ][ vy] ) >> shift1 (8‑769)

gradientVL1[ x ][ y ]  =   (predSampleL1[ hx ][ vy + 1 ] − predSampleL1[ hx][vy − 1 ] ) >> shift1 (8‑770)

1. The variables diff[ x ][ y ], tempH[ x ][ y ] and tempV[ x ][ y ] are derived as follows:

diff[ x ][ y ] = (predSamplesL0[ hx ][ vy ] >> shift2 ) − ( predSamplesL1[ hx ][ vy ] >> shift2 ) (8‑771)

tempH[ x ][ y ] = (gradientHL0[ x ][ y ] + gradientHL1[ x ][ y ] ) >> shift3 (8‑772)

tempV[ x ][ y ] = (gradientVL0[ x ][ y ] + gradientVL1[ x ][ y ] ) >> shift3 (8‑773)

* + The variables sGx2, sGy2, sGxGy, sGxdI and sGydI are derived as follows:

sGx2 = ΣiΣj( tempH[ xSb + i ][ ySb + j ] \* tempH[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑774)

sGy2 = ΣiΣj( tempV[ xSb + i ][ ySb + j ] \* tempV[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑775)

sGxGy = ΣiΣj( tempH[ xSb + i ][ ySb + j ] \* tempV[ xSb + i ][ ySb + j ] ) with i, j  −1..4 (8‑776)

sGxGym = sGxGy >> 12 (8‑777)

sGxGys =  sGxGy & ( ( 1 << 12 ) − 1 ) (8‑778)

sGxdI = ΣiΣj( −tempH[ xSb + i ][ ySb + j ] \* diff[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑779)

sGydI = ΣiΣj( −tempV[ xSb + i ][ ySb + j ] \* diff[ xSb + i ][ ySb + j ] ) with i, j = −1..4 (8‑780)

* + The horizontal and vertical motion offset of the current subblock are derived as:

vx = sGx2 > 0  ?  Clip3( −mvRefineThres, mvRefineThres, (8‑781)  
 −( sGxdI << 3 ) >> Floor( Log2( sGx2 ) ) )  :  0

vy = sGy2 > 0  ?  Clip3( −mvRefineThres, mvRefineThres, ( ( sGydI << 3 ) −  (8‑782)  
 ( ( vx \* sGxGym ) << 12 + vx \* sGxGys ) >> 1 ) >> Floor( Log2( sGx2 ) ) )  :  0

* + For x =xSb − 1..xSb + 2, y = ySb − 1..ySb + 2, the prediction sample values of the current sub-block are derived as follows:

bdofOffset = Round( ( vx \* ( gradientHL1[ x + 1 ][ y + 1 ] − gradientHL0[ x + 1 ][ y + 1 ] ) ) >> 1 ) (8‑783)  
 + Round( ( vy \* (gradientVL1[ x + 1 ][ y + 1 ] − gradientVL0[ x + 1 ][ y + 1 ] ) ) >> 1 )

[Ed. (JC): Round() operation is defined for float input. The Round() operation seems redundant here since the input is an integer value. To be confirmed by the proponent]

pbSamples[ x ][ y ] = Clip3( 0, ( 2bitDepth ) − 1, ( predSamplesL0[ x + 1 ][ y + 1 ]  + offset4 + (8‑784)  
 predSamplesL1[ x + 1 ][ y + 1 ] + bdofOffset )  >>  shift4 )

#### Weighted sample prediction process

##### General

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags, predFlagL0 and predFlagL1,
* the reference indices refIdxL0 and refIdxL1,
* the bi-prediction weight index bcwIdx,
* the variable cIdx specifying the colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

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 variable weightedPredFlag is derived as follows:

* If slice\_type is equal to P, weightedPredFlag is set equal to weighted\_pred\_flag.
* Otherwise (slice\_type is equal to B), weightedPredFlag is set equal to weighted\_bipred\_flag.

The following applies:

* If weightedPredFlag is equal to 0, the array pbSamples of the prediction samples is derived by invoking the default weighted sample prediction process as specified in clause 8.5.6.5.2 with the coding block width nCbW, the coding block height nCbH, two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1, the prediction list utilization flags predFlagL0 and predFlagL1, the bi-prediction weight index bcwIdx and the bit depth bitDepth as inputs.
* Otherwise (weightedPredFlag is equal to 1), the array pbSamples of the prediction samples is derived by invoking the weighted sample prediction process as specified in clause 8.5.6.5.3 with the coding block width nCbW, the coding block height nCbH, two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1, the prediction list utilization flags predFlagL0 and predFlagL1, the reference indices refIdxL0 and refIdxL1, the colour component index cIdx and the bit depth bitDepth as inputs.

##### Default weighted sample prediction process

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags predFlagL0 and predFlagL1,
* the reference indices refIdxL0 and refIdxL1,
* the bi-prediction weight index bcwIdx.
* the sample bit depth, bitDepth.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

Variables shift1, shift2, offset1, offset2, and offset3 are derived as follows:

* The variable shift1 is set equal to Max( 2, 14 − bitDepth ) and the variable shift2 is set equal to Max( 3, 15 − bitDepth ).
* The variable offset1 is set equal to 1  <<  ( shift1 − 1 ).
* The variable offset2 is set equal to 1  <<  ( shift2 − 1 ).
* The variable offset3 is set equal to 1  <<  ( shift2 + 2 ).

Depending on the values of predFlagL0 and predFlagL1, the prediction samples pbSamples[ x ][ y ] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

* If predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesL0[ x ][ y ] + offset1 )  >>  shift1 ) (8‑785)

* Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesL1[ x ][ y ] + offset1 )  >>  shift1 ) (8‑786)

* Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the following applies:
* If bcwIdx is equal to 0, the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, (8‑787)  
 ( predSamplesL0[ x ][ y ] + predSamplesL1[ x ][ y ] + offset2 )  >>  shift2 )

* Otherwise (bcwIdx is not equal to 0), the following applies:
* The variable w1 is set equal to bcwWLut[ bcwIdx ] with bcwWLut[ k ] = { 4, 5, 3, 10, −2 }.
* The variable w0 is set equal to ( 8 − w1 ).
* The prediction sample values are derived as follows.

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, (8‑788)  
 ( w0\*predSamplesL0[ x ][ y ] + w1\*predSamplesL1[ x ][ y ] + offset3 )  >>  (shift2+3) )

##### Explicit weighted sample prediction process

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW)x(nCbH) arrays predSamplesL0 and predSamplesL1,
* the prediction list utilization flags, predFlagL0 and predFlagL1,
* the reference indices, refIdxL0 and refIdxL1,
* the variable cIdx specifying the colour component index,
* the sample bit depth, bitDepth.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable shift1 is set equal to Max( 2, 14 − bitDepth ).

The variables log2Wd, o0, o1, w0 and w1 are derived as follows:

* If cIdx is equal to 0 for luma samples, the following applies:

log2Wd = luma\_log2\_weight\_denom + shift1 (8‑789)

w0 = LumaWeightL0[ refIdxL0 ] (8‑790)

w1 = LumaWeightL1[ refIdxL1 ] (8‑791)

o0 = luma\_offset\_l0[ refIdxL0 ] << ( BitDepthY − 8 ) (8‑792)

o1 = luma\_offset\_l1[ refIdxL1 ] << ( BitDepthY − 8 ) (8‑793)

* Otherwise (cIdx is not equal to 0 for chroma samples), the following applies:

log2Wd = ChromaLog2WeightDenom + shift1 (8‑794)

w0 = ChromaWeightL0[ refIdxL0 ][ cIdx − 1 ] (8‑795)

w1 = ChromaWeightL1[ refIdxL1 ][ cIdx − 1 ] (8‑796)

o0 = ChromaOffsetL0[ refIdxL0 ][ cIdx − 1 ] << ( BitDepthC − 8 ) (8‑797)

o1 = ChromaOffsetL1[ refIdxL1 ][ cIdx − 1 ] << ( BitDepthC − 8 ) (8‑798)

The prediction sample pbSamples[ x ][ y ] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

* If predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

if( log2Wd >= 1 )  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,  
 ( ( predSamplesL0[ x ][ y ] \* w0 + 2log2Wd − 1 ) >> log2Wd ) + o0 ) (8‑799)  
else  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1, predSamplesL0[ x ][ y ] \* w0 + o0 )

* Otherwise, if predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

if( log2Wd >= 1 )  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1,  
 ( ( predSamplesL1[ x ][ y ] \* w1 + 2log2Wd − 1 ) >> log2Wd ) + o1 ) (8‑800)  
else  
 pbSamples[ x ][ y ] = Clip3( 0, ( 1 << bitDepth ) − 1, predSamplesL1[ x ][ y ] \* w1 + o1 )

* Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1,  
 ( predSamplesL0[ x ][ y ] \* w0 + predSamplesL1[ x ][ y ] \* w1 +  
 ( ( o0 + o1 + 1 )  <<  log2Wd ) )  >>  ( log2Wd + 1 ) ) (8‑801)

#### Weighted sample prediction process for combined merge and intra prediction

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current luma coding block relative to the top left luma sample of the current picture,
* the width of the current coding block cbWidth,
* the height of the current coding block cbHeight,
* two (cbWidth)x(cbHeight) arrays predSamplesInter and predSamplesIntra,
* a variable cIdx specifying the colour component index.

Output of this process is the (cbWidth)x(cbHeight) array predSamplesComb of prediction sample values.

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 variable scallFact is derived as follows:

scallFact = ( cIdx = = 0 ) ? 0 : 1. (8‑802)

The neighbouring luma locations ( xNbA, yNbA ) and ( xNbB, yNbB ) are set equal to   
( xCb − 1, yCb − 1 + ( cbHeight << scallFact ) ) and ( xCb − 1  + (cbWidth << scallFact ), yCb − 1 ), respectively.

For X being replaced by either A or B, the variables availableX and isIntraCodedNeighbourX are derived as follows:

* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbX, yNbX ) as inputs, and the output is assigned to availableX.
* The variable isIntraCodedNeighbourX is derived as follows:
* If availableX is equal to TRUE and CuPredMode[ xNbX ][ yNbX ] is equal to MODE\_INTRA, isIntraCodedNeighbourX is set equal to TRUE.
* Otherwise, isIntraCodedNeighbourX is set equal to FALSE.

The weight w is derived as follows:

* If isIntraCodedNeighbourA and isIntraCodedNeighbourB are both equal to TRUE, w is set equal to 3.
* Otherwise, if isIntraCodedNeighbourA and isIntraCodedNeighbourB are both equal to to FALSE, w is set equal to 1.
* Otherwise, w is set equal to 2.

When cIdx is equal to 0 and slice\_lmcs\_enabled\_flag is equal to 1, predSamplesInter[ x ][ y ] with x = 0..cbWidth − 1 and y = 0..cbHeight − 1 are modified as follows:

idxY = predSamplesInter[ x ][ y ] >> Log2( OrgCW )   
predSamplesInter [ x ][ y ] = Clip1Y( LmcsPivot[ idxY ] +  (8‑803)  
 ( ScaleCoeff[ idxY ] \* ( predSamplesInter[ x ][ y ] − InputPivot[ idxY ] ) +   
 ( 1 << 10 ) ) >> 11 )

The prediction samples predSamplesComb[ x ][ y ] with x = 0..cbWidth − 1 and y = 0..cbHeight − 1 are derived as follows:

predSamplesComb[ x ][ y ] = ( w \* predSamplesIntra[ x ][ y ] +  (8‑804)  
 ( 4 − w ) \* predSamplesInter[ x ][ y ] + 2) >> 2

### Decoding process for triangle inter blocks

#### General

This process is invoked when decoding a coding unit with MergeTriangleFlag[ xCb ][ yCb ] equal to 1.

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* the luma motion vectors in 1/16 fractional-sample accuracy mvA and mvB,
* the chroma motion vectors mvCA and mvCB,
* the reference indices refIdxA and refIdxB,
* the prediction list flags predListFlagA and predListFlagB.

Outputs of this process are:

* an (cbWidth)x(cbHeight) array predSamplesL of luma prediction samples,
* an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCb of chroma prediction samples for the component Cb,
* an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCr of chroma prediction samples for the component Cr.

Let predSamplesLAL and predSamplesLBL be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesLACb, predSamplesLBCb, predSamplesLACr and predSamplesLBCr be (cbWidth / SubWidthC)x(cbHeight / SubHeightC) arrays of predicted chroma sample values.

The predSamplesL, predSamplesCb and predSamplesCr are derived by the following ordered steps:

1. For N being each of A and B, the following applies:

* The reference picture consisting of an ordered two-dimensional array refPicLNL of luma samples and two ordered two-dimensional arrays refPicLNCb and refPicLNCr of chroma samples is derived by invoking the process specified in clause 8.5.6.2 with X set equal to predListFlagN and refIdxX set equal to refIdxN as input.
* The array predSamplesLNL is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the luma coding block width sbWidth set equal to cbWidth, the luma coding block height sbHeight set equal to cbHeight, the motion vector offset mvOffset set equal to ( 0, 0 ), the motion vector mvLX set equal to mvN and the reference array refPicLXL set equal to refPicLNL, the variable bdofFlag set euqal to FALSE, and the variable cIdx is set equal to 0 as inputs.
* The array predSamplesLNCb is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the coding block width sbWidth set equal to cbWidth / SubWidthC, the coding block height sbHeight set equal to cbHeight / SubHeightC, the motion vector offset mvOffset set equal to ( 0, 0 ), the motion vector mvLX set equal to mvCN, and the reference array refPicLXCb set equal to refPicLNCb, the variable bdofFlag set euqal to FALSE, and the variable cIdx is set equal to 1 as inputs.
* The array predSamplesLNCr is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the coding block width sbWidth set equal to cbWidth / SubWidthC, the coding block height sbHeight set equal to cbHeight / SubHeightC, the motion vector offset mvOffset set equal to ( 0, 0 ), the motion vector mvLX set equal to mvCN, and the reference array refPicLXCr set equal to refPicLNCr, the variable bdofFlag set euqal to FALSE, and the variable cIdx is set equal to 2 as inputs.

1. The partition direction of merge triangle mode variable triangleDir is set equal to merge\_triangle\_split\_dir[ xCb ][ yCb ].
2. The prediction samples inside the current luma coding block, predSamplesL[ xL ][ yL ] with xL = 0..cbWidth − 1 and yL = 0..cbHeight − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.7.2 with the coding block width nCbW set equal to cbWidth, the coding block height nCbH set equal to cbHeight, the sample arrays predSamplesLAL and predSamplesLBL, and the variables triangleDir, and cIdx equal to 0 as inputs.
3. The prediction samples inside the current chroma component Cb coding block, predSamplesCb[ xC ][ yC ] with xC = 0..cbWidth / SubWidthC − 1 and yC = 0..cbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.7.2 with the coding block width nCbW set equal to cbWidth / SubWidthC, the coding block height nCbH set equal to cbHeight / SubHeightC, the sample arrays predSamplesLACb and predSamplesLBCb, and the variables triangleDir, and cIdx equal to 1 as inputs.
4. The prediction samples inside the current chroma component Cr coding block, predSamplesCr[ xC ][ yC ] with xC = 0..cbWidth / SubWidthC − 1 and yC = 0..cbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process for triangle merge mode specified in clause 8.5.7.2 with the coding block width nCbW set equal to cbWidth / SubWidthC, the coding block height nCbH set equal to cbHeight / SubHeightC, the sample arrays predSamplesLACr and predSamplesLBCr, and the variables triangleDir, and cIdx equal to 2 as inputs.
5. The motion vector storing process for merge triangle mode specified in clause 8.5.7.3 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth, the luma coding block height cbHeight, the partition direction triangleDir, the luma motion vectors mvA and mvB, the reference indices refIdxA and refIdxB, and the prediction list flags predListFlagA and predListFlagB as inputs.

#### Weighted sample prediction process for triangle merge mode

Inputs to this process are:

* two variables nCbW and nCbH specifying the width and the height of the current coding block,
* two (nCbW)x(nCbH) arrays predSamplesLA and predSamplesLB,
* a variable triangleDir specifying the partition direction,
* a variable cIdx specifying colour component index.

Output of this process is the (nCbW)x(nCbH) array pbSamples of prediction sample values.

The variable nCbR is derived as follows:

nCbR = ( nCbW > nCbH ) ? ( nCbW / nCbH ) : ( nCbH / nCbW ) (8‑805)

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.

Variables shift1 and offset1 are derived as follows:

– The variable shift1 is set equal to Max( 5, 17 − bitDepth).

– The variable offset1 is set equal to 1  <<  ( shift1 − 1 ).

Depending on the values of triangleDir, wS and cIdx, the prediction samples pbSamples[ x ][ y ] with x = 0..nCbW − 1 and y = 0..nCbH − 1 are derived as follows:

– The variable wIdx is derived as follows:

* If cIdx is equal to 0 and triangleDir is equal to 0, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 8, ( x / nCbR − y ) + 4 ) ) (8‑806)  
 : ( Clip3( 0, 8, ( x − y / nCbR ) + 4 ) )

* Otherwise, if cIdx is equal to 0 and triangleDir is equal to 1, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 8, ( nCbH − 1 − x / nCbR − y ) + 4 ) ) (8‑807)  
 ( Clip3( 0, 8, ( nCbW − 1 − x − y / nCbR ) + 4 ) )

* Otherwise, if cIdx is greater than 0 and triangleDir is equal to 0, the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 4, ( x / nCbR − y ) + 2 ) ) (8‑808)  
 : ( Clip3( 0, 4, ( x − y / nCbR ) + 2 ) )

* Otherwise (if cIdx is greater than 0 and triangleDir is equal to 1), the following applies:

wIdx = ( nCbW > nCbH ) ? ( Clip3( 0, 4, ( nCbH − 1 − x / nCbR − y ) + 2 ) ) (8‑809)  
 ( Clip3( 0, 4, ( nCbW − 1 − x − y / nCbR ) + 2 ) )

– The variable wValue specifying the weight of the prediction sample is derived using wIdx and cIdx as follows:

wValue = ( cIdx = = 0 ) ? Clip3( 0, 8, wIdx ) : Clip3( 0, 8, wIdx \* 2 ) (8‑810)

– The prediction sample values are derived as follows:

pbSamples[ x ][ y ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, ( predSamplesLA[ x ][ y ] \* wValue +  (8‑811)  
 predSamplesLB[ x ][ y ] \* ( 8 − wValue ) + offset1 ) >> shift1 )

#### Motion vector storing process for triangle merge mode

This process is invoked when decoding a coding unit with MergeTriangleFlag[ xCb ][ yCb ] equal to 1.

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable triangleDir specifying the partition direction,
* the luma motion vectors in 1/16 fractional-sample accuracy mvA and mvB,
* the reference indices refIdxA and refIdxB,
* the prediction list flags predListFlagA and predListFlagB.

The variables numSbX and numSbY specifying the number of 4x4 blocks in the current coding block in horizontal and vertical direction are set equal to numSbX = cbWidth >> 2 and numSbY = cbHeight >> 2.

The variable minSb is set equal to min( numSbX, numSbY ).

The variable cbRatio is derived as follows:

cbRatio = ( cbWidth > cbHeight ) ? ( cbWidth / cbHeight ) : ( cbHeight / cbWidth ) (8‑812)

The variable refIdxTempA is derived by invoking the reference picture mapping process for triangle merge mode specified in clause 8.5.7.4 with X set equal to predListFlagA and refIdxN set equal to refIdxA as inputs.

The variable refIdxTempB is derived by invoking the reference picture mapping process for triangle merge mode specified in clause 8.5.7.4 with the X set equal to predListFlagB and refIdxN set equal to refIdxB as inputs.

For each 4x4 subblock at subblock index ( xSbIdx, ySbIdx ) with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the following applies:

* The variables xIdx and yIdx are derived as follows:

xIdx = ( cbWidth > cbHeight ) ? ( xSbIdx / cbRatio ) : xSbIdx (8‑813)

yIdx = ( cbWidth > cbHeight ) ? ySbIdx : ( ySbIdx / cbRatio ) (8‑814)

* The variable sType is derived as follows:
* If triangleDir is equal to 0, the following applies:

sType = ( xIdx = = yIdx ) ? 2 : ( ( xIdx > yIdx ) ? 0 : 1 ) (8‑815)

* Otherwise (triangleDir is equal to 1), the following applies:

sType = ( xIdx + yIdx = = minSb ) ? 2 : ( ( xIdx + yIdx < minSb ) ? 0 : 1 ) (8‑816)

* Depending on the value of sType, the following assignments are made:
* If sType is equal to 0, the following applies:

predFlagL0 = ( predListFlagA = = 0 ) ? 1 : 0 (8‑817)

predFlagL1 = ( predListFlagA = = 0 ) ? 0 : 1 (8‑818)

refIdxL0 = ( predListFlagA = = 0 ) ? refIdxA : −1 (8‑819)

refIdxL1 = ( predListFlagA = = 0 ) ? −1 : refIdxA (8‑820)

mvL0[ 0 ] = ( predListFlagA = = 0 ) ? mvA[ 0 ] : 0 (8‑821)

mvL0[ 1 ] = ( predListFlagA = = 0 ) ? mvA[ 1 ] : 0 (8‑822)

mvL1[ 0 ] = ( predListFlagA = = 0 ) ? 0 : mvA[ 0 ] (8‑823)

mvL1[ 1 ] = ( predListFlagA = = 0 ) ? 0 : mvA[ 1 ] (8‑824)

* Otherwise, if sType is equal to 1, the following applies:

predFlagL0 = ( predListFlagB = = 0 ) ? 1 : 0 (8‑825)

predFlagL1 = ( predListFlagB = = 0 ) ? 0 : 1 (8‑826)

refIdxL0 = ( predListFlagB = = 0 ) ? refIdxB : −1 (8‑827)

refIdxL1 = ( predListFlagB = = 0 ) ? −1 : refIdxB (8‑828)

mvL0[ 0 ] = ( predListFlagB = = 0 ) ? mvB[ 0 ] : 0 (8‑829)

mvL0[ 1 ] = ( predListFlagB = = 0 ) ? mvB[ 1 ] : 0 (8‑830)

mvL1[ 0 ] = ( predListFlagB = = 0 ) ? 0 : mvB[ 0 ] (8‑831)

mvL1[ 1 ] = ( predListFlagB = = 0 ) ? 0 : mvB[ 1 ] (8‑832)

* Otherwise (sType is equal to 2), the following applies:
  + If predListFlagA + predListFlagB is equal to 1,

predFlagL0 = 1 (8‑833)

predFlagL1 = 1 (8‑834)

refIdxL0 = ( predListFlagA = = 0 ) ? refIdxA : refIdxB (8‑835)

refIdxL1 = ( predListFlagA = = 0 ) ? refIdxB : refIdxA (8‑836)

mvL0[ 0 ] = ( predListFlagA = = 0 ) ? mvA[ 0 ] : mvB[ 0 ] (8‑837)

mvL0[ 1 ] = ( predListFlagA = = 0 ) ? mvA[ 1 ] : mvB[ 1 ] (8‑838)

mvL1[ 0 ] = ( predListFlagA = = 0 ) ? mvB[ 0 ] : mvA[ 0 ] (8‑839)

mvL1[ 1 ] = ( predListFlagA = = 0 ) ? mvB[ 1 ] : mvA[ 1 ] (8‑840)

* + If predListFlagA + predListFlagB is equal to 0, the following applies:

predFlagL0 = 1 (8‑841)

predFlagL1 = ( refIdxTempA = = −1 && refIdxTempB = = −1 ) ? 0 : 1 (8‑842)

refIdxL0 = ( refIdxTempB != −1 ) ? refIdxA : (8‑843)  
 ( ( refIdxTempA != −1 ) ? refIdxB : refIdxA )

refIdxL1 = ( refIdxTempB != −1 ) ? refIdxTempB : (8‑844)  
 ( ( refIdxTempA != −1 ) ? refIdxTempA : −1 )

mvL0[ 0 ] = ( refIdxTempB != −1 ) ? mvA[ 0 ] : (8‑845)  
 ( ( refIdxTempA != −1 ) ? mvB[ 0 ] : mvA[ 0 ] )

mvL0[ 1 ] = ( refIdxTempB != −1 ) ? mvA[ 1 ] : (8‑846)  
 ( ( refIdxTempA != −1 ) ? mvB[ 1 ] : mvA[ 1 ] )

mvL1[ 0 ] = ( refIdxTempB != −1 ) ? mvB[ 0 ] : (8‑847)  
 ( ( refIdxTempA != −1 ) ? mvA[ 0 ] : 0 )

mvL1[ 1 ] = ( refIdxTempB != −1 ) ? mvB[ 1 ] : (8‑848)  
 ( ( refIdxTempA != −1 ) ? mvA[ 1 ] : 0 )

* + If predListFlagA + predListFlagB is equal to 2, the following applies:

predFlagL0 = ( refIdxTempA = = −1 && refIdxTempB = = −1 ) ? 0 : 1 (8‑849)

predFlagL1 = 1 (8‑850)

refIdxL0 = ( refIdxTempB != −1 ) ? refIdxTempB : (8‑851)  
 ( ( refIdxTempA != −1 ) ? refIdxTempA : −1 )

refIdxL1 = ( refIdxTempB != −1 ) ? refIdxA : (8‑852)  
 ( ( refIdxTempA != −1 ) ? refIdxB : refIdxA )

mvL0[ 0 ] = ( refIdxTempB != −1 ) ? mvB[ 0 ] : (8‑853)  
 ( ( refIdxTempA != −1 ) ? mvA[ 0 ] : 0 )

mvL0[ 1 ] = ( refIdxTempB != −1 ) ? mvB[ 1 ] : (8‑854)  
 ( ( refIdxTempA != −1 ) ? mvA[ 1 ] : 0 )

mvL1[ 0 ] = ( refIdxTempB != −1 ) ? mvA[ 0 ] : (8‑855)  
 ( ( refIdxTempA != −1 ) ? mvB[ 0 ] : mvA[ 0 ] )

mvL1[ 1 ] = ( refIdxTempB != −1 ) ? mvA[ 1 ] : (8‑856)  
 ( ( refIdxTempA != −1 ) ? mvB[ 1 ] : mvA[ 1 ] )

* The following assignments are made for x = 0..3 and y = 0..3:

MvL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = mvL0 (8‑857)

MvL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = mvL1 (8‑858)

RefIdxL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = refIdxL0 (8‑859)

RedIdxL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = refIdxL1 (8‑860)

PredFlagL0[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = predFlagL0 (8‑861)

PredFlagL1[ ( xSbIdx << 2 ) + x ][ ( ySbIdx << 2 ) + y] = predFlagL1 (8‑862)

#### Reference picture mapping process for triangle merge mode

Input to this process are:

* a variable X representing a reference list being equal to 0 or 1,
* a reference index refIdxN.

Output of this process is:

* a reference index refIdxTemp.

The variable refPicPoc is derived as follows:

refPicPoc = ( X = = 0 ) ? RefPicList[ 0 ][ refIdxN ] : RefPicList[ 1 ][ refIdxN ] (8‑863)

The reference picture list refPicListTemp is derived as follows:

refPicListTemp = ( X = = 0 ) ? RefPicList[ 1 ] : RefPicList[ 0 ] (8‑864)

The variable refIdxTemp is derived as follows:

* The variable mapStop is set equal to FALSE.
* For the variable refIdxm with m = 0..NumRefIdxActive[ 1 ] − 1, the following applies until mapStop is equal to FALSE:

refIdxTemp = ( refPicListTemp[ refIdxm ] = = refPicPoc ) ? refIdxm : −1 (8‑865)

mapStop = ( refIdxTemp != −1 ) ? TRUE : FALSE (8‑866)

### Decoding process for the residual signal of coding blocks coded in inter prediction mode

Inputs to this process are:

* a sample location ( xTb0, yTb0 ) specifying the top-left sample of the current transform block relative to the top‑left sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable cIdx specifying the colour component of the current block.

Output of this process is an (nTbW)x(nTbH) array resSamples.

The maximum transform block size maxTbSize is derived as follows:

maxTbSize = ( cIdx  = =  0 ) ? MaxTbSizeY : MaxTbSizeY / Min(SubWidthC, SubHeightC)[Ed. (JC), should be max TB size for chroma same to Luma for 422 and 444 case, to be confirmed] (8‑867)

The luma sample location is derived as follows:

( xTbY, yTbY ) = ( cIdx  = =  0 ) ? ( xTb0, yTb0 ) : ( xTb0 \* SubWidthC, yTb0 \* SubHeightC ) (8‑868)

[Ed. (JC), should bug fix needs to be confirmed]

Depending on maxTbSize, the following applies:

* If nTbW is greater than maxTbSize or nTbH is greater than maxTbSize, the following ordered steps apply.

1. The variables newTbW and newTbH are derived as follows:

newTbW = ( nTbW  >  maxTbSize ) ? ( nTbW / 2 ) : nTbW (8‑869)

newTbH = ( nTbH   >  maxTbSize ) ? ( nTbH / 2 ) :  nTbH (8‑870)

1. The decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
2. When nTbW is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, and the variable cIdx as inputs, and the output is a modified reconstructed picture .
3. When nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and the height nTbH set equal to newTbH, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.
4. Wwhen nTbW is greater than maxTbSize and nTbH is greater than maxTbSize, the decoding process process for the residual signal of coding units coded in inter prediction mode as specified in this clause is invoked with the location ( xTb0, yTb0 ) set equal to ( xTb0 + newTbW, yTb0 + newTbH ), the transform block width nTbW set equal to newTbW and height nTbH set equal to newTbH, and the variable cIdx as inputs, and the output is a modified reconstructed picture before in-loop filtering.

* Otherwise, the scaling and transformation process as specified in clause 8.7.2 is invoked with the luma location ( xTbY, yTbY ), the variable cIdx, the transform width nTbW and the transform height nTbH as inputs, and the output is an (nTbW)x(nTbH) array resSamples.

## Decoding process for coding units coded in IBC prediction mode

### General decoding process for coding units coded in IBC prediction mode

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable treeType specifying whether a single or a dual tree is used and if a dual tree is used, it specifies whether the current tree corresponds to the luma or chroma components.

Output of this process is a modified reconstructed picture before in-loop filtering.

The derivation process for quantization parameters as specified in clause 8.7.1 is invoked with the luma location ( xCb, yCb ), the width of the current coding block in luma samples cbWidth and the height of the current coding block in luma samples cbHeight, and the variable treeType as inputs.

The decoding process for coding units coded in ibc prediction mode consists of the following ordered steps:

1. The motion vector components of the current coding unit are derived as follows:
   * + If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the following applies:

* The derivation process for motion vector components as specified in clause 8.6.2.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight as inputs, and the luma motion vector mvL[ 0 ][ 0 ] as output.
* When treeType is equal to SINGLE\_TREE, the derivation process for chroma motion vectors in clause 8.6.2.5 is invoked with luma motion vector mvL[ 0 ][ 0 ] as input, and chroma motion vector mvC[ 0 ][ 0 ] as output.
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are both set equal to 1.
  + - Otherwise, if treeType is equal to DUAL\_TREE\_CHROMA, the following applies:
* The number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

numSbX = ( cbWidth >> 2 ) (8‑871)

numSbY = ( cbHeight >> 2 ) (8‑872)

* The chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] are derived as follows for xSbIdx = 0..numSbX − 1, ySbIdx = 0..numSbY − 1:
* The luma motion vector mvL[ xSbIdx ][ ySbIdx ] is derived as follows:
* The location ( xCuY, yCuY ) of the collocated luma coding unit is dervied as follows:

xCuY = xCb + xSbIdx\*4 (8‑873)

yCuY = yCb + ySbIdx\*4 (8‑874)

* If CuPredMode[ xCuY ][ yCuY ] is equal to MODE\_INTRA, the following applies.

mvL[ xSbIdx ][ ySbIdx ][ 0 ] = 0 (8‑875)

mvL[ xSbIdx ][ ySbIdx ][ 1 ] = 0 (8‑876)

predFlagL0[ xSbIdx ][ ySbIdx ] = 0 (8‑877)

predFlagL1[ xSbIdx ][ ySbIdx ] = 0 (8‑878)

* Otherwise ( CuPredMode[ xCuY ][ yCuY ] is equal to MODE\_IBC ), the following applies:

mvL[ xSbIdx ][ ySbIdx ][ 0 ]=MvL0[ xCuY ][ yCuY ][ 0 ] (8‑879)

mvL[ xSbIdx ][ ySbIdx ][ 1 ]=MvL0[ xCuY ][ yCuY ][ 1 ] (8‑880)

predFlagL0[ xSbIdx ][ ySbIdx ] = 1 (8‑881)

predFlagL1[ xSbIdx ][ ySbIdx ] = 0 (8‑882)

* The derivation process for chroma motion vectors in clause 8.6.2.5 is invoked with mvL[ xSbIdx ][ ySbIdx ] as inputs, and mvC[ xSbIdx ][ ySbIdx ] as output.
* It is a requirement of bitstream conformance that the chroma motion vector mvC[ xSbIdx ][ ySbIdx ] shall obey the following constraints:
* When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ) and the neighbouring chroma location ( xCb / SubWidthC + ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ), yCb / SubHeightC + ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) ) as inputs, the output shall be equal to TRUE.
* When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current chroma location ( xCurr, yCurr ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ) and the neighbouring chroma location ( xCb / SubWidthC + ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ) + cbWidth / SubWidthC − 1, yCb / SubHeightC + ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) + cbHeight / SubHeightC − 1 ) as inputs, the output shall be equal to TRUE.
* One or both of the following conditions shall be true:
* ( mvC[ xSbIdx ][ ySbIdx ][ 0 ] >> 5 ) + xSbIdx \* 2 + 2 is less than or equal to 0.
* ( mvC[ xSbIdx ][ ySbIdx ][ 1 ] >> 5 ) + ySbIdx \* 2 + 2 is less than or equal to 0.

1. The prediction samples of the current coding unit are derived as follows:

* If treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process for ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the luma motion vectors mvL[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1, the variable cIdx set equal to 0 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth)x(cbHeight) array predSamplesL of prediction luma samples as outputs.
* Otherwise if treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the prediction samples of the current coding unit are derived as follows:
  + - * The decoding process ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and the variable cIdx set equal to 1 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCb of prediction chroma samples for the chroma components Cb as outputs.
      * The decoding process for ibc blocks as specified in clause 8.6.3.1 is invoked with the luma coding block location ( xCb, yCb ), the luma coding block width cbWidth and the luma coding block height cbHeight, the number of luma coding subblocks in horizontal direction numSbX and in vertical direction numSbY, the chroma motion vectors mvC[ xSbIdx ][ ySbIdx ] with xSbIdx = 0..numSbX − 1, and ySbIdx = 0..numSbY − 1 and the variable cIdx set equal to 2 as inputs, and the ibc prediction samples (predSamples) that are an (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamplesCr of prediction chroma samples for the chroma components Cr as outputs.

1. The variables NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ] are set equal to numSbX and numSbY, respectively.
2. The residual samples of the current coding unit are derived as follows:

* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the luma location ( xCb, yCb ), the width nTbW set equal to the luma coding block width cbWidth, the height nTbH set equal to the luma coding block height cbHeight and the variable cIdxset equal to 0 as inputs, and the array resSamplesL as output.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC, yCb / SubHeightC ), the width nTbW set equal to the chroma coding block width cbWidth / SubWidthC, the height nTbH set equal to the chroma coding block height cbHeight / SubHeightC and the variable cIdxset equal to 1 as inputs, and the array resSamplesCb as output.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the decoding process for the residual signal of coding blocks coded in inter prediction mode as specified in clause 8.5.8 is invoked with the location ( xTb0, yTb0 ) set equal to the chroma location ( xCb / SubWidthC, yCb / SubHeightC ), the width nTbW set equal to the chroma coding block width cbWidth / SubWidthC, the height nTbH set equal to the chroma coding block height cbHeight / SubHeightC and the variable cIdxset equal to 2 as inputs, and the array resSamplesCr as output.

1. The reconstructed samples of the current coding unit are derived as follows:

* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_LUMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb, yCb ), the block width bWidth set equal to cbWidth, the block height bHeight set equal to cbHeight, the variable cIdx set equal to 0, the (cbWidth)x(cbHeight) array predSamples set equal to predSamplesL and the (cbWidth)x(cbHeight) array resSamples set equal to resSamplesL as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ), the block width bWidth set equal to cbWidth / SubWidthC, the block height bHeight set equal to cbHeight / SubHeightC, the variable cIdx set equal to 1, the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamples set equal to predSamplesCb and the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array resSamples set equal to resSamplesCb as inputs, and the output is a modified reconstructed picture before in-loop filtering.
* When treeType is equal to SINGLE\_TREE or treeType is equal to DUAL\_TREE\_CHROMA, the picture reconstruction process for a colour component as specified in clause 8.7.5 is invoked with the block location ( xB, yB ) set equal to ( xCb / SubWidthC, yCb / SubHeightC ), the block width bWidth set equal to cbWidth / SubWidthC, the block height bHeight set equal to cbHeight / SubHeightC, the variable cIdx set equal to 2, the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array predSamples set equal to predSamplesCr and the (cbWidth / SubWidthC)x(cbHeight / SubHeightC) array resSamples set equal to resSamplesCr as inputs, and the output is a modified reconstructed picture before in-loop filtering.

### Derivation process for motion vector components for IBC blocks

#### General

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vector in 1/16 fractional-sample accuracy mvL.

The luma motion vector mvL is derived as follows:

* The derivation process for IBC luma motion vector prediction as specified in clause 8.6.2.2 is invoked with the luma location ( xCb, yCb ), the variables cbWidth and cbHeight inputs, and the output being the luma motion vector mvL.
* When general\_merge\_flag[ xCb ][ yCb ] is equal to 0, the following applies:

1. The variable mvd is derived as follows:

mvd[ 0 ] = MvdL0[ xCb ][ yCb ][ 0 ] (8‑883)

mvd[ 1 ] = MvdL0[ xCb ][ yCb ][ 1 ] (8‑884)

1. The rounding process for motion vectors as specified in clause 8.5.2.14 is invoked with mvX set equal to mvL, rightShift set equal to MvShift + 2, and leftShift set equal to MvShift + 2 as inputs and the rounded mvL as output.
2. The luma motion vector mvL is modified as follows:

u[ 0 ] = ( mvL[ 0 ] + mvd[ 0 ] + 218 ) % 218 (8‑885)

mvL[ 0 ] = ( u[ 0 ] >= 217 ) ? ( u[ 0 ] − 218 ) : u[ 0 ] (8‑886)

u[ 1 ] = ( mvL[ 1 ] + mvd[ 1 ] + 218 ) % 218  (8‑887)

mvL[ 1 ] = ( u[ 1 ] >= 217 ) ? ( u[ 1 ] − 218 ) : u[ 1 ] (8‑888)

NOTE 1– The resulting values of mvL[ 0 ] and mvL[ 1 ] as specified above will always be in the range of −217 to 217 − 1, inclusive.

The updating process for the history-based motion vector predictor list as specified in clause 8.6.2.6 is invoked with luma motion vector mvL.

The top-left location inside the reference block ( xRefTL, yRefTL ) and the bottom-right location inside the reference block ( xRefBR, yRefBR ) are derived as follows:

( xRefTL, yRefTL ) = ( xCb + ( mvL[ 0 ] >> 4 ), yCb + ( mvL[ 1 ] >> 4 ) ) (8‑889)

( xRefBR, yRefBR ) = ( xRefTL + cbWidth − 1, yRefTL + cbHeight − 1 ) (8‑890)

It is a requirement of bitstream conformance that the luma motion vector mvL shall obey the following constraints:

* The motion vector mvL shall point to a reference block that is available, i.e. for ( xRefTL, yRefTL ) and ( xRefBR, yRefBR ), all of the following conditions shall be true:
  + When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xRefTL, yRefTL ) as inputs, the output shall be equal to TRUE.
  + When the derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xRefBR, yRefBR ) as inputs, the output shall be equal to TRUE.
* The motion vector mvL shall point to a reference block that is outside (left and/or above) and not overlapping with the current coding bock, i.e. one or both the following conditions shall be true:
  + The value of ( mvL[ 0 ] >> 4 ) + cbWidth is less than or equal to 0.
  + The value of ( mvL[ 1 ] >> 4 ) + cbHeight is less than or equal to 0.
* The motion vector mvL shall point to a reference block that is fully contained in the same coding tree block as the current block or fully contained in a block to the left with the same height of the current coding tree block and a width equal to 128 luma samples, i.e. all of the following conditions shall be true:

yRefTL >> CtbLog2SizeY = yCb >> CtbLog2SizeY (8‑891)

yRefBR >> CtbLog2SizeY = yCb >> CtbLog2SizeY (8‑892)

xRefTL >> CtbLog2SizeY >= ( xCb >> CtbLog2SizeY ) + Min( 1, 7 − CtbLog2SizeY ) (8‑893)  
 − ( 1 << ( (7 − CtbLog2SizeY ) << 1) ) )

xRefBR >> CtbLog2SizeY <= ( xCb >> CtbLog2SizeY ) (8‑894)

* When the motion vector mvL points to a reference block where the top-left sample is located inside the neighbouring coding tree block to the left, i.e. ( xRefTL ) >> CtbLog2SizeY is equal to ( xCb >> CtbLog2SizeY ) − 1, and CtbSizeY is equal to 128, the motion vector mvL shall not point to a reference block where the top-left sample is located left to or inside the same 64x64 area as the current block but inside the neighbouring coding tree block to the left, i.e. the following conditions shall be true:
  + The derivation process for block availability as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( ( xRefTL + 128 ) / 64 \* 64, yRefTL / 64 \* 64 ) as inputs, and the output shall be equal to FALSE.
  + The luma location ( ( xRefTL + 128 ) / 64 \* 64, yRefTL / 64 \* 64 ) shall not be equal to ( xCb, yCb ).

#### Derivation process for IBC luma motion vector prediction

This process is only invoked when CuPredMode[ xCb ][ yCb ] is equal to MODE\_IBC, where ( xCb, yCb ) specify the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are:

* the luma motion vectors in 1/16 fractional-sample accuracy mvL.

The variables xSmr, ySmr, smrWidth, smrHeight, and smrNumHmvpIbcCand are derived as follows:

xSmr = IsInSmr[ xCb ][ yCb ]  ?  SmrX[ xCb ][ yCb ]  :  xCb (8‑895)

ySmr = IsInSmr[ xCb ][ yCb ]  ?  SmrY[ xCb ][ yCb ]  :  yCb (8‑896)

smrWidth = IsInSmr[ xCb ][ yCb ]  ?  SmrW[ xCb ][ yCb ]  :  cbWidth (8‑897)

smrHeight = IsInSmr[ xCb ][ yCb ]  ?  SmrH[ xCb ][ yCb ]  :  cbHeight (8‑898)

smrNumHmvpIbcCand = IsInSmr[ xCb ][ yCb ]  ?  NumHmvpSmrIbcCand  :  NumHmvpIbcCand (8‑899)

The luma motion vector mvL is derived by the following ordered steps:

1. The derivation process for spatial motion vector candidates from neighbouring coding units as specified in clause 8.6.2.3 is invoked with the luma coding block location ( xCb, yCb ) set equal to ( xSmr, ySmr ), the luma coding block width cbWidth, and the luma coding block height cbHeight set equal to smrWidth and smrHeight as inputs, and the outputs being the availability flags availableFlagA1, availableFlagB1 and the motion vectors mvA1 and mvB1.
2. The motion vector candidate list, mvCandList, is constructed as follows:

i = 0  
if( availableFlagA1 )  
 mvCandList [ i++ ] = mvA1 (8‑900)  
if( availableFlagB1 )  
 mvCandList [ i++ ] = mvB1

1. The variable numCurrCand is set equal to the number of merging candidates in the mvCandList.
2. When numCurrCand is less than MaxNumMergeCand and smrNumHmvpIbcCand is greater than 0, the derivation process of IBC history-based motion vector candidates as specified in 8.6.2.4 is invoked with mvCandList, isInSmr set equal to IsInSmr[ xCb ][ yCb ], and numCurrCand as inputs, and modified mvCandList and numCurrCand as outputs.
3. When numCurrCand is less than MaxNumMergeCand, the following applies until numCurrCand is equal to MaxNumMergeCand:
   * + mvCandList[ numCurrCand ][ 0 ] is set equal to 0.
     + mvCandList[ numCurrCand ][ 1 ] is set equal to 0.
     + numCurrCand is increased by 1.
4. The variable mvIdx is derived as follows:

mvIdx = general\_merge\_flag[ xCb ][ yCb ] ? merge\_idx[ xCb ][ yCb ] : mvp\_l0\_flag[ xCb ][ yCb ] (8‑901)

1. The following assignments are made:

mvL[ 0 ] = mvCandList[ mvIdx ][ 0 ] (8‑902)

mvL[ 1 ] = mvCandList[ mvIdx ][ 1 ] (8‑903)

#### Derivation process for IBC spatial motion vector candidates

Inputs to this process are:

* a luma location ( xCb, yCb ) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples.

Outputs of this process are as follows:

* the availability flags availableFlagA1 and availableFlagB1 of the neighbouring coding units,
* the motion vectors in 1/16 fractional-sample accuracy mvA1, and mvB1 of the neighbouring coding units,

For the derivation of availableFlagA1 and mvA1 the following applies:

* The luma location ( xNbA1, yNbA1 ) inside the neighbouring luma coding block is set equal to ( xCb − 1,  yCb + cbHeight − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbA1, yNbA1 ) as inputs, and the output is assigned to the block availability flag availableA1.
* The variables availableFlagA1 and mvA1 are derived as follows:
* If availableA1 is equal to FALSE, availableFlagA1 is set equal to 0 and both components of mvA1 are set equal to 0.
* Otherwise, availableFlagA1 is set equal to 1 and the following assignments are made:

mvA1 = MvL0[ xNbA1 ][ yNbA1 ] (8‑904)

For the derivation of availableFlagB1 and mvB1 the following applies:

* The luma location ( xNbB1, yNbB1 ) inside the neighbouring luma coding block is set equal to ( xCb + cbWidth − 1, yCb − 1 ).
* The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the current luma location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring luma location ( xNbB1, yNbB1 ) as inputs, and the output is assigned to the block availability flag availableB1.
* The variables availableFlagB1 and mvB1 are derived as follows:
* If one or more of the following conditions are true, availableFlagB1 is set equal to 0 and both components of mvB1 are set equal to 0:
  + availableB1 is equal to FALSE.
  + availableA1 is equal to TRUE and the luma locations ( xNbA1, yNbA1 ) and ( xNbB1, yNbB1 ) have the same motion vectors.
* Otherwise, availableFlagB1 is set equal to 1 and the following assignments are made:

mvB1 = MvL0[ xNbB1 ][ yNbB1 ] (8‑905)

#### Derivation process for IBC history-based motion vector candidates

Inputs to this process are:

* a motion vector candidate list mvCandList,
* a variable isInSmr specifying whether the current coding unit is inside a shared merging candidate region,
* the number of available motion vector candidates in the list numCurrCand.

Outputs to this process are:

* the modified motion vector candidate list mvCandList,
* the modified number of motion vector candidates in the list numCurrCand.

The variables isPrunedA1 and isPrunedB1 are set both equal to FALSE.

The array smrHmvpIbcCandList and the variable smrNumHmvpIbcCand are derived as follows:

smrHmvpIbcCandList = isInSmr  ?  HmvpSmrIbcCandList  :  HmvpIbcCandList (8‑906)

smrNumHmvpIbcCand = isInSmr  ?  NumHmvpSmrIbcCand  :  NumHmvpIbcCand (8‑907)

For each candidate in smrHmvpIbcCandList[ hMvpIdx ] with index hMvpIdx = 1..smrNumHmvpIbcCand, the following ordered steps are repeated until numCurrCand is equal to MaxNumMergeCand:

1. The variable sameMotion is derived as follows:
   * If all of the following conditions are true for any motion vector candidate N with N being A1 or B1, sameMotion and isPrunedN are both set equal to TRUE:
   * hMvpIdx is less than or equal to 1.
   * The candidate smrHmvpIbcCandList[smrNumHmvpIbcCand − hMvpIdx] is equal to the motion vector candidate N.
   * isPrunedN is equal to FALSE.
   * Otherwise, sameMotion is set equal to FALSE.
2. When sameMotion is equal to FALSE, the candidate smrHmvpIbcCandList[smrNumHmvpIbcCand − hMvpIdx] is added to the motion vector candidate list as follows:

mvCandList[ numCurrCand++ ] = smrHmvpIbcCandList[ smrNumHmvpIbcCand − hMvpIdx ] (8‑908)

#### Derivation process for chroma motion vectors

Input to this process is:

* a luma motion vector in 1/16 fractional-sample accuracy mvL.

Output of this process is a chroma motion vector in 1/32 fractional-sample accuracy mvC.

A chroma motion vector is derived from the corresponding luma motion vector.

The chroma motion vector mvC is derived as follows:

mvC[ 0 ] = ( ( mvL[ 0 ] >> ( 3 + SubWidthC ) ) \* 32 (8‑909)

mvC[ 1 ] = ( ( mvL[ 1 ] >> ( 3 + SubHeightC ) ) \* 32 (8‑910)

#### Updating process for the history-based motion vector predictor candidate list

Inputs to this process are:

* luma motion vector mvL in 1/16 fractional-sample accuracy.

The candidate list HmvpIbcCandList is modified by the following ordered steps:

1. The variable identicalCandExist is set equal to FALSE and the variable removeIdx is set equal to 0.
2. When NumHmvpIbcCand is greater than 0, for each index hMvpIdx with hMvpIdx = 0..NumHmvpIbcCand − 1, the following steps apply until identicalCandExist is equal to TRUE:
   * When hMvpCand is equal to HmvpIbcCandList[ hMvpIdx ], identicalCandExist is set equal to TRUE and removeIdx is set equal to hMvpIdx.
3. The candidate list HmvpIbcCandList is updated as follows:
   * If identicalCandExist is equal to TRUE or NumHmvpIbcCand  is equal to 5, the following applies:
   * For each index i with i = ( removeIdx + 1 )..( NumHmvpIbcCand  − 1 ), HmvpIbcCandList[ i − 1] is set equal to HmvpIbcCandList [ i ].
   * HmvpIbcCandList[ NumHmvpIbcCand − 1 ] is set equal to mvCand.
   * Otherwise (identicalCandExist is equal to FALSE and NumHmvpIbcCand  is less than 5), the following applies:
   * HmvpIbcCandList[ NumHmvpIbcCand ++ ] is set equal to mvCand.

### Decoding process for ibc blocks

#### General

This process is invoked when decoding a coding unit coded in ibc prediction mode.

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top‑left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* variables numSbX and numSbY specifying the number of luma coding subblocks in horizontal and vertical direction,
* the motion vectors mv[ xSbIdx ][ ySbIdx ] with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,
* a variable cIdx specifying the colour component index of the current block.

Outputs of this process are:

* an array predSamples of prediction samples.

Let predSamplesL0L, and predSamplesL1L be (cbWidth)x(cbHeight) arrays of predicted luma sample values and, predSamplesL0Cb, predSamplesL1Cb, predSamplesL0Cr and predSamplesL1Cr be (cbWidth / SubWidthC)x(cbHeight / SubHeightC) arrays of predicted chroma sample values.

The reference indices refIdxL0 and refIdxL1 are set equal to −1.

The prediction list utilization flags predFlagL0[ xSbIdx ][ ySbIdx ] are set equal to 1 and predFlagL1[ xSbIdx ][ ySbIdx ] are set equal to 0 with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1,

The bi-prediction weight index bcwIdx is set equal to 0,

The width and the height of the current coding sublock sbWidth and sbHeight in luma samples are derived as follows:

sbWidth  =  cbWidth / numSbX (8‑911)

sbHeight  =  cbHeight / numSbY (8‑912)

For each coding subblock at subblock index ( xSbIdx, ySbIdx ) with xSbIdx = 0 .. numSbX − 1, and ySbIdx = 0 .. numSbY − 1, the following applies:

* The luma location ( xSb, ySb ) specifying the top-left sample of the current coding subblock relative to the top‑left luma sample of the current picture is derived as follows:

( xSb, ySb )  =  ( xCb + xSbIdx \* sbWidth, yCb + ySbIdx \* sbHeight ) (8‑913)

* The variable bdofFlag is set equal to FALSE.

The current decoded picture consists of a pic\_width\_in\_luma\_samples by pic\_height\_in\_luma\_samples array of luma samples currPicL and two PicWidthInSamplesC by PicHeightInSamplesC arrays of chroma samples currPicCb and currPicCr. The current decoded picture sample arrays currPicL, currPicCb and currPicCr correspond to decoded sample arrays SL, SCb and SCr derived in clause 8.8 for the current decoded picture.

* If cIdx is equal to 0, the following applies:
  + The array predSamplesL0L is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth, the coding subblock height sbHeight in luma samples, the motion vector offset mvOffset set equal to ( 0, 0 ), the luma motion vector mv[ xSb ][ ySb ], the reference array currPicL, bdofFlag, and cIdx as inputs.
* Otherwise if cIdx is equal to 1, the following applies:
  + The array predSamplesL0Cb is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / SubWidthC, the coding subblock height sbHeight / SubHeightC, the motion vector offset mvOffset set equal to ( 0, 0 ), the chroma motion vector mv[ xSb ][ ySb ], the reference array currPicCb, bdofFlag, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the following applies:
  + The array predSamplesL0Cr is derived by invoking the fractional sample interpolation process specified in clause 8.5.6.3 with the luma location ( xCb, yCb ), the coding subblock width sbWidth / SubWidthC, the coding subblock height sbHeight / SubHeightC, the motion vector offset mvOffset set equal to ( 0, 0 ), the chroma motion vector mv[ xSb ][ ySb ], the reference array currPicCr, bdofFlag, and cIdx as inputs.
* The array predSamples of prediction samples is derived as follows:
* If cIdx is equal to 0, the prediction samples inside the current luma coding subblock, predSamples[ xL + xSb ][ yL + ySb ] with xL = 0..sbWidth − 1 and yL = 0..sbHeight − 1, are derived by invoking the weighted sample prediction process as specified in clause 8.5.6.5 is invoked with the luma coding subblock width sbWidth, the luma coding subblock height sbHeight and the sample arrays predSamplesL0 L and predSamplesL1L, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs, and predSamples[ xL + xSb ][ yL + ySb ] as outputs.
* Otherwise, if cIdx is equal to 1, the prediction samples inside the current chroma component Cb coding block, predSamples[ xC + xCb / SubWidthC ][ yC + yCb / SubHeightC ] with xC = 0..cbWidth / SubWidthC − 1 and yC = 0..cbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.6.5 with the coding block width nCbW set equal to cbWidth / SubWidthC, the coding block height nCbH set equal to cbHeight / SubHeightC, the sample arrays predSamplesL0Cb and predSamplesL1Cb, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs.
* Otherwise (cIdx is equal to 2), the prediction samples inside the current chroma component Cr coding block, predSamples[ xC + xCb / SubWidthC ][ yC + yCb / SubHeightC ] with xC = 0..cbWidth / SubWidthC − 1 and yC = 0..cbHeight / SubHeightC − 1, are derived by invoking the weighted sample prediction process specified in clause 8.5.6.5 with the coding block width nCbW set equal to cbWidth / SubWidthC, the coding block height nCbH set equal to cbHeight / SubHeightC, the sample arrays predSamplesL0Cr and predSamplesL1Cr, and the variables predFlagL0[ xSbIdx ][ ySbIdx ], predFlagL1[ xSbIdx ][ ySbIdx ], refIdxL0, refIdxL1, bcwIdx, and cIdx as inputs.
* When cIdx is equal to 0, the following assignments are made for x = 0..sbWidth − 1 and y = 0..sbHeight − 1:

MvL0[ xSb + x ][ ySb + y ] = mv[ xSbIdx ][ ySbIdx ] (8‑914)

MvL1[ xSb + x ][ ySb + y ] = 0 (8‑915)

RefIdxL0[ xSb + x ][ ySb + y ] = refIdxL0 (8‑916)

RefIdxL1[ xSb + x ][ ySb + y ] = refIdxL1 (8‑917)

PredFlagL0[ xSb + x ][ ySb + y ] = predFlagL0[ xSbIdx ][ ySbIdx ] (8‑918)

PredFlagL1[ xSb + x ][ ySb + y ] = predFlagL1[ xSbIdx ][ ySbIdx ] (8‑919)

BcwIdx[ xSb + x ][ ySb + y ] = bcwIdx (8‑920)

## Scaling, transformation and array construction process

### Derivation process for quantization parameters

Inputs to this process are:

* a luma location ( xCb, yCb ) specifying the top-left luma sample of the current coding block relative to the top-left luma sample of the current picture,
* a variable cbWidth specifying the width of the current coding block in luma samples,
* a variable cbHeight specifying the height of the current coding block in luma samples,
* a variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed.

In this process, the luma quantization parameter Qp′Y and the chroma quantization parameters Qp′Cb and Qp′Cr are derived.

The luma location ( xQg, yQg ), specifies the top-left luma sample of the current quantization group relative to the top left luma sample of the current picture. The horizontal and vertical positions xQg and yQg are set equal to CuQgTopLeftX and CuQgTopLeftY, respectively.

NOTE – : The current quantization group is a rectangluar region inside a coding tree block that shares the same qPY\_PRED. Its width and height are equal to the width and height of the coding tree node of which the top-left luma sample position is assigned to the variables CuQgTopLeftX and CuQgTopLeftY.

When treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the predicted luma quantization parameter qPY\_PRED is derived by the following ordered steps:

1. The variable qPY\_PREV is derived as follows:

– If one or more of the following conditions are true, qPY\_PREV is set equal to SliceQpY:

– The current quantization group is the first quantization group in a slice.

– The current quantization group is the first quantization group in a brick.

– Otherwise, qPY\_PREV is set equal to the luma quantization parameter QpY of the last luma coding unit in the previous quantization group in decoding order.

1. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xQg − 1, yQg ) as inputs, and the output is assigned to availableA. The variable qPY\_A is derived as follows:

– If one or more of the following conditions are true, qPY\_A is set equal to qPY\_PREV:

– availableA is equal to FALSE.

– the CTB address ctbAddrA of the CTB containing the luma coding block covering the luma location ( xQg − 1, yQg ) is not equal to CtbAddrInBs, where ctbAddrA is derived as follows:

xTmp = ( xQg − 1 )  >>  MinTbLog2SizeY  
yTmp = yQg  >>  MinTbLog2SizeY  
minTbAddrA = MinTbAddrZs[ xTmp ][ yTmp ]  
ctbAddrA = minTbAddrA  >>  ( 2  \* ( CtbLog2SizeY − MinTbLog2SizeY ) ) (8‑921)

– Otherwise, qPY\_A is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg − 1, yQg ).

1. The availability derivation process for a block as specified in clause 6.4.X [Ed. (BB): Neighbouring blocks availability checking process tbd] is invoked with the location ( xCurr, yCurr ) set equal to ( xCb, yCb ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xQg, yQg − 1 ) as inputs, and the output is assigned to availableB. The variable qPY\_B is derived as follows:

– If one or more of the following conditions are true, qPY\_B is set equal to qPY\_PREV:

– availableB is equal to FALSE.

– the CTB address ctbAddrB of the CTB containing the luma coding block covering the luma location ( xQg,  yQg − 1 ) is not equal to CtbAddrInBs, where ctbAddrB is derived as follows:

xTmp = xQg  >>  MinTbLog2SizeY  
yTmp = ( yQg − 1 )  >>  MinTbLog2SizeY  
minTbAddrB = MinTbAddrZs[ xTmp ][ yTmp ]  
ctbAddrB = minTbAddrB  >>  ( 2 \* ( CtbLog2SizeY − MinTbLog2SizeY ) ) (8‑922)

– Otherwise, qPY\_B is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg, yQg − 1 ).

1. The predicted luma quantization parameter qPY\_PRED is derived as follows:

* If all the following conditions are true, then qPY\_PRED is set equal to the luma quantization parameter QpY of the coding unit containing the luma coding block covering ( xQg, yQg − 1 ):
* availableB is equal to TRUE.
* the current quantization group is the first quantization group in a CTB row within a brick
* Otherwise, qPY\_PRED is derived as follows:

qPY\_PRED = ( qPY\_A + qPY\_B + 1 ) >> 1 (8‑923)

The variable QpY is derived as follows:

QpY = ( ( qPY\_PRED + CuQpDeltaVal + 64 + 2 \* QpBdOffsetY )%( 64 + QpBdOffsetY ) ) − QpBdOffsetY (8‑924)

The luma quantization parameter Qp′Y is derived as follows:

Qp′Y = QpY + QpBdOffsetY (8‑925)

[Ed. (BB): Modify highlighted sections when bricks without slices are incorporated]

When ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the following applies:

– When treeType is equal to DUAL\_TREE\_CHROMA, the variable QpY is set equal to the luma quantization parameter QpY of the luma coding unit that covers the luma location ( xCb + cbWidth / 2, yCb + cbHeight / 2 ).

– The variables qPCb, qPCr and qPCbCr are derived as follows:

qPiCb = Clip3( −QpBdOffsetC, 69, QpY + pps\_cb\_qp\_offset + slice\_cb\_qp\_offset ) (8‑926)

qPiCr = Clip3( −QpBdOffsetC, 69, QpY + pps\_cr\_qp\_offset + slice\_cr\_qp\_offset ) (8‑927)

qPiCbCr = Clip3( −QpBdOffsetC, 69, QpY + pps\_joint\_cbcr\_qp\_offset + slice\_joint\_cbcr\_qp\_offset ) (8‑928)

– If ChromaArrayType is equal to 1, the variables qPCb, qPCr and qPCbCr are set equal to the value of QpC as specified in Table 8‑17 based on the index qPiequal to qPiCb, qPiCr and qPiCbCr, respectively.

– Otherwise, the variables qPCb, qPCr and qPCbCr are set equal to Min( qPi, 63 ), based on the index qPiequal to qPiCb, qPiCr and qPiCbCr, respectively.

– The chroma quantization parameters for the Cb and Cr components, Qp′Cb and Qp′Cr, and joint Cb-Cr coding Qp′CbCr are derived as follows:

Qp′Cb = qPCb + QpBdOffsetC (8‑929)

Qp′Cr = qPCr + QpBdOffsetC (8‑930)

Qp′CbCr = qPCbCr + QpBdOffsetC (8‑931)

Table 8‑17 – Specification of QpC as a function of qPi for ChromaArrayType equal to 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| qPi | < 30 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | > 43 |
| QpC | = qPi | 29 | 30 | 31 | 32 | 33 | 33 | 34 | 34 | 35 | 35 | 36 | 36 | 37 | 37 | = qPi − 6 |

### Scaling and transformation process

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable cIdx specifying the colour component of the current block,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height.

Output of this process is the (nTbW)x(nTbH) array of residual samples resSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1.

The variables bitDepth, bdShift and tsShift are derived as follows:

bitDepth = ( cIdx = = 0 ) ? BitDepthY : BitDepthC (8‑932)

bdShift = Max( 20 − bitDepth, 0 ) (8‑933)

tsShift = 5 + ( ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) (8‑934)

The (nTbW)x(nTbH) array of residual samples resSamples is derived as follows:

* If cIdx is equal to 2 and tu\_joint\_cbcr\_residual[ xTbY ][ yTbY ] is equal to 1, the residual samples resSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

resSamples[ x ][ y ] = −resSamplesCb[ x ][ y ] (8‑935)

where resSamplesCb is the residual sample array resSamples that is generated in the last invokation of this clause for the Cb component block with cIdx euqal to 1 at the transform block location ( xTbY, yTbY ).

* Otherwise (cIdx is less than 2 or tu\_joint\_cbcr\_residual[ xTbY ][ yTbY ] is equal to 0), the following ordered steps apply:
  1. The scaling process for transform coefficients as specified in clause 8.7.3 is invoked with the transform block location ( xTbY, yTbY ), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the bit depth of the current colour component bitDepth as inputs, and the output is an (nTbW)x(nTbH) array of scaled transform coefficients d.
  2. The (nTbW)x(nTbH) array of residual samples r is derived as follows:
* If transform\_skip\_flag[ xTbY ][ yTbY ] is equal to 1 and cIdx is equal to 0, the residual sample array values r[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

r[ x ][ y ] = d[ x ][ y ]  <<  tsShift (8‑936)

* Otherwise (transform\_skip\_flag[ xTbY ][ yTbY ] is equal to 0 or and cIdx is not equal to 0), the transformation process for scaled transform coefficients as specified in clause 8.7.4.1 is invoked with the transform block location ( xTbY, yTbY ), the transform block width nTbW and the transform block height nTbH, the colour component variable cIdx and the (nTbW)x(nTbH) array of scaled transform coefficients d as inputs, and the output is an (nTbW)x(nTbH) array of residual samples r.
  1. The residual samples resSamples[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 are derived as follows:

resSamples[ x ][ y ] = ( r[ x ][ y ] + ( 1 << ( bdShift − 1 ) ) ) >> bdShift (8‑937)

### Scaling process for transform coefficients

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable nTbW specifying the transform block width,
* a variable nTbH specifying the transform block height,
* a variable cIdx specifying the colour component of the current block,
* a variable bitDepth specifying the bit depth of the current colour component.

Output of this process is the (nTbW)x(nTbH) array d of scaled transform coefficients with elements d[ x ][ y ].

The quantization parameter qP is derived as follows:

* If cIdx is equal to 0, the following applies:

qP = Qp′Y  (8‑938)

* Otherwise, if cIdx is equal to 1 and tu\_joint\_cbcr\_residual[ xTbY ][ yTbY ] is equal to 1, the following applies:

qP = Qp′CbCr (8‑939)

* Otherwise, if cIdx is equal to 1, the following applies:

qP = Qp′Cb (8‑940)

* Otherwise (cIdx is equal to 2), the following applies:

qP = Qp′Cr (8‑941)

The variable rectNonTsFlag is derived as follows:

rectNonTsFlag = ( ( ( Log2( nTbW ) + Log2( nTbH ) ) & 1 )  = =  1 && (8‑942)  
 transform\_skip\_flag[ xTbY ][ yTbY ] = = 0 )

The variables bdShift, rectNorm and bdOffset are derived as follows:

bdShift = bitDepth + ( ( rectNonTsFlag  ?  1  :  0 ) + (8‑943)  
 ( Log2( nTbW ) + Log2( nTbH ) ) / 2 ) − 5 + dep\_quant\_enabled\_flag

bdOffset = ( 1 << bdShift ) >> 1 (8‑944)

The list levelScale[ ][ ] is specified as levelScale[ j ][ k ] = { { 40, 45, 51, 57, 64, 72 }, { 57, 64, 72, 80, 90, 102 } } with j = 0..1, k = 0..5.

The (nTbW)x(nTbH) array dz is set equal to the (nTbW)x(nTbH) array TransCoeffLevel[ xTbY ][ yTbY ][ cIdx ].

For the derivation of the scaled transform coefficients d[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1, the following applies:

* The intermediate scaling factor m[ x ][ y ] is derived as follows:

– If one or more of the following conditions are true, m[ x ][ y ] is set equal to 16:

* scaling\_list\_enabled\_flag is equal to 0.
* transform\_skip\_flag[ xTbY ][ yTbY ] is equal to 1.

– Otherwise, the following applies:

m[ x ][ y ] = ScalingFactor[ Log2( nTbW ) ][ Log2( nTbH ) ][ matrixId ][ x ][ y ],   
 with matrixId as specified in Table 7‑4 (8‑945)

* The scaling factor ls[ x ][ y ] is derived as follows:

– If dep\_quant\_enabled\_flag is equal to 1, the following applies:

ls[ x ][ y ] = ( m[ x ][ y ] \* levelScale[ rectNonTsFlag ][ (qP + 1) % 6 ] ) << ( (qP + 1) / 6 ) (8‑946)

– Otherwise (dep\_quant\_enabled\_flag is equal to 0), the following applies:

ls[ x ][ y ] = ( m[ x ][ y ] \* levelScale[ rectNonTsFlag ][ qP % 6 ] ) << ( qP / 6 ) (8‑947)

* When BdpcmFlag[ xTbY ][ yYbY ] is equal to 1, dz[ x ][ y ] is modified as follows:

– If BdpcmDir[ xTbY ][ yYbY ] is equal to 0 and x is greater than 0, the following applies:

dz[ x ][ y ] = Clip3( CoeffMin, CoeffMax, dz[ x − 1 ][ y ] + dz[ x ][ y ]) (8‑948)

– Otherwise, if BdpcmDir[ xTbY ][ yYbY ] is equal to 1 and y is greater than 0, the following applies:

dz[ x ][ y ] = Clip3( CoeffMin, CoeffMax, dz[ x ][ y − 1 ] + dz[ x ][ y ]) (8‑949)

* The value dnc[ x ][ y ] is derived as follows:

dnc[ x ][ y ] = ( dz[ x ][ y ] \* ls[ x ][ y ] +bdOffset )  >>  bdShift (8‑950)

* The scaled transform coefficient d[ x ][ y ] is derived as follows:

d[ x ][ y ] = Clip3( CoeffMin, CoeffMax, dnc[ x ][ y ] ) (8‑951)

### Transformation process for scaled transform coefficients

#### General

Inputs to this process are:

* a luma location ( xTbY, yTbY ) specifying the top-left sample of the current luma transform block relative to the top‑left luma sample of the current picture,
* a variable nTbW specifying the width of the current transform block,
* a variable nTbH specifying the height of the current transform block,
* a variable cIdx specifying the colour component of the current block,
* an (nTbW)x(nTbH) array d[ x ][ y ] of scaled transform coefficients with x = 0..nTbW − 1, y = 0..nTbH − 1.

Output of this process is the (nTbW)x(nTbH) array r[ x ][ y ] of residual samples with x = 0..nTbW − 1, y = 0..nTbH − 1.

When lfnst\_idx[ xTbY ][ yTbY ] is not equal to 0 and both nTbW and nTbH are greater than or equal to 4, the following applies:

* The variables predModeIntra, nLfnstOutSize, log2LfnstSize, nLfnstSize, numLfnstX, numLfnstY, and nonZeroSize are derived as follows:

predModeIntra = ( cIdx = = 0 )  ?  IntraPredModeY[ xTbY ][ yTbY ]  :  IntraPredModeC[ xTbY ][ yTbY ] (8‑952)

nLfnstOutSize = ( nTbW >= 8  &&  nTbH  >= 8 )  ?  48  :  16 (8‑953)

log2LfnstSize = ( nTbW >= 8  &&  nTbH  >= 8 )  ?  3  :  2 (8‑954)

nLfnstSize = 1  <<  log2LfnstSize (8‑955)

numLfnstX = ( nTbW > 8  &&  nTbH  = = 4 )  ?  2  :  1 (8‑956)

numLfnstY = ( nTbW = = 4  &&  nTbH  > 8 )  ?  2  :  1 (8‑957)

nonZeroSize = ( ( nTbW = = 4  &&  nTbH  = = 4 ) | | ( nTbW = = 8  &&  nTbH  = = 8 ) )  ?  8  :  16 (8‑958)

* The wide angle intra prediction mode mapping process as specified in clause 8.4.5.2.6 is invoked with predModeIntra, nTbW, nTbH and cIdx as inputs, and the modified predModeIntra as output.
* For xSbIdx = 0..numLfnstX − 1 and ySbIdx = 0..numLfnstY − 1, the following applies:
* The values of the list u[ x ] with x = 0..nonZeroSize − 1 are derived as follows:

xC = ( xSbIdx << log2LfnstSize ) + DiagScanOrder[ 2 ][ 2 ][ x ][ 0 ] (8‑959)

yC = ( ySbIdx << log2LfnstSize ) + DiagScanOrder[ 2 ][ 2 ][ x ][ 1 ] (8‑960)

u[ x ] = d[ xC ][ yC ] (8‑961)

* The one-dimensional low frequency non-separable transformation process as specified in clause 8.7.4.2 is invoked with the input length of the scaled transform coefficients nonZeroSize, the transform output length nTrS set equal to nLfnstOutSize, the list of scaled non-zero transform coefficients u[ x ] with x = 0..nonZeroSize − 1, the intra prediction mode for LFNST set selection predModeIntra, and the LFNST index for transform selection in the selected LFNST set lfnst\_idx[ xTbY ][ yTbY ] as inputs, and the list v[ x ] with x = 0..nLfnstOutSize − 1 as output.
* The array d[ ( xSbIdx << log2LfnstSize ) + x ][ ( ySbIdx << log2LfnstSize ) + y ] with x = 0..nLfnstSize − 1, y = 0..nLfnstSize − 1 is derived as follows:
* If predModeIntra is less than or equal to 34, or equal to INTRA\_LT\_CCLM, INTRA\_T\_CCLM, or INTRA\_L\_CCLM, the following applies:

d[ ( xSbIdx << log2LfnstSize ) + x ][ ( ySbIdx << log2LfnstSize ) + y ] =   
 ( y < 4 ) ? v[ x + ( y << log2LfnstSize ) ] : (8‑962)  
 ( ( x < 4 ) ? v[ 32 + x + ( ( y − 4 ) << 2 ) ] : d[ x ][ y ] )

* Otherwise, the following applies:

d[ ( xSbIdx << log2LfnstSize ) + x ][ ( ySbIdx << log2LfnstSize ) + y ] =   
 ( x < 4 ) ? v[ y + ( x << log2LfnstSize ) ] : (8‑963)  
 ( ( y < 4 ) ? v[ 32 + y + ( ( x − 4 ) << 2 ) ] : d[ x ][ y ] )

The variable implicitMtsEnabled is derived as follows:

* If sps\_mts\_enabled\_flag is equal to 1 and one of the following conditions is true, implicitMtsEnabled is set equal to 1:
* IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT
* cu\_sbt\_flag is equal to 1 and Max( nTbW, nTbH ) is less than or equal to 32
* sps\_explicit\_mts\_intra\_enabled\_flag and sps\_explicit\_mts\_inter\_enabled\_flag are both equal to 0 and CuPredMode[ xTbY ][ yTbY ] is equal to MODE\_INTRA
* Otherwise, implicitMtsEnabled is set equal to 0.

The variable trTypeHor specifying the horizontal transform kernel and the variable trTypeVer specifying the vertical transform kernel are derived as follows:

* If cIdx is greater than 0, trTypeHor and trTypeVer are set equal to 0.
* Otherwise, if implicitMtsEnabled is equal to 1, the following applies:
* If IntraSubPartitionsSplitType is not equal to ISP\_NO\_SPLIT or both sps\_explicit\_mts\_intra\_enabled\_flag and sps\_explicit\_mts\_inter\_enabled\_flag are equal to 0 and CuPredMode[ xTbY ][ yTbY ] is equal to MODE\_INTRA, trTypeHor and trTypeVer are derived as follows:

trTypeHor = ( nTbW >= 4 && nTbW <= 16 ) ? 1 : 0 (8‑964)

trTypeVer = ( nTbH >= 4 && nTbH <= 16 ) ? 1 : 0 (8‑965)

* Otherwise, if cu\_sbt\_flag is equal to 1, trTypeHor and trTypeVer are specified in Table 8‑19 depending on cu\_sbt\_horizontal\_flag and cu\_sbt\_pos\_flag.
* Otherwise, trTypeHor and trTypeVer are specified in Table 8‑18 depending on tu\_mts\_idx[ xTbY ][ yTbY ].

The variables nonZeroW and nonZeroH are derived as follows:

nonZeroW = Min( nTbW, ( trTypeHor > 0 )  ?  16  :  32 ) (8‑966)

nonZeroH = Min( nTbH, ( trTypeVer > 0 )  ?  16  :  32 ) (8‑967)

The (nTbW)x(nTbH) array r of residual samples is derived as follows:

1. When nTbH is greater than 1, each (vertical) column of scaled transform coefficients d[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nonZeroH − 1 is transformed to e[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.7.4.4 for each column x = 0..nonZeroW − 1 with the height of the transform block nTbH, the non-zero height of the scaled transform coefficients nonZeroH, the list d[ x ][ y ] with y = 0..nonZeroH − 1 and the transform type variable trType set equal to trTypeVer as inputs, and the output is the list e[ x ][ y ] with y = 0..nTbH − 1.
2. When nTbH and nTbW are both greater than 1, the intermediate sample values g[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 are derived as follows:

g[ x ][ y ] = Clip3( CoeffMin, CoeffMax, ( e[ x ][ y ] + 64 ) >> 7 ) (8‑968)

1. When nTbW is greater than 1, each (horizontal) row of the resulting array g[ x ][ y ] with x = 0..nonZeroW − 1, y = 0..nTbH − 1 is transformed to r[ x ][ y ] with x = 0..nTbW − 1, y = 0..nTbH − 1 by invoking the one-dimensional transformation process as specified in clause 8.7.4.4 for each row y = 0..nTbH − 1 with the width of the transform block nTbW, the non-zero width of the resulting array g[ x ][ y ] nonZeroW, the list g[ x ][ y ] with x = 0..nonZeroW − 1 and the transform type variable trType set equal to trTypeHor as inputs, and the output is the list r[ x ][ y ] with x = 0..nTbW − 1.

Table 8‑18 – Specification of trTypeHor and trTypeVer depending on tu\_mts\_idx[ x ][ y ]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **tu\_mts\_idx[ x0 ][ y0 ]** | **0** | **1** | **2** | **3** | **4** |
| trTypeHor | 0 | 1 | 2 | 1 | 2 |
| trTypeVer | 0 | 1 | 1 | 2 | 2 |

Table 8‑19 – Specification of trTypeHor and trTypeVer depending on cu\_sbt\_horizontal\_flag and cu\_sbt\_pos\_flag

|  |  |  |  |
| --- | --- | --- | --- |
| **cu\_sbt\_horizontal\_flag** | **cu\_sbt\_pos\_flag** | trTypeHor | trTypeVer |
| **0** | **0** | 2 | 1 |
| **0** | **1** | 1 | 1 |
| **1** | **0** | 1 | 2 |
| **1** | **1** | 1 | 1 |

#### Low frequency non-separable transformation process

Inputs to this process are:

* a variable nonZeroSize specifying the transform input length,
* a variable nTrS specifying the transform output length,
* a list of of scaled non-zero transform coefficients x[ j ] with j = 0..nonZeroSize − 1,
* a variable predModeIntra specifying the intra prediction mode for LFNST set selection,
* a variable lfnstIdx specifying the LFNST index for transform selection in the selected LFNST set.

Output of this process is the list of transformed samples y[ i ] with i = 0..nTrS − 1.

The transformation matrix derivation process as specified in clause 8.7.4.3 is invoked with the transform output length nTrS, the intra prediction mode for LFNST set selection predModeIntra, and the LFNST index for transform selection in the selected LFNST set lfnstIdx as inputs, and the (nTrS)x(nonZeroSize) LFNST matrix lowFreqTransMatrix as output.

The list of transformed samples y[ i ] with i = 0..nTrS − 1 is derived as follows:

y[i] = Clip3( CoeffMin, CoeffMax, ( ( ) + 64 ) >> 7 ) (8‑969)

#### Low frequency non-separable transformation matrix derivation process

Inputs to this process are:

* a variable nTrS specifying the transform output length,
* a variable predModeIntra specifying the intra prediction mode for LFNST set selection,
* a variable lfnstIdx specifying the LFNST index for transform selection in the selected LFNST set.

Output of this process is the transformation matrix lowFreqTransMatrix.

The variable lfnstTrSetIdx is specified in Table 8‑20 – Specification of lfnstTrSetIdx depending on predModeIntra.

Table 8‑20 – Specification of lfnstTrSetIdx

|  |  |
| --- | --- |
| **predModeIntra** | **lfnstTrSetIdx** |
| predModeIntra < 0 | 1 |
| 0 <= predModeIntra <= 1 | 0 |
| 2 <= predModeIntra <= 12 | 1 |
| 13 <= predModeIntra <= 23 | 2 |
| 24 <= predModeIntra <= 44 | 3 |
| 45 <= predModeIntra <= 55 | 2 |
| 56 <= predModeIntra <= 80 | 1 |
| 81 <= predModeIntra <= 83 | 0 |

The transformation matrix lowFreqTransMatrix is derived based on nTrS, lfnstTrSetIdx, and lfnstIdx as follows:

* If nTrS is equal to 16, lfnstTrSetIdx is equal to 0, and lfnstIdx is equal to 1, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ 108 -44 -15 1 -44 19 7 -1 -11 6 2 -1 0 -1 -1 0 }

{ -40 -97 56 12 -11 29 -12 -3 18 18 -15 -3 -1 -3 2 1 }

{ 25 -31 -1 7 100 -16 -29 1 -54 21 14 -4 -7 2 4 0 }

{ -32 -39 -92 51 -6 -16 36 -8 3 22 18 -15 4 1 -5 2 }

{ 8 -9 33 -8 -16 -102 36 23 -4 38 -27 -5 5 16 -8 -6 }

{ -25 5 16 -3 -38 14 11 -3 -97 7 26 1 55 -10 -19 3 }

{ 8 9 16 1 37 36 94 -38 -7 3 -47 11 -6 -13 -17 10 }

{ 2 34 -5 1 -7 24 -25 -3 8 99 -28 -29 6 -43 21 11 }

{ -16 -27 -39 -109 6 10 16 24 3 19 10 24 -4 -7 -2 -3 }

{ -9 -10 -34 4 -9 -5 -29 5 -33 -26 -96 33 14 4 39 -14 }

{ -13 1 4 -9 -30 -17 -3 -64 -35 11 17 19 -86 6 36 14 }

{ 8 -7 -5 -15 7 -30 -28 -87 31 4 4 33 61 -5 -17 22 }

{ -2 13 -6 -4 -2 28 -13 -14 -3 37 -15 -3 -2 107 -36 -24 }

{ 4 9 11 31 4 9 16 19 12 33 32 94 12 0 34 -45 }

{ 2 -2 8 -16 8 5 28 -17 6 -7 18 -45 40 36 97 -8 }

{ 0 -2 0 -10 -1 -7 -3 -35 -1 -7 -2 -32 -6 -33 -16 -112 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 0, and lfnstIdx is equal to 2, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ 119 -30 -22 -3 -23 -2 3 2 -16 3 6 0 -3 2 1 0 }

{ -27 -101 31 17 -47 2 22 3 19 30 -7 -9 5 3 -5 -1 }

{ 0 58 22 -15 -102 2 38 2 10 -13 -5 4 14 -1 -9 0 }

{ 23 4 66 -11 22 89 -2 -26 13 -8 -38 -1 -9 -20 -2 8 }

{ -19 -5 -89 2 -26 76 -11 -17 20 13 18 -4 1 -15 3 5 }

{ -10 -1 -1 6 23 25 87 -7 -74 4 39 -5 0 -1 -20 -1 }

{ -17 -28 12 -8 -32 14 -53 -6 -68 -67 17 29 2 6 25 4 }

{ 1 -24 -23 1 17 -7 52 9 50 -92 -15 27 -15 -10 -6 3 }

{ -6 -17 -2 -111 7 -17 8 -42 9 18 16 25 -4 2 -1 11 }

{ 9 5 35 0 6 21 -9 34 44 -3 102 11 -7 13 11 -20 }

{ 4 -5 -5 -10 15 19 -2 6 6 -12 -13 6 95 69 -29 -24 }

{ -6 -4 -9 -39 1 22 0 102 -19 19 -32 30 -16 -14 -8 -23 }

{ 4 -4 7 8 4 -13 -18 5 0 0 21 22 58 -88 -54 28 }

{ -4 -7 0 -24 -7 0 -25 3 -3 -30 8 -76 -34 4 -80 -26 }

{ 0 6 0 30 -6 1 -13 -23 1 20 -2 80 -44 37 -68 1 }

{ 0 0 -1 5 -1 -7 1 -34 -2 3 -6 19 5 -38 11 -115 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 1, and lfnstIdx is equal to 1, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ -111 39 4 3 44 11 -12 -1 7 -16 -5 2 3 -1 4 2 }

{ -47 -27 15 -1 -92 43 20 -2 20 39 -16 -5 10 -5 -13 2 }

{ -35 -23 4 4 -17 -72 32 6 -59 18 50 -6 0 40 0 -13 }

{ 13 93 -27 -4 -48 13 -34 4 -52 11 1 10 3 16 -3 1 }

{ -11 -27 1 2 -47 -4 -36 10 -2 -85 14 29 -20 -2 57 4 }

{ 0 -35 32 -2 26 60 -3 -17 -82 1 -30 0 -37 21 3 12 }

{ -17 -46 -92 14 7 -10 -39 29 -17 27 -28 17 1 -15 -13 17 }

{ 4 -10 -23 4 16 58 -17 26 30 21 67 2 -13 59 13 -40 }

{ 5 -20 32 -5 8 -3 -46 -7 -4 2 -15 24 100 44 0 5 }

{ -4 -1 38 -18 -7 -42 -63 -6 33 34 -23 15 -65 33 -20 2 }

{ -2 -10 35 -19 5 8 -44 14 -25 25 58 17 7 -84 -16 -18 }

{ 5 13 18 34 11 -4 18 18 5 58 -3 42 -2 -10 85 38 }

{ -5 -7 -34 -83 2 -1 -4 -73 4 20 15 -12 4 -3 44 12 }

{ 0 4 -2 -60 5 9 42 34 5 -14 9 80 -5 13 -38 37 }

{ -1 2 7 -57 3 -7 9 68 -9 6 -49 -20 6 -4 36 -64 }

{ -1 0 -12 23 1 -4 17 -53 -3 4 -21 72 -4 -8 -3 -83 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 1, and lfnstIdx is equal to 2, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ 88 -55 6 -3 -66 27 9 -2 11 11 -13 1 -2 -7 1 2 }

{ -58 -20 27 -2 -27 75 -29 0 47 -42 -11 11 -9 -3 19 -4 }

{ -51 23 -22 5 -63 3 37 -5 1 64 -35 -4 29 -31 -11 13 }

{ -27 -76 49 -2 40 14 9 -17 -56 36 -25 6 14 3 -6 8 }

{ 19 -4 -36 22 52 7 36 -23 28 -17 -64 15 -5 -44 48 9 }

{ 29 50 13 -10 1 34 -59 1 -51 4 -16 30 52 -33 24 -5 }

{ -12 -21 -74 43 -13 39 18 -5 -58 -35 27 -5 19 26 6 -5 }

{ 19 38 -10 -5 28 66 0 -5 -4 19 -30 -26 -40 28 -60 37 }

{ -6 27 18 -5 -37 -18 12 -25 -44 -10 -38 37 -66 45 40 -7 }

{ -13 -28 -45 -39 0 -5 -39 69 -23 16 -12 -18 -50 -31 24 13 }

{ -1 8 24 -51 -15 -9 44 10 -28 -70 -12 -39 24 -18 -4 51 }

{ -8 -22 -17 33 -18 -45 -57 -27 0 -31 -30 29 -2 -13 -53 49 }

{ 1 12 32 51 -8 8 -2 -31 -22 4 46 -39 -49 -67 14 17 }

{ 4 5 24 60 -5 -14 -23 38 9 8 -34 -59 24 47 42 28 }

{ -1 -5 -20 -34 4 4 -15 -46 18 31 42 10 10 27 49 78 }

{ -3 -7 -22 -34 -5 -11 -36 -69 -1 -3 -25 -73 5 4 4 -49 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 2, and lfnstIdx is equal to 1, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ -112 47 -2 2 -34 13 2 0 15 -7 1 0 8 -3 -1 0 }

{ 29 -7 1 -1 -108 40 2 0 -45 13 4 -1 8 -5 1 0 }

{ -36 -87 69 -10 -17 -33 26 -2 7 14 -11 2 6 8 -7 0 }

{ 28 -5 2 -2 -29 13 -2 0 103 -36 -4 1 48 -16 -4 1 }

{ -12 -24 15 -3 26 80 -61 9 15 54 -36 2 0 -4 6 -2 }

{ 18 53 69 -74 14 24 28 -30 -6 -7 -11 12 -5 -7 -6 8 }

{ 5 -1 2 0 -26 6 0 1 45 -9 -1 0 -113 28 8 -1 }

{ -13 -32 18 -2 15 34 -27 7 -25 -80 47 -1 -16 -50 28 2 }

{ -4 -13 -10 19 18 46 60 -48 16 33 60 -48 1 0 5 -2 }

{ 15 33 63 89 8 15 25 40 -4 -8 -15 -8 -2 -6 -9 -7 }

{ -8 -24 -27 15 12 41 26 -29 -17 -50 -39 27 0 35 -67 26 }

{ -2 -6 -24 13 -1 -8 37 -22 3 18 -51 22 -23 -95 17 17 }

{ -3 -7 -16 -21 10 24 46 75 8 20 38 72 1 2 1 7 }

{ 2 6 10 -3 -5 -16 -31 12 7 24 41 -16 -16 -41 -89 49 }

{ 4 8 21 40 -4 -11 -28 -57 5 14 31 70 7 18 32 52 }

{ 0 1 4 11 -2 -4 -13 -34 3 7 20 47 -6 -19 -42 -101 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 2, and lfnstIdx is equal to 2, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ -99 39 -1 2 65 -20 -5 0 -15 -2 5 -1 0 3 -1 0 }

{ 58 42 -33 3 33 -63 23 -1 -55 32 3 -5 21 -2 -8 3 }

{ -15 71 -44 5 -58 -29 25 3 62 -7 -4 -4 -19 4 0 1 }

{ 46 5 4 -6 71 -12 -15 5 52 -38 13 -2 -63 23 3 -3 }

{ -14 -54 -29 29 25 -9 61 -29 27 44 -48 5 -27 -21 12 7 }

{ -3 3 69 -42 -11 -50 -26 26 24 63 -19 -5 -18 -22 12 0 }

{ 17 16 -2 1 38 18 -12 0 62 1 -14 5 89 -42 8 -2 }

{ 15 54 -8 6 6 60 -26 -8 -30 17 -38 22 -43 -45 42 -7 }

{ -6 -17 -55 -28 9 30 -8 58 4 34 41 -52 -16 -36 -20 16 }

{ -2 -1 -9 -79 7 11 48 44 -13 -34 -55 6 12 23 20 -11 }

{ 7 29 14 -6 12 53 10 -11 14 59 -15 -3 5 71 -54 13 }

{ -5 -24 -53 15 -3 -15 -61 26 6 30 -16 23 13 56 44 -35 }

{ 4 8 21 52 -1 -1 -5 29 -7 -17 -44 -84 8 20 31 39 }

{ -2 -11 -25 -4 -4 -21 -53 2 -5 -26 -64 19 -8 -19 -73 39 }

{ -3 -5 -23 -57 -2 -4 -24 -75 1 3 9 -25 6 15 41 61 }

{ 1 1 7 18 1 2 16 47 2 5 24 67 3 9 25 88 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 3, and lfnstIdx is equal to 1, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ -114 37 3 2 -22 -23 14 0 21 -17 -5 2 5 2 -4 -1 }

{ -19 -41 19 -2 85 -60 -11 7 17 31 -34 2 -11 19 2 -8 }

{ 36 -25 18 -2 -42 -53 35 5 46 -60 -25 19 8 21 -33 -1 }

{ -27 -80 44 -3 -58 1 -29 19 -41 18 -12 -7 12 -17 7 -6 }

{ -11 -21 37 -10 44 -4 47 -12 -37 -41 58 18 10 -46 -16 31 }

{ 15 47 10 -6 -16 -44 42 10 -80 25 -40 21 -23 -2 3 -14 }

{ 13 25 79 -39 -13 10 31 -4 49 45 12 -8 3 -1 43 7 }

{ 16 11 -26 13 -13 -74 -20 -1 5 -6 29 -47 26 -49 54 2 }

{ -8 -34 -26 7 -26 -19 29 -37 1 22 46 -9 -81 37 14 20 }

{ -6 -30 -42 -12 -3 5 57 -52 -2 37 -12 6 74 10 6 -15 }

{ 5 9 -6 42 -15 -18 -9 26 15 58 14 43 23 -10 -37 75 }

{ -5 -23 -23 36 3 22 36 40 27 -4 -16 56 -25 -46 56 -24 }

{ 1 3 23 73 8 5 34 46 -12 2 35 -38 26 52 2 -31 }

{ -3 -2 -21 -52 1 -10 -17 44 -19 -20 30 45 27 61 49 21 }

{ -2 -7 -33 -56 -4 -6 21 63 15 31 32 -22 -10 -26 -52 -38 }

{ -5 -12 -18 -12 8 22 38 36 -5 -15 -51 -63 -5 0 15 73 }

},

* Otherwise, if nTrS is equal to 16, lfnstTrSetIdx is equal to 3, and lfnstIdx is equal to 2, the following applies:

lowFreqTransMatrix[ m ][ n ] =

{

{ -102 22 7 2 66 -25 -6 -1 -15 14 1 -1 2 -2 1 0 }

{ 12 93 -27 -6 -27 -64 36 6 13 5 -23 0 -2 6 5 -3 }

{ -59 -24 17 1 -62 -2 -3 2 83 -12 -17 -2 -24 14 7 -2 }

{ -33 23 -36 11 -21 50 35 -16 -23 -78 16 19 22 15 -30 -5 }

{ 0 -38 -81 30 27 5 51 -32 24 36 -16 12 -24 -8 9 1 }

{ 28 38 8 -9 62 32 -13 2 51 -32 15 5 -66 28 0 -1 }

{ 11 -35 21 -17 30 -18 31 18 -11 -36 -80 12 16 49 13 -32 }

{ -13 23 22 -36 -12 64 39 25 -19 23 -36 9 -30 -58 33 -7 }

{ -9 -20 -55 -83 3 -2 1 62 8 2 27 -28 7 15 -11 5 }

{ -6 24 -38 23 -8 40 -49 0 -7 9 -25 -44 23 39 70 -3 }

{ 12 17 17 0 32 27 21 2 67 11 -6 -10 89 -22 -12 16 }

{ 2 -9 8 45 7 -8 27 35 -9 -31 -17 -87 -23 -22 -19 44 }

{ -1 -9 28 -24 -1 -10 49 -30 -8 -7 40 1 4 33 65 67 }

{ 5 -12 -24 -17 13 -34 -32 -16 14 -67 -7 9 7 -74 49 1 }

{ 2 -6 11 45 3 -10 33 55 8 -5 59 4 7 -4 44 -66 }

{ -1 1 -14 36 -1 2 -20 69 0 0 -15 72 3 4 5 65 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 0, and lfnstIdx is equal to 1 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ -117 28 18 2 4 1 2 1 32 -18 -2 0 -1 0 0 0 }

{ -29 -91 47 1 9 0 3 0 -54 26 -8 3 0 1 0 0 }

{ -10 62 -11 -8 -2 -2 -1 -1 -95 3 32 0 4 0 2 0 }

{ -15 15 -10 -2 1 0 1 0 10 112 -20 -17 -4 -4 -1 -2 }

{ 32 39 92 -44 4 -10 1 -4 26 12 -15 13 -5 2 -2 0 }

{ -10 1 50 -15 2 -3 1 -1 -28 -15 14 6 1 1 1 0 }

{ 1 -33 -11 -14 7 -2 2 0 29 -12 37 -7 -4 0 -1 0 }

{ 0 6 -6 21 -4 2 0 0 -20 -24 -104 30 5 5 1 2 }

{ -13 -13 -37 -101 29 -11 8 -3 -12 -15 -20 2 -11 5 -2 1 }

{ 6 1 -14 -36 9 -3 2 0 10 9 -18 -1 -3 1 0 0 }

{ -12 -2 -26 -12 -9 2 -1 1 -3 30 4 34 -4 0 -1 0 }

{ 0 -3 0 -4 -15 6 -3 1 -7 -15 -28 -86 19 -5 4 -1 }

{ -1 9 13 5 14 -2 2 -1 -8 3 -4 -62 4 1 1 0 }

{ 6 2 -3 2 10 -1 2 0 8 3 -1 -20 0 1 0 0 }

{ 6 9 -2 35 110 -22 11 -4 -2 0 -3 1 -18 12 -3 2 }

{ -1 7 -2 9 -11 5 -1 1 -7 2 -22 4 -13 0 -1 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 14 -1 -3 0 -1 0 0 0 2 0 0 0 0 0 0 0 }

{ 33 5 -9 -1 -2 0 -1 0 -3 3 0 0 0 0 0 0 }

{ 32 -30 -4 4 -1 1 0 0 6 2 -5 0 0 0 0 0 }

{ -20 -26 31 1 0 0 0 0 2 -16 -1 6 0 1 0 0 }

{ 29 -16 -22 8 0 1 0 1 -20 6 4 -3 1 0 0 0 }

{ -99 -4 9 5 5 2 2 1 44 -10 -11 1 -2 0 -1 0 }

{ 6 -99 3 26 -1 5 0 2 14 30 -27 -2 1 -1 0 -1 }

{ -7 -46 10 -14 7 0 1 0 9 21 7 -6 -2 -1 0 -1 }

{ -12 10 26 12 -6 0 -1 0 -32 -2 11 3 3 -1 1 0 }

{ 38 26 -13 -1 -5 -1 -1 0 102 3 -14 -1 -5 -1 -2 0 }

{ -30 3 -92 14 19 0 3 0 -11 34 21 -33 1 -2 0 -1 }

{ -5 -17 -41 42 -6 2 -1 1 -1 -40 37 13 -4 2 -1 1 }

{ -12 23 16 -11 -17 0 -1 0 -11 97 -3 -3 0 -6 0 -2 }

{ -4 4 -16 0 -2 0 1 0 34 23 6 -7 -4 -2 -1 0 }

{ -5 -4 -22 8 -25 3 0 0 -3 -21 2 -3 9 -2 1 0 }

{ 0 28 0 76 4 -6 0 -2 -13 5 -76 -4 33 -1 3 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 3 0 -1 0 1 0 0 0 1 0 0 0 1 0 0 0 }

{ 7 2 -2 0 -1 1 0 0 2 1 -1 0 0 0 0 0 }

{ 6 -3 0 0 2 0 -1 0 2 -1 0 0 1 0 0 0 }

{ 1 -4 0 0 0 -3 0 1 0 -1 0 0 0 -2 0 0 }

{ 1 -4 -3 2 -4 1 0 0 1 -1 -2 1 -2 0 0 0 }

{ -5 4 -3 0 8 -1 -2 0 -2 1 -1 0 4 0 -1 0 }

{ -6 6 6 -3 1 3 -3 0 -1 1 1 0 0 1 -1 0 }

{ 2 2 5 -2 0 3 4 -1 0 0 1 0 0 1 2 -1 }

{ 11 -5 -1 6 -4 2 1 0 3 -1 1 2 -1 0 0 0 }

{ -29 10 10 0 10 -4 -1 1 -7 1 2 1 2 -1 0 0 }

{ -9 -4 18 3 2 0 0 -2 -1 -1 3 0 0 0 0 -1 }

{ -10 13 -1 -4 4 -4 3 4 -2 2 -1 -1 1 -1 1 2 }

{ -21 -5 23 0 2 -2 -1 6 -3 -3 1 0 0 0 0 2 }

{ 108 -5 -30 6 -27 10 7 -2 11 -3 -1 1 -4 1 0 1 }

{ -7 1 3 -5 3 0 -1 0 0 1 0 -1 1 0 0 0 }

{ 9 18 -3 -35 -4 -1 6 1 1 2 0 -3 -1 0 2 0 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 0, and lfnstIdx is equal to 2 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ -108 48 9 1 1 1 0 0 44 -6 -9 -1 -1 0 -1 0 }

{ 55 66 -37 -5 -6 -1 -2 0 67 -30 -20 4 -2 0 -1 0 }

{ 2 86 -21 -13 -4 -2 -1 -1 -88 5 6 4 5 1 1 0 }

{ -24 -21 -38 19 0 4 -1 2 -23 -89 31 20 2 3 1 1 }

{ 9 20 98 -26 -3 -5 0 -2 -9 -26 15 -16 2 0 1 0 }

{ -21 -7 -37 10 2 2 -1 1 -10 69 -5 -7 -2 -2 0 -1 }

{ -10 -25 4 -17 8 -2 2 -1 -27 -17 -71 25 8 2 1 1 }

{ 2 5 10 64 -9 4 -3 1 -4 8 62 3 -17 1 -2 0 }

{ -11 -15 -28 -97 6 -1 4 -1 7 3 57 -15 10 -2 0 -1 }

{ 9 13 24 -6 7 -2 1 -1 16 39 20 47 -2 -2 -2 0 }

{ -7 11 12 7 2 -1 0 -1 -14 -1 -24 11 2 0 0 0 }

{ 0 0 7 -6 23 -3 3 -1 5 1 18 96 13 -9 -1 -1 }

{ -2 -6 -1 -10 0 1 1 0 -7 -2 -28 20 -15 4 -3 1 }

{ -1 6 -16 0 24 -3 1 -1 2 6 6 16 18 -7 1 -1 }

{ -5 -6 -3 -19 -104 18 -4 3 0 6 0 35 -41 20 -2 2 }

{ -1 -2 0 23 -9 0 -2 0 1 1 8 -1 29 1 1 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 9 -9 -1 1 0 0 0 0 3 -1 1 0 0 0 0 0 }

{ -31 -19 14 4 1 1 1 0 -6 3 5 -2 0 0 0 0 }

{ 14 -5 0 3 0 0 0 0 10 -5 -2 0 -1 0 0 0 }

{ -30 26 36 -8 -2 -2 0 -1 14 18 -7 -9 -1 -1 0 0 }

{ -61 -3 -2 3 7 1 1 0 12 16 -6 -1 0 -1 0 0 }

{ -93 2 19 0 3 0 2 0 17 4 0 0 -1 0 0 0 }

{ -4 -66 28 36 -5 3 0 1 -10 20 33 -13 -8 0 0 -1 }

{ -3 -75 5 -14 1 4 0 1 -36 3 18 -4 4 0 1 0 }

{ -1 -27 13 6 1 -1 0 0 -34 -6 0 3 4 1 2 0 }

{ 28 23 76 -5 -25 -3 -3 -1 6 36 -7 -39 -4 -1 0 -1 }

{ -20 48 11 -13 -5 -2 0 -1 -105 -19 17 0 6 2 3 0 }

{ -21 -7 -42 14 -24 -3 0 0 11 -47 -7 3 -5 9 1 2 }

{ -2 -32 -2 -66 3 7 1 2 -11 13 -70 5 43 -2 3 0 }

{ -3 11 -63 9 4 -5 2 -1 -22 94 -4 -6 -4 -4 1 -2 }

{ -2 10 -18 16 21 3 -2 0 -2 11 6 -10 6 -3 -1 0 }

{ 3 -6 13 76 30 -11 -1 -2 -26 -8 -69 7 -9 -7 3 -1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 1 -1 0 0 1 0 0 0 0 -1 0 0 0 0 0 0 }

{ -7 -1 1 0 -1 1 1 0 -2 -1 1 0 0 0 0 0 }

{ 6 -5 0 1 2 -1 0 0 1 -1 0 0 1 0 0 0 }

{ 1 3 -2 -1 3 2 -2 -1 0 1 0 0 1 1 -1 0 }

{ 2 0 -8 1 3 1 -1 1 0 -1 -2 0 1 0 -1 0 }

{ 5 -4 -2 0 4 -2 0 1 0 0 0 0 2 -1 0 0 }

{ 3 6 -3 -7 -1 3 3 -1 1 0 -1 0 0 1 1 -1 }

{ 1 14 -2 -8 -2 1 -3 0 2 2 -1 -2 0 1 -1 0 }

{ -2 8 1 5 -2 0 -3 1 1 1 0 2 -1 0 -1 0 }

{ 2 -4 -18 -3 -1 -1 -2 -2 1 -2 -2 0 0 0 -1 -1 }

{ -14 8 8 2 1 2 -1 -2 3 0 -1 0 0 0 0 0 }

{ 0 -1 19 -1 1 0 -1 -6 -1 1 2 0 1 0 0 -2 }

{ 8 -14 -3 43 -1 2 7 -1 1 -2 1 3 -1 1 1 0 }

{ 10 23 -19 -5 0 -6 -4 6 3 -2 1 1 0 -1 0 0 }

{ -1 5 -1 -6 -1 -1 -1 -1 -1 0 0 0 0 0 0 -1 }

{ -10 -34 -25 13 -1 0 11 5 1 -1 1 -2 0 0 2 0 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 1, and lfnstIdx is equal to 1 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ 110 -49 -3 -4 -1 -1 0 -1 -38 -1 10 0 2 0 1 0 }

{ -43 -19 17 -1 3 0 1 0 -98 46 14 -1 2 0 1 0 }

{ -19 17 -7 3 -2 1 -1 0 -32 -59 29 3 4 0 2 0 }

{ -35 -103 39 1 7 0 2 0 38 -13 25 -6 1 -1 0 0 }

{ 9 5 -6 -1 -1 0 -1 0 42 4 21 -11 1 -3 1 -1 }

{ -5 -5 -28 9 -3 2 -1 1 -20 -78 22 16 1 3 0 1 }

{ 14 17 27 -12 1 -3 1 -1 8 19 -13 4 -2 1 -1 0 }

{ 7 35 17 -4 -1 0 0 0 3 8 54 -17 1 -2 1 -1 }

{ -13 -27 -101 24 -8 6 -3 2 11 43 6 28 -6 3 -1 1 }

{ -11 -13 -3 -10 3 -1 1 0 -19 -19 -37 8 4 2 0 1 }

{ -4 -10 -24 -11 3 -2 0 -1 -6 -37 -45 -17 8 -2 2 -1 }

{ -2 1 13 -17 3 -5 1 -2 3 0 -55 22 6 1 1 0 }

{ 3 1 5 -15 1 -2 1 -1 7 4 -7 29 -1 2 -1 1 }

{ -4 -8 -1 -50 6 -4 2 -2 -1 5 -22 20 6 1 0 0 }

{ 5 -1 26 102 -13 12 -4 4 -4 -2 -40 -7 -23 3 -5 1 }

{ -5 -6 -27 -22 -12 0 -3 0 -5 8 -20 -83 0 0 0 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ -9 13 1 -2 0 0 0 0 -4 2 -3 0 0 0 0 0 }

{ 26 26 -15 -3 -2 -1 -1 0 11 -7 -9 2 0 0 0 0 }

{ -72 43 34 -9 3 -2 1 -1 13 36 -18 -10 0 -2 0 -1 }

{ -1 7 6 -7 1 -1 0 0 -13 14 2 -4 2 -1 0 0 }

{ 21 70 -32 -21 0 -4 -1 -1 34 -26 -57 11 4 2 0 1 }

{ 80 -6 25 -5 -4 -1 -1 0 6 -24 7 -9 0 0 0 0 }

{ 48 -1 48 -15 -4 -2 -1 -1 1 60 -28 -42 5 -6 1 -2 }

{ 10 14 -11 -34 4 -4 1 -1 -80 -7 -6 2 15 0 3 0 }

{ -3 14 21 -12 -7 -2 -1 -1 -23 10 -4 -12 3 0 1 0 }

{ -12 -30 3 -9 5 0 1 0 -56 -9 -47 8 21 1 4 1 }

{ 17 14 -58 14 15 0 2 0 -10 34 -7 28 4 -1 1 0 }

{ 8 74 21 40 -14 0 -2 0 -36 -8 11 -13 -23 1 -3 0 }

{ 8 3 12 -14 -9 -1 -1 0 4 29 -15 31 10 4 1 1 }

{ -16 -15 18 -29 -11 2 -2 1 40 -45 -19 -22 31 2 4 1 }

{ -1 5 8 -23 7 2 1 1 10 -11 -13 -3 12 -3 2 0 }

{ 9 7 24 -20 41 3 6 1 15 20 12 11 17 -9 1 -2 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ -2 2 0 1 -1 1 0 0 -1 1 0 0 -1 0 0 0 }

{ 9 -3 -1 2 3 -3 0 0 4 -1 0 0 2 -1 0 0 }

{ 3 0 -12 3 6 1 -3 2 1 -1 -2 0 3 1 -1 1 }

{ -2 11 -6 -2 -2 4 -3 0 0 3 -2 0 -1 1 -1 0 }

{ -4 -32 5 24 1 -6 12 4 -3 -2 4 -2 0 -1 0 0 }

{ -7 3 13 -4 -3 5 1 -5 -2 3 1 -2 -1 2 -1 -2 }

{ 11 -11 -51 11 -2 -10 -2 13 2 -6 -4 4 -2 -3 2 2 }

{ -16 46 1 3 2 7 -24 0 2 -2 -5 8 1 -1 -2 2 }

{ 2 9 -10 0 1 -5 -4 4 2 -2 2 2 0 -2 1 0 }

{ -11 -30 10 59 -2 8 41 8 2 5 6 -7 -1 3 5 -2 }

{ 23 34 -31 4 10 -22 -30 22 4 -15 9 20 2 -5 9 4 }

{ -36 6 16 -14 2 19 -4 -12 -1 0 -7 -3 0 2 -2 -1 }

{ 61 22 55 14 13 3 -9 -65 1 -11 -21 -7 0 0 -1 3 }

{ -25 41 0 12 9 7 -42 12 -3 -14 2 28 5 1 6 2 }

{ -9 23 4 9 14 9 -14 -4 0 -12 -7 6 3 0 6 3 }

{ -26 -1 18 -1 -12 32 3 -18 -5 10 -25 -5 -2 1 -8 10 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 1, and lfnstIdx is equal to 2 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ 80 -49 6 -4 1 -1 1 -1 -72 36 4 0 1 0 0 0 }

{ -72 -6 17 0 3 0 1 0 -23 58 -21 2 -3 1 -1 0 }

{ -50 19 -15 4 -1 1 -1 1 -58 -2 30 -3 4 -1 2 0 }

{ -33 -43 28 -7 4 -2 2 -1 -38 11 -8 4 1 1 0 0 }

{ 10 66 -21 -3 -3 0 -1 0 -53 -41 -2 16 -1 4 -1 1 }

{ 18 14 13 -9 2 -2 1 -1 34 32 -31 12 -5 2 -2 1 }

{ 21 66 -1 9 -4 2 -1 1 -21 41 -30 -10 0 -2 0 -1 }

{ 1 -6 -24 17 -5 3 -2 1 24 10 39 -21 5 -4 2 -1 }

{ 9 33 -24 1 4 0 1 0 6 50 26 1 -10 0 -2 0 }

{ -7 -9 -32 14 -3 3 -1 1 -23 -28 0 -5 -1 0 0 0 }

{ 6 30 69 -18 5 -4 3 -1 -3 -11 -34 -16 9 -4 2 -1 }

{ 1 -8 24 -3 7 -2 2 -1 -6 -51 -6 -4 -5 0 -1 0 }

{ 4 10 4 17 -9 4 -2 1 5 14 32 -15 9 -3 2 -1 }

{ -3 -9 -23 10 -10 3 -3 1 -5 -14 -16 -27 13 -5 2 -1 }

{ 2 11 22 2 9 -2 2 0 -6 -7 20 -32 -3 -4 0 -1 }

{ 2 -3 8 14 -5 3 -1 1 -2 -11 5 -18 8 -3 2 -1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 26 0 -12 2 -2 1 -1 0 -7 -9 6 1 0 0 0 0 }

{ 55 -46 -1 6 -2 1 -1 0 -22 7 17 -7 2 -1 1 0 }

{ 6 57 -34 0 -2 0 -1 0 34 -48 -2 14 -4 3 -1 1 }

{ -55 24 26 -5 2 -1 1 0 15 46 -40 -1 -1 0 -1 0 }

{ 36 -5 41 -20 3 -3 1 -1 -30 26 -32 -3 7 -2 2 -1 }

{ 40 4 -4 -9 -3 -2 -1 -1 27 -31 -43 19 -2 3 -1 1 }

{ -35 -17 -3 26 -6 5 -2 2 56 3 18 -25 -1 -2 -1 -1 }

{ 33 32 -30 4 -3 -1 -1 0 -4 13 -16 -10 0 -1 0 0 }

{ -27 1 -28 -21 16 -5 3 -2 -23 36 -2 40 -17 4 -3 1 }

{ -36 -59 -24 14 4 2 1 1 -23 -26 23 26 -3 5 0 2 }

{ -16 35 -35 30 -9 3 -2 1 -57 -13 6 4 -5 5 -1 1 }

{ 38 -1 0 25 6 2 1 1 47 20 35 1 -27 1 -5 0 }

{ 7 13 19 15 -8 1 -1 0 3 25 30 -18 1 -2 0 -1 }

{ -1 -13 -30 11 -5 2 -1 0 -5 -8 -22 -16 10 0 1 0 }

{ 13 -5 -28 6 18 -4 3 -1 -26 27 -14 6 -20 0 -2 0 }

{ 12 -23 -19 22 2 0 1 0 23 41 -7 35 -10 4 -1 1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 3 5 -1 -2 -2 -2 -1 1 1 1 0 0 -1 -1 0 0 }

{ 9 5 -12 1 -3 -4 4 2 4 1 -2 -1 -1 -1 1 0 }

{ -10 7 21 -10 6 1 -11 0 -1 -1 4 2 3 0 -2 -1 }

{ 17 -38 1 17 -3 11 15 -11 3 -1 -10 1 0 1 3 2 }

{ 15 -8 1 17 -1 -2 4 -8 2 0 -1 3 0 0 0 -1 }

{ 7 -49 52 10 -11 22 7 -26 -1 -6 -9 6 -2 2 4 -2 }

{ -15 -13 -27 9 9 -6 20 5 -3 2 -6 -9 3 -3 1 5 }

{ 24 -26 -37 33 5 -32 55 -5 -7 22 -14 -22 1 -9 -3 13 }

{ 43 -13 4 -41 -19 -2 -24 17 11 -4 8 4 -3 -3 -3 -3 }

{ 10 -26 38 7 -12 11 42 -22 -5 20 -14 -15 -1 -2 1 6 }

{ 28 10 4 7 0 -15 7 -10 -1 7 -2 2 1 -3 0 0 }

{ 37 -37 -9 -47 -28 5 0 18 8 6 0 -8 -4 -3 -3 1 }

{ 11 24 22 -11 -3 37 -13 -58 -5 12 -63 26 9 -15 11 8 }

{ 0 -29 -27 6 -27 -10 -30 9 -3 -10 -7 77 9 -13 45 -8 }

{ -76 -26 -4 -7 12 51 5 24 7 -17 -16 -12 -5 4 2 13 }

{ 5 7 23 5 69 -38 -8 -32 -15 -31 24 11 2 18 11 -15 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 2, and lfnstIdx is equal to 1 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ -121 33 4 4 1 2 0 1 -1 -1 1 0 0 0 0 0 }

{ 0 -2 0 0 0 0 0 0 121 -23 -7 -3 -2 -1 -1 0 }

{ -20 19 -5 2 -1 1 0 0 16 3 -2 0 0 0 0 0 }

{ 32 108 -43 10 -9 3 -3 1 4 19 -7 1 -1 0 0 0 }

{ -3 0 -1 0 0 0 0 0 -29 11 -2 1 0 0 0 0 }

{ -4 -12 -3 1 -1 0 0 0 19 105 -31 7 -6 1 -2 0 }

{ 7 1 2 0 0 0 0 0 4 3 -2 0 0 0 0 0 }

{ -8 -31 14 -4 3 -1 1 0 9 43 0 1 -1 0 0 0 }

{ -15 -43 -100 23 -12 6 -4 2 -6 -17 -48 10 -5 2 -1 1 }

{ -3 1 2 0 0 0 0 0 -6 3 1 0 0 0 0 0 }

{ -1 -6 -3 2 -1 0 0 0 -6 -35 9 0 2 0 0 0 }

{ -5 -14 -48 2 -5 1 -2 0 10 24 99 -17 10 -4 3 -1 }

{ -2 0 2 0 0 0 0 0 -2 0 1 0 0 0 0 0 }

{ -2 -10 -4 0 0 0 0 0 3 11 -1 -1 0 0 0 0 }

{ -2 -3 -25 -2 -3 0 -1 0 -1 -3 -1 4 -2 2 0 1 }

{ 4 -4 28 103 -42 24 -9 7 1 2 4 0 3 -1 0 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 24 -5 -1 -1 0 0 0 0 5 -1 0 0 0 0 0 0 }

{ 17 1 -2 0 0 0 0 0 -27 4 2 0 0 0 0 0 }

{ -120 14 8 1 3 1 1 0 -18 -2 3 0 1 0 0 0 }

{ 11 -30 9 -2 1 -1 0 0 0 -8 2 0 0 0 0 0 }

{ 12 7 -1 0 0 0 0 0 -117 12 9 1 3 0 1 0 }

{ 9 46 -6 0 0 0 0 0 8 -29 9 -3 1 0 0 0 }

{ 22 -8 1 -1 0 0 0 0 -28 -9 4 0 1 0 0 0 }

{ -13 -105 17 -2 2 0 0 0 -8 -25 -3 0 0 0 0 0 }

{ 1 -5 19 -6 3 -1 1 0 2 7 15 -3 1 -1 0 0 }

{ 0 3 -2 0 0 0 0 0 -20 8 -2 0 0 0 0 0 }

{ 1 -6 11 -2 2 0 1 0 -9 -100 17 -1 1 0 0 0 }

{ 4 14 32 0 2 0 1 0 -4 0 -39 6 -4 1 -1 0 }

{ -1 -1 1 -1 0 0 0 0 -1 -4 2 0 0 0 0 0 }

{ -6 -40 -15 6 -2 1 0 0 5 57 -6 2 0 0 0 0 }

{ -7 -8 -97 17 -9 3 -3 1 -8 -26 -61 -1 -3 -1 -1 -1 }

{ -1 0 -9 -42 17 -9 3 -2 -1 1 -14 6 -4 2 -1 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 3 -1 0 0 2 -1 0 0 2 -1 0 0 1 0 0 0 }

{ -12 2 1 0 -5 1 0 0 -1 0 0 0 -2 0 0 0 }

{ 17 -3 -1 0 6 -1 -1 0 2 0 0 0 2 0 0 0 }

{ -7 -1 2 0 -3 -1 1 0 -2 -2 1 0 0 0 0 0 }

{ -32 -3 3 0 12 -2 -1 0 7 0 0 0 1 0 0 0 }

{ -3 -19 3 0 -4 -6 1 0 0 0 0 0 0 -1 0 0 }

{ 117 -10 -8 0 32 1 -4 0 3 1 -1 0 -3 1 0 0 }

{ -7 32 -5 1 -1 4 0 0 2 -1 0 0 1 0 -1 0 }

{ 4 10 5 -1 0 3 1 0 -2 1 2 0 -1 1 1 0 }

{ 30 13 -3 0 -116 6 10 0 -35 -5 4 0 -3 -1 0 0 }

{ -10 -63 1 2 -17 3 -4 0 -1 9 -1 0 3 4 -1 0 }

{ 2 -3 -4 0 2 -2 -2 0 0 0 -1 0 0 -1 -1 0 }

{ -8 -2 -1 1 30 4 -4 1 -102 4 8 -1 -69 -2 6 -1 }

{ 1 -95 18 -6 -10 -34 -2 0 -4 17 -2 0 0 2 1 0 }

{ 2 10 24 -7 5 9 19 -1 0 1 4 0 -2 0 1 0 }

{ -1 -2 -4 4 0 3 1 -1 0 2 0 -2 2 0 0 0 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 2, and lfnstIdx is equal to 2 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ 87 -41 3 -4 1 -1 0 -1 -73 28 2 1 1 1 0 0 }

{ -75 4 7 0 2 0 1 0 -41 36 -7 3 -1 1 0 0 }

{ 26 -44 22 -6 4 -2 1 -1 77 24 -22 2 -4 0 -1 0 }

{ -39 -68 37 -7 6 -2 2 0 -9 56 -21 1 -2 0 -1 0 }

{ 10 -20 2 0 1 0 0 0 50 -1 8 -5 1 -1 0 0 }

{ -21 -45 8 -2 3 -1 1 0 -7 -30 26 -8 3 -1 1 -1 }

{ -4 -2 -55 28 -8 5 -3 2 -2 37 43 -19 1 -2 1 -1 }

{ 2 19 47 -23 6 -4 2 -1 -23 -22 -44 17 -2 2 -1 0 }

{ -19 -62 -9 3 0 0 0 0 -12 -56 27 -7 3 -1 1 0 }

{ 1 9 -5 0 -1 0 0 0 0 22 -1 2 0 1 0 0 }

{ 5 17 -9 0 -2 1 0 0 13 54 -2 7 -1 1 0 0 }

{ 7 27 56 -2 10 -3 3 -1 -2 -6 8 -28 3 -4 1 -1 }

{ 0 0 19 -4 3 -2 2 -1 -3 -13 10 -4 1 0 0 0 }

{ -3 0 -27 -80 40 -16 6 -4 4 3 31 61 -22 7 -1 1 }

{ 1 2 -8 6 -1 1 0 0 2 8 -5 -1 0 0 0 0 }

{ -4 -18 -57 8 -8 1 -3 0 -5 -20 -69 7 -6 2 -2 1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 30 -5 -6 1 -1 0 0 0 -8 -3 3 0 0 0 0 0 }

{ 72 -29 -2 0 -1 0 -1 0 -37 6 7 -2 1 0 0 0 }

{ 7 -38 10 0 1 0 0 0 -51 27 4 -3 2 -1 1 0 }

{ -45 4 -3 6 -1 2 0 1 49 -13 3 -3 -1 0 0 0 }

{ 66 17 -24 4 -3 1 -1 0 13 -49 15 1 0 0 0 0 }

{ -9 69 -33 5 -2 0 -1 0 -44 -31 10 7 -2 2 0 1 }

{ -47 -34 -27 5 4 -1 1 0 -39 -2 27 4 -2 1 0 0 }

{ -33 3 22 -2 -4 1 -1 0 -58 -17 6 -6 7 -1 1 0 }

{ 7 -8 16 -6 4 -2 1 -1 -15 54 -23 2 -1 0 0 0 }

{ -13 17 0 -2 0 -1 0 0 -46 -10 -10 4 -1 1 0 0 }

{ 4 51 -3 -6 -1 -1 0 0 -20 6 -34 9 -2 2 -1 0 }

{ -1 -4 -68 35 -5 5 -2 1 0 35 43 -4 -6 1 -1 0 }

{ -6 -37 -18 -5 2 -2 1 -1 6 -6 -7 25 -6 4 -1 1 }

{ -4 -7 -26 -6 -10 6 -4 1 3 8 14 -18 15 -5 2 -1 }

{ 1 24 3 5 -1 1 0 0 -3 12 6 -10 1 -1 0 0 }

{ 1 4 0 33 -7 5 -2 1 0 -9 53 -22 3 -1 0 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 3 2 -1 0 -2 -1 0 0 1 1 0 0 -1 0 0 0 }

{ 12 3 -4 0 -3 -2 1 0 4 0 0 0 -1 0 0 0 }

{ 31 -5 -8 3 -14 0 5 -1 6 1 -3 0 -4 -1 1 0 }

{ -19 2 0 0 5 1 1 0 -2 0 -1 0 1 0 0 0 }

{ -53 34 6 -5 30 -7 -11 3 -11 -2 5 1 4 2 -1 -1 }

{ 49 7 2 -6 -23 -3 -2 2 9 4 0 0 -2 -1 -1 0 }

{ -11 32 -8 -7 27 -12 -6 6 -13 0 4 -3 3 -1 -2 1 }

{ -23 40 -2 5 43 -11 -8 -1 -18 -4 5 2 4 3 0 -1 }

{ -42 -25 4 6 34 8 2 -2 -15 -1 0 -1 3 2 0 1 }

{ -80 -27 20 -4 -66 23 -2 -2 20 -3 -2 3 -14 2 3 -1 }

{ 16 -52 28 1 59 15 -8 -5 -28 -7 2 2 10 3 0 -1 }

{ -14 -38 -12 -10 9 5 7 6 -9 7 -4 -3 4 -4 0 3 }

{ 16 10 55 -24 15 46 -52 1 35 -43 10 12 -23 13 5 -8 }

{ -2 -4 -1 13 0 2 -4 -3 3 -1 2 1 -2 0 -2 -1 }

{ -9 -1 -25 10 45 -11 18 2 86 1 -13 -4 -65 -6 7 2 }

{ 4 -27 -2 -9 5 36 -13 5 -7 -17 1 2 4 6 4 -1 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 3, and lfnstIdx is equal to 1 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ -115 37 9 2 2 1 1 0 10 -29 8 0 1 0 1 0 }

{ 15 51 -18 0 -3 0 -1 0 -95 7 34 -3 5 -1 2 0 }

{ 29 -22 16 -6 3 -2 1 -1 -4 -80 12 15 0 3 0 1 }

{ -36 -98 25 5 4 1 2 1 -59 11 -17 1 1 1 0 0 }

{ -6 18 3 -3 -1 0 0 0 -50 -5 -38 12 0 2 0 1 }

{ 4 15 52 -13 5 -3 2 -1 -17 -45 16 24 -2 4 -1 2 }

{ -20 -7 -43 4 0 1 -1 1 -7 35 0 12 -4 1 -1 0 }

{ 4 29 1 26 -5 4 -2 1 -17 -7 -73 6 6 2 1 1 }

{ 12 13 10 2 -1 3 -1 1 17 -2 -46 12 7 0 2 0 }

{ 5 20 90 -17 4 -3 2 -1 6 66 8 28 -7 3 -1 1 }

{ -3 -4 -34 -12 2 -1 -1 0 5 25 11 43 -10 4 -2 1 }

{ -1 -3 2 19 -2 4 -1 2 9 3 -35 22 11 1 2 0 }

{ 10 -4 -6 12 5 1 1 0 11 -9 -12 -2 -7 0 -1 0 }

{ 4 6 14 53 -4 4 0 2 0 -1 -20 -13 3 2 -1 1 }

{ 2 9 13 37 19 6 2 2 -9 -3 -9 -28 -20 -4 -3 -1 }

{ 3 -3 12 84 -12 8 -2 3 6 13 50 -1 45 1 7 0 }

},

lowFreqTransMatrix [ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ 23 -8 -8 1 -1 0 0 0 3 3 -2 -1 0 0 0 0 }

{ 23 -47 1 6 0 1 0 1 8 5 -12 0 -1 0 0 0 }

{ 45 7 -59 7 -2 1 -1 0 -15 41 -3 -16 2 -3 0 -1 }

{ 6 -13 7 -3 0 0 0 0 14 -4 -14 3 -1 0 0 0 }

{ 3 67 -7 -40 3 -6 1 -3 -12 -13 65 -3 -10 0 -1 0 }

{ -87 -8 -14 7 8 1 2 0 23 -35 -6 -3 1 1 0 0 }

{ -51 -2 -57 5 15 0 4 0 7 39 5 -55 1 -7 1 -3 }

{ -5 21 -3 5 -1 -3 0 -1 -11 2 -52 -3 27 -2 5 0 }

{ 16 -45 -9 -53 6 1 1 0 70 16 8 -4 -37 1 -7 0 }

{ 29 5 -19 12 9 -1 1 0 -10 14 -1 -13 7 0 1 0 }

{ 23 20 -40 12 21 -3 4 -1 25 -28 -10 5 8 6 0 2 }

{ -7 -65 -19 -22 11 4 2 1 -75 -18 3 -1 -10 2 0 1 }

{ 33 -10 -4 18 18 -4 4 -1 28 -72 1 -49 15 2 2 1 }

{ -3 1 -5 35 -16 -6 -1 -2 46 29 13 21 37 -5 4 -1 }

{ 1 18 9 28 24 6 2 2 -20 -5 -25 -33 -36 9 -2 2 }

{ -2 18 -22 -37 -13 14 0 3 1 -12 -3 2 -15 -8 1 -1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ 4 0 0 -1 1 1 0 0 2 0 0 0 0 0 0 0 }

{ 3 -3 1 -1 2 1 -2 0 1 -1 0 0 1 1 -1 0 }

{ 1 0 7 -2 -3 6 1 -2 0 0 1 0 -1 2 0 -1 }

{ 2 8 -3 -5 2 0 0 0 0 3 0 -1 1 0 0 0 }

{ 9 -20 -5 22 -2 0 0 -1 2 -3 -2 3 -1 0 1 0 }

{ 2 5 -17 0 3 -1 -1 -5 0 1 -4 0 1 0 0 -2 }

{ 1 -10 41 2 4 -3 -2 3 -1 -2 7 1 1 -1 -1 0 }

{ 0 27 8 -58 2 -5 25 3 0 3 0 -5 0 -2 7 0 }

{ -12 29 3 21 4 0 5 -1 -3 4 1 4 2 0 1 0 }

{ 0 -6 13 -4 0 -4 1 5 0 -1 -1 1 0 -1 0 0 }

{ -4 21 -64 -8 -5 19 10 -48 3 -1 10 -3 0 4 3 -6 }

{ 2 -35 -27 4 1 8 -17 -19 3 0 3 -6 0 2 -1 -2 }

{ 56 -23 22 -1 4 -1 -15 26 6 4 -10 0 0 2 -3 2 }

{ -10 -53 -18 8 9 12 -41 -25 -2 2 13 -16 4 1 -5 1 }

{ -13 42 1 57 -22 -2 -25 -28 5 6 19 -12 -5 -3 -2 4 }

{ 19 14 -4 -12 -4 5 17 8 2 -4 -4 4 -2 2 1 0 }

},

* Otherwise, if nTrS is equal to 48, lfnstTrSetIdx is equal to 3, and lfnstIdx is equal to 2 the following applies:

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15

lowFreqTransMatrixCol0to15 =

{

{ 109 -26 -8 -3 -2 -1 -1 0 -50 28 2 1 0 0 0 0 }

{ -39 31 -5 2 -1 1 0 0 -95 6 18 0 4 0 1 0 }

{ 29 -3 -2 -2 0 0 0 0 0 -41 9 0 2 0 1 0 }

{ 18 96 -23 2 -5 1 -2 0 -10 6 10 -2 1 -1 1 0 }

{ -29 -60 16 -2 3 -1 1 0 -52 9 -17 5 -2 1 -1 1 }

{ -23 -5 -15 5 -2 1 -1 1 2 79 -13 -4 -2 -1 -1 0 }

{ -7 -3 12 -3 3 -1 1 0 -31 -62 8 7 0 2 0 1 }

{ 1 -26 5 0 1 0 1 0 24 -3 43 -6 4 -2 1 -1 }

{ 11 14 6 -3 1 -1 1 0 10 -7 -9 3 -2 1 -1 0 }

{ -10 -11 -47 3 -4 1 -1 0 5 28 11 -2 -1 0 0 0 }

{ -8 -24 -99 11 -10 3 -4 1 -5 -36 19 -26 4 -5 1 -2 }

{ -5 1 -1 0 1 0 0 0 -10 -14 -6 8 0 1 0 0 }

{ 1 12 -20 21 -4 5 -2 2 -5 -2 -75 9 -1 2 -1 1 }

{ 2 -9 -18 8 -3 3 -1 1 3 -25 -62 -6 0 -2 0 -1 }

{ 4 9 39 18 0 2 0 1 -6 -16 -22 -37 5 -5 1 -2 }

{ -7 -2 15 -6 1 -1 1 -1 -11 -3 22 -14 0 -2 1 -1 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15

lowFreqTransMatrixCol16to31 =

{

{ -18 -8 6 0 1 0 1 0 6 -2 -3 0 0 0 0 0 }

{ 32 -49 5 1 1 0 0 0 27 -1 -14 2 -2 1 -1 0 }

{ 86 4 -33 2 -6 1 -2 0 -32 58 1 -7 0 -2 0 -1 }

{ -14 26 2 -4 1 -1 0 0 -43 -9 35 -2 4 -1 1 0 }

{ 13 56 -2 -9 0 -2 0 -1 -34 -18 41 0 3 0 1 0 }

{ -9 1 5 -1 1 0 0 0 -4 49 2 -14 1 -3 0 -1 }

{ -75 9 -45 5 -1 1 -1 0 14 35 0 -23 2 -5 1 -2 }

{ -7 -64 9 14 0 3 0 1 -12 -4 5 3 -1 1 0 0 }

{ 22 21 1 -21 2 -4 1 -2 92 1 53 0 -9 1 -2 0 }

{ -12 -2 -38 2 0 1 0 0 16 38 11 -16 -1 -3 0 -2 }

{ 0 25 41 5 -3 1 0 0 10 -5 -7 12 2 1 0 0 }

{ -17 -2 7 -5 3 -1 0 0 -16 13 3 31 -1 6 0 2 }

{ -1 -2 -16 -4 0 -1 0 0 -7 7 -31 0 3 0 0 0 }

{ -6 -61 14 -51 2 -6 0 -2 -19 0 40 -7 -17 0 -3 0 }

{ -5 15 63 9 -16 0 -3 0 18 42 -18 27 15 1 3 1 }

{ -18 -7 30 -9 -4 0 -1 0 -35 23 23 10 -17 1 -3 0 }

},

lowFreqTransMatrix[ m ][ n ] = lowFreqTransMatrixCol32to47[ m − 32 ][ n ] with m = 32..47, n = 0..15

lowFreqTransMatrixCol32to47 =

{

{ -3 2 1 -1 0 0 0 0 -2 0 0 0 0 0 0 0 }

{ 3 5 -3 -2 4 1 -1 -1 2 0 0 0 2 0 0 0 }

{ -14 -8 20 0 -2 -3 0 4 -1 -1 0 0 -1 1 0 0 }

{ 14 -40 1 10 2 1 -10 1 2 -4 -1 -1 0 0 -1 0 }

{ 19 -36 -10 13 3 6 -14 -1 3 1 -1 -3 1 1 -1 -1 }

{ -31 -14 56 -1 13 -37 -4 20 -2 2 -10 0 2 -4 0 -1 }

{ 1 -8 32 -1 7 -12 -4 10 0 2 -6 -1 2 0 0 -2 }

{ 8 -59 -3 26 14 6 -58 6 -5 17 -7 -18 3 3 -1 -5 }

{ -21 -11 1 40 -5 -4 -24 5 -4 5 -6 -5 0 0 0 -3 }

{ 12 -9 -22 7 -8 60 4 -36 -6 -15 54 7 3 -7 -8 14 }

{ -1 1 9 -3 -3 -14 -3 12 2 4 -13 -2 -1 3 2 -4 }

{ -93 -15 -46 -3 23 -19 0 -47 8 4 8 3 2 3 0 0 }

{ 4 11 -12 4 -12 14 -50 -1 -8 32 -4 -54 2 0 30 -15 }

{ 13 -4 11 9 17 0 24 5 1 -12 4 28 0 0 -15 8 }

{ 12 -34 9 -24 4 28 -2 4 -11 -4 30 2 5 -13 -4 18 }

{ -19 53 6 48 -65 12 -12 11 -8 -16 10 -21 -2 -12 6 2 }

},

#### Transformation process

Inputs to this process are:

* a variable nTbS specifying the horizontal sample size of transformed samples,
* a variable nonZeroS specifying the horizontal sample size of non-zero scaled transform coefficients,
* a list of scaled transform coefficients x[ j ] with j = 0..nonZeroS − 1,
* a transform kernel type variable trType.

Output of this process is the list of transformed samples y[ i ] with i = 0..nTbS − 1.

The transformation matrix derivation process as specified in clause 8.7.4.5 in invoked with the transform size nTbS and the transform kernel Type trType as inputs, and the transformation maxtrix transMatrix as output.

Depending on the value of trType, the following applies:, the list of transformed samples y[ i ] with i = 0..nTbS − 1 is derived as follows:

* If trType is equal to 0, the following transform matrix multiplication applies:

y[i]= with i = 0..nTbS − 1 (8‑970)

* Otherwise (trType is equal to 1 or trType is equal to 2), the following transform matrix multiplication applies:

y[i]= with i = 0..nTbS − 1 (8‑971)

#### Transformation matrix derivation process

Inputs to this process are:

* a variable nTbS specifying the horizontal sample size of scaled transform coefficients,
* the transformation kernel type trType.

Output of this process is the transformation matrix transMatrix.

The transformation matrix transMatrix is derived based on trType and nTbs as follows:

* If trType is equal to 0, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..63 (8‑972)

transMatrixCol0to15 = (8‑973)

{

{ 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 }

{ 91 90 90 90 88 87 86 84 83 81 79 77 73 71 69 65 }

{ 90 90 88 85 82 78 73 67 61 54 46 38 31 22 13 4 }

{ 90 88 84 79 71 62 52 41 28 15 2 −11 −24 −37 −48 −59 }

{ 90 87 80 70 57 43 25 9 −9 −25 −43 −57 −70 −80 −87 −90 }

{ 90 84 73 59 41 20 −2 −24 −44 −62 −77 −86 −90 −90 −83 −71 }

{ 90 82 67 46 22 −4 −31 −54 −73 −85 −90 −88 −78 −61 −38 −13 }

{ 90 79 59 33 2 −28 −56 −77 −88 −90 −81 −62 −37 −7 24 52 }

{ 89 75 50 18 −18 −50 −75 −89 −89 −75 −50 −18 18 50 75 89 }

{ 88 71 41 2 −37 −69 −87 −90 −73 −44 −7 33 65 86 90 77 }

{ 88 67 31 −13 −54 −82 −90 −78 −46 −4 38 73 90 85 61 22 }

{ 87 62 20 −28 −69 −90 −84 −56 −11 37 73 90 81 48 2 −44 }

{ 87 57 9 −43 −80 −90 −70 −25 25 70 90 80 43 −9 −57 −87 }

{ 86 52 −2 −56 −87 −84 −48 7 59 88 83 44 −11 −62 −90 −81 }

{ 85 46 −13 −67 −90 −73 −22 38 82 88 54 −4 −61 −90 −78 −31 }

{ 84 41 −24 −77 −90 −56 7 65 91 69 11 −52 −88 −79 −28 37 }

{ 83 36 −36 −83 −83 −36 36 83 83 36 −36 −83 −83 −36 36 83 }

{ 83 28 −44 −88 −73 −11 59 91 62 −7 −71 −90 −48 24 81 84 }

{ 82 22 −54 −90 −61 13 78 85 31 −46 −90 −67 4 73 88 38 }

{ 81 15 −62 −90 −44 37 88 69 −7 −77 −84 −24 56 91 52 −28 }

{ 80 9 −70 −87 −25 57 90 43 −43 −90 −57 25 87 70 −9 −80 }

{ 79 2 −77 −81 −7 73 83 11 −71 −84 −15 69 86 20 −65 −87 }

{ 78 −4 −82 −73 13 85 67 −22 −88 −61 31 90 54 −38 −90 −46 }

{ 77 −11 −86 −62 33 90 44 −52 −90 −24 69 83 2 −81 −71 20 }

{ 75 −18 −89 −50 50 89 18 −75 −75 18 89 50 −50 −89 −18 75 }

{ 73 −24 −90 −37 65 81 −11 −88 −48 56 86 2 −84 −59 44 90 }

{ 73 −31 −90 −22 78 67 −38 −90 −13 82 61 −46 −88 −4 85 54 }

{ 71 −37 −90 −7 86 48 −62 −79 24 91 20 −81 −59 52 84 −11 }

{ 70 −43 −87 9 90 25 −80 −57 57 80 −25 −90 −9 87 43 −70 }

{ 69 −48 −83 24 90 2 −90 −28 81 52 −65 −71 44 84 −20 −90 }

{ 67 −54 −78 38 85 −22 −90 4 90 13 −88 −31 82 46 −73 −61 }

{ 65 −59 −71 52 77 −44 −81 37 84 −28 −87 20 90 −11 −90 2 }

{ 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 }

{ 62 −69 −56 73 48 −79 −41 83 33 −86 −24 88 15 −90 −7 91 }

{ 61 −73 −46 82 31 −88 −13 90 −4 −90 22 85 −38 −78 54 67 }

{ 59 −77 −37 87 11 −91 15 86 −41 −73 62 56 −79 −33 88 7 }

{ 57 −80 −25 90 −9 −87 43 70 −70 −43 87 9 −90 25 80 −57 }

{ 56 −83 −15 90 −28 −77 65 44 −87 −2 88 −41 −69 73 33 −90 }

{ 54 −85 −4 88 −46 −61 82 13 −90 38 67 −78 −22 90 −31 −73 }

{ 52 −87 7 83 −62 −41 90 −20 −77 71 28 −91 33 69 −79 −15 }

{ 50 −89 18 75 −75 −18 89 −50 −50 89 −18 −75 75 18 −89 50 }

{ 48 −90 28 65 −84 7 79 −73 −15 87 −59 −37 91 −41 −56 88 }

{ 46 −90 38 54 −90 31 61 −88 22 67 −85 13 73 −82 4 78 }

{ 44 −91 48 41 −90 52 37 −90 56 33 −90 59 28 −88 62 24 }

{ 43 −90 57 25 −87 70 9 −80 80 −9 −70 87 −25 −57 90 −43 }

{ 41 −90 65 11 −79 83 −20 −59 90 −48 −33 87 −71 −2 73 −86 }

{ 38 −88 73 −4 −67 90 −46 −31 85 −78 13 61 −90 54 22 −82 }

{ 37 −86 79 −20 −52 90 −69 2 65 −90 56 15 −77 87 −41 −33 }

{ 36 −83 83 −36 −36 83 −83 36 36 −83 83 −36 −36 83 −83 36 }

{ 33 −81 87 −48 −15 71 −90 62 −2 −59 90 −73 20 44 −86 83 }

{ 31 −78 90 −61 4 54 −88 82 −38 −22 73 −90 67 −13 −46 85 }

{ 28 −73 91 −71 24 33 −77 90 −69 20 37 −79 90 −65 15 41 }

{ 25 −70 90 −80 43 9 −57 87 −87 57 −9 −43 80 −90 70 −25 }

{ 24 −65 88 −86 59 −15 −33 71 −90 83 −52 7 41 −77 91 −79 }

{ 22 −61 85 −90 73 −38 −4 46 −78 90 −82 54 −13 −31 67 −88 }

{ 20 −56 81 −91 83 −59 24 15 −52 79 −90 84 −62 28 11 −48 }

{ 18 −50 75 −89 89 −75 50 −18 −18 50 −75 89 −89 75 −50 18 }

{ 15 −44 69 −84 91 −86 71 −48 20 11 −41 65 −83 90 −87 73 }

{ 13 −38 61 −78 88 −90 85 −73 54 −31 4 22 −46 67 −82 90 }

{ 11 −33 52 −69 81 −88 91 −87 79 −65 48 −28 7 15 −37 56 }

{ 9 −25 43 −57 70 −80 87 −90 90 −87 80 −70 57 −43 25 −9 }

{ 7 −20 33 −44 56 −65 73 −81 86 −90 91 −90 87 −83 77 −69 }

{ 4 −13 22 −31 38 −46 54 −61 67 −73 78 −82 85 −88 90 −90 }

{ 2 −7 11 −15 20 −24 28 −33 37 −41 44 −48 52 −56 59 −62 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..63 (8‑974)

transMatrixCol16to31 = (8‑975)

{

{ 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 }

{ 62 59 56 52 48 44 41 37 33 28 24 20 15 11 7 2 }

{ −4 −13 −22 −31 −38 −46 −54 −61 −67 −73 −78 −82 −85 −88 −90 −90 }

{ −69 −77 −83 −87 −90 −91 −90 −86 −81 −73 −65 −56 −44 −33 −20 −7 }

{ −90 −87 −80 −70 −57 −43 −25 −9 9 25 43 57 70 80 87 90 }

{ −56 −37 −15 7 28 48 65 79 87 91 88 81 69 52 33 11 }

{ 13 38 61 78 88 90 85 73 54 31 4 −22 −46 −67 −82 −90 }

{ 73 87 90 83 65 41 11 −20 −48 −71 −86 −91 −84 −69 −44 −15 }

{ 89 75 50 18 −18 −50 −75 −89 −89 −75 −50 −18 18 50 75 89 }

{ 48 11 −28 −62 −84 −90 −79 −52 −15 24 59 83 91 81 56 20 }

{ −22 −61 −85 −90 −73 −38 4 46 78 90 82 54 13 −31 −67 −88 }

{ −79 −91 −77 −41 7 52 83 90 71 33 −15 −59 −86 −88 −65 −24 }

{ −87 −57 −9 43 80 90 70 25 −25 −70 −90 −80 −43 9 57 87 }

{ −41 15 65 90 79 37 −20 −69 −90 −77 −33 24 71 91 73 28 }

{ 31 78 90 61 4 −54 −88 −82 −38 22 73 90 67 13 −46 −85 }

{ 83 86 44 −20 −73 −90 −59 2 62 90 71 15 −48 −87 −81 −33 }

{ 83 36 −36 −83 −83 −36 36 83 83 36 −36 −83 −83 −36 36 83 }

{ 33 −41 −87 −77 −15 56 90 65 −2 −69 −90 −52 20 79 86 37 }

{ −38 −88 −73 −4 67 90 46 −31 −85 −78 −13 61 90 54 −22 −82 }

{ −86 −73 −2 71 87 33 −48 −90 −59 20 83 79 11 −65 −90 −41 }

{ −80 −9 70 87 25 −57 −90 −43 43 90 57 −25 −87 −70 9 80 }

{ −24 62 88 28 −59 −90 −33 56 90 37 −52 −90 −41 48 91 44 }

{ 46 90 38 −54 −90 −31 61 88 22 −67 −85 −13 73 82 4 −78 }

{ 88 56 −41 −91 −37 59 87 15 −73 −79 7 84 65 −28 −90 −48 }

{ 75 −18 −89 −50 50 89 18 −75 −75 18 89 50 −50 −89 −18 75 }

{ 15 −79 −69 33 91 28 −71 −77 20 90 41 −62 −83 7 87 52 }

{ −54 −85 4 88 46 −61 −82 13 90 38 −67 −78 22 90 31 −73 }

{ −90 −33 73 69 −41 −88 −2 87 44 −65 −77 28 90 15 −83 −56 }

{ −70 43 87 −9 −90 −25 80 57 −57 −80 25 90 9 −87 −43 70 }

{ −7 88 33 −79 −56 62 73 −41 −86 15 91 11 −87 −37 77 59 }

{ 61 73 −46 −82 31 88 −13 −90 −4 90 22 −85 −38 78 54 −67 }

{ 91 7 −90 −15 88 24 −86 −33 83 41 −79 −48 73 56 −69 −62 }

{ 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 64 −64 −64 64 }

{ −2 −90 11 90 −20 −87 28 84 −37 −81 44 77 −52 −71 59 65 }

{ −67 −54 78 38 −85 −22 90 4 −90 13 88 −31 −82 46 73 −61 }

{ −90 20 84 −44 −71 65 52 −81 −28 90 2 −90 24 83 −48 −69 }

{ −57 80 25 −90 9 87 −43 −70 70 43 −87 −9 90 −25 −80 57 }

{ 11 84 −52 −59 81 20 −91 24 79 −62 −48 86 7 −90 37 71 }

{ 73 31 −90 22 78 −67 −38 90 −13 −82 61 46 −88 4 85 −54 }

{ 90 −44 −59 84 2 −86 56 48 −88 11 81 −65 −37 90 −24 −73 }

{ 50 −89 18 75 −75 −18 89 −50 −50 89 −18 −75 75 18 −89 50 }

{ −20 −71 81 2 −83 69 24 −90 52 44 −90 33 62 −86 11 77 }

{ −78 −4 82 −73 −13 85 −67 −22 88 −61 −31 90 −54 −38 90 −46 }

{ −87 65 20 −86 69 15 −84 71 11 −83 73 7 −81 77 2 −79 }

{ −43 90 −57 −25 87 −70 −9 80 −80 9 70 −87 25 57 −90 43 }

{ 28 52 −91 56 24 −84 77 −7 −69 88 −37 −44 90 −62 −15 81 }

{ 82 −22 −54 90 −61 −13 78 −85 31 46 −90 67 4 −73 88 −38 }

{ 84 −81 24 48 −90 71 −7 −62 91 −59 −11 73 −88 44 28 −83 }

{ 36 −83 83 −36 −36 83 −83 36 36 −83 83 −36 −36 83 −83 36 }

{ −37 −28 79 −88 52 11 −69 91 −65 7 56 −90 77 −24 −41 84 }

{ −85 46 13 −67 90 −73 22 38 −82 88 −54 −4 61 −90 78 −31 }

{ −81 90 −62 11 44 −83 88 −59 7 48 −84 87 −56 2 52 −86 }

{ −25 70 −90 80 −43 −9 57 −87 87 −57 9 43 −80 90 −70 25 }

{ 44 2 −48 81 −90 73 −37 −11 56 −84 90 −69 28 20 −62 87 }

{ 88 −67 31 13 −54 82 −90 78 −46 4 38 −73 90 −85 61 −22 }

{ 77 −90 86 −65 33 7 −44 73 −90 87 −69 37 2 −41 71 −88 }

{ 18 −50 75 −89 89 −75 50 −18 −18 50 −75 89 −89 75 −50 18 }

{ −52 24 7 −37 62 −81 90 −88 77 −56 28 2 −33 59 −79 90 }

{ −90 82 −67 46 −22 −4 31 −54 73 −85 90 −88 78 −61 38 −13 }

{ −71 83 −90 90 −86 77 −62 44 −24 2 20 −41 59 −73 84 −90 }

{ −9 25 −43 57 −70 80 −87 90 −90 87 −80 70 −57 43 −25 9 }

{ 59 −48 37 −24 11 2 −15 28 −41 52 −62 71 −79 84 −88 90 }

{ 90 −90 88 −85 82 −78 73 −67 61 −54 46 −38 31 −22 13 −4 }

{ 65 −69 71 −73 77 −79 81 −83 84 −86 87 −88 90 −90 90 −91 }

},

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol16to31[47 −m ][ n ] (8‑976)  
 with m =32..47, n = 0..63

transMatrix[ m ][ n ] = ( n & 1 ? −1 : 1 ) \* transMatrixCol0to15[63 −m ][ n ] (8‑977)  
 with m =48..63, n = 0..63

* Otherwise, if trType is equal to 1 and nTbs is equal to 4, the following applies:

transMatrix[ m ][ n ] = (8‑978)

{

{ 29 55 74 84 }

{ 74 74 0 −74 }

{ 84 −29 −74 55 }

{ 55 −84 74 −29 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑979)

{

{ 17 32 46 60 71 78 85 86 }

{ 46 78 86 71 32 −17 −60 −85 }

{ 71 85 32 −46 −86 −60 17 78 }

{ 85 46 −60 −78 17 86 32 −71 }

{ 86 −17 −85 32 78 −46 −71 60 }

{ 78 −71 −17 85 −60 −32 86 −46 }

{ 60 −86 71 −17 −46 85 −78 32 }

{ 32 −60 78 −86 85 −71 46 −17 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑980)

{

{ 8 17 25 33 40 48 55 62 68 73 77 81 85 87 88 88 }

{ 25 48 68 81 88 88 81 68 48 25 0 −25 −48 −68 −81 −88 }

{ 40 73 88 85 62 25 −17 −55 −81 −88 −77 −48 −8 33 68 87 }

{ 55 87 81 40 −17 −68 −88 −73 −25 33 77 88 62 8 −48 −85 }

{ 68 88 48 −25 −81 −81 −25 48 88 68 0 −68 −88 −48 25 81 }

{ 77 77 0 −77 −77 0 77 77 0 −77 −77 0 77 77 0 −77 }

{ 85 55 −48 −87 −8 81 62 −40 −88 −17 77 68 −33 −88 −25 73 }

{ 88 25 −81 −48 68 68 −48 −81 25 88 0 −88 −25 81 48 −68 }

{ 88 −8 −88 17 87 −25 −85 33 81 −40 −77 48 73 −55 −68 62 }

{ 87 −40 −68 73 33 −88 8 85 −48 −62 77 25 −88 17 81 −55 }

{ 81 −68 −25 88 −48 −48 88 −25 −68 81 0 −81 68 25 −88 48 }

{ 73 −85 25 55 −88 48 33 −87 68 8 −77 81 −17 −62 88 −40 }

{ 62 −88 68 −8 −55 88 −73 17 48 −87 77 −25 −40 85 −81 33 }

{ 48 −81 88 −68 25 25 −68 88 −81 48 0 −48 81 −88 68 −25 }

{ 33 −62 81 −88 85 −68 40 −8 −25 55 −77 88 −87 73 −48 17 }

{ 17 −33 48 −62 73 −81 87 −88 88 −85 77 −68 55 −40 25 −8 }

},

* Otherwise, if trType is equal to 1 and nTbs is equal to 32, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15 (8‑981)

transMatrixCol0to15 = (8‑982)

{

{ 4 9 13 17 21 26 30 34 38 42 46 50 53 56 60 63 }

{ 13 26 38 50 60 68 77 82 86 89 90 88 85 80 74 66 }

{ 21 42 60 74 84 89 89 84 74 60 42 21 0 −21 −42 −60 }

{ 30 56 77 87 89 80 63 38 9 −21 −50 −72 −85 −90 −84 −68 }

{ 38 68 86 88 74 46 9 −30 −63 −84 −90 −78 −53 −17 21 56 }

{ 46 78 90 77 42 −4 −50 −80 −90 −74 −38 9 53 82 89 72 }

{ 53 85 85 53 0 −53 −85 −85 −53 0 53 85 85 53 0 −53 }

{ 60 89 74 21 −42 −84 −84 −42 21 74 89 60 0 −60 −89 −74 }

{ 66 90 56 −13 −74 −87 −46 26 80 84 34 −38 −85 −78 −21 50 }

{ 72 86 34 −46 −89 −63 13 78 82 21 −56 −90 −53 26 84 77 }

{ 77 80 9 −72 −84 −17 66 86 26 −60 −88 −34 53 90 42 −46 }

{ 80 72 −17 −86 −60 34 90 46 −50 −89 −30 63 85 13 −74 −78 }

{ 84 60 −42 −89 −21 74 74 −21 −89 −42 60 84 0 −84 −60 42 }

{ 86 46 −63 −78 21 90 26 −77 −66 42 87 4 −85 −50 60 80 }

{ 88 30 −78 −56 60 77 −34 −87 4 89 26 −80 −53 63 74 −38 }

{ 90 13 −87 −26 84 38 −78 −50 72 60 −63 −68 53 77 −42 −82 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15 (8‑983)

transMatrixCol16to31 = (8‑984)

{

{ 66 68 72 74 77 78 80 82 84 85 86 87 88 89 90 90 }

{ 56 46 34 21 9 −4 −17 −30 −42 −53 −63 −72 −78 −84 −87 −90 }

{ −74 −84 −89 −89 −84 −74 −60 −42 −21 0 21 42 60 74 84 89 }

{ −46 −17 13 42 66 82 90 86 74 53 26 −4 −34 −60 −78 −88 }

{ 80 90 82 60 26 −13 −50 −77 −89 −85 −66 −34 4 42 72 87 }

{ 34 −13 −56 −84 −88 −68 −30 17 60 85 87 66 26 −21 −63 −86 }

{ −85 −85 −53 0 53 85 85 53 0 −53 −85 −85 −53 0 53 85 }

{ −21 42 84 84 42 −21 −74 −89 −60 0 60 89 74 21 −42 −84 }

{ 88 72 9 −60 −90 −63 4 68 89 53 −17 −77 −86 −42 30 82 }

{ 9 −66 −88 −42 38 87 68 −4 −74 −85 −30 50 90 60 −17 −80 }

{ −90 −50 38 89 56 −30 −87 −63 21 85 68 −13 −82 −74 4 78 }

{ 4 82 68 −21 −87 −56 38 90 42 −53 −88 −26 66 84 9 −77 }

{ 89 21 −74 −74 21 89 42 −60 −84 0 84 60 −42 −89 −21 74 }

{ −17 −90 −30 74 68 −38 −88 −9 84 53 −56 −82 13 89 34 −72 }

{ −86 9 90 21 −82 −50 66 72 −42 −85 13 90 17 −84 −46 68 }

{ 30 86 −17 −89 4 90 9 −88 −21 85 34 −80 −46 74 56 −66 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 4, the following applies:

transMatrix[ m ][ n ] = (8‑985)

{

{ 84 74 55 29 }

{ 74 0 −74 −74 }

{ 55 −74 −29 84 }

{ 29 −74 84 −55 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 8, the following applies:

transMatrix[ m ][ n ] = (8‑986)

{

{ 86 85 78 71 60 46 32 17 }

{ 85 60 17 −32 −71 −86 −78 −46 }

{ 78 17 −60 −86 −46 32 85 71 }

{ 71 −32 −86 −17 78 60 −46 −85 }

{ 60 −71 −46 78 32 −85 −17 86 }

{ 46 −86 32 60 −85 17 71 −78 }

{ 32 −78 85 −46 −17 71 −86 60 }

{ 17 −46 71 −85 86 −78 60 −32 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 16, the following applies:

transMatrix[ m ][ n ] = (8‑987)

{

{ 88 88 87 85 81 77 73 68 62 55 48 40 33 25 17 8 }

{ 88 81 68 48 25 0 −25 −48 −68 −81 −88 −88 −81 −68 −48 −25 }

{ 87 68 33 −8 −48 −77 −88 −81 −55 −17 25 62 85 88 73 40 }

{ 85 48 −8 −62 −88 −77 −33 25 73 88 68 17 −40 −81 −87 −55 }

{ 81 25 −48 −88 −68 0 68 88 48 −25 −81 −81 −25 48 88 68 }

{ 77 0 −77 −77 0 77 77 0 −77 −77 0 77 77 0 −77 −77 }

{ 73 −25 −88 −33 68 77 −17 −88 −40 62 81 −8 −87 −48 55 85 }

{ 68 −48 −81 25 88 0 −88 −25 81 48 −68 −68 48 81 −25 −88 }

{ 62 −68 −55 73 48 −77 −40 81 33 −85 −25 87 17 −88 −8 88 }

{ 55 −81 −17 88 −25 −77 62 48 −85 −8 88 −33 −73 68 40 −87 }

{ 48 −88 25 68 −81 0 81 −68 −25 88 −48 −48 88 −25 −68 81 }

{ 40 −88 62 17 −81 77 −8 −68 87 −33 −48 88 −55 −25 85 −73 }

{ 33 −81 85 −40 −25 77 −87 48 17 −73 88 −55 −8 68 −88 62 }

{ 25 −68 88 −81 48 0 −48 81 −88 68 −25 −25 68 −88 81 −48 }

{ 17 −48 73 −87 88 −77 55 −25 −8 40 −68 85 −88 81 −62 33 }

{ 8 −25 40 −55 68 −77 85 −88 88 −87 81 −73 62 −48 33 −17 }

},

* Otherwise, if trType is equal to 2 and nTbs is equal to 32, the following applies:

transMatrix[ m ][ n ] = transMatrixCol0to15[ m ][ n ] with m = 0..15, n = 0..15 (8‑988)

transMatrixCol0to15 = (8‑989)

{

{ 90 90 89 88 87 86 85 84 82 80 78 77 74 72 68 66 }

{ 90 87 84 78 72 63 53 42 30 17 4 −9 −21 −34 −46 −56 }

{ 89 84 74 60 42 21 0 −21 −42 −60 −74 −84 −89 −89 −84 −74 }

{ 88 78 60 34 4 −26 −53 −74 −86 −90 −82 −66 −42 −13 17 46 }

{ 87 72 42 4 −34 −66 −85 −89 −77 −50 −13 26 60 82 90 80 }

{ 86 63 21 −26 −66 −87 −85 −60 −17 30 68 88 84 56 13 −34 }

{ 85 53 0 −53 −85 −85 −53 0 53 85 85 53 0 −53 −85 −85 }

{ 84 42 −21 −74 −89 −60 0 60 89 74 21 −42 −84 −84 −42 21 }

{ 82 30 −42 −86 −77 −17 53 89 68 4 −63 −90 −60 9 72 88 }

{ 80 17 −60 −90 −50 30 85 74 4 −68 −87 −38 42 88 66 −9 }

{ 78 4 −74 −82 −13 68 85 21 −63 −87 −30 56 89 38 −50 −90 }

{ 77 −9 −84 −66 26 88 53 −42 −90 −38 56 87 21 −68 −82 −4 }

{ 74 −21 −89 −42 60 84 0 −84 −60 42 89 21 −74 −74 21 89 }

{ 72 −34 −89 −13 82 56 −53 −84 9 88 38 −68 −74 30 90 17 }

{ 68 −46 −84 17 90 13 −85 −42 72 66 −50 −82 21 90 9 −86 }

{ 66 −56 −74 46 80 −34 −85 21 88 −9 −90 −4 89 17 −86 −30 }

},

transMatrix[ m ][ n ] = transMatrixCol16to31[ m − 16 ][ n ] with m = 16..31, n = 0..15 (8‑990)

transMatrixCol16to31 = (8‑991)

{

{ 63 60 56 53 50 46 42 38 34 30 26 21 17 13 9 4 }

{ −66 −74 −80 −85 −88 −90 −89 −86 −82 −77 −68 −60 −50 −38 −26 −13 }

{ −60 −42 −21 0 21 42 60 74 84 89 89 84 74 60 42 21 }

{ 68 84 90 85 72 50 21 −9 −38 −63 −80 −89 −87 −77 −56 −30 }

{ 56 21 −17 −53 −78 −90 −84 −63 −30 9 46 74 88 86 68 38 }

{ −72 −89 −82 −53 −9 38 74 90 80 50 4 −42 −77 −90 −78 −46 }

{ −53 0 53 85 85 53 0 −53 −85 −85 −53 0 53 85 85 53 }

{ 74 89 60 0 −60 −89 −74 −21 42 84 84 42 −21 −74 −89 −60 }

{ 50 −21 −78 −85 −38 34 84 80 26 −46 −87 −74 −13 56 90 66 }

{ −77 −84 −26 53 90 56 −21 −82 −78 −13 63 89 46 −34 −86 −72 }

{ −46 42 90 53 −34 −88 −60 26 86 66 −17 −84 −72 9 80 77 }

{ 78 74 −13 −85 −63 30 89 50 −46 −90 −34 60 86 17 −72 −80 }

{ 42 −60 −84 0 84 60 −42 −89 −21 74 74 −21 −89 −42 60 84 }

{ −80 −60 50 85 −4 −87 −42 66 77 −26 −90 −21 78 63 −46 −86 }

{ −38 74 63 −53 −80 26 89 4 −87 −34 77 60 −56 −78 30 88 }

{ 82 42 −77 −53 68 63 −60 −72 50 78 −38 −84 26 87 −13 −90 }

},

### Picture reconstruction process

#### General

Inputs to this process are:

* a location ( xCurr, yCurr ) specifying the top-left sample of the current block relative to the top‑left sample of the current picture component,
* the variables nCurrSw and nCurrSh specifying the width and height, respectively, of the current block,
* a variable cIdx specifying the colour component of the current block,
* an (nCurrSw) x (nCurrSh) array predSamples specifying the predicted samples of the current block,
* an (nCurrSw) x (nCurrSh) array resSamples specifying the residual samples of the current block.

Output of this process is a reconstructed picture sample array recSamples.

Depending on the value of the colour component cIdx, the following assignments are made:

* If cIdx is equal to 0, recSamples corresponds to the reconstructed picture sample array SL and the function clipCidx1 corresponds to Clip1Y.
* Otherwise, if cIdx is equal to 1, tuCbfChroma is set equal to tu\_cbf\_cb[ xCurr ][ yCurr ], recSamples corresponds to the reconstructed chroma sample array SCb and the function clipCidx1 corresponds to Clip1C.
* Otherwise (cIdx is equal to 2), tuCbfChroma is set equal to tu\_cbf\_cr[ xCurr ][ yCurr ], recSamples corresponds to the reconstructed chroma sample array SCr and the function clipCidx1 corresponds to Clip1C.

Depending on the value of slice\_lmcs\_enabled\_flag, the following applies:

* If slice\_lmcs\_enabled\_flag is equal to 0, the (nCurrSw)x(nCurrSh) block of the reconstructed samples recSamples at location ( xCurr, yCurr ) is derived as follows for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:

recSamples[ xCurr + i ][ yCurr + j ] = clipCidx1( predSamples[ i ][ j ] + resSamples[ i ][ j ] ) (8‑992)

* Otherwise (slice\_lmcs\_enabled\_flag is equal to 1), the following applies:
* If cIdx is equal to 0, the following applies:
* The picture reconstruction with mapping process for luma samples as specified in clause 8.7.5.2 is invoked with the luma location ( xCurr, yCurr ), the block width nCurrSw and height nCurrSh, the predicted luma sample array predSamples, and the residual luma sample array resSamples as inputs, and the output is the reconstructed luma sample array recSamples.
* Otherwise (cIdx is greater than 0), the picture reconstruction with luma dependent chroma residual scaling process for chroma samples as specified in clause 8.7.5.3 is invoked with the chroma location ( xCurr, yCurr ), the transform block width nCurrSw and height nCurrSh, the coded block flag of the current chroma transform block tuCbfChroma, the predicted chroma sample array predSamples, and the residual chroma sample array resSamples as inputs, and the output is the reconstructed chroma sample array recSamples.

#### Picture reconstruction with mapping process for luma samples

Inputs to this process are:

* a location ( xCurr, yCurr ) of the top-left sample of the current block relative to the top-left sample of the current picture,
* a variable nCurrSw specifying the block width,
* a variable nCurrSh specifying the block height,
* an (nCurrSw)x(nCurrSh) array predSamples specifying the luma predicted samples of the current block,
* an (nCurrSw)x(nCurrSh) array resSamples specifying the luma residual samples of the current block.

Outputs of this process is:

* a reconstructed luma picture sample array recSamples.

The (nCurrSw)x(nCurrSh) array of mapped predicted luma samples PredMapSamples, which is used in the picture reconstruction with luma dependent chroma residual scaling process for chroma samples as specified in clause 8.7.5.3, is derived as follows:

* If one of the following conditions is true, PredMapSamples[ i ][ j ] is set equal to predSamples[ i ][ j ] for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTRA.
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_IBC.
* CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTER and ciip\_flag[ xCurr ][ yCurr ] is equal to 1.
* Otherwise (CuPredMode[ xCurr ][ yCurr ] is equal to MODE\_INTER and ciip\_flag[ xCurr ][ yCurr ] is equal to 0), the following applies:

idxY = predSamples[ i ][ j ] >> Log2( OrgCW )  
PredMapSamples[ i ][ j ] =  LmcsPivot[ idxY ]   
  + ( ScaleCoeff[ idxY ] \* ( predSamples[ i ][ j ] − InputPivot[ idxY ] ) + ( 1 << 10 ) ) >> 11  (8‑993)  
 with i = 0..nCurrSw − 1, j = 0..nCurrSh − 1

The reconstructed luma picture sample recSamples is derived as follows:

recSamples[ xCurr + i ][ yCurr + j ] = Clip1Y( PredMapSamples[ i ][ j ]+ resSamples[ i ][ j ] ] ) (8‑994)  
 with i = 0..nCurrSw − 1, j = 0..nCurrSh − 1

#### Picture reconstruction with luma dependent chroma residual scaling process for chroma samples

Inputs to this process are:

* a location ( xCurr, yCurr ) of the top-left sample of the current transform block relative to the top-left sample of the current picture,
* a variable nCurrSw specifying the transform block width,
* a variable nCurrSh specifying the transform block height,
* a variable tuCbfChroma specifying the coded block flag of the current chroma transform block,
* an (nCurrSw)x(nCurrSh) array predSamples specifying the chroma prediction samples of the current block,
* an (nCurrSw)x(nCurrSh) array resSamples specifying the chroma residual samples of the current block.

Output of this process is a reconstructed chroma picture sample array recSamples.

The reconstructed chroma picture sample recSamples is derived as follows for i = 0..nCurrSw − 1, j = 0..nCurrSh − 1:

* If one of the following conditions is true, recSamples[ xCurr + i ][ yCurr + j ] is set equal to Clip1C( predSamples[ i ][ j ] + resSamples[ i ][ j ] ):
* slice\_chroma\_residual\_scale\_flag is equal to 0
* nCurrSw \* nCurrSh is less than or euqal to 4
* tu\_cbf\_cb [ xCurr ][ yCurr ] is equal to 0 and tu\_cbf\_cr [ xCurr ][ yCurr ] is equal to 0
* Otherwise, the following applies:
* For the derivation of the variable varScale the following ordered steps apply:

1. The variable invAvgLuma is derived as follows:

invAvgLuma = Clip1Y( (   
  + nCurrSw \* nCurrSh \* ( ( SubWidthC \* SubHeightC ) / 2 ) ) /  (8‑995)  
 ( nCurrSw \* nCurrSh \*SubWidthC \* SubHeightC ) )

1. The variable idxYInv is derived by invoking the identification of piece-wise function index process for a luma sample as specified in clause 8.8.2.3 with the variable lumaSample set equal to invAvgLuma as the input and idxYInv as the output.
2. The variable varScale is derived as follows:

varScale = ChromaScaleCoeff[ idxYInv ] (8‑996)

* The recSamples is derived as follows:
* If tuCbfChroma is equal to 1, the following applies:

resSamples[ i ][ j ] = Clip3( −( 1 << BitDepthC ), ( 1 << BitDepthC ) − 1, resSamples[ i ][ j ] ) (8‑997)

recSamples[ xCurr + i ][ yCurr + j ] = Clip1C( predSamples[ i ][ j ] +  (8‑998)  
 Sign( resSamples[ i ][ j ] ) \* ( ( Abs( resSamples[ i ][ j ] ) \* varScale + ( 1 << 10 ) ) >> 11 ) )

* Otherwise (tu\_cbf is equal to 0), the following applies:

recSamples[ xCurr + i ][ yCurr + j ] = Clip1C(predSamples[ i ][ j ] ) (8‑999)

## In-loop Filter Process

### General

The picture inverse mapping process with luma samples and the three in-loop filters, namely deblocking filter, sample adaptive offset and adaptive loop filter, are applied as specified by the following ordered steps:

1. When sps\_lmcs\_enabled\_flag is equal to 1, the following applies:

* The picture inverse mapping process for luma samples as specified in clause 8.8.2.1 is invoked with the reconstructed luma sample array SL as inputs, and the modified reconstructed luma sample array S′L after picture inverse mapping process for luma samples as outputs.
* The array S′L is assigned to the array SL (which represent the decoded picture).

1. For the deblocking filter, the following applies:

* The deblocking filter process as specified in clause 8.8.3.1 is invoked with the reconstructed picture sample array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr as inputs, and the modified reconstructed picture sample array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr after deblocking as outputs.
* The array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr are assigned to the array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr (which represent the decoded picture), respectively.

1. When sps\_sao\_enabled\_flag is equal to 1, the following applies:

* The sample adaptive offset process as specified in clause 8.8.4.1 is invoked with the reconstructed picture sample array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr as inputs, and the modified reconstructed picture sample array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr after sample adaptive offset as outputs.
* The array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr are assigned to the array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr (which represent the decoded picture), respectively.

1. When sps\_alf\_enabled\_flag is equal to 1, the following applies:

* The adaptive loop filter process as specified in clause 8.8.5.1 is invoked with the reconstructed picture sample array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr as inputs, and the modified reconstructed picture sample array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr after sample adaptive offset as outputs.
* The array S′L and, when ChromaArrayType is not equal to 0, the arrays S′Cb and S′Cr are assigned to the array SL and, when ChromaArrayType is not equal to 0, the arrays SCb and SCr (which represent the decoded picture), respectively.

### Picture inverse mapping process for luma samples

#### General

Input to this process is a reconstructed picture luma sample array SL.

The output to this process is a modified reconstructed picture luma sample array SL.

The inverse mapping process for a luma sample SL[ x ][ y ] with x = 0..pic\_width\_in\_luma\_samples − 1, y = 0..pic\_height\_in\_luma\_samples − 1 is invoked as specified in clause 8.8.2.2 with the variable lumaSample set equal to SL[ x ][ y ] as the input and the output is assigned to the luma sample SL[ x ][ y ].

#### Inverse mapping process for a luma sample

Input to this process is a luma sample lumaSample.

Output of this process is a modified luma sample invLumaSample .

The value of invLumaSample is derived as follows:

* If slice\_lmcs\_enabled\_flag of the slice that contains the luma sample lumaSample is equal to 1, the following ordered steps apply:

1. The variable idxYInv is derived by invoking the identification of piece-wise function index process for a luma sample as specified in clause 8.8.2.3 with lumaSample as the input and idxYInv as the output.
2. The variable invSample is derived as follows:

invSample = InputPivot[ idxYInv ] + ( InvScaleCoeff[ idxYInv ] \*  (8‑1000)  
 ( lumaSample − LmcsPivot[ idxYInv ] ) + ( 1 << 10 ) ) >> 11

1. The inverse mapped luma sample invLumaSample is derived as follows:

invLumaSample = Clip1Y( invSample ) (8‑1001)

* Otherwise, invLumaSample is set equal to lumaSample.

#### Identification of piecewise function index process for a luma sample

Input to this process is a luma sample lumaSample.

Output of this process is an index idxYInv identifing the piece to which the luma sample lumaSample belongs.

The variable idxYInv is derived as follows:

if ( lumaSample < LmcsPivot[ lmcs\_min\_bin\_idx + 1 ] )  
 idxYInv = lmcs\_min\_bin\_idx  
else if ( lumaSample >= LmcsPivot[ LmcsMaxBinIdx ] )   
 idxYInv = LmcsMaxBinIdx  
else { (8‑1002)  
 for( idxYInv = lmcs\_min\_bin\_idx; idxYInv < LmcsMaxBinIdx; idxYInv++ ) {  
 if( lumaSample < LmcsPivot [ idxYInv + 1 ] )  
 break  
 }  
}

### Deblocking filter process

#### General

Inputs to this process are the reconstructed picture prior to deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0, the arrays 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 the vertical edge filtering process as input. The vertical and horizontal edges in the CTBs of each CTU 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, provided the decoder properly accounts for the processing dependency order so as to produce the same output values.

The deblocking filter process is applied to all coding subblock edges and transform block edges of a picture, except the following types of edges:

* Edges that are at the boundary of the picture,
* Edges that coincide with the virtual boundaries of the picture when pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1,
* Edges that coincide with brick boundaries when loop\_filter\_across\_bricks\_enabled\_flag is equal to 0,
* Edges that coincide with slice boundaries when loop\_filter\_across\_slices\_enabled\_flag is equal to 0,
* Edges that coincide with upper or left boundaries of slices with slice\_deblocking\_filter\_disabled\_flag equal to 1,
* Edges within slices with slice\_deblocking\_filter\_disabled\_flag equal to 1,
* Edges that do not correspond to 8x8 sample grid boundaries of the considered component,
* Edges within the luma component for which both sides of the edge have intra\_bdpcm\_flag equal to 1,
* Edges of chroma subblocks that are not edges of the associated transform unit.

The edge type, vertical or horizontal, is represented by the variable edgeType as specified in Table 8‑21.

Table 8‑21 – Name of association to edgeType

|  |  |
| --- | --- |
| edgeType | Name of edgeType |
| 0 (vertical edge) | EDGE\_VER |
| 1 (horizontal edge) | EDGE\_HOR |

When slice\_deblocking\_filter\_disabled\_flag of the current slice is equal to 0, the following applies:

* The variable treeType is derived as follows:
* If slice\_type is equal to I and qtbtt\_dual\_tree\_intra\_flag is equal to 1, treeType is set equal to DUAL\_TREE\_LUMA.
* Otherwise, treeType is set equal to SINGLE\_TREE.
* The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.3.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr as outputs.
* The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.3.2 with the variable treeType, the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the array recPictureL and, when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE, the arrays recPictureCb and recPictureCr as outputs.
* When slice\_type is equal to I and qtbtt\_dual\_tree\_intra\_flag is equal to 1, the following applies:
* The variable treeType is set equal to DUAL\_TREE\_CHROMA
* The vertical edges are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.3.2 with the variable treeType, the reconstructed picture prior to deblocking, i.e., the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_VER as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr as outputs.
* The horizontal edge are filtered by invoking the deblocking filter process for one direction as specified in clause 8.8.3.2 with the variable treeType, the modified reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr, and the variable edgeType set equal to EDGE\_HOR as inputs, and the modified reconstructed picture after deblocking, i.e., the arrays recPictureCb and recPictureCr as outputs.

#### Deblocking filter process for one direction

Inputs to this process are:

* the variable treeType specifying whether a single tree (SINGLE\_TREE) or a dual tree is used to partition the CTUs and, when a dual tree is used, whether the luma (DUAL\_TREE\_LUMA) or chroma components (DUAL\_TREE\_CHROMA) are currently processed,
* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the reconstructed picture prior to deblocking, i.e., the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Outputs of this process are the modified reconstructed picture after deblocking, i.e:

* when treeType is equal to SINGLE\_TREE or DUAL\_TREE\_LUMA, the array recPictureL,
* when ChromaArrayType is not equal to 0 and treeType is equal to SINGLE\_TREE or DUAL\_TREE\_CHROMA, the arrays recPictureCb and recPictureCr.

The variables firstCompIdx and lastCompIdx are derived as follows:

firstCompIdx = ( treeType = = DUAL\_TREE\_CHROMA ) ? 1 : 0 (8‑1003)

lastCompIdx = ( treeType = = DUAL\_TREE\_LUMA | | ChromaArrayType = = 0 ) ? 0 : 2 (8‑1004)

For each coding unit and each coding block per colour component of a coding unit indicated by the colour component index cIdx ranging from firstCompIdx to lastCompIdx, inclusive, with coding block width nCbW, coding block height nCbH and location of top-left sample of the coding block ( xCb, yCb ), when edgeType is equal to EDGE\_VER and xCb % 8 is equal 0 or when edgeType is equal to EDGE\_HOR and yCb % 8 is equal to 0, the edges are filtered by the following ordered steps:

1. The variable filterEdgeFlag is derived as follows:

* If edgeType is equal to EDGE\_VER and one or more of the following conditions are true, filterEdgeFlag is set equal to 0:
* The left boundary of the current coding block is the left boundary of the picture.
* The left boundary of the current coding block is the left boundary of the brick and loop\_filter\_across\_bricks\_enabled\_flag is equal to 0.
* The left boundary of the current coding block is the left boundary of the slice and loop\_filter\_across\_slices\_enabled\_flag is equal to 0.
* The left boundary of the current coding block is one of the vertical virtual boundaries of the picture and pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1.
* Otherwise if edgeType is equal to EDGE\_HOR and one or more of the following conditions are true, the variable filterEdgeFlag is set equal to 0:
* The top boundary of the current luma coding block is the top boundary of the picture.
* The top boundary of the current coding block is the top boundary of the brick and loop\_filter\_across\_bricks\_enabled\_flag is equal to 0.
* The top boundary of the current coding block is the top boundary of the slice and loop\_filter\_across\_slices\_enabled\_flag is equal to 0.
* The top boundary of the current coding block is one of the horizontal virtual boundaries of the picture and pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1.
* Otherwise, filterEdgeFlag is set equal to 1.

1. All elements of the two-dimensional (nCbW)x(nCbH) array edgeFlags, maxFilterLengthQs and maxFilterlengthPs are initialized to be equal to zero.
2. The derivation process of transform block boundary specified in clause 8.8.3.3 is invoked with the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, the variable cIdx, the variable filterEdgeFlag, the array edgeFlags, the maximum filter length arrays maxFilterLengthPs and maxFilterLengthQs, and the variable edgeType as inputs, and the modified array edgeFlags, the modified maximum filter length arrays maxFilterLengthPs and maxFilterLengthQs as outputs.
3. When cIdx is equal to 0, the derivation process of coding subblock boundary specified in clause 8.8.3.4 is invoked with the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, the array edgeFlags, the maximum filter length arrays maxFilterLengthPs and maxFilterLengthQs, and the variable edgeType as inputs, and the modified array edgeFlags, the modified maximum filter length arrays maxFilterLengthPs and maxFilterLengthQs as outputs.
4. The picture sample array recPicture is derived as follows:

* If cIdx is equal to 0, recPicture is set equal to the reconstructed luma picture sample array prior to deblocking recPictureL.
* Otherwise, if cIdx is equal to 1, recPicture is set equal to the reconstructed chroma picture sample array prior to deblocking recPictureCb.
* Otherwise (cIdx is equal to 2), recPicture is set equal to the reconstructed chroma picture sample array prior to deblocking recPictureCr.

1. The derivation process of the boundary filtering strength specified in clause 8.8.3.5 is invoked with the picture sample array recPicture, the luma location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, the variable edgeType, the variable cIdx, and the array edgeFlags as inputs, and an (nCbW)x(nCbH) array bS as output.
2. The edge filtering process for one direction is invoked for a coding block as specified in clause 8.8.3.6 with the variable edgeType, the variable cIdx, the reconstructed picture prior to deblocking recPicture, the location ( xCb, yCb ), the coding block width nCbW, the coding block height nCbH, and the arrays bS, maxFilterLengthPs, and maxFilterLengthQs, as inputs, and the modified reconstructed picture recPicture as output.

#### Derivation process of transform block boundary

Inputs to this process are:

* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* a variable cIdx specifying the colour component of the current coding block,
* a variable filterEdgeFlag,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags,
* two-dimensional (nCbW)x(nCbH) arrays maxFilterLengthQs and maxFilterLengthPs,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Outputs of this process are:

* the modified two-dimensional (nCbW)x(nCbH) array edgeFlags,
* the modified two-dimensional (nCbW)x(nCbH) arrays maxFilterLengthQs, maxFilterLengthPs.

Depending on edgeType, the arrays edgeFlags, maxFilterLengthPs and maxFilterLengthQs are derived as follows:

* If edgeType is equal to EDGE\_VER, the following applies:
* The variable numEdges is set equal to Max( 1, nCbW / 8 ).
* For xEdge = 0..numEdges − 1 and y = 0..nCbH − 1, the following applies:
* The horizontal position x inside the current coding block is set equal to xEdge \*8.
* The value of edgeFlags[ x ][ y ] is derived as follows:
  + - If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag equal to 1 and ( xCb + x ) is equal to PpsVirtualBoundariesPosX[ n ] for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, edgeFlags[ x ][ y ] is set equal to 0.
    - Otherwise, if x is equal to 0, edgeFlags[ x ][ y ] is set equal to filterEdgeFlag.
    - Otherwise, if the location ( xCb + x , yCb + y ) is at a transform block edge, edgeFlags[ x ][ y ] is set equal to 1.
* When edgeFlags[ x ][ y ] is equal to 1,the following applies:
  + - If cIdx is equal to 0, the following applies:
    - The value of maxFilterLengthQs[ x ][ y ] is derived as follows:
    - If the width in luma samples of the transform block at luma location ( xCb + x, yCb + y ) is equal to or greater than 32, maxFilterLengthQs[ x ][ y ] is set equal to 7.
    - Otherwise, maxFilterLengthQs[ x ][ y ] is set equal to 3.
    - The value of maxFilterLengthPs[ x ][ y ] is derived as follows:
    - If the width in luma samples of the transform block at luma location ( xCb + x − 1, yCb + y ) is equal to or greater than 32, maxFilterLengthPs[ x ][ y ] is set equal to 7.
    - Otherwise, maxFilterLengthPs[ x ][ y ] is set equal to 3.
    - Otherwise (cIdx is not equal to 0), the values of maxFilterLengthPs[ x][ y ] and maxFilterLengthQs[ x ][ y ] are derived as follows:
    - If the width in chroma samples of the transform block at chroma location ( xCb + x, yCb + y ) and the width at chroma location ( xCb + x − 1,  yCb + y ) are both equal to or greater than 8, maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are set equal to 3.
    - Otherwise, maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are set equal to 1.
* Otherwise (edgeType is equal to EDGE\_HOR), the following applies:
* The variable numEdges is set equal to Max( 1, nCbH / 8 ).
* For yEdge = 0..numEdges − 1 and x = 0..nCbW − 1, the following applies:
* The vertical position y inside the current coding block is set equal to yEdge \*8.
* The value of edgeFlags[ x ][ y ] is derived as follows:
  + - If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag equal to 1 and ( yCb + y ) is equal to PpsVirtualBoundariesPosY[ n ] for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, edgeFlags[ x ][ y ] is set equal to 0.
    - Otherwise, if y is equal to 0, edgeFlags[ x ][ y ] is set equal to filterEdgeFlag.
    - Otherwise, if the location ( xCb + x , yCb + y ) is at a transform block edge, edgeFlags[ x ][ y ] is set equal to 1.
* When edgeFlags[ x ][ y ] is equal to 1,the following applies:
  + - If cIdx is equal to 0, the following applies:
    - The value of maxFilterLengthQs[ x ][ y ] is derived as follows:
    - If the height in luma samples of the transform block at luma location ( xCb + x, yCb + y ) is equal to or greater than 32, maxFilterLengthQs[ x ][ y ] is set equal to 7.
    - Otherwise, maxFilterLengthQs[ x ][ y ] is set equal to 3.
    - The value of maxFilterLengthPs[ x ][ y ] is derived as follows:
    - If the height in luma samples of the transform block at luma location ( xCb + x, yCb + y − 1 ) is equal to or greater than 32, maxFilterLengthPs[ x ][ y ] is set equal to 7.
    - Otherwise, maxFilterLengthPs[ x ][ y ] is set equal to 3.
    - Otherwise (cIdx is not equal to 0), the values of maxFilterLengthPs[ x][ y ] and maxFilterLengthQs[ x ][ y ] are derived as follows:
    - If all of the following conditions are true, maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are set equal to 3:
    - The height in chroma samples of the transform block at chroma location ( xCb + x, yCb + y ) and the height at chroma location ( xCb + x,  yCb + y − 1 ) are both equal to or greater than 8.
    - ( yCb + y ) % CtbHeightC is greater than 0, i.e. the horizontal edge do not overlap with the upper chroma CTB boundary.
    - Otherwise, maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are set equal to 1.

#### Derivation process of coding subblock boundary

Inputs to this process are:

* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags,
* two-dimensional (nCbW)x(nCbH) arrays maxFilterLengthQs and maxFilterLengthPs,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered.

Outputs of this process are:

* the modified two-dimensional (nCbW)x(nCbH) array edgeFlags,
* the modified two-dimensional (nCbW)x(nCbH) arrays maxFilterLengthQs and maxFilterLengthPs.

The number of coding subblock in horizontal direction numSbX and in vertical direction numSbY are derived as follows:

* If inter\_affine\_flag[ xCb ][ yCb ] is equal to 1 or merge\_subblock\_flag[ xCb ][ yCb ] is equal to 1, numSbX and numSbY are set equal to NumSbX[ xCb ][ yCb ] and NumSbY[ xCb ][ yCb ], respectively.
* Otherwise, numSbX and numSbY are both set equal to 1.

Depending on the value of edgeType the following applies:

* If edgeType is equal to EDGE\_VER, the following applies:
* The variable sbW is set equal to Max( 8, nCbW / numSbX ).
* The array edgeTbFlags is set equal to edgeFlags.
* For xEdge = 0..min( ( nCbW / 8 ) − 1, numSbX − 1), y = 0..nCbH − 1:
* The horizontal position x inside the current coding block is set equal to xEdge \*sbW.
* The value of edgeFlags[ x ][ y ] is derived as follows:
* If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and x is equal to PpsVirtualBoundariesPosX[ n ] for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

edgeFlags[ x ][ y ] = 0 (8‑1005)

* Otherwise, the following applies:

edgeFlags[ x ][ y ] = 1 (8‑1006)

* When edgeFlags[ x ][ y ] is equal to 1, the values of maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are modified as follows:
* If x is equal to 0, the following applies:
* When numSbX is greater than 1, the following applies:

maxFilterLengthQs[ x ][ y ] = Min( 5, maxFilterLengthQs[ x ][ y ] ) (8‑1007)

* When inter\_affine\_flag[ xCb − 1 ][ yCb ] is equal to 1 or merge\_subblock\_flag[ xCb − 1 ][ yCb ] is equal to 1, the following applies:

maxFilterLengthPs[ x ][ y ] = Min( 5, maxFilterLengthPs[ x ][ y ] ) (8‑1008)

* Otherwise, if edgeTbFlags[ x ][ y ] is equal to 1, the following applies:

maxFilterLengthPs[ x ][ y ] = Min( 5, maxFilterLengthPs[ x ][ y ] ) (8‑1009)

maxFilterLengthQs[ x ][ y ] = Min( 5, maxFilterLengthQs[ x ][ y ] ) (8‑1010)

* Otherwise, if one or more of the following conditions are true:
* xEdge is equal to 1,
* xEdge is equal to ( nCbW / 8 ) − 1,
* edgeTbFlags[ x − sbW ][ y ] is equal to 1,
* edgeTbFlags[ x + sbW ][ y ] is equal to 1,

the following applies:

maxFilterLengthPs[ x ][ y ] = 2 (8‑1011)

maxFilterLengthQs[ x ][ y ] = 2 (8‑1012)

* Otherwise, the following applies:

maxFilterLengthPs[ x ][ y ] = 3 (8‑1013)

maxFilterLengthQs[ x ][ y ] = 3 (8‑1014)

* Otherwise if edgeType is equal to EDGE\_HOR, the following applies:
* The variable sbH is set equal to Max( 8, nCbH / numSbY ).
* The array edgeTbFlags is set equal to edgeFlags.
* For yEdge = 0..min( ( nCbH / 8 ) − 1, numSbY − 1 ), x = 0..nCbW − 1:
* The vertical position y inside the current coding block is set equal to yEdge \*sbH.
* The value of edgeFlags[ x ][ y ] is derived as follows:
* If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and y is equal to PpsVirtualBoundariesPosY[ n ] for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

edgeFlags[ x ][ y ] = 0 (8‑1015)

* Otherwise, the following applies:

edgeFlags[ x ][ y ] = 1 (8‑1016)

* When edgeFlags[ x ][ y ] is equal to 1, the values of maxFilterLengthPs[ x ][ y ] and maxFilterLengthQs[ x ][ y ] are modified as follows:
* If y is equal to 0 and edgeFlags[ x ][ y ] is equal to 1, the following applies:
* When numSbY is greater than 1, the following applies:

maxFilterLengthQs[ x ][ y ] = Min( 5, maxFilterLengthQs[ x ][ y ] ) (8‑1017)

* When inter\_affine\_flag[ xCb ][ yCb − 1 ] is equal to 1 or merge\_subblock\_flag[ xCb ][ yCb − 1 ] is equal to 1, the following applies:

maxFilterLengthPs[ x ][ y ] = Min( 5, maxFilterLengthPs[ x ][ y ] ) (8‑1018)

* Otherwise, if edgeTbFlags[ x ][ y ] is equal to 1, the following applies:

maxFilterLengthPs[ x ][ y ] = Min( 5, maxFilterLengthPs[ x ][ y ] ) (8‑1019)

maxFilterLengthQs[ x ][ y ] = Min( 5, maxFilterLengthQs[ x ][ y ] ) (8‑1020)

* Otherwise, if one or more of the following conditions are true:
* yEdge is equal to 1,
* yEdge is equal to ( nCbH / 8 ) − 1,
* edgeTbFlags[ x ][ y − sbH ] is equal to 1,
* edgeTbFlags[ x ][ y + sbH ] is equal to 1,

the following applies:

maxFilterLengthPs[ x ][ y ] = 2 (8‑1021)

maxFilterLengthQs[ x ][ y ] = 2 (8‑1022)

* Otherwise, the following applies:

maxFilterLengthPs[ x ][ y ] = 3 (8‑1023)

maxFilterLengthQs[ x ][ y ] = 3 (8‑1024)

#### Derivation process of boundary filtering strength

Inputs to this process are:

* a picture sample array recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* a variable cIdx specifying the colour component of the current coding block,
* a two-dimensional (nCbW)x(nCbH) array edgeFlags.

Output of this process is a two-dimensional (nCbW)x(nCbH) array bS specifying the boundary filtering strength.

The variables xDi, yDj, xN and yN are derived as follows:

* If edgeType is equal to EDGE\_VER,

xDi = ( i  <<  3 ) (8‑1025)

yDj = cIdx = = 0 ? ( j  <<  2 ) : ( j  <<  1 ) (8‑1026)

xN is set equal to Max( 0, ( nCbW / 8 ) − 1 ) (8‑1027)

yN = cIdx = = 0 ? ( nCbH / 4 ) − 1 : ( nCbH / 2 ) − 1 (8‑1028)

* Otherwise (edgeType is equal to EDGE\_HOR),

xDi = cIdx = = 0 ? ( i  <<  2 ) : ( i  <<  1 ) (8‑1029)

yDj = ( j  <<  3 ) (8‑1030)

xN = cIdx = = 0 ? ( nCbW / 4 ) − 1 : ( nCbW / 2 ) − 1 (8‑1031)

yN = Max( 0, ( nCbH / 8 ) − 1 ) (8‑1032)

For xDi with i = 0..xN and yDj with j = 0..yN, the following applies:

* If edgeFlags[ xDi ][ yDj ] is equal to 0, the variable bS[ xDi ][ yDj ] is set equal to 0.
* Otherwise, the following applies:
* The sample values p0 and q0 are derived as follows:
  + - If edgeType is equal to EDGE\_VER, p0 is set equal to recPicture[ xCb + xDi − 1 ][ yCb + yDj ] and q0 is set equal to recPicture[ xCb + xDi ][ yCb + yDj ].
    - Otherwise (edgeType is equal to EDGE\_HOR), p0 is set equal to recPicture[ xCb + xDi ][ yCb + yDj − 1 ] and q0 is set equal to recPicture[ xCb + xDi ][ yCb + yDj ].
* The variable bS[ xDi ][ yDj ] is derived as follows:
  + - If cIdx is equal to 0 and both samples p0 and q0 are in a coding block with intra\_bdpcm\_flag equal to 1, bS[ xDi ][ yDj ] is set equal to 0.
    - Otherwise, if the sample p0 or q0 is in the coding block of a coding unit coded with intra prediction mode, 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 coding block with ciip\_flag equal to 1, 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 transform block which contains one or more non-zero transform coefficient levels, bS[ xDi ][ yDj ] is set equal to 1.
    - Otherwise, if the prediction mode of the coding subblock containing the sample p0 is different from the prediction mode of the coding subblock containing the sample q0, bS[ xDi ][ yDj ] is set equal to 1.
    - Otherwise, if cIdx is equal to 0 and one or more of the following conditions are true, bS[ xDi ][ yDj ] is set equal to 1:
      * The coding subblock containing the sample p0 and the coding subblock containing the sample q0 are both coded in IBC prediction mode, and the absolute difference between the horizontal or vertical component of the motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.
      * For the prediction of the coding subblock containing the sample p0 different reference pictures or a different number of motion vectors are used than for the prediction of the coding subblock containing the sample q0.

NOTE 1 – The determination of whether the reference pictures used for the two coding sublocks 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 coding subblock with top-left sample covering ( xSb, ySb ), is equal to PredFlagL0[ xSb ][ ySb ] + PredFlagL1[ xSb ][ ySb ].

* + - * One motion vector is used to predict the coding subblock containing the sample p0 and one motion vector is used to predict the coding subblock 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 samples.
      * Two motion vectors and two different reference pictures are used to predict the coding subblock containing the sample p0, two motion vectors for the same two reference pictures are used to predict the coding subblock 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 coding subblocks for the same reference picture is greater than or equal to 4 in units of quarter luma samples.
      * Two motion vectors for the same reference picture are used to predict the coding subblock containing the sample p0, two motion vectors for the same reference picture are used to predict the coding subblock containing the sample q0 and both 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 coding subblocks is greater than or equal to 4 in quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vectors used in the prediction of the two coding subblocks is greater than or equal to 4 in units of quarter luma samples.
        + The absolute difference between the horizontal or vertical component of list 0 motion vector used in the prediction of the coding subblock containing the sample p0 and the list 1 motion vector used in the prediction of the coding subblock containing the sample q0 is greater than or equal to 4 in units of quarter luma samples, or the absolute difference between the horizontal or vertical component of the list 1 motion vector used in the prediction of the coding subblock containing the sample p0 and list 0 motion vector used in the prediction of the coding subblock containing the sample q0 is greater than or equal to 4 in units of quarter luma samples.
    - Otherwise, the variable bS[ xDi ][ yDj ] is set equal to 0.

#### Edge filtering process for one direction

Inputs to this process are:

* a variable edgeType specifying whether vertical edges (EDGE\_VER) or horizontal edges (EDGE\_HOR) are currently processed,
* a variable cIdx specifying the current colour component,
* the reconstructed picture prior to deblocking recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a variable nCbW specifying the width of the current coding block,
* a variable nCbH specifying the height of the current coding block,
* the array bS specifying the boundary strength,
* the arrays maxFilterLengthPs and maxFilterLengthQs.

Output of this process is the modified reconstructed picture after deblocking recPicture.

For the edge filtering process, the following applies:

* The variables subW, subH, xN, yN, xDk and yDm are derived as follows:

subW = cIdx = = 0 ? 1 : SubWidthC (8‑1033)

subH = cIdx = = 0 ? 1 : SubHeightC (8‑1034)

xN = edgeType = = EDGE\_VER ? Max( 0, ( nCbW / 8 ) − 1 ) : ( nCbW / 4 / subW ) − 1 (8‑1035)

yN = edgeType = = EDGE\_VER ? ( nCbH / 4 / subH ) − 1 : Max( 0, ( nCbH / 8 ) − 1 ) (8‑1036)

xDk = edgeType = = EDGE\_VER ? ( k  <<  3 ) : ( k  <<  ( 2 / subW ) ) (8‑1037)

yDm = edgeType = = EDGE\_VER ? ( m  <<  ( 2 / subH ) ) : ( m  <<  3 ) (8‑1038)

* For xDk with k = 0..xN and yDm with m = 0..yN, the following applies:
* When bS[ xDk ][ yDm ] is greater than 0, the following ordered steps apply:
* If cIdx is equal to 0, the filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The decision process for luma block edges as specified in clause 8.8.3.6.1 is invoked with the luma picture sample array recPicture, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xBl, yBl ) set equal to ( xDk, yDm ), the edge direction edgeType, the boundary filtering strength bS[ xDk ][ yDm ], the maximum filter lengths maxFilterLengthP set equal to maxFilterLengthPs[ xDk ][ yDm ] and maxFilterLengthQ set equal to maxFilterLengthQs[ xDk ][ yDm ] as inputs, and the decisions dE, dEp and dEq, the modified maximum filter lengths maxFilterLengthP and maxFilterLengthQ, and the variable tC as outputs.
2. The filtering process for block edges as specified in clause 8.8.3.6.2 is invoked with the luma picture sample array recPicture, the location of the luma coding block ( xCb, yCb ), the luma location of the block ( xBl, yBl ) set equal to ( xDk, yDm ), the edge direction edgeType, the decisions dE, dEp and dEq, the maximum filter lengths maxFilterLengthP and maxFilterLengthQ, and the variable tC as inputs, and the modified luma picture sample array recPicture as output.

* Otherwise (cIdx is not equal to 0), the filtering process for edges in the chroma coding block of current coding unit specified by cIdx consists of the following ordered steps:

1. The variable cQpPicOffset is derived as follows:

cQpPicOffset = cIdx = = 1 ? pps\_cb\_qp\_offset : pps\_cr\_qp\_offset (8‑1039)

1. The decision process for chroma block edges as specified in clause 8.8.3.6.3 is invoked the chroma picture sample array recPicture, the location of the chroma coding block ( xCb, yCb ), the location of the chroma block ( xBl, yBl ) set equal to ( xDk, yDm ), the edge direction edgeType, the variable cQpPicOffset, the boundary filtering strength bS[ xDk ][ yDm ], and the variable maxFilterLengthCbCr set equal to maxFilterLengthPs[ xDk ][ yDm ] as inputs, the modified variable maxFilterLengthCbCr, and the variable tC as outputs.
2. When maxFilterLengthCbCr is greater than 0, the filtering process for chroma block edges as specified in clause 8.8.3.6.4 is invoked with the chroma picture sample array recPicture, the location of the chroma coding block ( xCb, yCb ), the chroma location of the block ( xBl, yBl ) set equal to ( xDk, yDm ), the edge direction edgeType, the variable maxFilterLengthCbCr, and the variable tC as inputs, and the modified chroma picture sample array recPicture as output.

##### Decision process for luma block edges

Inputs to this process are:

* a picture sample array recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a location ( xBl, yBl ) specifying the top-left sample of the current block relative to the top-left sample of the current 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,
* a variable maxFilterLengthP specifying the max filter length,
* a variable maxFilterLengthQ specifying the max filter length.

Outputs of this process are:

* the variables dE, dEp and dEq containing decisions,
* the modified filter length variables maxFilterLengthP and maxFilterLengthQ,
* the variable tC.

The sample values pi,k and qj,k with i = 0..maxFilterLengthP, j = 0..maxFilterLengthQ and k = 0 and 3 are derived as follows:

* If edgeType is equal to EDGE\_VER, the following applies:

qj,k = recPictureL[ xCb + xBl + j ][ yCb + yBl + k ] (8‑1040)

pi,k = recPictureL[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1041)

* Otherwise (edgeType is equal to EDGE\_HOR), the following applies:

qj,k = recPicture[ xCb + xBl + k ][ yCb + yBl + j ] (8‑1042)

pi,k = recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1043)

The variable qpOffset is derived as follows:

* If sps\_ladf\_enabled\_flag is equal to 1, the following applies:
* The variable lumaLevel of the reconstructed luma level is derived as follow:

lumaLevel = ( ( p0,0 + p0,3 + q0,0 + q0,3 ) >> 2 ), (8‑1044)

* The variable qpOffset is set equal to sps\_ladf\_lowest\_interval\_qp\_offset and modified as follows:

for( i = 0; i < sps\_num\_ladf\_intervals\_minus2 + 1; i++ ) {  
 if( lumaLevel > SpsLadfIntervalLowerBound[ i + 1 ] )  
 qpOffset = sps\_ladf\_qp\_offset[ i ] (8‑1045)  
 else  
 break  
}

* Otherwise, qpOffset is set equal to 0.

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 qP is derived as follows:

qP = ( ( QpQ + QpP + 1 )  >>  1 ) + qpOffset (8‑1046)

The value of the variable β′ is determined as specified in Table 8‑22 based on the quantization parameter Q derived as follows:

Q = Clip3( 0, 63, qP + ( slice\_beta\_offset\_div2  <<  1 ) ) (8‑1047)

where slice\_beta\_offset\_div2 is the value of the syntax element slice\_beta\_offset\_div2 for the slice that contains sample q0,0.

The variable β is derived as follows:

β = β′ \* ( 1  <<  ( BitDepthY − 8 ) ) (8‑1048)

The value of the variable tC′ is determined as specified in Table 8‑22 based on the quantization parameter Q derived as follows:

Q = Clip3( 0, 65, qP + 2 \* ( bS − 1 ) + ( slice\_tc\_offset\_div2  <<  1 ) ) (8‑1049)

where slice\_tc\_offset\_div2 is the value of the syntax element slice\_tc\_offset\_div2 for the slice that contains sample q0,0.

The variable tC is derived as follows:

tC = tC′ \* ( 1  <<  ( BitDepthY − 8 ) ) (8‑1050)

The following ordered steps apply:

1. The variables dp0, dp3, dq0 and dq3 are derived as follows:

dp0 = Abs( p2,0 − 2 \* p1,0 + p0,0 ) (8‑1051)

dp3 = Abs( p2,3 − 2 \* p1,3 + p0,3 ) (8‑1052)

dq0 = Abs( q2,0 − 2 \* q1,0 + q0,0 ) (8‑1053)

dq3 = Abs( q2,3 − 2 \* q1,3 + q0,3 ) (8‑1054)

1. When maxFilterLengthP and maxFilterLengthQ both are equal to or greater than 3 the variables sp0, sq0, spq0, sp3, sq3 and spq3 are derived as follows:

sp0 = Abs( p3,0 − p0,0 ) (8‑1055)

sq0 = Abs( q0,0 − q3,0 ) (8‑1056)

spq0 = Abs( p0,0 − q0,0 ) (8‑1057)

sp3 = Abs( p3,3 − p0,3 ) (8‑1058)

sq3 = Abs( q0,3 − q3,3 ) (8‑1059)

spq3 = Abs( p0,3 − q0,3 ) (8‑1060)

1. The variables sidePisLargeBlk and sideQisLargeBlk are set equal to 0.
2. When maxFilterLengthP is larger than 3, sidePisLargeBlk is set equal to 1:
3. When maxFilterLengthQ is larger than 3, sideQisLargeBlk is set equal to 1:
4. When edgeType is equal to EDGE\_HOR and (yCb + yBl ) % CtbSizeY is equal to 0, sidePisLargeBlk is set equal to 0.
5. The variables dSam0 and dSam3 are initialized to 0.
6. When sidePisLargeBlk or sideQisLargeBlk is greater than 0, the following applies:
7. The variables dp0L, dp3L are derived and maxFilterLengthP is modified as follows:

* If sidePisLargeBlk is equal to 1, the following applies:

dp0L = ( dp0 + Abs( p5,0 − 2 \* p4,0 + p3,0 ) + 1 ) >> 1 (8‑1061)

dp3L = ( dp3 + Abs( p5,3 − 2 \* p4,3 + p3,3 ) + 1 ) >> 1 (8‑1062)

* Otherwise, the following applies:

dp0L = dp0 (8‑1063)

dp3L = dp3 (8‑1064)

maxFilterLengthP = 3 (8‑1065)

1. The variables dq0L and dq3L are derived as follows:

* If sideQisLargeBlk is equal to 1, the following applies:

dq0L = ( dq0 + Abs( q5,0 − 2 \* q4,0 + q3,0 ) + 1 ) >> 1 (8‑1066)

dq3L = ( dq3 + Abs( q5,3 − 2 \* q4,3 + q3,3 ) + 1 ) >> 1 (8‑1067)

* Otherwise, the following applies:

dq0L = dq0 (8‑1068)

dq3L = dq3 (8‑1069)

1. The variables dpq0L, dpq3L, and dL are derived as follows:

dpq0L = dp0L + dq0L (8‑1070)

dpq3L = dp3L + dq3L (8‑1071)

dL = dpq0L + dpq3L (8‑1072)

1. When dL is less than β, the following ordered steps apply:
   * 1. The variable dpq is set equal to 2 \* dpq0L.
     2. The variable sp is set equal to sp0, the variable sq is set equal to sq0 and the variable spq is set equal to spq0.
     3. The variables p0 p3 qo and q3 are first initialized to 0 and then modified according to sidePisLargeBlk and sideQisLargeBlk as follows:

* When sidePisLargeBlk is equal to 1, the following applies:

p3 = p3,0 (8‑1073)

p0 = pmaxFilterLengthP,0 (8‑1074)

* When sideQisLargeBlk is equal to 1, the following applies:

q3 = q3,0 (8‑1075)

q0 = qmaxFilterLengthQ,0 (8‑1076)

* + 1. For the sample location ( xCb + xBl, yCb + yBl ), the decision process for a luma sample as specified in clause 8.8.3.6.5 is invoked with the sample values p0, p3, q0, q3, the variables dpq, sp, sq, spq, sidePisLargeBlk, sideQisLargeBlk, β and tC as inputs, and the output is assigned to the decision dSam0.
    2. The variable dpq is set equal to 2 \* dpq3L.
    3. The variable sp is set equal to sp3, the variable sq is set equal to sq3 and the variable spq is set equal to spq3.
    4. The variables p0 p3 q0 and q3 are first initialized to 0 and are then modified according to sidePisLargeBlk and sideQisLargeBlk as follows:
* When sidePisLargeBlk is equal to 1, the following applies:

p3 = p3,3 (8‑1077)

p0 = pmaxFilterLengthP,3 (8‑1078)

* When sideQisLargeBlk is equal to 1, the following applies:

q3 = q3,3 (8‑1079)

q0 = qmaxFilterLengthQ,3 (8‑1080)

* + 1. When edgeType is equal to EDGE\_VER for the sample location ( xCb + xBl, yCb + yBl + 3 ) or when edgeType is equal to EDGE\_HOR for the sample location ( xCb + xBl + 3, yCb + yBl ), the decision process for a luma sample as specified in clause 8.8.3.6.5 is invoked with the sample values p0, p3, q0, q3, the variables dpq, sp, sq, spq, sidePisLargeBlk, sideQisLargeBlk, β and tC as inputs, and the output is assigned to the decision dSam3.

1. The variables dE, dEp and dEq are derived as follows:

* If dSam0 and dSam3 are both equal to 1, the variable dE is set equal to 3, dEp is set equal to 1, and dEq is set equal to 1.
* Otherwise, the following ordered steps apply:

1. The variables dpq0, dpq3, dp, dq and d are derived as follows:

dpq0 = dp0 + dq0 (8‑1081)

dpq3 = dp3 + dq3 (8‑1082)

dp = dp0 + dp3 (8‑1083)

dq = dq0 + dq3 (8‑1084)

d = dpq0 + dpq3 (8‑1085)

1. The variables dE, dEp, dEq, sidePisLargeBlk and sideQisLargeBlk are set equal to 0.
2. When d is less than β and both maxFilterLengthP and maxFilterLengthQ are greater than 2, the following ordered steps apply:
3. The variable dpq is set equal to 2 \* dpq0.
4. The variable sp is set equal to sp0, the variable sq is set equal to sq0 and the variable spq is set equal to spq0.
5. For the sample location ( xCb + xBl, yCb + yBl ), the decision process for a luma sample as specified in clause 8.8.3.6.5 is invoked with the variables p0, p3, q0, q3 all set equal to 0, the variables dpq, sp, sq, spq, sidePisLargeBlk, sideQisLargeBlk, β and tC as inputs, and the output is assigned to the decision dSam0.
6. The variable dpq is set equal to 2 \* dpq3.
7. The variable sp is set equal to sp3, the variable sq is set equal to sq3 and the variable spq is set equal to spq3.
8. When edgeType is equal to EDGE\_VER for the sample location ( xCb + xBl, yCb + yBl + 3 ) or when edgeType is equal to EDGE\_HOR for the sample location ( xCb + xBl + 3, yCb + yBl ), the decision process for a sample as specified in clause 8.8.3.6.5 is invoked with the variables p0, p3, q0, q3 all set equal to 0, the variables dpq, sp, sq, spq, sidePisLargeBlk, sideQisLargeBlk, β and tC as inputs, and the output is assigned to the decision dSam3.
9. When d is less than β, the following ordered steps apply:
10. The variable dE is set equal to 1.
11. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
12. When dp is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEp is set equal to 1.
13. When dq is less than ( β + ( β  >>  1 ) )  >>  3, the variable dEq is set equal to 1.

Table 8‑22 – 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 |
| **β**′ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| **tC**′ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Q** | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |
| **β**′ | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 | 22 | 24 | 26 | 28 |
| **tC**′ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| **Q** | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| **β**′ | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 |
| **tC**′ | 3 | 4 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 8 | 9 | 10 | 11 | 13 | 14 | 16 | 18 |
| **Q** | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 |  |  |
| **β**′ | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 | 80 | 82 | 84 | 86 | 88 | - | - |  |  |
| **tC**′ | 20 | 22 | 25 | 28 | 31 | 35 | 39 | 44 | 50 | 56 | 63 | 70 | 79 | 88 | 99 |  |  |

##### Filtering process for luma block edges

Inputs to this process are:

* a picture sample array recPicture,
* a location ( xCb, yCb ) specifying the top-left sample of the current coding block relative to the top-left sample of the current picture,
* a location ( xBl, yBl ) specifying the top-left sample of the current block relative to the top-left sample of the current coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* the variables dE, dEp and dEq containing decisions,
* the variables maxFilterLengthP and maxFilterLengthQ containing max filter lengths,
* the variable tC.

Output of this process is the modified picture sample array recPicture.

Depending on the value of 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..maxFilterLengthP, j = 0..maxFilterLengthQ and k = 0..3 are derived as follows:

qj,k = recPictureL[ xCb + xBl + j ][ yCb + yBl + k ] (8‑1086)

pi,k = recPictureL[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1087)

1. When dE is not equal to 0 and dE is not equal to 3, for each sample location ( xCb + xBl, yCb + yBl + k ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample using short filters as specified in clause 8.8.3.6.6 is invoked with the sample values pi,k, qi,k with i = 0..3, the locations ( xPi, yPi ) set equal to ( xCb + xBl − i − 1, yCb + yBl + k ) and ( xQi, yQi ) set equal to ( xCb + xBl + i, yCb + yBl + k ) with i = 0..2, the decision dE, the variables dEp and dEq and 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 recPicture as follows:

recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] = pi' (8‑1088)

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 recPicture as follows:

recPicture[ xCb + xBl + j ][ yCb + yBl + k ] = qj' (8‑1089)

1. When dE is equal to 3, for each sample location ( xCb + xBl, yCb + yBl + k ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample using long filters as specified in clause 8.8.3.6.7 is invoked with the sample values pi,k, qj,k with i = 0..maxFilterLengthP and j = 0..maxFilterLengthQ, the locations ( xPi, yPi ) set equal to ( xCb + xBl − i − 1, yCb + yBl + k ) with i = 0..maxFilterLengthP − 1 and ( xQj, yQj ) set equal to ( xCb + xBl + j, yCb + yBl + k ) with j = 0..maxFilterLengthQ − 1, the variables maxFilterLengthP, maxFilterLengthQ and tC as inputs and the filtered samples values pi' and qj' as outputs.
3. The filtered sample values pi' with i = 0..maxFilterLengthP − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] = pi'  **(**8‑1090**)**

1. The filtered sample values qj' with j = 0..maxFilterLengthQ − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + j ][ yCb + yBl + k ] = qj' **(**8‑1091**)**

* Otherwise (edgeType is equal to EDGE\_HOR), the following ordered steps apply:

1. The sample values pi,k and qi,k with i = 0..maxFilterLengthP, j = 0..maxFilterLengthQ and k = 0..3 are derived as follows:

qj,k = recPictureL[ xCb + xBl + k ][ yCb + yBl + j ] (8‑1092)

pi,k = recPictureL[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1093)

1. When dE is not equal to 0 and dE is not equal to 3, for each sample location ( xCb + xBl + k, yCb + yBl ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample using short filters as specified in clause 8.8.3.6.6 is invoked with the sample values pi,k, qi,k with i = 0..3, the locations ( xPi, yPi ) set equal to ( xCb + xBl + k, yCb + yBl − i − 1 ) and ( xQi, yQi ) set equal to ( xCb + xBl + k, yCb + yBl + i ) with i = 0..2, the decision dE, the variables dEp and dEq, and 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 recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] = pi' (8‑1094)

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 recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl + j ] = qj' (8‑1095)

1. When dE is equal to 3, for each sample location ( xCb + xBl + k, yCb + yBl ), k = 0..3, the following ordered steps apply:
2. The filtering process for a luma sample using long filters as specified in clause 8.8.3.6.7 is invoked with the sample values pi,k, qj,k with i = 0..maxFilterLengthP and j = 0..maxFilterLengthQ, the locations ( xPi, yPi ) set equal to ( xCb + xBl + k, yCb + yBl − i − 1 ) with i = 0..maxFilterLengthP − 1 and ( xQj, yQj ) set equal to ( xCb + xBl + k, yCb + yBl + j ) with j = 0..maxFilterLengthQ − 1, the variables maxFilterLengthP, maxFilterLengthQ, and the variable tC as inputs, and the filtered sample values pi' and qj' as outputs.
3. The filtered sample values pi' with i = 0..maxFilterLengthP − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] = pi' (8‑1096)

1. The filtered sample values qj' with j = 0..maxFilterLengthQ − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl + j ] = qj' (8‑1097)

##### Decision process for chroma block edges

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

* a chroma picture sample array recPicture,
* a chroma location ( xCb, yCb ) specifying the top-left sample of the current chroma coding block relative to the top-left chroma sample of the current picture,
* a chroma location ( xBl, yBl ) specifying the top-left sample of the current chroma block relative to the top-left sample of the current chroma coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* a variable cQpPicOffset specifying the picture-level chroma quantization parameter offset,
* a variable bS specifying the boundary filtering strength,
* a variable maxFilterLengthCbCr.

Outputs of this process are

* the modified variable maxFilterLengthCbCr,
* the variable tC.

The values pi and qi with i = 0.. maxFilterLengthCbCr and k = 0..1 are derived as follows:

* If edgeType is equal to EDGE\_VER, the following applies::

qi,k = recPicture[ xCb + xBl + i ][ yCb + yBl + k ] (8‑1098)

pi,k = recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1099)

* Otherwise (edgeType is equal to EDGE\_HOR), the following applies:

qi,k = recPicture[ xCb + xBl + k ][ yCb + yBl + i ] (8‑1100)

pi,k = recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1101)

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 derived as follows:

If ChromaArrayType is equal to 1, the variable QpC is determined as specified in Table 8‑17 based on the index qPi derived as follows:

qPi = ( ( QpQ + QpP + 1 )  >>  1 ) + cQpPicOffset (8‑1102)

* Otherwise (ChromaArrayType is greater than 1), the variable QpC is set equal to Min( qPi, 63 ).

NOTE – The variable cQpPicOffset provides an adjustment for the value of pps\_cb\_qp\_offset or pps\_cr\_qp\_offset, according to whether the filtered chroma component is the Cb or Cr component. However, to avoid the need to vary the amount of the adjustment within the picture, the filtering process does not include an adjustment for the value of slice\_cb\_qp\_offset or slice\_cr\_qp\_offset.

The value of the variable β′ is determined as specified in Table 8‑22 based on the quantization parameter Q derived as follows:

Q = Clip3( 0, 63, QpC + ( slice\_beta\_offset\_div2  <<  1 ) ) (8‑1103)

where slice\_beta\_offset\_div2 is the value of the syntax element slice\_beta\_offset\_div2 for the slice that contains sample q0,0.

The variable β is derived as follows:

β = β′ \* ( 1  <<  ( BitDepthC − 8 ) ) (8‑1104)

The value of the variable tC′ is determined as specified in Table 8‑22 based on the chroma quantization parameter Q derived as follows:

Q = Clip3( 0, 65, QpC + 2 \* ( bS − 1 ) + ( slice\_tc\_offset\_div2  <<  1 ) ) (8‑1105)

where slice\_tc\_offset\_div2 is the value of the syntax element slice\_tc\_offset\_div2 for the slice that contains sample q0,0.

The variable tC is derived as follows:

tC = tC′ \* ( 1  <<  ( BitDepthC − 8 ) ) (8‑1106)

When maxFilterLengthCbCr is equal to 1 and bS is not equal to 2, maxFilterLengthCbCr is set equal to 0.

When maxFilterLengthCbCr is equal to 3, the following ordered steps apply:

1. The variables dpq0, dpq1, dp, dq and d are derived as follows:

dp0 = Abs( p2,0 − 2 \* p1,0 + p0,0 ) (8‑1107)

dp1 = Abs( p2,1 − 2 \* p1,1 + p0,1 ) (8‑1108)

dq0 = Abs( q2,0 − 2 \* q1,0 + q0,0 ) (8‑1109)

dq1 = Abs( q2,1 − 2 \* q1,1 + q0,1 ) (8‑1110)

dpq0 = dp0 + dq0 (8‑1111)

dpq1 = dp1 + dq1 (8‑1112)

dp = dp0 + dp1 (8‑1113)

dq = dq0 + dq1 (8‑1114)

d = dpq0 + dpq1 (8‑1115)

1. The variables dSam0 and dSam1 are both set equal to 0.
2. When d is less than β, the following ordered steps apply:
3. The variable dpq is set equal to 2 \* dpq0.
4. The variable dSam0 is derived by invoking the decision process for a chroma sample as specified in clause 8.8.3.6.8 for the sample location ( xCb + xBl, yCb + yBl ) with sample values p0,0, p3,0, q0,0, and q3,0, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam0.
5. The variable dpq is set equal to 2 \* dpq1.
6. The variable dSam1 is modified as follows:

* If edgeType is equal to EDGE\_VER, for the sample location ( xCb + xBl, yCb + yBl + 1 ), the decision process for a chroma sample as specified in clause 8.8.3.6.8 is invoked with sample values p0,1, p3,1, q0,1, and q3,1, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam1.
* Otherwise (edgeType is equal to EDGE\_HOR), f or the sample location ( xCb + xBl + 1, yCb + yBl ), the decision process for a chroma sample as specified in clause 8.8.3.6.8 is invoked with sample values p0,1, p3,1, q0,1 and q3,1, the variables dpq, β and tC as inputs, and the output is assigned to the decision dSam1.

1. The variable maxFilterLengthCbCr is modified as follows:

* If dSam0 is equal to 1 and dSam1 is equal to 1, maxFilterLengthCbCr is set equal to 3.
* Otherwise, maxFilterLengthCbCr is set equal to 1.

##### Filtering process for chroma block edges

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

* a chroma picture sample array recPicture,
* a chroma location ( xCb, yCb ) specifying the top-left sample of the current chroma coding block relative to the top-left chroma sample of the current picture,
* a chroma location ( xBl, yBl ) specifying the top-left sample of the current chroma block relative to the top-left sample of the current chroma coding block,
* a variable edgeType specifying whether a vertical (EDGE\_VER) or a horizontal (EDGE\_HOR) edge is filtered,
* a variable maxFilterLengthCbCr containing the maximum chroma filter length,
* the variable tC.

Output of this process is the modified chroma picture sample array recPicture.

The maxK is derived as follows:

* If edgeType is equal to EDGE\_VER, the following applies:

maxK = ( SubHeightC = = 1 ) ? 3 : 1 (8‑1116)

* Otherwise (edgeType is equal to EDGE\_HOR), the following applies:

maxK = ( SubWidthC = = 1 ) ? 3 : 1 (8‑1117)

The values pi and qi with i = 0..maxFilterLengthCbCr and k = 0..maxK are derived as follows:

* If edgeType is equal to EDGE\_VER, the following applies:

qi,k = recPicture[ xCb + xBl + i ][ yCb + yBl + k ] (8‑1118)

pi,k = recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] (8‑1119)

* Otherwise (edgeType is equal to EDGE\_HOR), the following applies:

qi,k = recPicture[ xCb + xBl + k ][ yCb + yBl + i ] (8‑1120)

pi,k = recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] (8‑1121)

Depending on the value of edgeType, the following applies:

* If edgeType is equal to EDGE\_VER, for each sample location ( xCb + xBl, yCb + yBl + k ), k = 0..maxK, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.8.3.6.9 is invoked with the variable maxFilterLengthCbCr, the sample values pi,k, qi,k with i = 0..maxFilterLengthCbCr, the locations ( xCb + xBl − i − 1, yCb + yBl + k ) and ( xCb + xBl + i, yCb + yBl + k ) with i = 0..maxFilterLengthCbCr − 1, and the variable tC as inputs, and the filtered sample values pi′ and qi′ with i = 0..maxFilterLengthCbCr − 1 as outputs.
2. The filtered sample values pi′ and qi′ with i = 0..maxFilterLengthCbCr − 1 replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + i ][ yCb + yBl + k ] = qi′ (8‑1122)

recPicture[ xCb + xBl − i − 1 ][ yCb + yBl + k ] = pi′ (8‑1123)

* Otherwise (edgeType is equal to EDGE\_HOR), for each sample location ( xCb + xBl + k, yCb + yBl ), k = 0..maxK, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in clause 8.8.3.6.9 is invoked with the variable maxFilterLengthCbCr, the sample values pi,k, qi,k, with i = 0..maxFilterLengthCbCr , the locations ( xCb + xBl + k, yCb + yBl − i − 1 ) and ( xCb + xBl + k, yCb + yBl + i ), and the variable tC as inputs, and the filtered sample values pi′ and qi′ as outputs.
2. The filtered sample values pi′ and qi′ replace the corresponding samples inside the sample array recPicture as follows:

recPicture[ xCb + xBl + k ][ yCb + yBl + i ] = qi′ (8‑1124)

recPicture[ xCb + xBl + k ][ yCb + yBl − i − 1 ] = pi′ (8‑1125)

##### Decision process for a luma sample

Inputs to this process are:

* the sample values p0, p3, q0 and q3,
* the variables dpq, sp, sq, spq, sidePisLargeBlk, sideQisLargeBlk, β and tC.

Output of this process is the variable dSam containing a decision.

The variables sp and sq are modified as follows:

* When sidePisLargeBlk is equal to 1, the following applies:

sp = ( sp + Abs( p3 − p0 ) + 1 )  >>  1 (8‑1126)

* When sideQisLargeBlk is equal to 1, the following applies:

sq = ( sq + Abs( q3 − q0 ) + 1 )  >>  1 (8‑1127)

The variable sThr is derived as follows:

* If sidePisLargeBlk is equal to 1 or sideQisLargeBlk is equal to 1, the following applies:

sThr = 3 \* β  >>  5 (8‑1128)

* Otherwise, the following applies:

sThr = β  >>  3 (8‑1129)

The variable dSam is specified as follows:

* If all of the following conditions are true, dSam is set equal to 1:
* dpq is less than ( β  >>  2 ),
* sp + sq is less than sThr,
* spq is less than ( 5 \* tC + 1 )  >>  1.
* Otherwise, dSam is set equal to 0.

##### Filtering process for a luma sample using short filters

Inputs to this process are:

* the sample values pi and qi with i = 0..3,
* the locations of pi and qi, ( xPi, yPi ) and ( xQi, yQi ) with i = 0..2,
* a variable dE,
* the variables dEp and dEq containing decisions to filter samples p1 and q1, respectively,
* a variable tC.

Outputs of this process are:

* the number of filtered samples nDp and nDq,
* the filtered sample values pi′ and qj′ with i = 0..nDp − 1, j = 0..nDq − 1.

Depending on the value of dE, the following applies:

* If the variable dE is equal to 2, nDp and nDq are both set equal to 3 and the following strong filtering applies:

p0′ = Clip3( p0 − 3 \* tC, p0 + 3 \* tC, ( p2 + 2 \* p1 + 2 \* p0 + 2 \* q0 + q1 + 4 )  >>  3 ) (8‑1130)

p1′ = Clip3( p1 − 2 \* tC, p1 + 2 \* tC, ( p2 + p1 + p0 + q0 + 2 )  >>  2 ) (8‑1131)

p2′ = Clip3( p2 − 1 \* tC, p2 + 1\*tC, ( 2 \* p3 + 3 \* p2 + p1 + p0 + q0 + 4 )  >>  3 ) (8‑1132)

q0′ = Clip3( q0 − 3 \* tC, q0 + 3 \* tC, ( p1 + 2 \* p0 + 2 \* q0 + 2 \* q1 + q2 + 4 )  >>  3 ) (8‑1133)

q1′ = Clip3( q1 − 2 \* tC, q1 + 2 \* tC, ( p0 + q0 + q1 + q2 + 2 )  >>  2 ) (8‑1134)

q2′= Clip3( q2 − 1 \* tC, q2 + 1 \* tC, ( p0 + q0 + q1 + 3 \* q2 + 2 \* q3 + 4 )  >>  3 ) (8‑1135)

* Otherwise, nDp and nDq are set both equal to 0 and the following weak filtering applies:
  + The following applies:

Δ = ( 9 \* ( q0 −  p0 ) − 3 \* ( q1 − p1 ) + 8 )  >>  4 (8‑1136)

* + 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 ) (8‑1137)

p0′ = Clip1Y( p0 + Δ ) (8‑1138)

q0′ = Clip1Y( q0 − Δ ) (8‑1139)

* + - When dEp 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 ) (8‑1140)

p1′ = Clip1Y( p1 + Δp ) (8‑1141)

* + - When dEq 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 ) (8‑1142)

q1′ = Clip1Y( q1 + Δq ) (8‑1143)

* + - nDp is set equal to dEp + 1 and nDq is set equal to dEq + 1.

When nDp is greater than 0 and one or more of the following conditions are true, nDp is set equal to 0:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xP0 ][ yP0 ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample p0 is equal to 1.

When nDq is greater than 0 and one or more of the following conditions are true, nDq is set equal to 0:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xQ0 ][ yQ0 ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample q0 is equal to 1.

##### Filtering process for a luma sample using long filters

Inputs to this process are:

* the variables maxFilterLengthP and maxFilterLengthQ,
* the sample values pi and qj with i = 0..maxFilterLengthP and j = 0..maxFilterLengthQ,
* the locations of pi and qj, ( xPi, yPi ) and ( xQj, yQj ) with i = 0..maxFilterLengthP − 1 and j = 0..maxFilterLengthQ − 1,
* a variable tC.

Outputs of this process are:

* the filtered sample values pi′ and qj′ with i = 0..maxFilterLengthP − 1, j = 0..maxFilterLenghtQ − 1.

The variable refMiddle is derived as follows:

* If maxFilterLengthP is equal to maxFilterLengthQ and maxFilterLengthP is equal to 5, the following applies:

refMiddle = ( p4 + p3 + 2\* ( p2 + p1 + p0 + q0 + q1 + q2 ) + q3 + q4 + 8)  >>  4 (8‑1144)

* Otherwise, if maxFilterLengthP is equal to maxFilterLengthQ and maxFilterLengthP is not equal to 5, the following applies:

refMiddle = ( p6 + p5 + p4 + p3 + p2 + p1 + 2\* ( p0 + q0 ) + q1 + q2 + q3 + q4 + q5 + q6 + 8 )  >>  4 (8‑1145)

* Otherwise, if one of the following conditions are true,
* maxFilterLengthQ is equal to 7 and maxFilterLengthP is equal to 5,
* maxFilterLengthQ is equal to 5 and maxFilterLengthP is equal to 7,

the following applies:

refMiddle = ( p4 + p3 + 2\* ( p2 + p1 + p0 + q0 + q1 + q2 ) + q3 + q4 + 8 )  >>  4 (8‑1146)

* Otherwise, if one of the following conditions are true,
* maxFilterLengthQ is equal to 5 and maxFilterLengthP is equal to 3,
* maxFilterLengthQ is equal to 3 and maxFilterLengthP is equal to 5,

the following applies:

refMiddle = ( p3 + p2 + p1 + p0 + q0 + q1 + q2 + q3 + 4)  >>  3 (8‑1147)

* Otherwise, if maxFilterLengthQ is equal to 7 and maxFilterLengthP is equal to 3, the following applies:

refMiddle = ( 2 \* ( p2 + p1 + p0 + q0 ) + p0 + p1 + q1 + q2 + q3 + q4 + q5 + q6 + 8 )  >>  4 (8‑1148)

* Otherwise, the following applies:

refMiddle = ( p6 + p5 + p4 + p3 + p2 + p1 + 2\*( q2 + q1 + q0 + p0)  + q0 + q1 + 8 )  >>  4 (8‑1149)

The variables refP and refQ are derived as follows:

refP = ( pmaxFilterLengtP + pmaxFilterLengthP-1 + 1 )  >>  1 (8‑1150)

refQ = ( qmaxFilterLengtQ + qmaxFilterLengthQ-1 + 1 )  >>  1 (8‑1151)

The variables fi and tCPDi are defined as follows:

* If maxFilterLengthP is equal to 7, the following applies:

f0..6 = { 59, 50, 41, 32, 23, 14, 5 } (8‑1152)

tCPD0..6 = { 6,  5,  4,  3,  2,  1,  1 } (8‑1153)

* Otherwise, if maxFilterLengthP is equal to 5, the following applies:

f0..4 = { 58, 45, 32, 19, 6 } (8‑1154)

tCPD0..4 = { 6, 5, 4, 3, 2 } (8‑1155)

* Otherwise, the following applies:

f0..2 = { 53, 32, 11 } (8‑1156)

tCPD0..2 = { 6, 4, 2} (8‑1157)

The variables gj and tCQDj are defined as follows:

* If maxFilterLengthQ is equal to 7, the following applies:

g0..6 = { 59, 50, 41, 32, 23, 14, 5 } (8‑1158)

tCQD0..6 = { 6, 5, 4, 3, 2, 1, 1 } (8‑1159)

* Otherwise, if maxFilterLengthQ is equal to 5, the following applies:

g0..4 = { 58, 45, 32, 19, 6 } (8‑1160)

tCQD0..4 = { 6, 5, 4, 3, 2 } (8‑1161)

* Otherwise, the following applies:

g0..2 = { 53, 32, 11 } (8‑1162)

tCQD0..2 = { 6, 4, 2 } (8‑1163)

The filtered sample values pi′ and qj′ with i = 0..maxFilterLengthP − 1 and j = 0..maxFilterLengthQ − 1 are derived as follows:

pi′ = Clip3( pi − ( tC\*tCPDi )  >> 1, pi + ( tC\*tCPDi ) >> 1, ( refMiddle\*fi + refP\*( 64 − fi )  +  32)  >>  6 ) (8‑1164)

qj′ = Clip3( qj − ( tC\*tCQDj ) >> 1, qj + ( tC\*tCQDj ) >> 1, ( refMiddle\*gj + refQ\*(64 − gj ) +  32)  >>  6 ) (8‑1165)

When one or more of the following conditions are true, the filtered sample value, p i′ is substituted by the corresponding input sample value p i with i = 0..maxFilterLengthP − 1:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xPi ][ yPi ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that 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, q i′ is substituted by the corresponding input sample value q j with j = 0..maxFilterLengthQ − 1:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xQi ][ yQi ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample qi is equal to 1.

##### Decision process for a chroma sample

Inputs to this process are:

* the sample values p0, p3, q0 and q3,
* the variables dpq, β and tC.

Output of this process is the variable dSam containing a decision.

The variable dSam is specified as follows:

* If all of the following conditions are true, dSam is set equal to 1:
* dpq is less than ( β  >>  2 ),
* Abs( p3 − p0 ) + Abs( q0 − q3 ) is less than ( β  >>  3 ),
* Abs( p0 − q0 ) is less than ( 5 \* tC + 1 )  >>  1.
* Otherwise, dSam is set equal to 0.

##### Filtering process for a chroma sample

This process is only invoked when ChromaArrayType is not equal to 0.

Inputs to this process are:

* the variable maxFilterLength,
* the chroma sample values pi and qi with i = 0..maxFilterLengthCbCr,
* the chroma locations of pi and qi, ( xPi, yPi ) and ( xQi, yQi ) with i = 0..maxFilterLengthCbCr − 1,
* a variable tC.

Outputs of this process are the filtered sample values pi′ and qi′ with i = 0..maxFilterLengthCbCr − 1.

The filtered sample values pi′ and qi′ with i = 0..maxFilterLengthCbCr − 1 are derived as follows:

* If maxFilterLengthCbCr is equal to 3, the following strong filtering applies:

p0′ = Clip3( p0 − tC, p0 + tC, ( p3 + p2 + p1 + 2 \* p0 + q0 + q1 + q2  + 4 )  >>  3 ) (8‑1166)

p1′ = Clip3( p1 − tC, p1 + tC, ( 2 \* p3 + p2 + 2 \* p1 + p0 + q0 + q1  + 4 )  >>  3 ) (8‑1167)

p2′ = Clip3( p2 − tC, p2 + tC, ( 3 \* p3 + 2 \* p2 + p1 + p0 + q0 + 4 )  >>  3 ) (8‑1168)

q0′ = Clip3( q0 − tC, q0 + tC, ( p2 + p1 + p0 + 2 \* q0 + q1 + q2 + q3  + 4 )  >>  3 ) (8‑1169)

q1′ = Clip3( q1 − tC, q1 + tC, ( p1 + p0 + q0 + 2 \* q1 + q2 + 2 \* q3  + 4 )  >>  3 ) (8‑1170)

q2′= Clip3( q2 − tC, q2 + tC, ( p0 + q0 + q1 + 2 \* q2 + 3 \* q3 + 4 )  >>  3 ) (8‑1171)

* Otherwise, the following weak filtering applies:

Δ = Clip3( −tC, tC, ( ( ( ( q0 − p0 )  <<  2 ) + p1 − q1 + 4 )  >>  3 ) ) (8‑1172)

p0′ = Clip1C( p0 + Δ ) (8‑1173)

q0′ = Clip1C( q0 − Δ ) (8‑1174)

When one or more of the following conditions are true, the filtered sample value, pi′ is substituted by the corresponding input sample value pi with i = 0..maxFilterLengthCbCr − 1:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xPi \* SubWidthC ][ yPi \* SubHeightC ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that 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, qi′ is substituted by the corresponding input sample value qi with i = 0..maxFilterLengthCbCr − 1:

* pcm\_loop\_filter\_disabled\_flag is equal to 1 and pcm\_flag[ xQi \* SubWidthC ][ yQi \* SubHeightC ] is equal to 1.
* cu\_transquant\_bypass\_flag of the coding unit that includes the coding block containing the sample qi is equal to 1.

### Sample adaptive offset process

#### General

Inputs to this process are the reconstructed picture sample array prior to sample adaptive offset recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture sample array after sample adaptive offset saoPictureL and, when ChromaArrayType is not equal to 0, the arrays saoPictureCb and saoPictureCr.

This process is performed on a CTB basis after the completion of the deblocking filter process for the decoded picture.

The sample values in the modified reconstructed picture sample array saoPictureL and, when ChromaArrayType is not equal to 0, the arrays saoPictureCb and saoPictureCr are initially set equal to the sample values in the reconstructed picture sample array recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr, respectively.

For every CTU with CTB location ( rx, ry ), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

– When slice\_sao\_luma\_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.8.4.2 is invoked with recPicture set equal to recPictureL, cIdx set equal to 0, ( rx, ry ), and both nCtbSw and nCtbSh set equal to CtbSizeY as inputs, and the modified luma picture sample array saoPictureL as output.

– When ChromaArrayType is not equal to 0 and slice\_sao\_chroma\_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.8.4.2 is invoked with recPicture set equal to recPictureCb, cIdx set equal to 1, ( rx, ry ), nCtbSw set equal to ( 1  <<  CtbLog2SizeY ) / SubWidthC and nCtbSh set equal to ( 1  <<  CtbLog2SizeY ) / SubHeightC as inputs, and the modified chroma picture sample array saoPictureCb as output.

– When ChromaArrayType is not equal to 0 and slice\_sao\_chroma\_flag of the current slice is equal to 1, the CTB modification process as specified in clause 8.8.4.2 is invoked with recPicture set equal to recPictureCr, cIdx set equal to 2, ( rx, ry ), nCtbSw set equal to ( 1  <<  CtbLog2SizeY ) / SubWidthC and nCtbSh set equal to ( 1  <<  CtbLog2SizeY ) / SubHeightC as inputs, and the modified chroma picture sample array saoPictureCr as output.

#### CTB modification process

Inputs to this process are:

– the picture sample array recPicture for the colour component cIdx,

– a variable cIdx specifying the colour component index,

– a pair of variables ( rx, ry ) specifying the CTB location,

– the CTB width nCtbSw and height nCtbSh.

Output of this process is a modified picture sample array saoPicture for the colour component cIdx.

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 scaleWidth and scaleHeight are derived as follows:

scaleWidth = ( cIdx = = 0 ) ? 1 : SubWidthC (8‑1175)

scaleHeight = ( cIdx = = 0 ) ? 1 : SubHeightC (8‑1176)

The location ( xCtb, yCtb ), specifying the top-left sample of the current CTB for the colour component cIdx relative to the top-left sample of the current picture component cIdx, is derived as follows:

( xCtb, yCtb ) = ( rx \* nCtbSw, ry \* nCtbSh ) (8‑1177)

The sample locations inside the current CTB are derived as follows:

( xSi, ySj ) = ( xCtb + i, yCtb + j ) (8‑1178)

( xYi, yYj ) = ( cIdx  = =  0 ) ? ( xSi, ySj ) : ( xSi \* SubWidthC, ySj \* SubHeightC ) (8‑1179)

For all sample locations ( xSi, ySj ) and ( xYi, yYj ) with i = 0..nCtbSw − 1 and j = 0..nCtbSh − 1, depending on the values of pcm\_loop\_filter\_disabled\_flag, pcm\_flag[ xYi ][ yYj ] and cu\_transquant\_bypass\_flag of the coding unit which includes the coding block covering recPicture[ xSi ][ ySj ], the following applies:

– If one or more of the following conditions are true, saoPicture[ xSi ][ ySj ] is not modified:

* pcm\_loop\_filter\_disabled\_flag and pcm\_flag[ xYi ][ yYj ] are both equal to 1.
* cu\_transquant\_bypass\_flag is equal to 1.
* SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 0.
* pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and xSj is equal to ( ( PpsVirtualBoundariesPosX[ n ] / scaleWidth ) − 1 ) for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1 and SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2 and SaoEoClass[ cIdx ][ rx ][ ry ] is not equal to 1.
* pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and xSj is equal to ( PpsVirtualBoundariesPosX[ n ] / scaleWidth ) for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1 and SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2 and SaoEoClass[ cIdx ][ rx ][ ry ] is not equal to 1.
* pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and ySj is equal to ( ( PpsVirtualBoundariesPosY[ n ] / scaleHeight ) − 1 ) for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1 and SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2 and SaoEoClass[ cIdx ][ rx ][ ry ] is not equal to 0.
* pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and ySj is equal to ( PpsVirtualBoundariesPosY[ n ] / scaleHeight ) for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1 and SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2 and SaoEoClass[ cIdx ][ rx ][ ry ] is not equal to 0.

[Ed. (BB): Modify highlighted sections prending on future decision transform/quantizaion bypass.]

– Otherwise, if SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 2, the following ordered steps apply:

1. The values of hPos[ k ] and vPos[ k ] for k = 0..1 are specified in Table 8‑23 based on SaoEoClass[ cIdx ][ rx ][ ry ].
2. The variable edgeIdx is derived as follows:

* The modified sample locations ( xSik′, ySjk′ ) and ( xYik′, yYjk′ ) are derived as follows:

( xSik′, ySjk′ ) = ( xSi + hPos[ k ], ySj + vPos[ k ] ) (8‑1180)

( xYik′, yYjk′ ) = ( cIdx  = =  0 ) ? ( xSik′, ySjk′ ) : ( xSik′ \* SubWidthC, ySjk′ \* SubHeightC ) (8‑1181)

* If one or more of the following conditions for all sample locations ( xSik′, ySjk′ ) and ( xYik′, yYjk′ ) with k = 0..1 are true, edgeIdx is set equal to 0:
* The sample at location ( xSik′, ySjk′ ) is outside the picture boundaries.
* loop\_filter\_across\_slices\_enabled\_flag is equal to 0 and the sample at location ( xSik′, ySjk′ ) belongs to a different slice.
* loop\_filter\_across\_bricks\_enabled\_flag is equal to 0 and the sample at location ( xSik′, ySjk′ ) belongs to a different brick.
* Otherwise, edgeIdx is derived as follows:
* The following applies:

edgeIdx = 2 + Sign( recPicture[ xSi ][ ySj ] − recPicture[ xSi + hPos[ 0 ] ][ ySj + vPos[ 0 ] ] ) +  
 Sign( recPicture[ xSi ][ ySj ] − recPicture[ xSi + hPos[ 1 ] ][ ySj + vPos[ 1 ] ] ) (8‑1182)

* When edgeIdx is equal to 0, 1, or 2, edgeIdx is modified as follows:

edgeIdx = ( edgeIdx = = 2 ) ? 0 : ( edgeIdx + 1 ) (8‑1183)

1. The modified picture sample array saoPicture[ xSi ][ ySj ] is derived as follows:

saoPicture[ xSi ][ ySj ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, recPicture[ xSi ][ ySj ] +  
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ edgeIdx ] ) (8‑1184)

* Otherwise (SaoTypeIdx[ cIdx ][ rx ][ ry ] is equal to 1), 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 list bandTable is defined with 32 elements and all elements are initially set equal 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 (8‑1185)

1. The variable bandIdx is set equal to bandTable[ recPicture[ xSi ][ ySj ]  >>  bandShift ].
2. The modified picture sample array saoPicture[ xSi ][ ySj ] is derived as follows:

saoPicture[ xSi ][ ySj ] = Clip3( 0, ( 1  <<  bitDepth ) − 1, recPicture[ xSi ][ ySj ] +  
 SaoOffsetVal[ cIdx ][ rx ][ ry ][ bandIdx ] ) (8‑1186)

Table 8‑23 – Specification of hPos and vPos according to the sample adaptive offset class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SaoEoClass[ cIdx ][ rx ][ ry ] | 0 | 1 | 2 | 3 |
| 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

#### General

Inputs of this process are the reconstructed picture sample array prior to adaptive loop filter recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr.

Outputs of this process are the modified reconstructed picture sample array after adaptive loop filter alfPictureL and, when ChromaArrayType is not equal to 0, the arrays alfPictureCb and alfPictureCr.

The sample values in the modified reconstructed picture sample array after adaptive loop filter alfPictureL and, when ChromaArrayType is not equal to 0, the arrays alfPictureCb and alfPictureCr are initially set equal to the sample values in the reconstructed picture sample array prior to adaptive loop filter recPictureL and, when ChromaArrayType is not equal to 0, the arrays recPictureCb and recPictureCr, respectively.

When slice\_alf\_enabled\_flag is equal to 1, for every coding tree unit with luma coding tree block location ( rx, ry ), where rx = 0..PicWidthInCtbsY − 1 and ry = 0..PicHeightInCtbsY − 1, the following applies:

* + When alf\_ctb\_flag[ 0 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for luma samples as specified in clause 8.8.5.2 is invoked with recPictureL, alfPictureL, and the luma coding tree block location ( xCtb, yCtb ) set equal to ( rx  <<  CtbLog2SizeY, ry  <<  CtbLog2SizeY ) as inputs, and the output is the modified filtered picture alfPictureL.
  + When ChromaArrayType is not equal to 0 and alf\_ctb\_flag[ 1 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.8.5.4 is invoked with recPicture set equal to recPictureCb, alfPicture set equal to alfPictureCb, and the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( ( rx  <<  CtbLog2SizeY ) / SubWidthC, ( ry  <<  CtbLog2SizeY ) / SubHeightC ) as inputs, and the output is the modified filtered picture alfPictureCb.
  + When ChromaArrayType is not equal to 0 and alf\_ctb\_flag[ 2 ][ rx ][ ry ] is equal to 1, the coding tree block filtering process for chroma samples as specified in clause 8.8.5.4 is invoked with recPicture set equal to recPictureCr, alfPicture set equal to alfPictureCr, and the chroma coding tree block location ( xCtbC, yCtbC ) set equal to ( ( rx  <<  CtbLog2SizeY ) / SubWidthC, ( ry  <<  CtbLog2SizeY ) / SubHeightC ) as inputs, and the output is the modified filtered picture alfPictureCr.

#### Coding tree block filtering process for luma samples

Inputs of this process are:

* a reconstructed luma picture sample array recPictureL prior to the adaptive loop filtering process,
* a filtered reconstructed luma picture sample array alfPictureL,
* a luma location ( xCtb, yCtb ) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed luma picture sample array alfPictureL.

The derivation process for filter index clause 8.8.5.3 is invoked with the location ( xCtb, yCtb ) and the reconstructed luma picture sample array recPictureL as inputs, and filtIdx[ x ][ y ] and transposeIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1 as outputs.

For the derivation of the filtered reconstructed luma samples alfPictureL[ x ][ y ], each reconstructed luma sample inside the current luma coding tree block recPictureL[ x ][ y ] is filtered as follows with x, y = 0..CtbSizeY − 1:

* + The array of luma filter coefficients f[ j ] and the array of luma clipping values c[ j ] corresponding to the filter specified by filtIdx[ x ][ y ] is derived as follows with j = 0..11:
  + If AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is less than 16, the following applies:

i = AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] (8‑1187)

f[ j ] = AlfFixFiltCoeff[ AlfClassToFiltMap[ i ][ filtIdx[ x ][ y ] ] ][ j ] (8‑1188)

c[ j ] = 2BitdepthY (8‑1189)

* + Otherwise (AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] is greater than or equal to 16, the following applies:

i = slice\_alf\_aps\_id\_luma[ AlfCtbFiltSetIdxY[ xCtb >> Log2CtbSize ][ yCtb >> Log2CtbSize ] − 16 ] (8‑1190)

f[ j ] = AlfCoeffL[ i ][ filtIdx[ x ][ y ] ][ j ] (8‑1191)

c[ j ] = AlfClipL[ i ][ filtIdx[ x ][ y ] ][ j ] (8‑1192)

* + The luma filter coefficients and clipping values index idx are derived depending on transposeIdx[ x ][ y ] as follows:
  + If transposeIndex[ x ][ y ] is equal to 1, the following applies:

idx[ ] = { 9, 4, 10, 8, 1, 5, 11, 7, 3, 0, 2, 6 } (8‑1193)

* + Otherwise, if transposeIndex[ x ][ y ] is equal to 2, the following applies:

idx[ ] = { 0, 3, 2, 1, 8, 7, 6, 5, 4, 9, 10, 11 } (8‑1194)

* + Otherwise, if transposeIndex[ x ][ y ] is equal to 3, the following applies:

idx[ ] = { 9, 8, 10, 4, 3, 7, 11, 5, 1, 0, 2, 6 } (8‑1195)

* + Otherwise, the following applies:

idx[ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 } (8‑1196)

* + The locations ( hx + i, vy + j ) for each of the corresponding luma samples ( x, y ) inside the given array recPicture of luma samples with i, j = −3..3 are derived as follows:
  + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and xCtb + x − PpsVirtualBoundariesPosX[ n ] is greater than or equal to 0 and less than 3 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( PpsVirtualBoundariesPosX[ n ], pic\_width\_in\_luma\_samples − 1, xCtb + x + i ) (8‑1197)

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosX[ n ]− xCtb − x is greater than 0 and less than 4 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( 0, PpsVirtualBoundariesPosX[ n ] − 1, xCtb + x + i ) (8‑1198)

* + Otherwise, the following applies:

hx + i = Clip3( 0, pic\_width\_in\_luma\_samples − 1, xCtb + x + i ) (8‑1199)

* + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and yCtb + y − PpsVirtualBoundariesPosY[ n ] is greater than or equal to 0 and less than 3 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( PpsVirtualBoundariesPosY[ n ], pic\_height\_in\_luma\_samples − 1, yCtb + y + j ) (8‑1200)

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosY[ n ] − yCtb − y is greater than 0 and less than 4 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( 0, PpsVirtualBoundariesPosY[ n ] − 1, yCtb + y + j ) (8‑1201)

* + Otherwise, the following applies:

vy + j = Clip3( 0, pic\_height\_in\_luma\_samples − 1, yCtb + y + j ) (8‑1202)

* + The variable applyVirtualBoundary is derived as follows:
  + If one or more of the following conditions are true, applyVirtualBoundary is set equal to 0:
  + The bottom boundary of the current coding tree block is the bottom boundary of the picture.
  + The bottom boundary of the current coding tree block is the bottom boundary of the brick and loop\_filter\_across\_bricks\_enabled\_flag is equal to 0.
  + The bottom boundary of the current coding tree block is the bottom boundary of the slice and loop\_filter\_across\_slices\_enabled\_flag is equal to 0.
  + The bottom boundary of the current coding tree block is one of the bottom virtual boundaries of the picture and pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1.
  + Otherwise, applyVirtualBoundary is set equal to 1.
  + The reconstructed sample offsets r1, r2 and r3 are specified in Table 8‑24 according to the horizontal luma sample position y and applyVirtualBoundary.
  + The variable curr is derived as follows:

curr = recPictureL[ hx, vy ] (8‑1203)

* + The variable sum is derived as follows:

sum = f[ idx[ 0 ] ]   \* (  Clip3( −c[ idx[ 0 ] ], c[ idx[ 0 ] ],     recPictureL[ hx, vy + r3 ] − curr ) +   
 Clip3( −c[ idx[ 0 ] ], c[ idx[ 0 ] ],     recPictureL[ hx, vy − r3 ] − curr ) ) +   
 f[ idx[ 1 ] ]   \* (  Clip3( −c[ idx[ 1 ] ], c[ idx[ 1 ] ],     recPictureL[ hx + 1, vy + r2 ] − curr ) +   
 Clip3( −c[ idx[ 1 ] ], c[ idx[ 1 ] ],     recPictureL[ hx − 1, vy − r2 ] − curr ) ) +   
 f[ idx[ 2 ] ]   \* (  Clip3( −c[ idx[ 2 ] ], c[ idx[ 2 ] ],     recPictureL[ hx, vy + r2 ] − curr ) +   
 Clip3( −c[ idx[ 2 ] ], c[ idx[ 2 ] ],     recPictureL[ hx, vy − r2 ] − curr ) ) +   
 f[ idx[ 3 ] ]   \* (  Clip3( −c[ idx[ 3 ] ], c[ idx[ 3 ] ],     recPictureL[ hx − 1, vy + r2 ] − curr ) +   
 Clip3( −c[ idx[ 3 ] ], c[ idx[ 3 ] ],     recPictureL[ hx + 1, vy − r2 ] − curr ) ) +   
 f[ idx[ 4 ] ]   \* (  Clip3( −c[ idx[ 4 ] ], c[ idx[ 4 ] ],     recPictureL[ hx + 2, vy + r1 ] − curr ) +   
 Clip3( −c[ idx[ 4 ] ], c[ idx[ 4 ] ],     recPictureL[ hx − 2, vy − r1 ] − curr ) ) +   
 f[ idx[ 5 ] ]   \* (  Clip3( −c[ idx[ 5 ] ], c[ idx[ 5 ] ],     recPictureL[ hx + 1, vy + r1 ] − curr ) +   
 Clip3( −c[ idx[ 5 ] ], c[ idx[ 5 ] ],     recPictureL[ hx − 1, vy − r1 ] − curr ) ) +   
 f[ idx[ 6 ] ]   \* (  Clip3( −c[ idx[ 6 ] ], c[ idx[ 6 ] ],     recPictureL[ hx, vy + r1 ] − curr ) +   
 Clip3( −c[ idx[ 6 ] ], c[ idx[ 6 ] ],     recPictureL[ hx, vy − r1 ] − curr ) ) + (8‑1204)  
 f[ idx[ 7 ] ]   \* (  Clip3( −c[ idx[ 7 ] ], c[ idx[ 7 ] ],     recPictureL[ hx − 1, vy + r1 ] − curr ) +   
 Clip3( −c[ idx[ 7 ] ], c[ idx[ 7 ] ],     recPictureL[ hx + 1, vy − r1 ] − curr ) ) +   
 f[ idx[ 8 ] ]   \* (  Clip3( −c[ idx[ 8 ] ], c[ idx[ 8 ] ],     recPictureL[ hx − 2, vy + r1 ] − curr ) +   
 Clip3( −c[ idx[ 8 ] ], c[ idx[ 8 ] ],     recPictureL[ hx + 2, vy − r1 ] − curr ) ) +   
 f[ idx[ 9 ] ]   \* (  Clip3( −c[ idx[ 9 ] ], c[ idx[ 9 ] ],     recPictureL[ hx + 3, vy ] − curr ) +   
 Clip3( −c[ idx[ 9 ] ], c[ idx[ 9 ] ],     recPictureL[ hx − 3, vy ] − curr ) ) +   
 f[ idx[ 10 ] ] \* (  Clip3( −c[ idx[ 10 ] ], c[ idx[ 10 ] ], recPictureL[ hx + 2, vy ] − curr ) +   
 Clip3( −c[ idx[ 10 ] ], c[ idx[ 10 ] ], recPictureL[ hx − 2, vy ] − curr ) ) +   
 f[ idx[ 11 ] ] \* (  Clip3( −c[ idx[ 11 ] ], c[ idx[ 11 ] ], recPictureL[ hx + 1, vy ] − curr ) +   
 Clip3( −c[ idx[ 11 ] ], c[ idx[ 11 ] ], recPictureL[ hx − 1, vy ] − curr ) )

sum = curr + ( ( sum + 64 ) >> 7 ) (8‑1205)

* + The modified filtered reconstructed luma picture sample alfPictureL[ xCtb + x ][ yCtb + y ] is derived as follows:
  + If pcm\_loop\_filter\_disabled\_flag and pcm\_flag[ xCtb+ x ][ yCtb + y ] are both equal to 1, the following applies:

alfPictureL[ xCtb + x ][ yCtb + y ] = recPictureL[ hx, vy ] (8‑1206)

* + Otherwise (pcm\_loop\_filter\_disabled\_flag is equal to 0 or pcm\_flag[ x ][ y ] is equal 0), the following applies:

alfPictureL[ xCtb + x ][ yCtb + y ] = Clip3( 0, ( 1 << BitDepthY ) − 1, sum ) (8‑1207)

Table 8‑24 – Specification of r1, r2, and r3 according to the horizontal luma sample position y and applyVirtualBoundary

|  |  |  |  |
| --- | --- | --- | --- |
| condition | r1 | r2 | r3 |
| ( y = = CtbSizeY − 5 | | y = = CtbSizeY − 4 ) && ( applyVirtualBoundary = = 1 ) | 0 | 0 | 0 |
| ( y = = CtbSizeY − 6 | | y = = CtbSizeY − 3 ) && ( applyVirtualBoundary = = 1 ) | 1 | 1 | 1 |
| ( y = = CtbSizeY − 7 | | y = = CtbSizeY − 2 ) && ( applyVirtualBoundary = = 1 ) | 1 | 2 | 2 |
| otherwise | 1 | 2 | 3 |

#### Derivation process for ALF transpose and filter index for luma samples

Inputs of this process are:

* a luma location ( xCtb, yCtb ) specifying the top-left sample of the current luma coding tree block relative to the top left sample of the current picture,
* a reconstructed luma picture sample array recPictureL prior to the adaptive loop filtering process.

Outputs of this process are

* the classification filter index array filtIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1,
* the transpose index array transposeIdx[ x ][ y ] with x, y = 0..CtbSizeY − 1.

The locations ( hx + i, vy + j ) for each of the corresponding luma samples ( x, y ) inside the given array recPicture of luma samples with i, j = −2..5 are derived as follows:

* + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and xCtb + x − PpsVirtualBoundariesPosX[ n ] is greater than or equal to 0 and less than 2 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( PpsVirtualBoundariesPosX[ n ], pic\_width\_in\_luma\_samples − 1, xCtb + x + i ) (8‑1208)

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosX[ n ] − xCtb − x is greater than 0 and less than 6 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( 0, PpsVirtualBoundariesPosX[ n ] − 1, xCtb + x + i ) (8‑1209)

* + Otherwise, the following applies:

hx + i = Clip3( 0, pic\_width\_in\_luma\_samples − 1, xCtb + x + i ) (8‑1210)

* + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and yCtb + y − PpsVirtualBoundariesPosY[ n ] is greater than or equal to 0 and less than 2 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( PpsVirtualBoundariesPosY[ n ], pic\_height\_in\_luma\_samples − 1, yCtb + y + j ) (8‑1211)

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosY[ n ] − yCtb − y is greater than 0 and less than 6 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( 0, PpsVirtualBoundariesPosY[ n ] − 1, yCtb + y + j ) (8‑1212)

* + Otherwise, the following applies:
  + If yCtb + CtbSizeY is greater than or equal to pic\_height\_in\_luma\_samples, the following applies:

vy + j = Clip3( 0, pic\_height\_in\_luma\_samples − 1, yCtb + y + j ) (8‑1213)

* + Otherwise, if y is less than CtbSizeY − 4, the following applies:

vy + j = Clip3( 0, yCtb + CtbSizeY − 5, yCtb + y + j ) (8‑1214)

* + Otherwise, the following applies:

vy + j = Clip3(  yCtb + CtbSizeY − 4, pic\_height\_in\_luma\_samples − 1, yCtb + y + j ) (8‑1215)

The classification filter index array filtIdx and the transpose index array transposeIdx are derived by the following ordered steps:

1. The variables filtH[ x ][ y ], filtV[ x ][ y ], filtD0[ x ][ y ] and filtD1[ x ][ y ] with x, y = − 2..CtbSizeY + 1 are derived as follows:

* If both x and y are even numbers or both x and y are uneven numbers, the following applies:

filtH[ x ][ y ] = Abs( ( recPicture[ hx, vy ]  <<  1 ) − recPicture[ hx − 1, vy ] −  (8‑1216)  
  recPicture[ hx + 1, vy ] )

filtV[ x ][ y ] = Abs( ( recPicture[ hx, vy ]  <<  1 ) − recPicture[ hx, vy − 1 ] −  (8‑1217)  
  recPicture[ hx, vy + 1 ] )

filtD0[ x ][ y ] = Abs( ( recPicture[ hx, vy ]  <<  1 ) − recPicture[ hx − 1, vy − 1 ] −  (8‑1218)  
 recPicture[ hx + 1, vy + 1 ] )

filtD1[ x ][ y ] = Abs( ( recPicture[ hx, vy ]  <<  1 ) − recPicture[ hx + 1, vy − 1 ] −  (8‑1219)  
 recPicture[ hx − 1, vy + 1 ] )

* Otherwise, filtH[ x ][ y ], filtV[ x ][ y ], filtD0[ x ][ y ] and filtD1[ x ][ y ] are set equal to 0.

1. The variables minY, maxY and ac are derived as follows:

* If ( y << 2 ) is equal to ( CtbSizeY − 8 ) and ( yCtb + CtbSizeY ) is less than pic\_height\_in\_luma\_samples − 1, minY is set equal to −2, maxY is set equal to 3 and ac is set equal to 96.
* Otherwise, if ( y << 2 ) is equal to ( CtbSizeY − 4 ) and ( yCtb + CtbSizeY ) is less than pic\_height\_in\_luma\_samples − 1, minY is set equal to 0, maxY is set equal to 5 and ac is set equal to 96.
* Otherwise, minY is set equal to −2 and maxY is set equal to 5 and ac is set equal to 64.

1. The variables sumH[ x ][ y ], sumV[ x ][ y ], sumD0[ x ][ y ], sumD1[ x ][ y ] and sumOfHV[ x ][ y ] with x, y = 0..( CtbSizeY − 1 ) >> 2 are derived as follows:

sumH[ x ][ y ] = ΣiΣj filtH[ h(x  <<  2 ) + i − xCtb ][ v(y  <<  2) + j − yCtb ] with i = −2..5, j = minY..maxY (8‑1220)

sumV[ x ][ y ] = ΣiΣj filtV[ h(x  <<  2 ) + i − xCtb ][ v(y  <<  2) + j − yCtb ] with i = −2..5, j = minY..maxY (8‑1221)

sumD0[ x ][ y ] = ΣiΣj filtD0[ h(x  <<  2 ) + i − xCtb ][ v(y  <<  2) + j − yCtb ] with i = −2..5, j = minY..maxY (8‑1222)

sumD1[ x ][ y ] = ΣiΣj filtD1[ h(x  <<  2 ) + i − xCtb ][ v(y  <<  2) + j − yCtb ] with i = −2..5, j = minY..maxY (8‑1223)

sumOfHV[ x ][ y ] = sumH[ x ][ y ] + sumV[ x ][ y ] (8‑1224)

1. The variables dir1[ x ][ y ], dir2[ x ][ y ] and dirS[ x ][ y ] with x, y = 0..CtbSizeY − 1 are derived as follows:

* The variables hv1, hv0 and dirHV are derived as follows:
* If sumV[ x >> 2 ][ y >> 2 ] is greater than sumH[ x >> 2 ][ y >> 2 ], the following applies:

hv1 = sumV[ x >> 2 ][ y >> 2 ] (8‑1225)

hv0 = sumH[ x >> 2 ][ y >> 2 ]  8‑1226)

dirHV = 1 (8‑1227)

* Otherwise, the following applies:

hv1 = sumH[ x >> 2 ][ y >> 2 ] (8‑1228)

hv0 = sumV[ x >> 2 ][ y >> 2 ] (8‑1229)

dirHV = 3 (8‑1230)

* The variables d1, d0 and dirD are derived as follows:
* If sumD0[ x >> 2 ][ y >> 2 ] is greater than sumD1[ x >> 2 ][ y >> 2 ], the following applies:

d1 = sumD0[ x >> 2 ][ y >> 2 ] (8‑1231)

d0 = sumD1[ x >> 2 ][ y >> 2 ] (8‑1232)

dirD = 0 (8‑1233)

* Otherwise, the following applies:

d1 = sumD1[ x >> 2 ][ y >> 2 ] (8‑1234)

d0 = sumD0[ x >> 2 ][ y >> 2 ] (8‑1235)

dirD = 2 (8‑1236)

* The variables hvd1, hvd0, are derived as follows:

hvd1 = ( d1 \* hv0 > hv1 \* d0 )  ?  d1  :  hv1 (8‑1237)

hvd0 = ( d1 \* hv0 > hv1 \* d0 )  ?  d0  :  hv0 (8‑1238)

* The variables dirS[ x ][ y ], dir1[ x ][ y ] and dir2[ x ][ y ] derived as follows:

dir1[ x ][ y ] = ( d1 \* hv0 > hv1 \* d0 )  ?  dirD  :  dirHV (8‑1239)

dir2[ x ][ y ] = ( d1 \* hv0 > hv1 \* d0 )  ?  dirHV  :  dirD (8‑1240)

dirS[ x ][ y ] = ( hvd1 > 2 \* hvd0 )  ?  1  :  ( ( hvd1 \* 2 > 9 \* hvd0 )  ?  2  :  0 ) (8‑1241)

1. The variable avgVar[ x ][ y ] with x, y = 0..CtbSizeY − 1 is derived as follows:

varTab[ ] = { 0, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4 } (8‑1242)

avgVar[ x ][ y ] = varTab[ Clip3( 0, 15, ( sumOfHV[ x >> 2 ][ y >> 2 ] \* ac ) >> ( 3 + BitDepthY ) ) ] (8‑1243)

1. The classification filter index array filtIdx[ x ][ y ] and the transpose index array transposeIdx[ x ][ y ] with x = y = 0..CtbSizeY − 1 are derived as follows:

transposeTable[ ] = { 0, 1, 0, 2, 2, 3, 1, 3 }

transposeIdx[ x ][ y ] = transposeTable[ dir1[ x ][ y ] \* 2 + ( dir2[ x ][ y ] >> 1 ) ]

filtIdx[ x ][ y ] = avgVar[ x ][ y ]

When dirS[ x ][ y ] is not equal 0, filtIdx[ x ][ y ] is modified as follows:

filtIdx[ x ][ y ] += ( ( ( dir1[ x ][ y ] & 0x1 ) << 1 ) + dirS[ x ][ y ] ) \* 5 (8‑1244)

#### Coding tree block filtering process for chroma samples

Inputs of this process are:

* a reconstructed chroma picture sample array recPicture prior to the adaptive loop filtering process,
* a filtered reconstructed chroma picture sample array alfPicture,
* a chroma location ( xCtbC, yCtbC ) specifying the top-left sample of the current chroma coding tree block relative to the top left sample of the current picture.

Output of this process is the modified filtered reconstructed chroma picture sample array alfPicture.

The width and height of the current chroma coding tree block ctbWidthC and ctbHeightC is derived as follows:

ctbWidthC = CtbSizeY / SubWidthC (8‑1245)

ctbHeightC = CtbSizeY / SubHeightC (8‑1246)

For the derivation of the filtered reconstructed chroma samples alfPicture[ x ][ y ], each reconstructed chroma sample inside the current chroma coding tree block recPicture[ x ][ y ] is filtered as follows with x = 0..ctbWidthC − 1, y = 0..ctbHeightC − 1:

* + The locations ( hx + i, vy + j ) for each of the corresponding chroma samples ( x, y ) inside the given array recPicture of chroma samples with i, j = −2..2 are derived as follows:
  + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and xCtbC + x − PpsVirtualBoundariesPosX[ n ] / SubWidthC is greater than or equal to 0 and less than 2 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( PpsVirtualBoundariesPosX[ n ] / SubWidthC, (8‑1247)  
 pic\_width\_in\_luma\_samples / SubWidthC − 1, xCtbC + x + i )

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosX[ n ] / SubWidthC − xCtbC − x is greater than 0 and less than 3 for any n = 0..pps\_num\_ver\_virtual\_boundaries − 1, the following applies:

hx + i = Clip3( 0, PpsVirtualBoundariesPosX[ n ] / SubWidthC − 1, xCtbC + x + i ) (8‑1248)

* + Otherwise, the following applies:

hx + i = Clip3( 0, pic\_width\_in\_luma\_samples / SubWidthC − 1, xCtbC + x + i ) (8‑1249)

* + If pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and yCtbC + y − PpsVirtualBoundariesPosY[ n ] / SubHeightC is greater than or equal to 0 and less than 2 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( PpsVirtualBoundariesPosY[ n ] / SubHeightC , (8‑1250)  
 pic\_height\_in\_luma\_samples / SubHeightC − 1, yCtbC + y + j )

* + Otherwise, if pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1 and PpsVirtualBoundariesPosY[ n ] / SubHeightC − yCtbC − y is greater than 0 and less than 3 for any n = 0..pps\_num\_hor\_virtual\_boundaries − 1, the following applies:

vy + j = Clip3( 0, PpsVirtualBoundariesPosY[ n ] / SubHeightC − 1, yCtbC + y + j ) (8‑1251)

* + Otherwise, the following applies:

vy + j = Clip3( 0, pic\_height\_in\_luma\_samples / SubHeightC − 1, yCtbC + y + j ) (8‑1252)

* + The variable applyVirtualBoundary is derived as follows:
  + If one or more of the following conditions are true, applyVirtualBoundary is set equal to 0:
  + The bottom boundary of the current coding tree block is the bottom boundary of the picture.
  + The bottom boundary of the current coding tree block is the bottom boundary of the brick and loop\_filter\_across\_bricks\_enabled\_flag is equal to 0.
  + The bottom boundary of the current coding tree block is the bottom boundary of the slice and loop\_filter\_across\_slices\_enabled\_flag is equal to 0.
  + The bottom boundary of the current coding tree block is one of the bottom virtual boundaries of the picture and pps\_loop\_filter\_across\_virtual\_boundaries\_disabled\_flag is equal to 1.
  + Otherwise, applyVirtualBoundary is set equal to 1.
  + The reconstructed sample offsets r1 and r2 are specified in Table 8‑24 according to the horizontal luma sample position y and applyVirtualBoundary.
  + The variable curr is derived as follows:

curr = recPicture[ hx, vy ] (8‑1253)

* + The array of chroma filter coefficients f[ j ] and the array of chroma clipping values c[ j ] is derived as follows with j = 0..5:

f[ j ] = AlfCoeffC[ slice\_alf\_aps\_id\_chroma ][ j ] (8‑1254)

c[ j ] = AlfClipC[ slice\_alf\_aps\_id\_chroma ][ j ] (8‑1255)

* + The variable sum is derived as follows:

sum = f[ 0 ] \* (  Clip3( −c[ 0 ], c[ 0 ], recPicture[ hx, vy + r2 ] − curr ) +   
 Clip3( −c[ 0 ], c[ 0 ], recPicture[ hx, vy− r2 ] − curr ) ) +   
 f[ 1 ] \* ( Clip3( −c[ 1 ], c[ 1 ], recPicture[ hx + 1, vy + r1 ] − curr ) +   
 Clip3( −c[ 1 ], c[ 1 ], recPicture[ hx − 1, vy − r1 ] − curr ) ) +   
 f[ 2 ] \* ( Clip3( −c[ 2 ], c[ 2 ], recPicture[ hx, vy + r1 ] − curr ) +   
 Clip3( −c[ 2 ], c[ 2 ], recPicture[ hx, vy − r1 ] − curr ) ) + (8‑1256)  
 f[ 3 ] \* ( Clip3( −c[ 3 ], c[ 3 ], recPicture[ hx − 1, vy + r1 ] − curr ) +   
 Clip3( −c[ 3 ], c[ 3 ], recPicture[ hx + 1, vy − r1 ] − curr ) ) +   
 f[ 4 ] \* ( Clip3( −c[ 4 ], c[ 4 ], recPicture[ hx + 2, vy ] − curr ) +   
 Clip3( −c[ 4 ], c[ 4 ], recPicture[ hx − 2, vy ] − curr ) ) +   
 f[ 5 ] \* ( Clip3( −c[ 5 ], c[ 5 ], recPicture[ hx + 1, vy ] − curr ) +   
 Clip3( −c[ 5 ], c[ 5 ], recPicture[ hx − 1, vy ] − curr ) )

sum = curr + ( sum + 64 ) >> 7 ) (8‑1257)

* + The modified filtered reconstructed chroma picture sample alfPicture[ xCtbC + x ][ yCtbC + y ] is derived as follows:
  + If pcm\_loop\_filter\_disabled\_flag and pcm\_flag[ ( xCtbC + x ) \* SubWidthC ][ ( yCtbC + y ) \* SubHeightC ] are both equal to 1, the following applies:

alfPicture[ xCtbC + x ][ yCtbC + y ] = recPictureL[ hx, vy ] (8‑1258)

* + Otherwise (pcm\_loop\_filter\_disabled\_flag is equal to 0 or pcm\_flag[ x ][ y ] is equal 0), the following applies:

alfPicture[ xCtbC + x ][ yCtbC + y ] = Clip3( 0, ( 1 << BitDepthC ) − 1, sum ) (8‑1259)

Table 8‑25 – Specification of r1 and r2 according to the horizontal luma sample position y and applyVirtualBoundary

|  |  |  |
| --- | --- | --- |
| condition | r1 | r2 |
| ( y = = ctbHeightC − 2 | | y = = ctbHeightC − 3 ) && ( applyVirtualBoundary = = 1 ) | 0 | 0 |
| ( y = = ctbHeightC − 1 | | y = = ctbHeightC − 4 ) && ( applyVirtualBoundary = = 1 ) | 1 | 1 |
| otherwise | 1 | 2 |

# Parsing process

## General

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 is equal to ue(v), se(v), uek(v) (see clause 9.2), tu(v) (see clause 9.3), tb(v) (see clause 9.4), or ae(v) (see clause 9.5).

## Parsing process for k-th order Exp-Golomb codes

### General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), uek(v) or se(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 with order k equal to 0 and syntax elements coded as uek(v) are Exp-Golomb-coded with order k. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

leadingZeroBits = −1  
for( b = 0; !b; leadingZeroBits++ ) (9‑1)  
 b = read\_bits( 1 )

The variable codeNum is then assigned as follows:

codeNum = ( 2leadingZeroBits − 1 ) \* 2k + read\_bits( leadingZeroBits + k ) (9‑2)

where the value returned from read\_bits( leadingZeroBits ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9‑1 illustrates the structure of the 0-th order Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed as specified above for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 9‑1. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as xi in Table 9‑1, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each xi is equal to either 0 or 1.

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

|  |  |
| --- | --- |
| **Bit string form** | **Range of codeNum** |
| 1 | 0 |
| 0 1 x0 | 1..2 |
| 0 0 1 x1 x0 | 3..6 |
| 0 0 0 1 x2 x1 x0 | 7..14 |
| 0 0 0 0 1 x3 x2 x1 x0 | 15..30 |
| 0 0 0 0 0 1 x4 x3 x2 x1 x0 | 31..62 |
| ... | ... |

Table 9‑2 illustrates explicitly the assignment of bit strings to codeNum values.

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

|  |  |
| --- | --- |
| **Bit string** | **codeNum** |
| 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 2 |
| 0 0 1 0 0 | 3 |
| 0 0 1 0 1 | 4 |
| 0 0 1 1 0 | 5 |
| 0 0 1 1 1 | 6 |
| 0 0 0 1 0 0 0 | 7 |
| 0 0 0 1 0 0 1 | 8 |
| 0 0 0 1 0 1 0 | 9 |
| ... | ... |

Depending on the descriptor, the value of a syntax element is derived as follows:

* If the syntax element is coded as ue(v), the value of the syntax element is equal to codeNum.
* Otherwise (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 clause 9.2.2 with codeNum as input.

### Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in clause 9.2.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 9‑3 provides the assignment rule.

Table 9‑3 – 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 ) |

## Parsing process for truncated unary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tu(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tu(v) are truncated unary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxVal inclusive, with maxVal being greater than or equal to 1. codeNum which is equal to the value of the syntax element is given by a process specified as follows:

codeNum = 0  
keepGoing = 1  
for(i = 0; i < maxVal && keepGoing; i++){  
 keepGoing = read\_bits( 1 ) (9‑3)  
 if( keepGoing )  
 codeNum ++  
}

## Parsing process for truncated binary codes

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to tb(v).

Inputs to this process are bits from the RBSP and the maximum value maxVal.

Outputs of this process are syntax element values.

Syntax elements coded as tb(v) are truncated binary coded. The range of possible values for the syntax element is determined first. The range of this syntax element is 0 to maxVal, inclusive, with maxVal being greater than or equal to 1. synVal which is equal to the value of the syntax element is given by a process specified as follows:

thVal = 1  
th = −1  
while( thVal <= maxVal ) {  
 th++  
 thVal <<= 1  
}  
val = 1 << th (9‑4)  
b = maxVal − val  
synVal = read\_bits( th )  
if( synVal >= val − b ) {  
 synVal <<= 1  
 synVal += read\_bits( 1 )  
 synVal −= val − b  
}

where the value returned from read\_bits( th ) is interpreted as a binary representation of an unsigned integer with most significant bit written first.

## CABAC parsing process for slice data

### General

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.

The initialization process as specified in clause 9.5.2 is invoked when starting the parsing of the CTU syntax specified in clause 7.3.7.2 and one or more of the following conditions are true:

* The CTU is the first CTU in a brick. [Ed. (YK): The start of the slice data is also covered by this sentence as each start of the slice data is the start of a brick.]
* The value of entropy\_coding\_sync\_enabled\_flag is equal to 1 and the CTU is the first CTU in a CTU row of a brick.

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as specified in subclause 9.5.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 9.5.4.

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 initialized after the decoding of any pcm\_alignment\_zero\_bit and all pcm\_sample\_luma and pcm\_sample\_chroma data as specified in subclause 9.5.2.5.

The storage process for context variables is applied as follows:

* When ending the parsing of the CTU syntax in clause 7.3.7.2, entropy\_coding\_sync\_enabled\_flag is equal to 1, and either CtbAddrInRs % PicWidthInCtbsY is equal to 0 or BrickId[ CtbAddrInBs ] is not equal to BrickId[ CtbAddrRsToBs[ CtbAddrInRs − 1 ] ], the storage process for context variables as specified in clause 9.5.2.3 is invoked with TableStateIdx0Wpp, TableStateIdx1Wpp, and TableMpsValWpp as outputs.

The whole CABAC parsing process for a syntax element synEl is illustrated in Figure 9‑1.



Figure 9‑1 – Illustration of CABAC parsing process for a syntax element synEl (informative)

### Initialization process

#### General

Outputs of this process are initialized CABAC internal variables.

The context variables of the arithmetic decoding engine are initialized as follows:

– If the CTU is the first CTU in a brick, the initialization process for context variables is invoked as specified in clause 9.5.2.2.

– Otherwise, if entropy\_coding\_sync\_enabled\_flag is equal to 1 and either CtbAddrInRs % PicWidthInCtbsY is equal to 0 or BrickId[ CtbAddrInBs ] is not equal to BrickId[ CtbAddrRsToBs[ CtbAddrInRs − 1 ] ], the following applies:

– The location ( xNbT, yNbT ) of the top-left luma sample of the spatial neighbouring block T (Figure 9-x) is derived using the location ( x0, y0 ) of the top-left luma sample of the current CTB as follows:

( xNbT, yNbT ) = ( x0, y0 − CtbSizeY ) (9‑5)

– The availability derivation process for a block as specified in clause 6.4.4 is invoked with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbT, yNbT ) as inputs, and the output is assigned to availableFlagT.

– The synchronization process for context variables is invoked as follows:

– If availableFlagT is equal to 1, the synchronization process for context variables as specified in clause 9.5.2.4 is invoked with TableStateIdx0Wpp, TableStateIdx1Wpp, TableMpsValWpp as inputs.

– Otherwise, the initialization process for context variables is invoked as specified in clause 9.5.2.2.

– Otherwise, the initialization process for context variables is invoked as specified in clause 9.5.2.2. [Ed. (YK): Check whether this can be merged with the condition "If the CTU is the first CTU in a brick", as the operation is the same. Note that this is also the case in HEVC.]

The decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision are initialized by invoking the initialization process for the arithmetic decoding engine as specified in subclause 9.5.2.5.

The whole initialization process for a syntax element synEl is illustrated in the flowchart of Figure 9‑2.



Figure 9‑2 – Illustration of CABAC initialization process (informative)

#### Initialization process for context variables

Outputs of this process are the initialized CABAC context variables indexed by ctxTable and ctxIdx.

[Ed. (BB): Align with ctxIdxOffset and ctxIdxInc used later]

[Ed. (BB): Add tables with current init values]

For each context variable, the two variables pStateIdx0 and pStateIdx1 are initialized as follows:

* Table 9‑6 to Table X-X 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 7.3.7.1 through 7.3.7.11, except end\_of\_brick\_one\_bit and pcm\_flag.
* From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

slopeIdx = initValue >> 4  
offsetIdx = initValue & 15 (9‑6)

* The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

m = slopeIdx \* 5 − 45  
n = ( offsetIdx << 3 ) − 16 (9‑7)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived from SliceQpY, which is derived in Equation 7‑89. Given the variables m and n, the initialization is specified as follows:

preCtxState = Clip3( 0, 127, ( ( m \* Clip3( 0, 51, SliceQpY ) ) >> 4 ) + n ) (9‑8)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived as follows from initial state mapping table initStateIdxToState[ preCtxState ] specified in Table 9‑4:

pStateIdx0 = initStateIdxToState[preCtxState] >> 4  
pStateIdx1 = initStateIdxToState[preCtxState] (9‑9)

NOTE 1 – The variables pStateIdx0 and pStateIdx1 correspond to probability state indices as further described in subclause 9.5.4.3.

In Table 9‑5, the ctxIdxOffset for which initialization is needed for each of the three initialization types, specified by the variable initType, are listed. Also listed is the table number that includes the values of initValue needed for the initialization for each value of ctxIdxInc. For P and B slice types, the derivation of initType depends on the value of the cabac\_init\_flag syntax element. The variable initType is derived as follows:

if( slice\_type = = I )  
 initType = 0  
else if( slice\_type = = P )  
 initType = cabac\_init\_flag ? 2 : 1 (9‑10)  
else  
 initType = cabac\_init\_flag ? 1 : 2

Table 9‑4 – Initial state mapping table initStateIdxToState[ preCtxState ]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **preCtxState** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **initStateIdxToState** | 307 | 323 | 340 | 359 | 378 | 398 | 419 | 442 | 466 | 491 | 517 | 544 | 574 | 604 | 637 | 671 |
| **preCtxState** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **initStateIdxToState** | 707 | 745 | 784 | 826 | 871 | 917 | 966 | 1018 | 1073 | 1130 | 1191 | 1254 | 1321 | 1392 | 1467 | 1545 |
| **preCtxState** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **initStateIdxToState** | 1628 | 1715 | 1807 | 1903 | 2005 | 2112 | 2226 | 2345 | 2470 | 2602 | 2741 | 2888 | 3043 | 3206 | 3377 | 3558 |
| **preCtxState** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **initStateIdxToState** | 3748 | 3949 | 4160 | 4383 | 4617 | 4864 | 5124 | 5399 | 5687 | 5992 | 6312 | 6650 | 7006 | 7381 | 7775 | 8192 |
| **preCtxState** | **64** | **65** | **66** | **67** | **68** | **69** | **70** | **71** | **72** | **73** | **74** | **75** | **76** | **77** | **78** | **79** |
| **initStateIdxToState** | 8192 | 8608 | 9002 | 9377 | 9733 | 10071 | 10391 | 10696 | 10984 | 11259 | 11519 | 11766 | 12000 | 12223 | 12434 | 12635 |
| **preCtxState** | **80** | **81** | **82** | **83** | **84** | **85** | **86** | **87** | **88** | **89** | **90** | **91** | **92** | **93** | **94** | **95** |
| **initStateIdxToState** | 12825 | 13006 | 13177 | 13340 | 13495 | 13642 | 13781 | 13913 | 14038 | 14157 | 14271 | 14378 | 14480 | 14576 | 14668 | 14755 |
| **preCtxState** | **96** | **97** | **98** | **99** | **100** | **101** | **102** | **103** | **104** | **105** | **106** | **107** | **108** | **109** | **110** | **111** |
| **initStateIdxToState** | 14838 | 14916 | 14991 | 15062 | 15129 | 15192 | 15253 | 15310 | 15365 | 15417 | 15466 | 15512 | 15557 | 15599 | 15638 | 15676 |
| **preCtxState** | **112** | **113** | **114** | **115** | **116** | **117** | **118** | **119** | **120** | **121** | **122** | **123** | **124** | **125** | **126** | **127** |
| **initStateIdxToState** | 15712 | 15746 | 15779 | 15809 | 15839 | 15866 | 15892 | 15917 | 15941 | 15964 | 15985 | 16005 | 16024 | 16043 | 16060 | 16076 |

| Table 9‑5 – Association of ctxIdxOffset and syntax elements for each initializationType in the initialization process | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Syntax structure** | **Syntax element** | **ctxTable** | **initType** | | |
| **0** | **1** | **2** |
| coding\_tree\_unit( ) | alf\_ctb\_flag[ ][ ][ ] | Table 9‑6 | 0 | 9 | 18 |
| alf\_ctb\_use\_first\_aps\_flag | Table 9‑7 |  |  |  |
| alf\_use\_aps\_flag | Table 9‑8 |  |  |  |
| sao( ) | sao\_merge\_left\_flag sao\_merge\_up\_flag | Table 9‑9 | 0 | 1 | 2 |
| sao\_type\_idx\_luma sao\_type\_idx\_chroma | Table 9‑10 | 0 | 1 | 2 |
| coding\_tree( ) | split\_cu\_flag |  |  |  |  |
| split\_qt\_flag |  |  |  |  |
| mtt\_split\_cu\_vertical\_flag |  |  |  |  |
| mtt\_split\_cu\_binary\_flag |  |  |  |  |
| coding\_unit( ) | cu\_skip\_flag[ ][ ] |  |  | 0 | 3 |
| pred\_mode\_ibc\_flag |  |  |  |  |
| pred\_mode\_flag |  |  |  |  |
| intra\_bdpcm\_flag[ ][ ] |  |  |  |  |
| intra\_bdpcm\_dir\_flag[ ][ ] |  |  |  |  |
| intra\_mip\_flag[ ][ ] |  |  |  |  |
| intra\_mip\_mpm\_flag[ ][ ] |  |  |  |  |
| intra\_luma\_ref\_idx[ ][ ] |  |  |  |  |
| intra\_subpartitions\_mode\_flag |  |  |  |  |
| intra\_subpartitions\_split\_flag |  |  |  |  |
| intra\_luma\_mpm\_flag[ ][ ] |  |  |  |  |
| intra\_chroma\_pred\_mode[ ][ ] |  |  |  |  |
| general\_merge\_flag[ ][ ] |  |  |  |  |
| inter\_pred\_idc[ x0 ][ y0 ] |  |  |  |  |
| inter\_affine\_flag[ ][ ] |  |  |  |  |
| cu\_affine\_type\_flag[ ][ ] |  |  |  |  |
| sym\_mvd\_flag[ ][ ] |  |  |  |  |
| ref\_idx\_l0[ ][ ] |  |  |  |  |
| mvp\_l0\_flag[ ][ ] |  |  |  |  |
| ref\_idx\_l1[ ][ ] |  |  |  |  |
| mvp\_l1\_flag[ ][ ] |  |  |  |  |
| avmr\_flag[ ][ ] |  |  |  |  |
| amvr\_precision\_flag[ ][ ] |  |  |  |  |
| bcw\_idx[ ][ ] |  |  |  |  |
| cu\_cbf |  |  |  |  |
| cu\_sbt\_flag |  |  |  |  |
| cu\_sbt\_quad\_flag |  |  |  |  |
| cu\_sbt\_horizontal\_flag |  |  |  |  |
| cu\_sbt\_pos\_flag |  |  |  |  |
| lfnst\_idx[ ][ ] |  |  |  |  |
| merge\_data( ) | regular\_merge\_flag[ ][ ] |  |  |  |  |
| mmvd\_merge\_flag[ ][ ] |  |  |  |  |
| mmvd\_cand\_flag[ ][ ] |  |  |  |  |
| mmvd\_distance\_idx[ ][ ] |  |  |  |  |
| ciip\_flag[ ][ ] |  |  |  |  |
| merge\_subblock\_flag[ ][ ] |  |  |  |  |
| merge\_subblock\_idx[ ][ ] |  |  |  |  |
| merge\_triangle\_idx0[ ][ ] |  |  |  |  |
| merge\_triangle\_idx1[ ][ ] |  |  |  |  |
| merge\_idx[ ][ ] |  |  |  |  |
| mvd\_coding( ) | abs\_mvd\_greater0\_flag[ ] |  |  |  |  |
| abs\_mvd\_greater1\_flag[ ] |  |  |  |  |
| transform\_unit( ) | tu\_cbf\_luma[ ][ ][ ] |  |  |  |  |
| tu\_cbf\_cb[ ][ ][ ] |  |  |  |  |
| tu\_cbf\_cr[ ][ ][ ] |  |  |  |  |
| cu\_qp\_delta\_abs |  |  |  |  |
| transform\_skip\_flag[ ][ ] |  |  |  |  |
| tu\_mts\_idx[ ][ ] |  |  |  |  |
| tu\_joint\_cbcr\_residual[ ][ ] |  |  |  |  |
| residual\_coding( ) | last\_sig\_coeff\_x\_prefix |  |  |  |  |
| last\_sig\_coeff\_y\_prefix |  |  |  |  |
| coded\_sub\_block\_flag[ ][ ] |  |  |  |  |
| sig\_coeff\_flag[ ][ ] |  |  |  |  |
| par\_level\_flag[ ] |  |  |  |  |
| abs\_level\_gtx\_flag[ ][ ] |  |  |  |  |

Table 9‑6 – Specification of initValue and shiftIdx for ctxIdxInc of alf\_ctb\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of alf\_ctb\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 154 | 139 | 219 | 0 |
| **1** | 186 | 186 | 236 | 0 |
| **2** | 174 | 203 | 238 | 4 |
| **3** | 183 | 183 | 232 | 0 |
| **4** | 233 | 247 | 249 | 0 |
| **5** | 250 | 249 | 235 | 1 |
| **6** | 168 | 183 | 246 | 0 |
| **7** | 248 | 232 | 234 | 0 |
| **8** | 250 | 249 | 251 | 1 |

Table 9‑7 – Specification of initValue and shiftIdx for ctxIdxInc of alf\_ctb\_use\_first\_aps\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of alf\_ctb\_use\_first\_aps\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 169 | 183 | 159 | 0 |

Table 9‑8 – Specification of initValue and shiftIdx for ctxIdxInc of alf\_use\_aps\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of alf\_use\_aps\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 201 | 200 | 154 | 0 |

Table 9‑9 – Specification of initValue and shiftIdx for ctxInc of sao\_merge\_left\_flag and sao\_merge\_up\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of sao\_merge\_left\_flag and sao\_merge\_up\_flag** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 47 | 233 | 199 | 0 |

Table 9‑10 – Specification of initValue and shiftIdx for ctxInc of sao\_type\_idx\_luma and sao\_type\_idx\_chroma

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ctxInc** | **initValue of sao\_type\_idx\_luma and sao\_type\_idx\_chroma** | | | **shiftIdx** |
| initType = = 0 | initType = = 1 | initType = = 2 |
| **0** | 47 | 95 | 95 | 0 |

#### Storage process for context variables

Inputs to this process are:

– The CABAC context variables indexed by ctxTable and ctxIdx.

Outputs of this process are:

– The variables tableStateSync0, tableStateSync1, and tableMPSSync containing the values of the variables pStateIdx0, pStateIdx1, and valMps used in the initialization process of context variables that are assigned to all syntax elements in clauses 7.3.7.1 through 7.3.7.11, except end\_of\_brick\_one\_bit, end\_of\_subset\_one\_bit, and pcm\_flag.

For each context variable, the corresponding entries pStateIdx0, pStateIdx1, and valMps of tables tableStateSync0, tableStateSync1, and tableMPSSync are initialized to the corresponding pStateIdx0, pStateIdx1, and valMps.

#### Synchronization process for context variables

Inputs to this process are:

– The variables tableStateSync0, tableStateSync1, and tableMPSSync containing the values of the variables pStateIdx0, pStateIdx1, and valMps used in the storage process of context variables that are assigned to all syntax elements in clauses 7.3.7.1 through 7.3.7.11, except end\_of\_brick\_one\_bit, end\_of\_subset\_one\_bit, and pcm\_flag.

Outputs of this process are:

– The initialized CABAC context variables indexed by ctxTable and ctxIdx.

For each context variable, the corresponding context variables pStateIdx0, pStateIdx1, and valMps are initialized to the corresponding entries pStateIdx0, pStateIdx1, and valMps of tables tableStateSync0, tableStateSync1, and tableMPSSync.

#### Initialization process for the arithmetic decoding engine

Outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables ivlCurrRange and ivlOffset. In the initialization procedure of the arithmetic decoding process, ivlCurrRange is set equal to 510 and ivlOffset is set equal to the value returned from read\_bits( 9 ) interpreted as a 9 bit binary representation of an unsigned integer with the most significant bit written first.

The bitstream shall not contain data that result in a value of ivlOffset being equal to 510 or 511.

NOTE – The description of the arithmetic decoding engine in this Specification utilizes 16 bit register precision. However, a minimum register precision of 9 bits is required for storing the values of the variables ivlCurrRange and ivlOffset after invocation of the arithmetic decoding process (DecodeBin) as specified in subclause 9.5.4.3. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in subclause 9.5.4.3.2 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in subclause 9.5.4.3.5 require a minimum register precision of 9 bits for the variables ivlCurrRange and ivlOffset. The bypass decoding process for binary decisions (DecodeBypass) as specified in subclause 9.5.4.3.4 requires a minimum register precision of 10 bits for the variable ivlOffset and a minimum register precision of 9 bits for the variable ivlCurrRange.

### Binarization process

#### General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9‑11 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the truncated binary (TB) binarization process, the k-th order Exp-Golomb (EGk) binarization process and the fixed-length (FL) binarization process are given in clauses 9.5.3.3 through 9.5.3.7, respectively.

| Table 9‑11 – Syntax elements and associated binarizations | | | |
| --- | --- | --- | --- |
| **Syntax structure** | **Syntax element** | **Binarization** | |
| **Process** | **Input parameters** |
| slice\_data( ) | end\_of\_brick\_one\_bit | FL | cMax = 1 |
| coding\_tree\_unit( ) | alf\_ctb\_flag[ ][ ][ ] | FL | cMax = 1 |
| alf\_ctb\_use\_first\_aps\_flag | FL | cMax = 1 |
| alf\_use\_aps\_flag | FL | cMax = 1 |
| alf\_luma\_fixed\_filter\_idx | TB | cMax = 15 |
| alf\_luma\_prev\_filter\_idx\_minus1 | TB | cMax = slice\_num\_alf\_aps\_ids\_luma − 2 |
| sao( ) | sao\_merge\_left\_flag | FL | cMax = 1 |
| sao\_merge\_up\_flag | FL | cMax = 1 |
| sao\_type\_idx\_luma | TR | cMax = 2, cRiceParam = 0 |
| sao\_type\_idx\_chroma | TR | cMax = 2, cRiceParam = 0 |
| sao\_offset\_abs[ ][ ][ ][ ] | TR | cMax = ( 1  <<  ( Min( bitDepth, 10 ) − 5 ) ) − 1, cRiceParam = 0 |
| sao\_offset\_sign[ ][ ][ ][ ] | FL | cMax = 1 |
| sao\_band\_position[ ][ ][ ] | FL | cMax = 31 |
| sao\_eo\_class\_luma | FL | cMax = 3 |
| sao\_eo\_class\_chroma | FL | cMax = 3 |
| coding\_tree( ) | split\_cu\_flag | FL | cMax = 1 |
| split\_qt\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_vertical\_flag | FL | cMax = 1 |
| mtt\_split\_cu\_binary\_flag | FL | cMax = 1 |
| coding\_unit( ) | cu\_skip\_flag[ ][ ] | FL | cMax = 1 |
| pred\_mode\_ibc\_flag | FL | cMax = 1 |
| pred\_mode\_flag | FL | cMax = 1 |
| pcm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_bdpcm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_bdpcm\_dir\_flag[ ][ ] | FL | cMax = 1 |
| intra\_mip\_flag[ ][ ] | FL | cMax = 1 |
| intra\_mip\_mpm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_mip\_mpm\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| intra\_mip\_mpm\_remainder[ ][ ] | FL | cMax = (cbWidth = = 4 && cbHeight = = 4) ? 31 : ( (cbWidth <= 8 && cbHeight <= 8) ? 15 : 7) |
| intra\_luma\_ref\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| intra\_subpartitions\_mode\_flag | FL | cMax = 1 |
| intra\_subpartitions\_split\_flag | FL | cMax = 1 |
| intra\_luma\_mpm\_flag[ ][ ] | FL | cMax = 1 |
| intra\_luma\_not\_planar\_flag[ ][ ] | FL | cMax = 1 |
| intra\_luma\_mpm\_idx[ ][ ] | TR | cMax = 4, cRiceParam = 0 |
| intra\_luma\_mpm\_remainder[ ][ ] | TB | cMax = 60 |
| intra\_chroma\_pred\_mode[ ][ ] | 9.5.3.8 | - |
| general\_merge\_flag[ ][ ] | FL | cMax = 1 |
| inter\_pred\_idc[ x0 ][ y0 ] | 9.5.3.9 | cbWidth, cbHeight |
| inter\_affine\_flag[ ][ ] | FL | cMax = 1 |
| cu\_affine\_type\_flag[ ][ ] | FL | cMax = 1 |
| sym\_mvd\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l0[ ][ ] | TR | cMax = NumRefIdxActive[ 0 ] − 1, cRiceParam = 0 |
| mvp\_l0\_flag[ ][ ] | FL | cMax = 1 |
| ref\_idx\_l1[ ][ ] | TR | cMax = NumRefIdxActive[ 1 ] − 1, cRiceParam = 0 |
| mvp\_l1\_flag[ ][ ] | FL | cMax = 1 |
| avmr\_flag[ ][ ] | FL | cMax = 1 |
| amvr\_precision\_flag[ ][ ] | FL | cMax = 1 |
| bcw\_idx[ ][ ] | TR | cMax = NoBackwardPredFlag ? 4: 2 |
| cu\_cbf | FL | cMax = 1 |
| cu\_sbt\_flag | FL | cMax = 1 |
| cu\_sbt\_quad\_flag | FL | cMax = 1 |
| cu\_sbt\_horizontal\_flag | FL | cMax = 1 |
| cu\_sbt\_pos\_flag | FL | cMax = 1 |
| lfnst\_idx[ ][ ] | TR | cMax = 2, cRiceParam = 0 |
| merge\_data( ) | regular\_merge\_flag[ ][ ] | FL | cMax = 1 |
| mmvd\_merge\_flag[ ][ ] | FL | cMax = 1 |
| mmvd\_cand\_flag[ ][ ] | FL | cMax = 1 |
| mmvd\_distance\_idx[ ][ ] | TR | cMax = 7, cRiceParam = 0 |
| mmvd\_direction\_idx[ ][ ] | FL | cMax = 3 |
| ciip\_flag[ ][ ] | FL | cMax = 1 |
| merge\_subblock\_flag[ ][ ] | FL | cMax = 1 |
| merge\_subblock\_idx[ ][ ] | TR | cMax = MaxNumSubblockMergeCand − 1, cRiceParam = 0 |
| merge\_triangle\_split\_dir[ ][ ] | FL | cMax = 1 |
| merge\_triangle\_idx0[ ][ ] | TR | cMax = MaxNumTriangleMergeCand − 1, cRiceParam = 0 |
| merge\_triangle\_idx1[ ][ ] | TR | cMax = MaxNumTriangleMergeCand − 2, cRiceParam = 0 |
| merge\_idx[ ][ ] | TR | cMax = MaxNumMergeCand − 1, cRiceParam = 0 |
| mvd\_coding( ) | abs\_mvd\_greater0\_flag[ ] | FL | cMax = 1 |
| abs\_mvd\_greater1\_flag[ ] | FL | cMax = 1 |
| abs\_mvd\_minus2[ ] | EG1 | - |
| mvd\_sign\_flag[ ] | FL | cMax = 1 |
| transform\_unit( ) | tu\_cbf\_luma[ ][ ][ ] | FL | cMax = 1 |
| tu\_cbf\_cb[ ][ ][ ] | FL | cMax = 1 |
| tu\_cbf\_cr[ ][ ][ ] | FL | cMax = 1 |
| cu\_qp\_delta\_abs | 9.5.3.10 | - |
| cu\_qp\_delta\_sign\_flag | FL | cMax = 1 |
| transform\_skip\_flag[ ][ ] | FL | cMax = 1 |
| tu\_mts\_idx[ ][ ] | TR | cMax = 4, cRiceParam = 0 |
| tu\_joint\_cbcr\_residual[ ][ ] | FL | cMax = 1 |
| residual\_coding( ) | last\_sig\_coeff\_x\_prefix | TR | cMax = ( log2ZoTbWidth << 1 ) − 1, cRiceParam = 0 |
| last\_sig\_coeff\_y\_prefix | TR | cMax = ( log2ZoTbHeight << 1 ) − 1, cRiceParam = 0 |
| last\_sig\_coeff\_x\_suffix | FL | cMax = ( 1  <<  ( ( last\_sig\_coeff\_x\_prefix  >>  1 ) − 1 ) − 1 ) |
| last\_sig\_coeff\_y\_suffix | FL | cMax = ( 1  <<  ( ( last\_sig\_coeff\_y\_prefix  >>  1 ) − 1 ) − 1 ) |
| coded\_sub\_block\_flag[ ][ ] | FL | cMax = 1 |
| sig\_coeff\_flag[ ][ ] | FL | cMax = 1 |
| par\_level\_flag[ ] | FL | cMax = 1 |
| abs\_level\_gtx\_flag[ ][ ] | FL | cMax = 1 |
| abs\_remainder[ ] | 9.5.3.11 | cIdx, current sub-block index i, x0, y0, xC, yC, log2TbWidth, log2TbHeight |
| dec\_abs\_level[ ] | 9.5.3.12 | cIdx, x0, y0, xC, yC, log2TbWidth, log2TbHeight |
| coeff\_sign\_flag[ ] | FL | cMax = 1 |

#### Rice parameter derivation process for abs\_remainder[ ] and dec\_abs\_level[ ]

Inputs to this process are the base level baseLevel, the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the Rice parameter cRiceParam.

Given the array AbsLevel[ x ][ y ] for the transform block with component index cIdx and the top-left luma location ( x0, y0 ), the variable locSumAbs is derived as specified by the following pseudo code:

* If transform\_skip\_flag[ x0 ][ y0 ] is equal to 1, trafoSkip is set equal to 1 and the following applies:

locSumAbs = 0  
if( xC > 0 )  
 locSumAbs += AbsLevel[ xC − 1 ][ yC ]  
if( yC > 0 ) (9‑11)  
 locSumAbs += AbsLevel[ xC ][ yC − 1 ]  
locSumAbs = Clip3( 0, 31, locSumAbs )

* Otherwise (transform\_skip\_flag[ x0 ][ y0 ] is equal to 0), trafoSkip is set equal to 0 and the following applies:

locSumAbs = 0  
if( xC < (1 << log2TbWidth) − 1 ) {  
 locSumAbs += AbsLevel[ xC + 1 ][ yC ]  
 if( xC < (1 << log2TbWidth) − 2 )  
 locSumAbs += AbsLevel[ xC + 2 ][ yC ]  
 if( yC < (1 << log2TbHeight) − 1 )  
 locSumAbs += AbsLevel[ xC + 1 ][ yC + 1 ] (9‑12)  
}  
if( yC < (1 << log2TbHeight) − 1 ) {  
 locSumAbs += AbsLevel[ xC ][ yC + 1 ]  
 if( yC < (1 << log2TbHeight) − 2 )  
 locSumAbs += AbsLevel [ xC ][ yC + 2 ]  
}   
locSumAbs = Clip3( 0, 31, locSumAbs − baseLevel \* 5 )

The following applies:

* If baseLevel is equal to 0, the variable s is set equal to Max( 0, QState − 1 ) and given the variables locSumAbs, trafoSkip and s, the Rice parameter cRiceParam and the variable ZeroPos[ n ] are derived as specified in Table 9‑12.
* Otherwise (baseLevel is greater than 0), given the variables locSumAbs and trafoSkip, the Rice parameter cRiceParam is derived as specified in Table 9‑12.

Table 9‑12 – Specification of cRiceParam and ZeroPos[ n ] based on locSumAbs, trafoSkip and s

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **trafoSkip** | **s** | **locSumAbs** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **0** |  | cRiceParam | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| **1** |  | cRiceParam | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|  | **0** | ZeroPos[ n ] | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 |
|  | **1** | ZeroPos[ n ] | 1 | 1 | 1 | 1 | 2 | 3 | 4 | 4 | 4 | 6 | 6 | 6 | 8 | 8 | 8 | 8 |
|  | **2** | ZeroPos[ n ] | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 6 | 6 | 6 | 8 | 8 | 8 | 8 |
|  |  | **locSumAbs** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **0** |  | cRiceParam | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| **1** |  | cRiceParam | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|  | **0** | ZeroPos[ n ] | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 16 | 16 | 16 | 16 |
|  | **1** | ZeroPos[ n ] | 4 | 4 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 16 | 16 | 16 | 16 | 16 | 16 |
|  | **2** | ZeroPos[ n ] | 8 | 8 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |

#### Truncated Rice binarization process

Input to this process is a request for a truncated Rice (TR) binarization, cMax and cRiceParam.

Output of this process is the TR binarization associating each value symbolVal with a corresponding bin string.

A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of symbolVal, prefixVal, is derived as follows:

prefixVal = symbolVal  >>  cRiceParam (9‑13)

* The prefix of the TR bin string is specified as follows:
* If prefixVal is less than cMax  >>  cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0. Table 9‑13 illustrates the bin strings of this unary binarization for prefixVal.
* Otherwise, the bin string is a bit string of length cMax  >>  cRiceParam with all bins being equal to 1.

Table 9‑13 – Bin string of the unary binarization (informative)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **prefixVal** | **Bin string** | | | | | |
| 0 | 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 |

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

* The suffix value suffixVal is derived as follows:

suffixVal = symbolVal − ( ( prefixVal )  <<  cRiceParam ) (9‑14)

* The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.5.3.7 for suffixVal with a cMax value equal to ( 1  <<  cRiceParam ) − 1.

NOTE – For the input parameter cRiceParam = 0, the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

#### Truncated Binary (TB) binarization process

Input to this process is a request for a TB binarization for a syntax element with value synVal and cMax. Output of this process is the TB binarization of the syntax element.The bin string of the TB binarization process of a syntax element synVal is specified as follows:

n = cMax + 1  
k = Floor( Log2( n ) ) (9‑15)  
u = ( 1  <<  ( k + 1) ) − n

* If synVal is less than u, the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.7 for synVal with a cMax value equal to ( 1  <<  k ) − 1.
* Otherwise (synVal is greater than or equal to u), the TB bin string is derived by invoking the FL binarization process specified in clause 9.5.3.7 for ( synVal + u ) with a cMax value equal to ( 1  <<  ( k + 1) ) − 1.

#### k-th order Exp-Golomb binarization process

Inputs to this process is a request for a k-th order Exp-Golomb (EGk) binarization.

Output of this process is the EGk binarization associating each value symbolVal with a corresponding bin string.

The bin string of the EGk binarization process for each value symbolVal is specified as follows, where each call of the function put( X ), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

absV = Abs( symbolVal )  
stopLoop = 0  
do  
 if( absV >= ( 1 << k ) ) {  
 put( 1 )  
 absV = absV − ( 1 << k )  
 k++  
 } else {  
 put( 0 ) (9‑16)  
 while( k− − )  
 put( ( absV >> k ) & 1 )  
 stopLoop = 1  
 }  
while( !stopLoop )

NOTE – 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 k-th order as specified in clause 9.2.

#### Limited k-th order Exp-Golomb binarization process

Inputs to this process is a request for a limited k-th order Exp-Golomb (EGk) binarization and the Rice parameter riceParam.

Output of this process is the limited EGk binarization associating each value symbolVal with a corresponding bin string.

The variables log2TransformRange and maxPreExtLen are derived as follows:

log2TransformRange = 15 (9‑17)

maxPreExtLen = 26 − log2TransformRange (9‑18)

The bin string of the limited EGk binarization process for each value symbolVal is specified as follows, where each call of the function put( X ), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

codeValue = symbolVal >> riceParam  
preExtLen = 0  
while( ( preExtLen < maxPreExtLen ) && ( codeValue > ( ( 2 << preExtLen ) − 2 ) ) ) {  
 preExtLen++  
 put( 1 )  
}  
if( preExtLen = = maxPreExtLen ) (9‑19)  
 escapeLength = log2TransformRange  
else {  
 escapeLength = preExtLen + riceParam  
 put( 0 )   
}  
symbolVal = symbolVal − ( ( ( 1 << preExtLen ) − 1 ) << riceParam )  
while( ( escapeLength− − ) > 0 )  
 put( ( symbolVal >> escapeLength ) & 1 )

#### Fixed-length binarization process

Inputs to this process are a request for a fixed-length (FL) binarization and cMax.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength‑bit unsigned integer bin string of the symbol value symbolVal, 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 intra\_chroma\_pred\_mode

Input to this process is a request for a binarization for the syntax element intra\_chroma\_pred\_mode.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element intra\_chroma\_pred\_mode is specified in Table 9‑14 and Table 9‑15.

Table 9‑14 – Binarization for intra\_chroma\_pred\_mode when sps\_cclm\_enabled\_flag is equal to 0

|  |  |
| --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **Bin string** |
| 4 | 0 |
| 0 | 100 |
| 1 | 101 |
| 2 | 110 |
| 3 | 111 |

Table 9‑15 – Binarization for intra\_chroma\_pred\_mode when sps\_cclm\_enabled\_flag is equal to 1

|  |  |
| --- | --- |
| **Value of intra\_chroma\_pred\_mode** | **Bin string** |
| 7 | 0 |
| 4 | 10 |
| 5 | 1110 |
| 6 | 1111 |
| 0 | 11000 |
| 1 | 11001 |
| 2 | 11010 |
| 3 | 11011 |

#### Binarization process for inter\_pred\_idc

Input to this process is a request for a binarization for the syntax element inter\_pred\_idc, the current luma coding block width cbWidth and the current luma coding block height cbHeight.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element inter\_pred\_idc is specified in Table 9‑16.

Table 9‑16 – Binarization for inter\_pred\_idc

|  |  |  |  |
| --- | --- | --- | --- |
| **Value of inter\_pred\_idc** | **Name of inter\_pred\_idc** | **Bin string** | |
| ( cbWidth + cbHeight )  >  12 | ( cbWidth + cbHeight )  = =  12 |
| 0 | PRED\_L0 | 00 | 0 |
| 1 | PRED\_L1 | 01 | 1 |
| 2 | PRED\_BI | 1 | - |

#### Binarization process for cu\_qp\_delta\_abs

Input to this process is a request for a binarization for the syntax element cu\_qp\_delta\_abs.

Output of this process is the binarization of the syntax element.

The binarization of the syntax element cu\_qp\_delta\_abs is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of cu\_qp\_delta\_abs, prefixVal, is derived as follows:

prefixVal = Min( cu\_qp\_delta\_abs, 5 ) (9‑20)

* The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with cMax = 5 and cRiceParam = 0.

When prefixVal is greater than 4, the suffix bin string is present and it is derived as follows:

* The suffix value of cu\_qp\_delta\_abs, suffixVal, is derived as follows:

suffixVal = cu\_qp\_delta\_abs − 5 (9‑21)

* The suffix bin string is specified by invoking the k-th order EGk binarization process as specified in clause 9.5.3.5 for suffixVal with the Exp-Golomb order k set equal to 0.

#### Binarization process for abs\_remainder[ ]

Input to this process is a request for a binarization for the syntax element abs\_remainder[ n ], the colour component cIdx, the current sub-block index i, and the luma location ( x0, y0 ) specifying the top-left sample of the current luma transform block relative to the top-left luma sample of the picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The variables lastAbsRemainder and lastRiceParam are derived as follows:

* If this process is invoked for the first time for the current sub-block index i, lastAbsRemainder and lastRiceParam are both set equal to 0.
* Otherwise (this process is not invoked for the first time for the current sub-block index i), lastAbsRemainder and lastRiceParam are set equal to the values of abs\_remainder[ n ] and cRiceParam, respectively, that have been derived during the last invocation of the binarization process for the syntax element abs\_remainder[ n ] as specified in this clause.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process for abs\_remainder[] as specified in clause 9.5.3.2 with the variable baseLevel set equal to 4, the colour component index cIdx, the luma location ( x0, y0 ), the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as inputs.

The variable cMax is derived from cRiceParam as:

cMax = 6  <<  cRiceParam (9‑22)

The binarization of the syntax element abs\_remainder[ n ] is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of abs\_remainder[ n ], prefixVal, is derived as follows:

prefixVal = Min( cMax, abs\_remainder[ n ] ) (9‑23)

* The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length 6 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

* The suffix value of abs\_remainder[ n ], suffixVal, is derived as follows:

suffixVal = abs\_remainder[ n ] − cMax (9‑24)

* The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in clause 9.5.3.6 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as input.

#### Binarization process for dec\_abs\_level[ ]

Input to this process is a request for a binarization of the syntax element dec\_abs\_level[ n ], the colour component cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left luma sample of the picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the binarization of the syntax element.

The rice parameter cRiceParam is derived by invoking the rice parameter derivation process for dec\_abs\_level[] as specified in clause 9.5.3.2 with the variable baseLevel set equal to 0, the colour component index cIdx, the luma location ( x0, y0 ), the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as inputs.

The variable cMax is derived from cRiceParam as:

cMax = 6  <<  cRiceParam (9‑25)

The binarization of dec\_abs\_level[ n ] is a concatenation of a prefix bin string and (when present) a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of dec\_abs\_level[ n ], prefixVal, is derived as follows:

prefixVal = Min( cMax, dec\_abs\_level[ n ] ) (9‑26)

* The prefix bin string is specified by invoking the TR binarization process as specified in clause 9.5.3.3 for prefixVal with the variables cMax and cRiceParam as inputs.

When the prefix bin string is equal to the bit string of length 6 with all bits equal to 1, the suffix bin string is present and it is derived as follows:

* The suffix value of dec\_abs\_level[ n ], suffixVal, is derived as follows:

suffixVal = dec\_abs\_level[ n ] − cMax (9‑27)

* The suffix bin string is specified by invoking the limited k-th order EGk binarization process as specified in clause 9.5.3.6 for the binarization of suffixVal with the Exp-Golomb order k set equal to cRiceParam + 1 and cRiceParam as input.

### Decoding process flow

#### General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 9.5.3.

Output of this process is the value of the syntax element.

This process specifies how each bin of a bin string is parsed for each syntax element. After parsing each bin, the resulting bin string is compared to all bin strings of the binarization of the syntax element and the following applies:

– If the bin string is equal to one of the bin strings, the corresponding value of the syntax element is the output.

– Otherwise (the bin 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.

The parsing of each bin is specified by the following two ordered steps:

1. The derivation process for ctxTable, ctxIdx, and bypassFlag as specified in clause 9.5.4.2 is invoked with binIdx as input and ctxTable, ctxIdx and bypassFlag as outputs.

2. The arithmetic decoding process as specified in clause 9.5.4.3 is invoked with ctxTable, ctxIdx and bypassFlag as inputs and the value of the bin as output.

#### Derivation process for ctxTable, ctxIdx and bypassFlag

##### General

Input to this process is the position of the current bin within the bin string, binIdx.

Outputs of this process are ctxTable, ctxIdx and bypassFlag.

The values of ctxTable, ctxIdx and bypassFlag are derived as follows based on the entries for binIdx of the corresponding syntax element in Table 9‑17:

* If the entry in Table 9‑17 is not equal to "bypass", "terminate" or "na", the values of binIdx are decoded by invoking the DecodeDecision process as specified in clause 9.5.4.3.2 and the following applies:
* ctxTable is specified in Table 9‑5
* The variable ctxInc is specified by the corresponding entry in Table 9‑17 and when more than one value is listed in Table 9‑17 for a binIdx, the assignment process for ctxInc for that binIdx is further specified in the clauses given in parenthesis.
* The variable ctxIdxOffset is specified in Table 9‑5 depending on the current value of initType.
* ctxIdx is set equal to the sum of ctxInc and ctxIdxOffset.
* bypassFlag is set equal to 0.
* Otherwise, if the entry in Table 9‑17 is equal to "bypass", the values of binIdx are decoded by invoking the DecodeBypass process as specified in clause 9.5.4.3.4 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 1.a
* Otherwise, if the entry in Table 9‑17 is equal to "terminate", the values of binIdx are decoded by invoking the DecodeTerminate process as specified in clause 9.5.4.3.5 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 0.
* Otherwise (the entry in Table 9‑17 is equal to "na"), the values of binIdx do not occur for the corresponding syntax element.

| Table 9‑17 – Assignment of ctxInc to syntax elements with context coded bins | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Syntax element** | **binIdx** | | | | | |
| **0** | **1** | **2** | **3** | **4** | **>= 5** |
| end\_of\_brick\_one\_bit | terminate | na | na | na | na | na |
| alf\_ctb\_flag[ ][ ][ ] | 0..8 (clause 9.5.4.2.2) | na | na | na | na | na |
| alf\_ctb\_use\_first\_aps\_flag | 0 | na | na | na | na | na |
| alf\_use\_aps\_flag | 0 | na | na | na | na | na |
| alf\_luma\_fixed\_filter\_idx | bypass | bypass | bypass | bypass | bypass | bypass |
| alf\_luma\_prev\_filter\_idx\_minus1 | bypass | bypass | bypass | bypass | bypass | bypass |
| sao\_merge\_left\_flag | 0 | na | na | na | na | na |
| sao\_merge\_up\_flag | 0 | na | na | na | na | na |
| sao\_type\_idx\_luma | 0 | bypass | na | na | na | na |
| sao\_type\_idx\_chroma | 0 | bypass | na | na | na | na |
| sao\_offset\_abs[ ][ ][ ][ ] | bypass | bypass | bypass | bypass | bypass | na |
| sao\_offset\_sign[ ][ ][ ][ ] | bypass | na | na | na | na | na |
| sao\_band\_position[ ][ ][ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| sao\_eo\_class\_luma | bypass | bypass | na | na | na | na |
| sao\_eo\_class\_chroma | bypass | bypass | na | na | na | na |
| split\_cu\_flag | 0..8 (clause 9.5.4.2.2) | na | na | na | na | na |
| split\_qt\_flag | 0..5 (clause 9.5.4.2.2) | na | na | na | na | na |
| mtt\_split\_cu\_vertical\_flag | 0..4 (clause 9.5.4.2.3) | na | na | na | na | na |
| mtt\_split\_cu\_binary\_flag | ( 2 \* mtt\_split\_cu\_vertical\_flag ) + ( mttDepth < = 1 ? 1 : 0 ) | na | na | na | na | na |
| cu\_skip\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| pred\_mode\_flag | 0,1 (clause 9.5.4.2.2) | na | na | na | na | na |
| pred\_mode\_ibc\_flag | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| pcm\_flag[ ][ ] | terminate | na | na | na | na | na |
| intra\_bdpcm\_flag[ ][ ] | 0 | na | na | na | na | na |
| intra\_bdpcm\_dir\_flag[ ][ ] | 0 | na | na | na | na | na |
| intra\_mip\_flag[ ][ ] | (Abs( Log2(cbWidth) − Log2(cbHeight) ) > 1) ?  3 : ( 0,1,2 (clause 9.5.4.2.2) ) | na | na | na | na | na |
| intra\_mip\_mpm\_flag[ ][ ] | 0 | na | na | na | na | na |
| intra\_mip\_mpm\_idx[ ][ ] | bypass | bypass | na | na | na | na |
| intra\_mip\_mpm\_remainder[ ][ ] | bypass | bypass | bypass | bypass | bypass | na |
| intra\_luma\_ref\_idx[ ][ ] | 0 | 1 | na | na | na | na |
| intra\_subpartitions\_mode\_flag | 0 | na | na | na | na | na |
| intra\_subpartitions\_split\_flag | 0 | na | na | na | na | na |
| intra\_luma\_mpm\_flag[ ][ ] | 0 | na | na | na | na | na |
| intra\_luma\_not\_planar\_flag[ ][ ] | intra\_subpartitions\_mode\_flag | na | na | na | na | na |
| intra\_luma\_mpm\_idx[ ][ ] | bypass | bypass | bypass | bypass | na | na |
| intra\_luma\_mpm\_remainder[ ][ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| intra\_chroma\_pred\_mode[ ][ ] sps\_cclm\_enabled\_flag = = 0 | 0 | bypass | bypass | na | na | na |
| intra\_chroma\_pred\_mode[ ][ ]  sps\_cclm\_enabled\_flag = = 1 &&  bin at binIdx equal to 2 = = 0 | 0 | 1 | 2 | bypass | bypass | na |
| intra\_chroma\_pred\_mode[ ][ ]  sps\_cclm\_enabled\_flag = = 1 &&  bin at binIdx equal to 2 = = 1 | 0 | 1 | 2 | 2 | na | na |
| general\_merge\_flag[ ][ ] | 0 | na | na | na | na | na |
| regular\_merge\_flag[ ][ ] | cu\_skip\_flag[ ][ ] ? 0 : 1 | na | na | na | na | na |
| mmvd\_merge\_flag[ ][ ] | 0 | na | na | na | na | na |
| mmvd\_cand\_flag[ ][ ] | 0 | na | na | na | na | na |
| mmvd\_distance\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | bypass |
| mmvd\_direction\_idx[ ][ ] | bypass | bypass | na | na | na | na |
| merge\_subblock\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| merge\_subblock\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| ciip\_flag[ ][ ] | 0 | na | na | na | na | na |
| merge\_idx[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| merge\_triangle\_split\_dir[ ][ ] | bypass | na | na | na | na | na |
| merge\_triangle\_idx0[ ][ ] | 0 | bypass | bypass | bypass | bypass | na |
| merge\_triangle\_idx1[ ][ ] | 0 | bypass | bypass | bypass | na | na |
| inter\_pred\_idc[ x0 ][ y0 ] | ( cbWidth + cbHeight ) > 12 ? 7 − ( ( 1 +  Log2( cbWidth ) + Log2( cbHeight ) ) >> 1 )   : 4 | 4 | na | na | na | na |
| inter\_affine\_flag[ ][ ] | 0,1,2 (clause 9.5.4.2.2) | na | na | na | na | na |
| cu\_affine\_type\_flag[ ][ ] | 0 | na | na | na | na | na |
| sym\_mvd\_flag[ ][ ] | 0 | na | na | na | na | na |
| ref\_idx\_l0[ ][ ] | 0 | 1 | bypass | bypass | bypass | bypass |
| ref\_idx\_l1[ ][ ] | 0 | 1 | bypass | bypass | bypass | bypass |
| mvp\_l0\_flag[ ][ ] | 0 | na | na | na | na | na |
| mvp\_l1\_flag[ ][ ] | 0 | na | na | na | na | na |
| amvr\_flag[ ][ ] | inter\_affine\_flag[ ][ ] ? 1 : 0 | na | na | na | na | na |
| amvr\_precision\_flag[ ][ ] | 0 | na | na | na | na | na |
| bcw\_idx[ ][ ] NoBackwardPredFlag = = 0 | 0 | bypass | na | na | na | na |
| bcw\_idx[ ][ ]  NoBackwardPredFlag = = 1 | 0 | bypass | bypass | bypass | na | na |
| cu\_cbf | 0 | na | na | na | na | na |
| cu\_sbt\_flag | ( cbWidth \*  cbHeight < 256 ) ? 1 : 0 | na | na | na | na | na |
| cu\_sbt\_quad\_flag | 0 | na | na | na | na | na |
| cu\_sbt\_horizontal\_flag | ( cbWidth = = cbHeight ) ? 0 : ( cbWidth < cbHeight ) ? 1 : 2 | na | na | na | na | na |
| cu\_sbt\_pos\_flag | 0 | na | na | na | na | na |
| lfnst\_idx[ ][ ] | ( tu\_mts\_idx[ x0 ][ y0 ]  = =  0 && treeType != SINGLE\_TREE ) ? 1 : 0 | bypass | na | na | na | na |
| abs\_mvd\_greater0\_flag[ ] | 0 | na | na | na | na | na |
| abs\_mvd\_greater1\_flag[ ] | 0 | na | na | na | na | na |
| abs\_mvd\_minus2[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| mvd\_sign\_flag[ ] | bypass | na | na | na | na | na |
| tu\_cbf\_luma[ ][ ][ ] | intra\_bdpcm\_flag[ ][ ] ? 4 : 0,1,2,3 (clause 9.5.4.2.5) | na | na | na | na | na |
| tu\_cbf\_cb[ ][ ][ ] | trDepth = = 0 ? 0 : 1 | na | na | na | na | na |
| tu\_cbf\_cr[ ][ ][ ] | tu\_cbf\_cb[ ][ ][ ] | na | na | na | na | na |
| cu\_qp\_delta\_abs | 0 | 1 | 1 | 1 | 1 | bypass |
| cu\_qp\_delta\_sign\_flag | bypass | na | na | na | na | na |
| transform\_skip\_flag[ ][ ] | 0 | na | na | na | na | na |
| tu\_mts\_idx[ ][ ] | cqtDepth | 6 | 7 | 8 | na | na |
| tu\_joint\_cbcr\_residual[ ][ ] | 0 | na | na | na | na | na |
| last\_sig\_coeff\_x\_prefix | 0..22 (clause 9.5.4.2.4) | | | | | |
| last\_sig\_coeff\_y\_prefix | 0..22 (clause 9.5.4.2.4) | | | | | |
| last\_sig\_coeff\_x\_suffix | bypass | bypass | bypass | bypass | bypass | bypass |
| last\_sig\_coeff\_y\_suffix | bypass | bypass | bypass | bypass | bypass | bypass |
| coded\_sub\_block\_flag[ ][ ] | ( MaxCcbs > 0)  ?  ( 0..7 (clause 9.5.4.2.6) ) : bypass | na | na | na | na | na |
| sig\_coeff\_flag[ ][ ] | ( MaxCcbs > 0)  ?  ( 0..93 (clause 9.5.4.2.8) ) : bypass | na | na | na | na | na |
| par\_level\_flag[ ] | ( MaxCcbs > 0)  ?  ( 0.332 (clause 9.5.4.2.9) ) : bypass | na | na | na | na | na |
| abs\_level\_gtx\_flag[ ] | ( MaxCcbs > 0)  ?  ( 0..70 (clause 9.5.4.2.9) ) : bypass | na | na | na | na | na |
| abs\_remainder[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| dec\_abs\_level[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| coeff\_sign\_flag[ ]  transform\_skip\_flag[ x0 ][ y0 ] = = 0 | bypass | na | na | na | na | na |
| coeff\_sign\_flag[ ] transform\_skip\_flag[ x0 ][ y0 ] = = 1 | ( MaxCcbs > 0)  ?   ( intra\_bdpcm\_flag  ?  1  :  0 )  :  bypass | na | na | na | na | na |

[Ed. (BB): In VTM, ctxInc of tu\_cbf\_cb is set to trDepth but trDepth can only takes values 0 and 1 because MaxTbSizeY is always equal to 64 and the maximum CTU size is 128. Thus, the depth resulting from implicit split cannot exceed 1.]

##### Derivation process of ctxInc using left and above syntax elements

Input to this process is the luma location ( x0, y0 ) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture, the colour component cIdx, the current coding quadtree depth cqDepth, the width and the height of the current coding block in luma samples cbWidth and cbHeight, and the variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer, allowSplitTtHor, and allowSplitQt as derived in the coding tree semantics in clause 7.4.8.4.

Output of this process is ctxInc.

The location ( xNbL, yNbL ) is set equal to ( x0 − 1, y0 ) and the variable availableL, specifying the availability of the block located directly to the left of the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbL, yNbL ) as inputs, and the output is assigned to availableL.

The location ( xNbA, yNbA ) is set equal to ( x0, y0 − 1 ) and the variable availableA specifying the availability of the coding block located directly above the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbA, yNbA ) as inputs, and the output is assigned to availableA.

The assignment of ctxInc is specified as follows with condL and condA specified in Table 9‑18:

* For the syntax elements alf\_ctb\_flag[ x0 ][ y0 ][ cIdx ], split\_qt\_flag, split\_cu\_flag, cu\_skip\_flag[ x0 ][ y0 ], pred\_mode\_ibc\_flag[ x0 ][ y0 ], inter\_affine\_flag[ x0 ][ y0 ] and merge\_subblock\_flag[ x0 ][ y0 ]:

ctxInc = ( condL  &&  availableL ) + ( condA  &&  availableA ) + ctxSetIdx \* 3 (9‑28)

* For the syntax element pred\_mode\_flag[ x0 ][ y0 ]:

ctxInc = ( condL  &&  availableL ) | | ( condA  &&  availableA ) (9‑29)

Table 9‑18 – Specification of ctxInc using left and above syntax elements

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax element** | **condL** | **condA** | **ctxSetIdx** |
| alf\_ctb\_flag[ x0 ][ y0 ][ cIdx ] | alf\_ctb\_flag[ xNbL ][ yNbL ][ cIdx ] | alf\_ctb\_flag[ xNbA ][ yNbA ][cIdx ] | cIdx |
| split\_qt\_flag | cqtDepth[ xNbL ][ yNbL ] > cqtDepth | cqtDepth[ xNbA ][ yNbA ] > cqtDepth | ( cqtDepth < 2) ? 0 : 1 |
| split\_cu\_flag | CbHeight[ xNbL ][ yNbL ] < cbHeight | CbWidth[ xNbA ][ yNbA ] < cbWidth | ( allowSplitBtVer +   allowSplitBtHor +   allowSplitTtVer +   allowSplitTtHor +   2 \* allowSplitQt − 1 ) / 3 |
| cu\_skip\_flag[ x0 ][ y0 ] | cu\_skip\_flag[ xNbL ][ yNbL ] | cu\_skip\_flag[ xNbA ][ yNbA ] | 0 |
| pred\_mode\_flag[ x0 ][ y0 ] | CuPredMode[ xNbL ][ yNbL ] = = MODE\_INTRA | CuPredMode[ xNbA ][ yNbA ] = = MODE\_INTRA | 0 |
| pred\_mode\_ibc\_flag[ x0 ][ y0 ] | CuPredMode[ xNbL ][ yNbL ] = = MODE\_IBC | CuPredMode[ xNbA ][ yNbA ] = = MODE\_IBC | 0 |
| intra\_mip\_flag[ x0 ][ y0 ] | intra\_mip\_flag[ xNbL ][ yNbL ] | intra\_mip\_flag[ xNbA ][ yNbA ] | 0 |
| merge\_subblock\_flag[ x0 ][ y0 ] | merge\_subblock\_flag[ xNbL ][ yNbL ] | inter\_affine\_flag[ xNbL ][ yNbL ] | merge\_subblock\_flag[ xNbA ][ yNbA ] | inter\_affine\_flag[ xNbA ][ yNbA ] | 0 |
| inter\_affine\_flag [ x0 ][ y0 ] | merge\_subblock\_flag[ xNbL ][ yNbL ] | inter\_affine\_flag[ xNbL ][ yNbL ] | merge\_subblock\_flag[ xNbA ][ yNbA ] | inter\_affine\_flag[ xNbA ][ yNbA ] | 0 |

##### Derivation process of ctxIncfor the syntax element mtt\_split\_cu\_vertical\_flag

Input to this process is the luma location ( x0, y0 ) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture, the width and the height of the current coding block in luma samples cbWidth and cbHeight, and the variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer, allowSplitTtHor, and allowSplitQt as derived in the coding tree semantics in clause 7.4.8.4.

Output of this process is ctxInc.

The location ( xNbL, yNbL ) is set equal to ( x0 − 1, y0 ) and the variable availableL, specifying the availability of the block located directly to the left of the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbL, yNbL ) as inputs, and the output is assigned to availableL.

The location ( xNbA, yNbA ) is set equal to ( x0, y0 − 1 ) and the variable availableA specifying the availability of the coding block located directly above the current block, is derived by invoking the availability derivation process for a block in z-scan order as specified in subclause 6.4 with the location ( xCurr, yCurr ) set equal to ( x0, y0 ) and the neighbouring location ( xNbY, yNbY ) set equal to ( xNbA, yNbA ) as inputs, and the output is assigned to availableA.

The assignment of ctxInc is specified as follows:

* If allowSplitBtVer + allowSplitBtHor is greater than allowSplitTtVer + allowSplitTtHor, ctxInc is set equal to 4.
* Otherwise, if allowSplitBtVer + allowSplitBtHor is less than allowSplitTtVer + allowSplitTtHor, ctxInc is set equal to 4.
* Otherwise, the following applies:
* The variables dA and dL are derived as follows

dA = cbWidth / ( availableA  ?  CbWidth[ xNbA ][ yNbA ]  :  1 ) (9‑30)

dL = cbHeight / ( availableL  ?  CbHeight[ xNbL ][ yNbL ]  :  1 ) (9‑31)

* If any of the following conditions is true, ctxInc is set equal to 0:
* dA is equal to dL,
* availableA is equal to FALSE,
* availableL is equal to FALSE.
* Otherwise, if dA is less then dL, ctxInc is set equal to 1.
* Otherwise, ctxInc is set equal to 0.

##### Derivation process of ctxInc for the syntax elements last\_sig\_coeff\_x\_prefix and last\_sig\_coeff\_y\_prefix

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable log2TbSize is derived as follows:

* If the syntax element to be parsed is last\_sig\_coeff\_x\_prefix, log2TbSize is set equal to log2TbWidth.
* Otherwise (the syntax element to be parsed is last\_sig\_coeff\_y\_prefix), log2TbSize is set equal to log2TbHeight.

The variables ctxOffset and ctxShift are derived as follows:

* If cIdx is equal to 0, ctxOffset is set equal to offsetY[ log2TbSize − 2 ] and ctxShift is set equal to ( log2TbSize + 1 )  >>  2 with the list offsetY specified as follows:

offsetY[] = {0, 3, 6, 10, 15} (9‑32)

* Otherwise (cIdx is greater than 0), ctxOffset is set equal to 20 and ctxShift is set equal to Clip3( 0, 2, 2log2TbSize >> 3 ).

The variable ctxInc is derived as follows:

ctxInc = ( binIdx >> ctxShift ) + ctxOffset (9‑33)

##### Derivation process of ctxInc for the syntax element tu\_cbf\_luma

Inputs to this process are the variable binIdx, the colour component index cIdx, the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight and the current transform tree depth trDepth.

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If IntraSubpartitionsSplitType is equal to ISP\_NO\_SPLIT or cIdx is not equal to 0, the following applies:

ctxInc = trDepth = = 0 ? 1 : 0 (9‑34)

* Otherwise ( IntraSubpartitionsSplitType is not equal to ISP\_NO\_SPLIT and cIdx is equal to 0 ), the following applies:
* The variable prevTuCbfY is derived as follows:
* If the current transform unit is the first one to be parsed in a coding unit, prevTuCbfY is set equal to 0.
* Otherwise, prevTuCbfY is set equal to the value of tu\_cbf\_luma of the previous luma transform unit in the current coding unit.
* The variable ctxInc is derived as follows:

ctxInc = 2 + prevTuCbfY (9‑35)

##### Derivation process of ctxInc for the syntax element coded\_sub\_block\_flag

Inputs to this process are the colour component index cIdx, the current sub-block scan location ( xS, yS ), the previously decoded bins of the syntax element coded\_sub\_block\_flag and the binary logarithm of the transform block width log2TbWidth and the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable csbfCtx is derived using the current location ( xS, yS ), two previously decoded bins of the syntax element coded\_sub\_block\_flag in scan order, log2TbWidth and log2TbHeight, as follows:

* The variables log2SbWidth and log2SbHeight are dervied as follows:

log2SbWidth = ( Min( log2TbWidth, log2TbHeight ) < 2 ? 1 : 2 ) (9‑36)

log2SbHeight = log2SbWidth (9‑37)

* The variables log2SbWidth and log2SbHeight are modified as follows:
* If log2TbWidth is less than 2 and cIdx is equal to 0, the following applies

log2SbWidth = log2TbWidth (9‑38)

log2SbHeight = 4 − log2SbWidth (9‑39)

* Otherwise, if log2TbHeight is less than 2 and cIdx is equal to 0, the following applies

log2SbHeight = log2TbHeight (9‑40)

log2SbWidth = 4 − log2SbHeight (9‑41)

* The variable csbfCtx is initialized with 0 and modified as follows:
* If transform\_skip\_flag[ xS ][ yS ] is equal to 1, the following applies:
* When xS is greater than 0, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS − 1 ][ yS ] (9‑42)

* When yS is greater than 0, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS ][ yS − 1 ] (9‑43)

* Otherwise (transform\_skip\_flag[ xS ][ yS ] is equal to 0), the following applies:
* When xS is less than ( 1  <<  ( log2TbWidth − log2SbWidth ) ) − 1, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS + 1 ][ yS ] (9‑44)

* When yS is less than ( 1  <<  ( log2TbHeight − log2SbHeight ) ) − 1, csbfCtx is modified as follows:

csbfCtx += coded\_sub\_block\_flag[ xS ][ yS + 1 ] (9‑45)

The context index increment ctxInc is derived using the colour component index cIdx and csbfCtx as follows:

* If cIdx is equal to 0, ctxInc is derived as follows:
* If transform\_skip\_flag[ xS ][ yS ] is equal to 1, the following applies:

ctxInc = 4 + csbfCtx (9‑46)

* Otherwise (transform\_skip\_flag[ xS ][ yS ] is equal to 0), the following applies:

ctxInc = Min( csbfCtx, 1 ) (9‑47)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 2 + Min( csbfCtx, 1 ) (9‑48)

##### Derivation process for the variables locNumSig, locSumAbsPass1

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Outputs of this process are the variables locNumSig and locSumAbsPass1.

Given the syntax elements sig\_coeff\_flag[ x ][ y ] and the array AbsLevelPass1[ x ][ C ] for the transform block with component index cIdx and the top-left luma location ( x0, y0 ), the variables locNumSig and locSumAbsPass1 are derived as specified by the following pseudo code:

locNumSig = 0  
locSumAbsPass1 = 0  
if( transform\_skip\_flag[ xS ][ yS ] ) {  
 if( xC > 0 ) {  
 locNumSig += sig\_coeff\_flag[ xC − 1 ][ yC ]  
 locSumAbsPass1 += AbsLevelPass1[ xC − 1 ][ yC ]  
 }  
 if( yC > 0 ) {  
 locNumSig += sig\_coeff\_flag[ xC ][ yC − 1 ]   
 locSumAbsPass1 += AbsLevelPass1[ xC ][ yC − 1 ]  
 }  
} else {  
 if( xC < (1 << log2TbWidth) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 1 ][ yC ]  
 locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC ]  
 if( xC < (1 << log2TbWidth) − 2 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 2 ][ yC ]   
 locSumAbsPass1 += AbsLevelPass1[ xC + 2 ][ yC ]  
 }  
 if( yC < (1 << log2TbHeight) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC + 1 ][ yC + 1 ] (9‑49)  
 locSumAbsPass1 += AbsLevelPass1[ xC + 1 ][ yC + 1 ]  
 }  
 }  
 if( yC < (1 << log2TbHeight) − 1 ) {  
 locNumSig += sig\_coeff\_flag[ xC ][ yC + 1 ]   
 locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 1 ]  
 if( yC < (1 << log2TbHeight) − 2 ) {  
 locNumSig += sig\_coeff\_flag[ xC ][ yC + 2 ]   
 locSumAbsPass1 += AbsLevelPass1[ xC ][ yC + 2 ]  
 }  
 }  
}

##### Derivation process of ctxInc for the syntax element sig\_coeff\_flag

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.7 with colour component index cIdx, the luma location ( x0, y0), the current coefficient scan location (xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.

The variable d is set equal to xC + yC.

The variable ctxInc is derived as follows:

* If transform\_skip\_flag[ xS ][ yS ] is equal to 1, the following applies:

ctxInc = 90 + locSumSig (9‑50)

* Otherwise ( transform\_skip\_flag[ xS ][ yS ] is equal to 1 ), the following applies:
* If cIdx is equal to 0, ctxInc is derived as follows:

ctxInc = 18 \* Max( 0, QState − 1) + Min( locSumAbsPass1, 5 ) + ( d < 2  ?  12  :  ( d < 5  ?  6  :  0 ) ) (9‑51)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 54 + 12 \* Max( 0, QState − 1) + Min( locSumAbsPass1, 5 ) + ( d < 2  ?  6  :  0 ) (9‑52)

##### Derivation process of ctxInc for the syntax elements par\_level\_flag and abs\_level\_gtx\_flag

Inputs to this process are the colour component index cIdx, the luma location ( x0, y0 ) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture, the current coefficient scan location ( xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight.

Output of this process is the variable ctxInc.

The variable ctxInc is derived as follows:

* If transform\_skip\_flag[ xS ][ yS ] is equal to 1, the following applies:
* If the syntax element is par\_level\_flag, the following applies:

ctxInc = 33 (9‑53)

* Otherwise, if the syntax element is abs\_level\_gtx\_flag[ n ][ j ], the following applies:

ctxInc = 65 + j (9‑54)

* Otherwise (transform\_skip\_flag[ xS ][ yS ] is equal to 0), the following applies:
* The variable locNumSig and locSumAbsPass1 is derived by invoking the derivation process for the variables locNumSig and locSumAbsPass1 specifies in clause 9.5.4.2.7 with colour component index cIdx, the luma location ( x0, y0), the current coefficient scan location (xC, yC ), the binary logarithm of the transform block width log2TbWidth, and the binary logarithm of the transform block height log2TbHeight as input.
* The variable ctxOffset is set equal to Min( locSumAbsPass1 − locNumSig, 4 ).
* The variable d is set equal to xC + yC.
* If xC is equal to LastSignificantCoeffX and yC is equal to LastSignificantCoeffY, ctxInc is derived as follows:

ctxInc = ( cIdx  = =  0  ?  0  :  21 ) (9‑55)

* Otherwise, if cIdx is equal to 0, ctxInc is derived as follows:

ctxInc = 1 + ctxOffset + ( d  = =  0  ?  15  :  ( d < 3  ?  10  :  ( d < 10  ?  5  :  0 ) ) ) (9‑56)

* Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

ctxInc = 22 + ctxOffset + ( d  = =  0  ?  5  :  0 ) (9‑57)

#### Arithmetic decoding process

##### General

Inputs to this process are ctxTable, ctxIdx, and bypassFlag, as derived in subclause 9.5.4.2, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 9‑3 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxTable, the ctxIdx and the bypassFlag are passed to the arithmetic decoding process DecodeBin( ctxTable, ctxIdx, bypassFlag ), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in subclause 9.5.4.3.4 is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxTable is equal to 0, and ctxIdx is equal to 0, DecodeTerminate( ) as specified in subclause 9.5.4.3.5 is invoked.

– Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ctxTable, ctxIdx ) as specified in subclause 9.5.4.3.2 is invoked.



Figure 9‑3 – 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 ivlCurrRange will be subdivided into two sub-intervals having range p( 0 ) \* ivlCurrRange and ivlCurrRange − p( 0 ) \* ivlCurrRange, 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 Specification 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 ivlCurrRange 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 ivlCurrRange \* 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

###### General

Inputs to this process are the variables ctxTable, ctxIdx, ivlCurrRange, and ivlOffset.

Outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

Figure 9‑4 shows the flowchart for decoding a single decision (DecodeDecision):

1. The value of the variable ivlLpsRange is derived as follows:

– Given the current value of ivlCurrRange, the variable qRangeIdx is derived as follows:

qRangeIdx = ivlCurrRange >> 5 (9‑58)

– Given qRangeIdx, pStateIdx0 and pStateIdx1 associated with ctxTable and ctxIdx, valMps and ivlLpsRange are derived as follows:

pState = pStateIdx1 + 16 \* pStateIdx0  
valMps = pState >> 14  
ivlLpsRange = ( qRangeIdx \* ( (valMps ? 32767 − pState : pState ) >> 9 ) >> 1 ) + 4 (9‑59)

1. The variable ivlCurrRange is set equal to ivlCurrRange − ivlLpsRange and the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 − valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

– Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state transition isperformed as specified in subclause 9.5.4.3.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in subclause 9.5.4.3.3.

###### State transition process

Inputs to this process are the current pStateIdx0 and pStateIdx1, and the decoded value binVal.

Outputs of this process are the updated pStateIdx0 and pStateIdx1 of the context variable associated with ctxIdx.

The variable ctxIdxOffset is specified in Table 9‑5 depending on the current value of initType and ctxInc set equal to ctxIdx − ctxIdxOffset.

The variables shift0 and shift1 are derived from the shiftIdx value associated with ctxTable and ctxInc in clause 9.5.2.2.

shift0 = (shiftIdx >> 2) + 2   
shift1 = (shiftIdx & 3) + 3 + shift0 (9‑60)

Depending on the decoded value binVal, the update of the two variables pStateIdx0 and pStateIdx1 associated with ctxIdx is derived as follows:

pStateIdx0 = pStateIdx0 − (pStateIdx0 >> shift0) + (1023 \* binVal >> shift0)  
pStateIdx1 = pStateIdx1 − (pStateIdx1 >> shift1) + (16383 \* binVal >> shift1) (9‑61)



Figure 9‑4 – Flowchart for decoding a decision

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset.

A flowchart of the renormalization is shown in Figure 9‑5. The current value of ivlCurrRange is first compared to 256 and further steps are specified as follows:

– If ivlCurrRange is greater than or equal to 256, no renormalization is needed and the RenormD process is finished;

– Otherwise (ivlCurrRange is less than 256), the renormalization loop is entered. Within this loop, the value of ivlCurrRange is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ).

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 9‑5 – Flowchart of renormalization

##### Bypass decoding process for binary decisions

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variable ivlOffset and the decoded value binVal.

The bypass decoding process is invoked when bypassFlag is equal to 1. Figure 9‑6 shows a flowchart of the corresponding process.

First, the value of ivlOffset is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ). Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset is decremented by ivlCurrRange.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0*.*

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 9‑6 – Flowchart of bypass decoding process

##### Decoding process for binary decisions before termination

Inputs to this process are bits from slice data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset, and the decoded value binVal.

This decoding process applies to decoding of end\_of\_brick\_one\_bit and pcm\_flag corresponding to ctxTable equal to 0 and ctxIdx equal to 0. Figure 9‑7 shows the flowchart of the corresponding decoding process, which is specified as follows:

First, the value of ivlCurrRange is decremented by 2. Then, the value of ivlOffset is compared to the value of ivlCurrRange and further steps are specified as follows:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register ivlOffset is equal to 1. When decoding end\_of\_brick \_one\_bit, this last bit inserted in register ivlOffset is interpreted as rbsp\_stop\_one\_bit.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0 and renormalization is performed as specified in subclause 9.5.4.3.3.

NOTE – This procedure may also be implemented using DecodeDecision( ctxTable, ctxIdx, bypassFlag ) with ctxTable = 0, ctxIdx = 0 and bypassFlag = 0. In the case where the decoded value is equal to 1, seven more bits would be read by DecodeDecision( ctxTable, ctxIdx, bypassFlag ) and a decoding process would have to adjust its bitstream pointer accordingly to properly decode following syntax elements.



Figure 9‑7 – Flowchart of decoding a decision before termination

# Sub-bitstream extraction process

Inputs to this process are a bitstream, a target NuhLayerId value lIdTarget, and a target highest TemporalId value tIdTarget.

Output of this process is a sub-bitstream.

It is a requirement of bitstream conformance for the input bitstream that any output sub-bitstream that is the output of the process specified in this clause with the bitstream, lIdTarget equal to any value in the range of 0 to 125, inclusive, and tIdTarget equal to any value in the range of 0 to 6, inclusive, as inputs, and that satisfies the following condition shall be a conforming bitstream:

– The output sub-bitstream contains at least one VCL NAL unit with TemporalId equal to tIdTarget and NuhLayerId equal to lIdTarget.

NOTE – A conforming bitstream contains one or more coded slice NAL units with TemporalId equal to 0.

The output sub-bitstream is derived as follows:

– Remove all NAL units with TemporalId greater than tIdTarget.

– Remove all NAL units with NuhLayerId not equal to lIdTarget.

1. Annex A  
     
   Profiles, tiers and levels

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

* 1. Overview of profiles, tiers and levels

Profiles, tiers and levels specify restrictions on bitstreams and hence limits on the capabilities needed to decode the bitstreams. Profiles, tiers and levels may also be used to indicate interoperability points between individual decoder implementations.

1. Annex B  
     
   Byte stream format

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

* 1. General

This annex specifies syntax and semantics of a byte stream format.

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 or bitstream subsets are subject to HRD conformance checking for this Specification. The first type, called a Type I bitstream, is a NAL unit stream containing only the VCL NAL units and NAL units with NalUnitType equal to FD\_NUT (filler data NAL units) for all access units in the bitstream. The second type, called a Type II bitstream, contains, in addition to the VCL NAL units and filler data NAL units for all access units in the bitstream, at least one of the following:

– additional non-VCL NAL units other than filler data NAL units,

– all leading\_zero\_8bits, zero\_byte, start\_code\_prefix\_one\_3bytes and trailing\_zero\_8bits syntax elements that form a byte stream from the NAL unit stream (as specified in Annex B).

NOTE 1 – Decoders conforming to profiles specified in Annex A do not use NAL units with NuhLayerId greater than 0 (e.g., access unit delimiter NAL units with NuhLayerId greater than 0) for access unit boundary detection, except for identification of whether a NAL unit is a VCL or non-VCL NAL unit. Consequently, hypothetical reference decoder (HRD) parameters carried in buffering period, picture timing and decoding unit information SEI messages apply to access units that are identified based on such access unit boundary detection.

Figure C.1 shows the types of bitstream conformance points checked by the HRD.

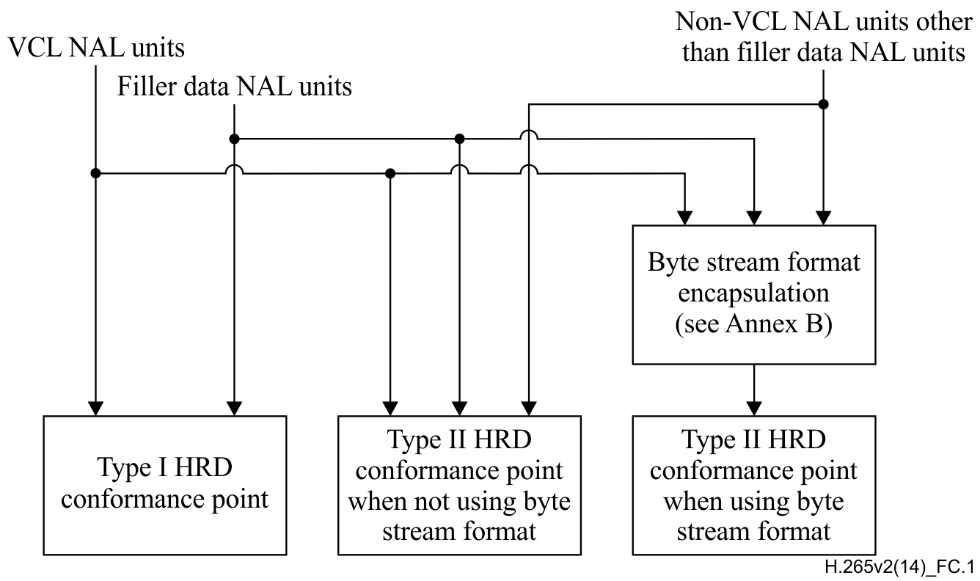


Figure C.1 – Structure of byte streams and NAL unit streams for HRD conformance checks

[Ed. (RS): This Figure C.1 needs to be updated by removing "H.265…".]

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 clauses of clause 7 and Annexes D and E.

Two types of HRD parameter sets (NAL HRD parameters and VCL HRD parameters) are used. The HRD parameter sets are signalled through the hrd\_parameters( ) syntax structure, which is part of the SPS syntax structure.

A set of bitstream conformance tests is needed for checking the conformance of a bitstream, which is referred to as the entire bitstream, denoted as entireBitstream. The set of bitstream conformance tests are for testing the conformance of the entire bitstream and its temporal subsets.

For each test, the following ordered steps apply in the order listed, followed by the processes described after these steps in this clause:

1. An operation point under test, denoted as TargetOp, is selected by selecting a target highest TemporalId value OpTid. The value of OpTid is in the range of 0 to sps\_max\_sub\_layers\_minus1, inclusive. The value of OpTid is such that the sub-bitstream BitstreamToDecode that is the output by invoking the sub-bitstream extraction process as specified in clause 10 with entireBitstream, lIdTarget equal to 0 [Ed. (YK): lIdTarget greater than 0 is currently not addressed.], and OpTid as inputs satisify the following condition:

– There is at least one VCL NAL unit with TemporalId equal to OpTid in BitstreamToDecode.

1. HighestTid is set equal to OpTid of TargetOp.
2. The hrd\_parameters( ) syntax structure and the sub\_layer\_hrd\_parameters( ) syntax structure applicable to TargetOp are selected. The hrd\_parameters( ) syntax structure in the active SPS (or provided through an external means not specified in this Specification) is selected. Within the selected hrd\_parameters( ) syntax structure, if BitstreamToDecode is a Type I bitstream, the sub\_layer\_hrd\_parameters( HighestTid ) syntax structure that immediately follows the condition "if( vcl\_hrd\_parameters\_present\_flag )" is selected and the variable NalHrdModeFlag is set equal to 0; otherwise (BitstreamToDecode is a Type II bitstream), the sub\_layer\_hrd\_parameters( HighestTid ) syntax structure that immediately follows either the condition "if( vcl\_hrd\_parameters\_present\_flag )" (in this case the variable NalHrdModeFlag is set equal to 0) or the condition "if( nal\_hrd\_parameters\_present\_flag )" (in this case the variable NalHrdModeFlag is set equal to 1) is selected. When BitstreamToDecode is a Type II bitstream and NalHrdModeFlag is equal to 0, all non-VCL NAL units except filler data NAL units, and all leading\_zero\_8bits, zero\_byte, start\_code\_prefix\_one\_3bytes and trailing\_zero\_8bits syntax elements that form a byte stream from the NAL unit stream (as specified in Annex B), when present, are discarded from BitstreamToDecode and the remaining bitstream is assigned to BitstreamToDecode.
3. An access unit associated with a buffering period SEI message (present in BitstreamToDecode or available through external means not specified in this Specification) applicable to TargetOp is selected as the HRD initialization point and referred to as access unit 0.
4. For each access unit in BitstreamToDecode starting from access unit 0, the buffering period SEI message (present in BitstreamToDecode or available through external means not specified in this Specification) that is associated with the access unit and applies to TargetOp is selected, and the picture timing SEI message (present in BitstreamToDecode or available through external means not specified in this Specification) that is associated with the access unit and applies to TargetOp is selected. The selected buffering period and picture timing SEI messages shall be either SEI messages or provided by external means.
5. A value of SchedSelIdx is selected. The selected SchedSelIdx shall be in the range of 0 to vui\_cpb\_cnt\_minus1[ HighestTid ], inclusive, where vui\_cpb\_cnt\_minus1[ HighestTid ] is found in the hrd\_parameters( ) syntax structure as selected above.

Each conformance test consists of a combination of one option in each of the above steps. When there is more than one option for a step, for any particular conformance test only one option is chosen. All possible combinations of all the steps form the entire set of conformance tests. For each operation point under test, the number of bitstream conformance tests to be performed is equal to n0 \* n1 \* n2, where the values of n0, n1, and n2 are specified as follows:

– n0 is derived as follows:

– If BitstreamToDecode is a Type I bitstream, n0 is equal to 1.

– Otherwise (BitstreamToDecode is a Type II bitstream), n0 is equal to 2.

– n1 is equal to vui\_cpb\_cnt\_minus1[ HighestTid ] + 1.

– n2 is the number of access units in BitstreamToDecode that each is associated with a buffering period SEI message applicable to TargetOp.

When BitstreamToDecode is a Type II bitstream, the following applies:

– If the sub\_layer\_hrd\_parameters( HighestTid ) syntax structure that immediately follows the condition "if( vcl\_hrd\_parameters\_present\_flag )" is selected, the test is conducted at the Type I conformance point shown in Figure C.1, and only VCL and filler data NAL units are counted for the input bit rate and CPB storage.

– Otherwise (the sub\_layer\_hrd\_parameters( HighestTid ) syntax structure that immediately follows the condition "if( nal\_hrd\_parameters\_present\_flag )" is selected), the test is conducted at the Type II conformance point shown in Figure C.1, and all bytes of the Type II bitstream, which may be a NAL unit stream or a byte stream, are counted for the input bit rate and CPB storage.

NOTE 2 – 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 variable bit rate (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.

All VPSs, SPSs, PPSs, and APSs referred to in the VCL NAL units and the 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 Specification.

In Annexes C, D and E, the specification for "presence" of non-VCL NAL units that contain VPSs, SPSs, PPSs, APSs, buffering period SEI messages or picture timing SEI messages 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 in this Specification. 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 such 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 such 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 as specified in this Specification.

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 clause 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 are supplied by some other means not specified in this Specification.

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.

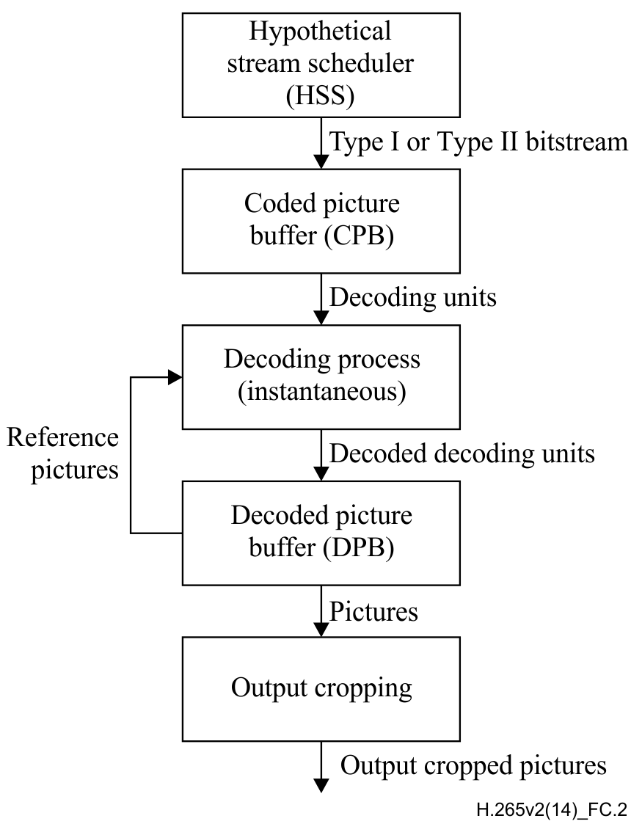


Figure C.2 – HRD buffer model

[Ed. (RS): Figure C.2 needs to be updated by removing "H.265…".]

For each bitstream conformance test, the CPB size (number of bits) is CpbSize[ SchedSelIdx ] as specified in clause E.3.3, where SchedSelIdx and the HRD parameters are specified above in this clause. The DPB size (number of picture storage buffers) is sps\_max\_dec\_pic\_buffering\_minus1[ HighestTid ] + 1.

The following is specified for expressing the constraints in this annex:

– Each access unit is referred to as access unit n, where the number n identifies the particular access unit. Access unit 0 is selected per step 4 above. The value of n is incremented by 1 for each subsequent access unit in decoding order.

– Picture n refers to the coded picture or the decoded picture of access unit n.

The HRD operates as follows:

– The HRD is initialized at access unit 0, with both the CPB and the DPB being set to be empty (the DPB fullness is set equal to 0).

NOTE 5 – After initialization, the HRD is not initialized again by subsequent buffering period SEI messages.

– Data associated with decoding units that flow into the CPB according to a specified arrival schedule are delivered by the hypothetical stream scheduler (HSS).

– The data associated with each access unit are removed and decoded instantaneously by the instantaneous decoding process at the CPB removal time of the access unit.

– Each decoded picture is placed in the DPB.

– A decoded picture is removed from the DPB when it becomes no longer needed for inter prediction reference and no longer needed for output.

For each bitstream conformance test, the operation of the CPB is specified in clause C.2, the instantaneous decoder operation is specified in clauses 2 through 10, the operation of the DPB is specified in clause C.3 and the output cropping is specified in clauses C.3.3 and C.5.2.2.

HSS and HRD information concerning the number of enumerated delivery schedules and their associated bit rates and buffer sizes is specified in clauses E.2.2 and E.3.2. The HRD is initialized as specified by the buffering period SEI message specified in clauses D.2.2 and D.3.2. The removal timing of access units from the CPB and output timing of decoded pictures from the DPB is specified using information in picture timing SEI messages (specified in clauses D.2.3 and D.3.3). All timing information relating to a specific access unit shall arrive prior to the CPB removal time of the access unit.

The requirements for bitstream conformance are specified in clause C.4 and the HRD is used to check conformance of bitstreams as specified above in this clause and to check conformance of decoders as specified in clause C.5.

NOTE 6 – While conformance is guaranteed under the assumption that all picture-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 performed 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 access unit is not necessarily an integer.

The variable ClockTick is derived as follows and is called a clock tick:

ClockTick = num\_units\_in\_tick ÷ time\_scale (C‑1)

* 1. Operation of coded picture buffer
     1. General

The specifications in this clause apply independently to each set of coded picture buffer (CPB) parameters that is present and to both the Type I and Type II conformance points shown in Figure C.1 and the set of CPB parameters is selected as specified in clause C.1.

* + 1. Timing of decoding unit arrival

The process specified in the remainder of this clause is invoked for derivation of the initial and final CPB arrival times for access unit n.

The variables InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are derived as follows:

– InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are set equal to the values of the buffering period SEI message syntax elements nal\_initial\_cpb\_removal\_delay[ SchedSelIdx ] and nal\_initial\_cpb\_removal\_offset[ SchedSelIdx ], respectively, when NalHrdModeFlag is equal to 1, or vcl\_initial\_cpb\_removal\_delay[ SchedSelIdx ] and vcl\_initial\_cpb\_removal\_offset[ SchedSelIdx ], respectively, when NalHrdModeFlag is equal to 0, where the buffering period SEI message syntax elements are selected as specified in clause C.1.

The time at which the first bit of access unit m begins to enter the CPB is referred to as the initial arrival time initArrivalTime[ m ].

The initial arrival time of access unit m is derived as follows:

– If the access unit is access unit 0 (i.e., when m is equal to 0), initArrivalTime[ 0 ] is set equal to 0.

– Otherwise (the decoding unit is access unit m with m > 0), the following applies:

– If cbr\_flag[ SchedSelIdx ] is equal to 1, the initial arrival time for access unit m is equal to the final arrival time (which is derived below) of decoding unit m − 1, i.e.,

initArrivalTime[ m ] = FinalArrivalTime[ m − 1 ] (C‑2)

– Otherwise (cbr\_flag[ SchedSelIdx ] is equal to 0), the initial arrival time for access unit m is derived as follows:

initArrivalTime[ m ] = Max( FinalArrivalTime[ m − 1 ], initArrivalEarliestTime[ m ] ) (C‑3)

where initArrivalEarliestTime[ m ] is derived as follows:

– The variable tmpNominalRemovalTime is derived as follows:

tmpNominalRemovalTime = NominalRemovalTime[ m ] (C‑4)

where NominalRemovalTime[ m ] is the nominal CPB removal time of access unit m, as specified in clause C.2.3.

– If access unit m is not the first access unit of a subsequent buffering period, initArrivalEarliestTime[ m ] is derived as follows:

initArrivalEarliestTime[ m ] = tmpNominalRemovalTime − ( InitCpbRemovalDelay[ SchedSelIdx ]  
 + InitCpbRemovalDelayOffset[ SchedSelIdx ] ) ÷ 90000 (C‑5)

– Otherwise (access unit m is the first access unit of a subsequent buffering period), initArrivalEarliestTime[ m ] is derived as follows:

initArrivalEarliestTime[ m ] = tmpNominalRemovalTime −  
 ( InitCpbRemovalDelay[ SchedSelIdx ] ÷ 90000 ) (C‑6)

The final arrival time for access unit m is derived as follows:

FinalArrivalTime[ m ] = initArrivalTime[ m ] + sizeInbits[ m ] ÷ BitRate[ SchedSelIdx ] (C‑7)

where sizeInbits[ m ] is the size in bits of access 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 selected hrd\_parameters( ) syntax structures for the access unit containing access 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 selected hrd\_parameters( ) syntax structures for the access unit containing access unit m that results in a BitRate[ SchedSelIdx1 ] or CpbSize[ SchedSelIdx1 ] for the access unit containing access 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 the initial CPB arrival time of the current access unit.

– The variable CpbSize[ SchedSelIdx ] comes into effect as follows:

– If the new value of CpbSize[ SchedSelIdx ] is greater than the old CPB size, it comes into effect at the initial CPB arrival time of the current access unit.

– Otherwise, the new value of CpbSize[ SchedSelIdx ] comes into effect at the CPB removal time of the current access unit.

* + 1. Timing of access unit removal and decoding of access unit

The variables InitCpbRemovalDelay[ SchedSelIdx ], and InitCpbRemovalDelayOffset[ SchedSelIdx ], are derived as follows: [Ed. (VD): InitCpbRemovalDelayOffset[ SchedSelIdx ] is derived here but never used in this clause. (YK): Hmm, this is simialr as in HEVC. I don’t know why it was like this in HEVC; maybe inherited from AVC. We should study whether the derivation of InitCpbRemovalDelayOffset[ SchedSelIdx ] is really not needed herein, and if not, we can just remove it. Probaly also for HEVC.]

– InitCpbRemovalDelay[ SchedSelIdx ] and InitCpbRemovalDelayOffset[ SchedSelIdx ] are set equal to the values of the buffering period SEI message syntax elements nal\_initial\_cpb\_removal\_delay[ SchedSelIdx ] and nal\_initial\_cpb\_removal\_offset[ SchedSelIdx ], respectively, when NalHrdModeFlag is equal to 1, or vcl\_initial\_cpb\_removal\_delay[ SchedSelIdx ] and vcl\_initial\_cpb\_removal\_offset[ SchedSelIdx ], respectively, when NalHrdModeFlag is equal to 0, where the buffering period SEI message containing the syntax elements is selected as specified in clause C.1.

The nominal removal time of the access unit n from the CPB is specified as follows:

– If access unit n is the access unit with n equal to 0 (the access unit that initializes the HRD), the nominal removal time of the access unit from the CPB is specified by:

NominalRemovalTime[ 0 ] = InitCpbRemovalDelay[ SchedSelIdx ] ÷ 90000 (C‑8)

– Otherwise, the following applies:

– When access unit n is the first access unit of a buffering period that does not initialize the HRD, the following applies:

The nominal removal time of the access unit n from the CPB is specified by:

if( !concatenationFlag ) {  
 baseTime = NominalRemovalTime[ firstPicInPrevBuffPeriod ]  
 tmpCpbRemovalDelay = CpbRemovalDelayVal  
} else {  
 baseTime = NominalRemovalTime[ prevNonDiscardablePic ]  
 tmpCpbRemovalDelay =  
 Max( ( CpbRemovalDelayDeltaMinus1 + 1 ), (C‑9)  
 Ceil( ( InitCpbRemovalDelay[ SchedSelIdx ] ÷ 90000 +  
 FinalArrivalTime[ n − 1 ] − NominalRemovalTime[ n − 1 ] ) ÷ ClockTick ) )  
}  
NominalRemovalTime[ n ] = baseTime + ClockTick \* tmpCpbRemovalDelay

where NominalRemovalTime[ firstPicInPrevBuffPeriod ] is the nominal removal time of the first access unit of the previous buffering period, NominalRemovalTime[ prevNonDiscardablePic ] is the nominal removal time of the preceding picture in decoding order with TemporalId equal to 0 that is not a RASL or RADL picture, CpbRemovalDelayVal is the value of CpbRemovalDelayVal[ OpTid ] derived according to cpb\_removal\_delay\_minus1[ OpTid ] and cpb\_removal\_delay\_delta\_idx[ OpTid ] in the picture timing SEI message, and cpb\_removal\_delay\_delta[ cpb\_removal\_delay\_delta\_idx[ OpTid ] ] in the buffering period SEI message, selected as specified in clause C.1, associated with access unit n and concatenationFlag and CpbRemovalDelayDeltaMinus1 are the values of the syntax elements concatenation\_flag and cpb\_removal\_delay\_delta\_minus1, respectively, in the buffering period SEI message, selected as specified in clause C.1, associated with access unit n.

– When access unit n is not the first access unit of a buffering period, the nominal removal time of the access unit n from the CPB is specified by:

NominalRemovalTime[ n ] = NominalRemovalTime[ firstPicInCurrBuffPeriod ] +  
 ClockTick \* CpbRemovalDelayVal (C‑10)

where NominalRemovalTime[ firstPicInCurrBuffPeriod ] is the nominal removal time of the first access unit of the current buffering period and CpbRemovalDelayVal is the value of CpbRemovalDelayVal[ OpTid ] derived according to cpb\_removal\_delay\_minus1[ OpTid ] and cpb\_removal\_delay\_delta\_idx[ OpTid ] in the picture timing SEI message, and cpb\_removal\_delay\_delta[ cpb\_removal\_delay\_delta\_idx[ OpTid ] ] in the buffering period SEI message, selected as specified in clause C.1, associated with access unit n.

The removal time of access unit n from the CPB is specified as follows, where FinalArrivalTime[ n ] and NominalRemovalTime[ n ] are the final CPB arrival time and nominal CPB removal time, respectively, of access unit n:

if( !low\_delay\_hrd\_flag[ HighestTid ] | | NominalRemovalTime[ n ] >= FinalArrivalTime[ n ] )  
 CpbRemovalTime[ n ] = NominalRemovalTime[ n ]  
else (C‑11)  
 CpbRemovalTime[ n ] = NominalRemovalTime[ n ] + ClockTick \*  
 Ceil( ( FinalArrivalTime[ n ] − NominalRemovalTime[ n ] ) ÷ ClockTick )

NOTE 1 – When low\_delay\_hrd\_flag[ HighestTid ] is equal to 1 and NominalRemovalTime[ n ] is less than FinalArrivalTime[ n ], the size of access unit n is so large that it prevents removal at the nominal removal time.

At the CPB removal time of access unit n, the access unit is instantaneously decoded.

* 1. Operation of the decoded picture buffer
     1. General

The specifications in this clause apply independently to each set of decoded picture buffer (DPB) parameters selected as specified in clause C.1.

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. The processes specified in clauses C.3.2, C.3.3, and C.3.4 are sequentially applied as specified below.

* + 1. Removal of pictures from the DPB before decoding of the current picture

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 list construction as specified in clause 8.3.2 is invoked and decoding process for reference picture marking as specified in clause 8.3.3 is invoked.

– When the current picture is an IRAP picture with NoIncorrectPicOutputFlag equal to 1 that is not picture 0, the following ordered steps are applied:

1. The variable NoOutputOfPriorPicsFlag is derived for the decoder under test as follows:

– If the current picture is a CRA picture, NoOutputOfPriorPicsFlag is set equal to 1 (regardless of the value of no\_output\_of\_prior\_pics\_flag).

– Otherwise, if the value of pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples, chroma\_format\_idc, separate\_colour\_plane\_flag, bit\_depth\_luma\_minus8, bit\_depth\_chroma\_minus8 or sps\_max\_dec\_pic\_‌buffering\_minus1[ HighestTid ] derived from the active SPS is different from the value of pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples, chroma\_format\_idc, separate\_colour\_plane\_‌flag, bit\_depth\_luma\_minus8, bit\_depth\_chroma\_minus8 or sps\_max\_dec\_pic\_buffering\_‌minus1[ HighestTid ], respectively, derived from the SPS active for the preceding picture, NoOutputOfPriorPicsFlag may (but should not) be set to 1 by the decoder under test, regardless of the value of no\_output\_of\_prior\_pics\_flag.

NOTE – Although setting NoOutputOfPriorPicsFlag equal to no\_output\_of\_prior\_pics\_flag is preferred under these conditions, the decoder under test is allowed to set NoOutputOfPriorPicsFlag to 1 in this case.

– Otherwise, NoOutputOfPriorPicsFlag is set equal to no\_output\_of\_prior\_pics\_flag.

2. The value of NoOutputOfPriorPicsFlag derived for the decoder under test is applied for the HRD, such that when the value of NoOutputOfPriorPicsFlag is equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain, and the DPB fullness is set equal to 0.

– When both of the following conditions are true for any pictures k in the DPB, all such pictures k in the DPB are removed from the DPB:

– picture k is marked as "unused for reference".

– picture k has PictureOutputFlag 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., DpbOutputTime[ k ] is less than or equal to DuCpbRemovalTime[ m ].

– For each picture that is removed from the DPB, the DPB fullness is decremented by one.

* + 1. Picture output

The processes specified in this clause happen instantaneously at the CPB removal time of access unit n, CpbRemovalTime[ n ].

When picture n has PictureOutputFlag equal to 1, its DPB output time DpbOutputTime[ n ] is derived as follows:

DpbOutputTime[ n ] = CpbRemovalTime[ n ] + ClockTick \* picDpbOutputDelay – (C‑12)  
 picDpbOutputDelta[ OpTid ]

where picDpbOutputDelay is the value of pic\_dpb\_output\_delay and picDpbOutputDelta is the value of picDpbOutputDelta[ OpTid ] derived according to cpb\_removal\_delay\_minus1[ OpTid ], and cpb\_removal\_delay\_delta\_idx[ OpTid ] in the picture timing SEI message associated with access unit n, and cpb\_removal\_delay\_delta[ cpb\_removal\_delay\_delta\_idx[ OpTid ] ] in the buffering period SEI message associated with access unit n.

The output of the current picture is specified as follows:

– If PictureOutputFlag is equal to 1 and DpbOutputTime[ n ] is equal to CpbRemovalTime[ n ], the current picture is output.

– Otherwise, if PictureOutputFlag is equal to 0, the current picture is not output, but will be stored in the DPB as specified in clause C.3.4.

– Otherwise (PictureOutputFlag is equal to 1 and DpbOutputTime[ n ] is greater than CpbRemovalTime[ n ] ), the current picture is output later and will be stored in the DPB (as specified in clause C.3.4) and is output at time DpbOutputTime[ 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 DpbOutputTime[ n ].

When output, the picture is cropped, using the conformance cropping window specified in the active SPS for the picture.

When picture n is a picture that is output and is not the last picture of the bitstream that is output, the value of the variable DpbOutputInterval[ n ] is derived as follows:

DpbOutputInterval[ n ] = DpbOutputTime[ nextPicInOutputOrder ] − DpbOutputTime[ n ] (C‑13)

where nextPicInOutputOrder is the picture that follows picture n in output order and has PictureOutputFlag equal to 1.

* + 1. Current decoded picture marking and storage

The current decoded picture is stored in the DPB in an empty picture storage buffer, the DPB fullness is incremented by one, and the current picture is marked as "used for short-term reference".

* 1. Bitstream conformance

A bitstream of coded data conforming to this Specification shall fulfil all requirements specified in this clause.

The bitstream shall be constructed according to the syntax, semantics and constraints specified in this Specification outside of this annex.

The first coded picture in a bitstream shall be an IRAP picture, i.e., an IDR picture or a CRA picture.

The bitstream is tested by the HRD for conformance as specified in clause C.1.

For each current picture, let the variables maxPicOrderCnt and minPicOrderCnt be set equal to the maximum and the minimum, respectively, of the PicOrderCntVal values of the following pictures:

– The current picture.

– The previous picture in decoding order that has NuhLayerId the same as the current picture and TemporalId equal to 0 and that is not a RASL or RADL picture.

– The STRPs referred to by all entries in RefPicList[ 0 ] and all entries in RefPicList[ 1 ] of the current picture.

– All pictures n that have NuhLayerId the same as the current picture, PictureOutputFlag equal to 1, CpbRemovalTime[ n ] less than CpbRemovalTime[ currPic ] and DpbOutputTime[ n ] greater than or equal to CpbRemovalTime[ currPic ], where currPic is the current picture.

All of the following conditions shall be fulfilled for each of the bitstream conformance tests:

1. For each access unit n, with n greater than 0, associated with a buffering period SEI message, let the variable deltaTime90k[ n ] be specified as follows:

deltaTime90k[ n ] = 90000 \* ( NominalRemovalTime[ n ] − FinalArrivalTime[ n − 1 ] ) (C‑14)

The value of InitCpbRemovalDelay[ SchedSelIdx ] is constrained as follows:

– If cbr\_flag[ SchedSelIdx ] is equal to 0, the following condition shall be true:

InitCpbRemovalDelay[ SchedSelIdx ] <= Ceil( deltaTime90k[ n ] ) (C‑15)

– Otherwise (cbr\_flag[ SchedSelIdx ] is equal to 1), the following condition shall be true:

Floor( deltaTime90k[ n ] ) <= InitCpbRemovalDelay[ SchedSelIdx ] <= Ceil( deltaTime90k[ n ] ) (C‑16)

NOTE 1 – 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 initialize 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 initialize the HRD, as the HRD may be initialized 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 greater than the CPB size. The CPB shall never overflow.
2. When low\_delay\_hrd\_flag[ HighestTid ] is equal to 0, the CPB shall never underflow. A CPB underflow is specified as follows:

– A CPB underflow is specified as the condition in which the nominal CPB removal time of access unit n NominalRemovalTime[ n ] is less than the final CPB arrival time of access unit n FinalArrivalTime[ n ] for at least one value of n.

1. The nominal removal times of pictures from the CPB (starting from the second picture in decoding order) shall satisfy the constraints on NominalRemovalTime[ n ] and CpbRemovalTime[ n ] expressed in clauses A.4.1 through A.4.2.
2. For each current picture, after invocation of the process for removal of pictures from the DPB as specified in clause C.3.2, the number of decoded pictures in the DPB, including all pictures n that are marked as "used for reference", or that have PictureOutputFlag equal to 1 and CpbRemovalTime[ n ] less than CpbRemovalTime[ currPic ], where currPic is the current picture, shall be less than or equal to sps\_max\_dec\_pic\_buffering\_minus1[ HighestTid ].
3. All reference pictures shall be present in the DPB when needed for prediction. Each picture that has PictureOutputFlag equal to 1 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 clause C.3.
4. For each current picture that is not a CLVSS picture, the value of maxPicOrderCnt − minPicOrderCnt shall be less than MaxPicOrderCntLsb / 2.
5. The value of DpbOutputInterval[ n ] as given by Equation C‑13, which is the difference between the output time of a picture and that of the first picture following it in output order and having PictureOutputFlag equal to 1, shall satisfy the constraint expressed in clause A.4.1 for the profile, tier and level specified in the bitstream using the decoding process specified in clauses 2 through 10.
6. For any two pictures m and n in the same CVS, when DpbOutputTime[ m ] is greater than DpbOutputTime[ n ], the PicOrderCntVal of picture m shall be greater than the PicOrderCntVal of picture n.

NOTE 2 – All pictures of an earlier CVS in decoding order that are output are output before any pictures of a later CVS in decoding order. Within any particular CVS, the pictures that are output are output in increasing PicOrderCntVal order.

* 1. Decoder conformance
     1. General

A decoder conforming to this Specification shall fulfil all requirements specified in this clause.

A decoder claiming conformance to a specific profile, tier and level shall be able to successfully decode all bitstreams that conform to the bitstream conformance requirements specified in clause C.4, in the manner specified in Annex A, provided that all VPSs, SPSs, PPSs and APSs 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 in this Specification.

When a bitstream contains syntax elements that have values that are specified as reserved and it is specified that decoders shall ignore values of the syntax elements or NAL units containing the syntax elements having the reserved values, and the bitstream is otherwise conforming to this Specification, a conforming decoder shall decode the bitstream in the same manner as it would decode a conforming bitstream and shall ignore the syntax elements or the NAL units containing the syntax elements having the reserved values as specified.

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, tier and level, as specified in clause C.4 are delivered by a hypothetical stream scheduler (HSS) both to the HRD and to the decoder under test (DUT). All cropped decoded pictures output by the HRD shall also be output by the DUT, each cropped decoded picture output by the DUT shall be a picture with PictureOutputFlag equal to 1, and, for each such cropped decoded picture output by the DUT, the values of all samples that are output shall be equal to the values of the samples produced by the specified decoding process.

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, tier 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 the DUT.

When the HRD parameters and the buffering period SEI messages are present with vui\_cpb\_cnt\_minus1[ HighestTid ] 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:

α = ( r − BitRate[ SchedSelIdx − 1 ] ) ÷ ( BitRate[ SchedSelIdx ] − BitRate[ SchedSelIdx − 1 ] ), (C‑17)

c( r ) = α \* CpbSize[ SchedSelIdx ] + ( 1 − α ) \* CpbSize[ SchedSelIdx − 1 ], (C‑18)

f( r ) = α \* InitCpbRemovalDelay[ SchedSelIdx ] \* BitRate[ SchedSelIdx ] +   
 ( 1 − α ) \* 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, tier 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 the HRD and the DUT up to a fixed delay.

For output order decoder conformance, the following applies:

– The HSS delivers the bitstream BitstreamToDecode 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 decoding 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 BitstreamToDecode 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 the HRD and the DUT.

– The HRD CPB size is given by CpbSize[ SchedSelIdx ] as specified in clause E.3.3, where SchedSelIdx and the HRD parameters are selected as specified in clause C.1. The DPB size is given by sps\_max\_dec\_pic\_buffering\_minus1[ HighestTid ] + 1. Removal time from the CPB for the HRD is the final bit arrival time and decoding is immediate. The operation of the DPB of this HRD is as described in clauses C.5.2 through C.5.2.3.

* + 1. Operation of the output order DPB
       1. General

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. The process for output and removal of pictures from the DPB before decoding of the current picture as specified in clause C.5.2.2 is invoked, the invocation of the process for current decoded picture marking and storage as specified in clause C.3.4, and finally followed by the invocation of the process for additional bumping as specified in clause C.5.2.3. The "bumping" process is specified in clause C.5.2.4 and is invoked as specified in clauses C.5.2.2 and C.5.2.3.

* + - 1. Output and removal of pictures from the DPB

The output and removal of pictures from the DPB before the 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 list construction as specified in clause 8.3.2 and decoding process for reference picture marking as specified in clause 8.3.3 is invoked.

– If the current picture is an IRAP picture with NoIncorrectPicOutputFlag equal to 1 that is not picture 0, the following ordered steps are applied:

1. The variable NoOutputOfPriorPicsFlag is derived for the decoder under test as follows:

– If the current picture is a CRA picture, NoOutputOfPriorPicsFlag is set equal to 1 (regardless of the value of no\_output\_of\_prior\_pics\_flag).

– Otherwise, if the value of pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples, chroma\_format\_idc, separate\_colour\_plane\_flag, bit\_depth\_luma\_minus8, bit\_depth\_chroma\_minus8 or sps\_max\_dec\_pic\_‌buffering\_minus1[ HighestTid ] derived from the active SPS is different from the value of pic\_width\_in\_luma\_samples, pic\_height\_in\_luma\_samples, chroma\_format\_idc, separate\_colour\_plane\_‌flag, bit\_depth\_luma\_minus8, bit\_depth\_chroma\_minus8 or sps\_max\_dec\_pic\_buffering\_‌minus1[ HighestTid ], respectively, derived from the SPS active for the preceding picture, NoOutputOfPriorPicsFlag may (but should not) be set to 1 by the decoder under test, regardless of the value of no\_output\_of\_prior\_pics\_flag.

NOTE – Although setting NoOutputOfPriorPicsFlag equal to no\_output\_of\_prior\_pics\_flag is preferred under these conditions, the decoder under test is allowed to set NoOutputOfPriorPicsFlag to 1 in this case.

– Otherwise, NoOutputOfPriorPicsFlag is set equal to no\_output\_of\_prior\_pics\_flag.

2. The value of NoOutputOfPriorPicsFlag derived for the decoder under test is applied for the HRD as follows:

– If NoOutputOfPriorPicsFlag is equal to 1, all picture storage buffers in the DPB are emptied without output of the pictures they contain and the DPB fullness is set equal to 0.

– Otherwise (NoOutputOfPriorPicsFlag is equal to 0), all picture storage buffers containing a picture that is marked as "not needed for output" and "unused for reference" are emptied (without output) and all non-empty picture storage buffers in the DPB are emptied by repeatedly invoking the "bumping" process specified in clause C.5.2.4 and the DPB fullness is set equal to 0.

– Otherwise (the current picture is not an IRAP picture with NoIncorrectPicOutputFlag equal to 1), all picture storage buffers containing a picture which are marked as "not needed for output" and "unused for reference" are emptied (without output). For each picture storage buffer that is emptied, the DPB fullness is decremented by one. When one or more of the following conditions are true, the "bumping" process specified in clause C.5.2.4 is invoked repeatedly while further decrementing the DPB fullness by one for each additional picture storage buffer that is emptied, until none of the following conditions are true:

* The number of pictures in the DPB that are marked as "needed for output" is greater than sps\_max\_num\_reorder\_pics[ HighestTid ].
* sps\_max\_latency\_increase\_plus1[ HighestTid ] is not equal to 0 and there is at least one picture in the DPB that is marked as "needed for output" for which the associated variable PicLatencyCount is greater than or equal to SpsMaxLatencyPictures[ HighestTid ].
* The number of pictures in the DPB is greater than or equal to sps\_max\_dec\_pic\_buffering\_minus1[ HighestTid ] + 1.
  + - 1. Additional bumping

The processes specified in this clause happen instantaneously when access unit n containing the current picture is removed from the CPB.

When the current picture has PictureOutputFlag equal to 1, for each picture in the DPB that is marked as "needed for output" and follows the current picture in output order, the associated variable PicLatencyCount is set equal to PicLatencyCount + 1.

The following applies:

– If the current decoded picture has PictureOutputFlag equal to 1, it is marked as "needed for output" and its associated variable PicLatencyCount is set equal to 0.

– Otherwise (the current decoded picture has PictureOutputFlag equal to 0), it is marked as "not needed for output".

When one or more of the following conditions are true, the "bumping" process specified in clause C.5.2.4 is invoked repeatedly until none of the following conditions are true:

– The number of pictures in the DPB that are marked as "needed for output" is greater than sps\_max\_num\_reorder\_pics[ HighestTid ].

– sps\_max\_latency\_increase\_plus1[ HighestTid ] is not equal to 0 and there is at least one picture in the DPB that is marked as "needed for output" for which the associated variable PicLatencyCount that is greater than or equal to SpsMaxLatencyPictures[ HighestTid ].

* + - 1. "Bumping" process

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 conformance cropping window specified in the active SPS for the picture, the cropped picture is output, and the picture is marked as "not needed for output".
3. When 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.

NOTE – For any two pictures picA and picB that belong to the same CVS and are output by the "bumping process", when picA is output earlier than picB, the value of PicOrderCntVal of picA is less than the value of PicOrderCntVal of picB.

1. Annex D  
     
   HRD related supplemental enhancement information

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

* 1. General

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 Specification (see Annex C for the specification of conformance). SEI messages specified in this Annex are required for checking bitstream conformance and for output timing decoder conformance. SEI messages specified in Annex F are not required to check bitstream conformance and for output timing decoder conformance.

In clause C.5.2, 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 in this Specification. When present in the bitstream, SEI messages shall obey the syntax and semantics specified in clause 7.3.4 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
     1. General SEI message syntax

|  |  |
| --- | --- |
| sei\_payload( payloadType, payloadSize ) { | Descriptor |
| if( NalUnitType = = PREFIX\_SEI\_NUT ) |  |
| if( payloadType = = 0 ) |  |
| buffering\_period( payloadSize ) |  |
| else if( payloadType = = 1 ) |  |
| pic\_timing( payloadSize ) |  |
| else if( payloadType = = 145 ) |  |
| dependent\_rap\_indication( payloadSize ) |  |
| else |  |
| reserved\_sei\_message( payloadSize ) |  |
| else /\* NalUnitType = = SUFFIX\_SEI\_NUT \*/ |  |
| if( payloadType = = 132 ) |  |
| decoded\_picture\_hash( payloadSize ) |  |
| else |  |
| reserved\_sei\_message( payloadSize ) |  |
| if( more\_data\_in\_payload( ) ) { |  |
| if( payload\_extension\_present( ) ) |  |
| **reserved\_payload\_extension\_data** | u(v) |
| **payload\_bit\_equal\_to\_one** /\* equal to 1 \*/ | f(1) |
| while( !byte\_aligned( ) ) |  |
| **payload\_bit\_equal\_to\_zero** /\* equal to 0 \*/ | f(1) |
| } |  |
| } |  |

* + 1. Buffering period SEI message syntax

|  |  |
| --- | --- |
| buffering\_period( payloadSize ) { | **Descriptor** |
| **bp\_seq\_parameter\_set\_id** | ue(v) |
| **bp\_nal\_hrd\_parameters\_present\_flag** | u(1) |
| **bp\_vcl\_hrd\_parameters\_present\_flag** | u(1) |
| if( bp\_nal\_hrd\_parameters\_present\_flag | | bp\_ vcl\_hrd\_parameters\_present\_flag ) { |  |
| **initial\_cpb\_removal\_delay\_length\_minus1** | u(5) |
| **cpb\_removal\_delay\_length\_minus1** | u(5) |
| **dpb\_output\_delay\_length\_minus1** | u(5) |
| } |  |
| **concatenation\_flag** | u(1) |
| **cpb\_removal\_delay\_delta\_minus1** | u(v) |
| **cpb\_removal\_delay\_deltas\_present\_flag** | u(1) |
| if( cpb\_removal\_delay\_deltas\_present\_flag ) { |  |
| **num\_cpb\_removal\_delay\_deltas\_minus1** | ue(v) |
| for( i = 0; i <= num\_cpb\_removal\_delay\_deltas\_minus1; i++ ) |  |
| **cpb\_removal\_delay\_delta**[ i ] | u(v) |
| } |  |
| **bp\_cpb\_cnt\_minus1** | ue(v) |
| if( bp\_nal\_hrd\_parameters\_present\_flag ) |  |
| for( i = 0; i < bp\_cpb\_cnt\_minus1 + 1; i++ ) { |  |
| **nal\_initial\_cpb\_removal\_delay**[ i ] | u(v) |
| **nal\_initial\_cpb\_removal\_offset**[ i ] | u(v) |
| } |  |
| if( bp\_vcl\_hrd\_parameters\_present\_flag ) |  |
| for( i = 0; i < bp\_cpb\_cnt\_minus1 + 1; i++ ) { |  |
| **vcl\_initial\_cpb\_removal\_delay**[ i ] | u(v) |
| **vcl\_initial\_cpb\_removal\_offset**[ i ] | u(v) |
| } |  |
| } |  |

* + 1. Picture timing SEI message syntax

|  |  |
| --- | --- |
| pic\_timing( payloadSize ) { | Descriptor |
| **frame\_field\_info\_present\_flag** | u(1) |
| if( frame\_field\_info\_present\_flag ) { |  |
| **pic\_struct** | u(4) |
| **source\_scan\_type** | u(2) |
| **duplicate\_flag** | u(1) |
| } |  |
| if( CpbDpbDelaysPresentFlag ) { |  |
| **cpb\_removal\_delay\_minus1**[ sps\_max\_sub\_layers\_minus1 ] | u(v) |
| for( i = TemporalId; i < sps\_max\_sub\_layers\_minus1; i++ ) { |  |
| **sub\_layer\_delays\_present\_flag**[ i ] | u(1) |
| if( sub\_layer\_delays\_present\_flag[ i ] ) { |  |
| **cpb\_removal\_delay\_delta\_enabled\_flag**[ i ] | u(1) |
| if( cpb\_removal\_delay\_delta\_enabled\_flag[ i ] ) |  |
| **cpb\_removal\_delay\_delta\_idx**[ i ] | u(v) |
| else |  |
| **cpb\_removal\_delay\_minus1**[ i ] | u(v) |
| } |  |
| } |  |
| **dpb\_output\_delay** | u(v) |
| } |  |
| } |  |

* 1. SEI payload semantics
     1. General SEI payload semantics

**reserved\_payload\_extension\_data** shall not be present in bitstreams conforming to this version of this Specification. However, decoders conforming to this version of this Specification shall ignore the presence and value of reserved\_payload\_extension\_data. When present, the length, in bits, of reserved\_payload\_extension\_data is equal to 8 \* payloadSize − nEarlierBits − nPayloadZeroBits − 1, where nEarlierBits is the number of bits in the sei\_payload( ) syntax structure that precede the reserved\_payload\_extension\_data syntax element, and nPayloadZeroBits is the number of payload\_bit\_equal\_to\_zero syntax elements at the end of the sei\_payload( ) syntax structure.

**payload\_bit\_equal\_to\_one** shall be equal to 1.

**payload\_bit\_equal\_to\_zero** shall be equal to 0.

NOTE 1 – SEI messages with the same value of payloadType are conceptually the same SEI message regardless of whether they are contained in prefix or suffix SEI NAL units.

NOTE 2 – For SEI messages with payloadType in the range of 0 to 47, inclusive, that are specified in this Specification, the payloadType values are aligned with similar SEI messages specified in Rec. ITU-T H.264 | ISO/IEC 14496-10.

The semantics and persistence scope for each SEI message are specified in the semantics specification for each particular SEI message.

NOTE 3 – Persistence information for SEI messages is informatively summarized in Table D.1.

Table D.1 – Persistence scope of SEI messages (informative)

|  |  |
| --- | --- |
| SEI message | Persistence scope |
| Buffering period | The remainder of the bitstream |
| Picture timing | The access unit containing the SEI message |
| Decoded picture hash | The access unit containing the SEI message |
| DRAP indication | The access unit containing the SEI message |

* + 1. Buffering period SEI message semantics

A buffering period SEI message provides initial CPB removal delay and initial CPB removal delay offset information for initialization of the HRD at the position of the associated access unit in decoding order.

When the buffering period SEI message is present, a picture is said to be a notDiscardablePic picture when the picture has TemporalId equal to 0 and is not a RASL or RADL picture.

When the current picture is not the first picture in the bitstream in decoding order, let prevNonDiscardablePic be the preceding picture in decoding order with TemporalId equal to 0 that is not a RASL or RADL picture.

The presence of buffering period SEI messages is specified as follows:

– If NalHrdBpPresentFlag is equal to 1 or VclHrdBpPresentFlag is equal to 1, the following applies for each access unit in the CVS:

* If the access unit is an IRAP access unit, a buffering period SEI message applicable to the operation point shall be associated with the access unit.
* Otherwise, if the access unit contains a notDiscardablePic, a buffering period SEI message applicable to the operation point may or may not be associated with the access unit.
* Otherwise, the access unit shall not be associated with a buffering period SEI message applicable to the operation point.

– Otherwise (NalHrdBpPresentFlag and VclHrdBpPresentFlag are both equal to 0), no access unit in the CVS shall be associated with a buffering period SEI message.

NOTE 1 – For some applications, frequent presence of buffering period SEI messages may be desirable (e.g., for random access at an IRAP picture or a non-IRAP picture or for bitstream splicing).

**bp\_seq\_parameter\_set\_id** indicates and shall be equal to the sps\_seq\_parameter\_set\_id for the SPS that is active for the coded picture associated with the buffering period SEI message. The value of bp\_seq\_parameter\_set\_id shall be equal to the value of pps\_seq\_parameter\_set\_id in the PPS referenced by the slice\_pic\_parameter\_set\_id of the slice headers of the coded picture associated with the buffering period SEI message. The value of bp\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**bp\_nal\_hrd\_parameters\_present\_flag** equal to 1 specifies that a list of syntax element pairs nal\_initial\_cpb\_removal\_delay[ i ] and nal\_initial\_cpb\_removal\_offset[ i ] are present in the buffering period SEI message. bp\_nal\_hrd\_parameters\_present\_flag equal to 0 specifies that no syntax element pairs nal\_initial\_cpb\_removal\_delay[ i ] and nal\_initial\_cpb\_removal\_offset[ i ] are present in the buffering period SEI message.

It is a requirement of bitstream conformance that the value of bp\_nal\_hrd\_parameters\_present\_flag in the buffering period SEI message assocaited with an access unit is equal to the value of vui\_nal\_hrd\_parameters\_present\_flag in the VUI parameters of the active SPS.

**bp\_vcl\_hrd\_parameters\_present\_flag** equal to 1 specifies that a list of syntax element pairs vcl\_initial\_cpb\_removal\_delay[ i ] and vcl\_initial\_cpb\_removal\_offset[ i ] are present in the buffering period SEI message. bp\_vcl\_hrd\_parameters\_present\_flag equal to 0 specifies that no syntax element pairs vcl\_initial\_cpb\_removal\_delay[ i ] and vcl\_initial\_cpb\_removal\_offset[ i ] are present in the buffering period SEI message.

It is a requirement of bitstream conformance that the value of bp\_vcl\_hrd\_parameters\_present\_flag in the buffering period SEI message associated with an access unit is equal to the value of vui\_vcl\_hrd\_parameters\_present\_flag in the VUI parameters of the active SPS.

**initial\_cpb\_removal\_delay\_length\_minus1** plus 1 specifies the length, in bits, of the syntax elements nal\_initial\_cpb\_removal\_delay[ i ], nal\_initial\_cpb\_removal\_offset[ i ], vcl\_initial\_cpb\_removal\_delay[ i ], and vcl\_initial\_cpb\_removal\_offset[ i ] of the buffering period SEI message. When not present, the value of initial\_cpb\_removal\_delay\_length\_minus1 is inferred to be equal to 23.

**cpb\_removal\_delay\_length\_minus1** plus 1 specifies the length, in bits, of the syntax elements cpb\_removal\_delay\_delta\_minus1 and cpb\_removal\_delay\_delta[ i ] in the buffering period SEI message and the syntax element cpb\_removal\_delay\_minus1[ i ] in the picture timing SEI message. When not present, the value of cpb\_removal\_delay\_length\_minus1 is inferred to be equal to 23.

**dpb\_output\_delay\_length\_minus1** plus 1 specifies the length, in bits, of the syntax element dpb\_output\_delay in the picture timing SEI message. When not present, the value of dpb\_output\_delay\_length\_minus1 is inferred to be equal to 23.

**concatenation\_flag** indicates, when the current picture is not the first picture in the bitstream in decoding order, whether the nominal CPB removal time of the current picture is determined relative to the nominal CPB removal time of the preceding picture with a buffering period SEI message or relative to the nominal CPB removal time of the picture prevNonDiscardablePic.

**cpb\_removal\_delay\_delta\_minus1** plus 1, when the current picture is not the first picture in the bitstream in decoding order, specifies a CPB removal delay increment value relative to the nominal CPB removal time of the picture prevNonDiscardablePic. The lenght of this syntax element is cpb\_removal\_delay\_length\_minus1 + 1 bits.

When the current picture contains a buffering period SEI message and concatenation\_flag is equal to 0 and the current picture is not the first picture in the bitstream in decoding order, it is a requirement of bitstream conformance that the following constraint applies:

* If the picture prevNonDiscardablePic is not associated with a buffering period SEI message, the cpb\_removal\_delay\_minus1 of the current picture shall be equal to the cpb\_removal\_delay\_minus1 of prevNonDiscardablePic plus cpb\_removal\_delay\_delta\_minus1 + 1.
* Otherwise, cpb\_removal\_delay\_minus1 shall be equal to cpb\_removal\_delay\_delta\_minus1.

NOTE 2 – When the current picture contains a buffering period SEI message and concatenation\_flag is equal to 1, the cpb\_removal\_delay\_minus1 for the current picture is not used. The above-specified constraint can, under some circumstances, make it possible to splice bitstreams (that use suitably-designed referencing structures) by simply changing the value of concatenation\_flag from 0 to 1 in the buffering period SEI message for an IRAP picture at the splicing point. When concatenation\_flag is equal to 0, the above-specified constraint enables the decoder to check whether the constraint is satisfied as a way to detect the loss of the picture prevNonDiscardablePic.

**cpb\_removal\_delay\_deltas\_present\_flag** equal to 1 specifies that the buffering period SEI message contains CPB removal delay deltas. cpb\_removal\_delay\_deltas\_present\_flag equal to 0 specifies that no CPB removal delay deltas are present in the buffering period SEI message.

**num\_cpb\_removal\_delay\_deltas\_minus1** plus 1 specifies the number of syntax elements cpb\_removal\_delay\_delta[ i ] in the buffering period SEI message. The value of num\_cpb\_removal\_offsets\_minus1shall be in the range of 0 to 15, inclusive.

**cpb\_removal\_delay\_delta**[ i ] specifies the i-th CPB removal delay delta. The lenght of this syntax element is cpb\_removal\_delay\_length\_minus1 + 1 bits.

**bp\_cpb\_cnt\_minus1** plus 1 specifies the number of syntax element pairs nal\_initial\_cpb\_removal\_delay[ i ] and nal\_initial\_cpb\_removal\_offset[ i ] when bp\_nal\_hrd\_parameters\_present\_flag is equal to 1, and the number of syntax element pairs vcl\_initial\_cpb\_removal\_delay[ i ] and vcl\_initial\_cpb\_removal\_offset[ i ] when bp\_vcl\_hrd\_parameters\_present\_flag is equal to 1. The value of bp\_cpb\_cnt\_minus1 shall be in the range of 0 to 31, inclusive.

**nal\_initial\_cpb\_removal\_delay**[ i ] specify the i-th initial CPB removal delay for the NAL HRD in units of a 90 kHz clock. The length of nal\_initial\_cpb\_removal\_delay[ i ] is initial\_cpb\_removal\_delay\_length\_minus1 + 1 bits. The value of nal\_initial\_cpb\_removal\_delay[ i ] shall not be equal to 0 and shall be less than or equal to 90000 \* ( CpbSize[ i ] ÷ BitRate[ i ] ), the time-equivalent of the CPB size in 90 kHz clock units. [Ed. (YK): Add a clarification of CpbSize[ i ] ÷ BitRate[ i ] at the beginning of this clause.]

**nal\_initial\_cpb\_removal\_offset**[ i ] specify the i-th initial CPB removal offset for the NAL HRD in units of a 90 kHz clock. The length of nal\_initial\_cpb\_removal\_offset[ i ] is initial\_cpb\_removal\_delay\_length\_minus1 + 1 bits.

Over the entire CVS, the sum of nal\_initial\_cpb\_removal\_delay[ i ] and nal\_initial\_cpb\_removal\_offset[ i ] shall be constant for each value of i.

**vcl\_initial\_cpb\_removal\_delay**[ i ] specify the i-th initial CPB removal delay for the VCL HRD in units of a 90 kHz clock. The length of vcl\_initial\_cpb\_removal\_delay[ i ] is initial\_cpb\_removal\_delay\_length\_minus1 + 1 bits. The value of vcl\_initial\_cpb\_removal\_delay[ i ] shall not be equal to 0 and shall be less than or equal to 90000 \* ( CpbSize[ i ] ÷ BitRate[ i ] ), the time-equivalent of the CPB size in 90 kHz clock units.

**vcl\_initial\_cpb\_removal\_offset**[ i ] specify the i-th initial CPB removal offset for the VCL HRD in units of a 90 kHz clock. The length of vcl\_initial\_cpb\_removal\_offset[ i ] is initial\_cpb\_removal\_delay\_length\_minus1 + 1 bits.

Over the entire CVS, the sum of vcl\_initial\_cpb\_removal\_delay[ i ] and vcl\_initial\_cpb\_removal\_offset[ i ] shall be constant for each value of i.

* + 1. Picture timing SEI message semantics

The picture timing SEI message provides CPB removal delay and DPB output delay information for the access unit associated with the SEI message.

If nal\_hrd\_parameters\_present\_flag or vcl\_hrd\_parameters\_present\_flag of the buffering period SEI mesage applicable for the current access unit is equal to 1, the variable CpbDpbDelaysPresentFlag is set equal to 1. Otherwise, CpbDpbDelaysPresentFlag is set equal to 0.

The presence of picture timing SEI messages is specified as follows:

– If CpbDpbDelaysPresentFlag is equal to 1, a picture timing SEI message shall be associated with the current access unit.

– Otherwise (CpbDpbDelaysPresentFlag is equal to 0), there may or may not be a picture timing SEI message associated with the current access unit.

NOTE 1 – When CpbDpbDelaysPresentFlag is equal to 0, a picture timing SEI message containing only the frame field information syntax elements pic\_struct, source\_scan\_type, and duplicate\_flag may be present.

**frame\_field\_info\_present\_flag** equal to 1 specifies that the syntax elements pic\_struct, source\_scan\_type, and duplicate\_flag are present in the picture timing SEI message. frame\_field\_info\_present\_flag equal to 0 specifies that the syntax elements pic\_struct, source\_scan\_type, and duplicate\_flag are not present in the picture timing SEI message.

**pic\_struct** indicates whether a picture should be displayed as a frame or as one or more fields and, for the display of frames when fixed\_pic\_rate\_within\_cvs\_flag is equal to 1, may indicate a frame doubling or tripling repetition period for displays that use a fixed frame refresh interval equal to DpbOutputElementalInterval[ n ] as given by Equation E‑65. The interpretation of pic\_struct is specified in Table D.2. Values of pic\_struct that are not listed in Table D.2 are reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders shall ignore reserved values of pic\_struct.

When present, it is a requirement of bitstream conformance that the value of pic\_struct shall be constrained such that exactly one of the following conditions is true:

– The value of pic\_struct is equal to 0, 7 or 8 for all pictures in the CVS.

– The value of pic\_struct is equal to 1, 2, 9, 10, 11 or 12 for all pictures in the CVS.

– The value of pic\_struct is equal to 3, 4, 5 or 6 for all pictures in the CVS.

When fixed\_pic\_rate\_within\_cvs\_flag is equal to 1, frame doubling is indicated by pic\_struct equal to 7, which indicates that the frame should be displayed two times consecutively on displays with a frame refresh interval equal to DpbOutputElementalInterval[ n ] as given by Equation E‑65, and frame tripling is indicated by pic\_struct equal to 8, which indicates that the frame should be displayed three times consecutively on displays with a frame refresh interval equal to DpbOutputElementalInterval[ n ] as given by Equation E‑65.

NOTE 2 – Frame doubling can be used to facilitate the display, for example, of 25 Hz progressive-scan video on a 50 Hz progressive-scan display or 30 Hz progressive-scan video on a 60 Hz progressive-scan display. Using frame doubling and frame tripling in alternating combination on every other frame can be used to facilitate the display of 24 Hz progressive-scan video on a 60 Hz progressive-scan display.

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 Figure D.1, Figure D.2 and Figure D.3, respectively.

Association indicators for fields (pic\_struct equal to 9 through 12) provide hints to associate fields of complementary parity together as frames. The parity of a field can be top or bottom, and the parity of two fields is considered complementary when the parity of one field is top and the parity of the other field is bottom.

When frame\_field\_info\_present\_flag is equal to 1, it is a requirement of bitstream conformance that the constraints specified in the third column of Table D.2 shall apply.

NOTE 3 – When frame\_field\_info\_present\_flag is equal to 0, then in many cases default values may be inferred or indicated by other means. In the absence of other indications of the intended display type of a picture, the decoder should infer the value of pic\_struct as equal to 0 when frame\_field\_info\_present\_flag is equal to 0.

**source\_scan\_type** equal to 1 indicates that the source scan type of the associated picture should be interpreted as progressive. source\_scan\_type equal to 0 indicates that the source scan type of the associated picture should be interpreted as interlaced. source\_scan\_type equal to 2 indicates that the source scan type of the associated picture is unknown or unspecified. source\_scan\_type equal to 3 is reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders conforming to this version of this Specification shall interpret the value 3 for source\_scan\_type as equivalent to the value 2.

The following applies to the semantics of source\_scan\_type:

– If general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 1, the value of source\_scan\_type shall be equal to 0 when present, and should be inferred to be equal to 0 when not present.

– Otherwise, if general\_progressive\_source\_flag is equal to 1 and general\_interlaced\_source\_flag is equal to 0, the value of source\_scan\_type shall be equal to 1 when present and should be inferred to be equal to 1 when not present.

– Otherwise, when general\_progressive\_source\_flag is equal to 0 and general\_interlaced\_source\_flag is equal to 0, the value of source\_scan\_type shall be equal to 2 when present and should be inferred to be equal to 2 when not present.

**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 of a previous picture in output order.

NOTE 4 – The duplicate\_flag should be used to mark coded pictures known to have originated from a repetition process such as 3:2 pull-down or other such duplication and picture rate interpolation methods. This flag would commonly be used when a video feed is encoded as a field sequence in a "transport pass-through" fashion, with known duplicate pictures tagged by setting duplicate\_flag equal to 1.

NOTE 5 – When field\_seq\_flag is equal to 1 and duplicate\_flag is equal to 1, this should be interpreted as an indication that the access unit contains a duplicated field of the previous field in output order with the same parity as the current field unless a pairing is otherwise indicated by the use of a pic\_struct value in the range of 9 to 12, inclusive.

Table D.2 – Interpretation of pic\_struct

|  |  |  |
| --- | --- | --- |
| **Value** | **Indicated display of picture** | **Restrictions** |
| 0 | (progressive) Frame | field\_seq\_flag shall be equal to 0 |
| 1 | Top field | field\_seq\_flag shall be equal to 1 |
| 2 | Bottom field | field\_seq\_flag shall be equal to 1 |
| 3 | Top field, bottom field, in that order | field\_seq\_flag shall be equal to 0 |
| 4 | Bottom field, top field, in that order | field\_seq\_flag shall be equal to 0 |
| 5 | Top field, bottom field, top field repeated, in that order | field\_seq\_flag shall be equal to 0 |
| 6 | Bottom field, top field, bottom field repeated, in that order | field\_seq\_flag shall be equal to 0 |
| 7 | Frame doubling | field\_seq\_flag shall be equal to 0 fixed\_pic\_rate\_within\_cvs\_flag shall be equal to 1 |
| 8 | Frame tripling | field\_seq\_flag shall be equal to 0 fixed\_pic\_rate\_within\_cvs\_flag shall be equal to 1 |
| 9 | Top field paired with previous bottom field in output order | field\_seq\_flag shall be equal to 1 |
| 10 | Bottom field paired with previous top field in output order | field\_seq\_flag shall be equal to 1 |
| 11 | Top field paired with next bottom field in output order | field\_seq\_flag shall be equal to 1 |
| 12 | Bottom field paired with next top field in output order | field\_seq\_flag shall be equal to 1 |

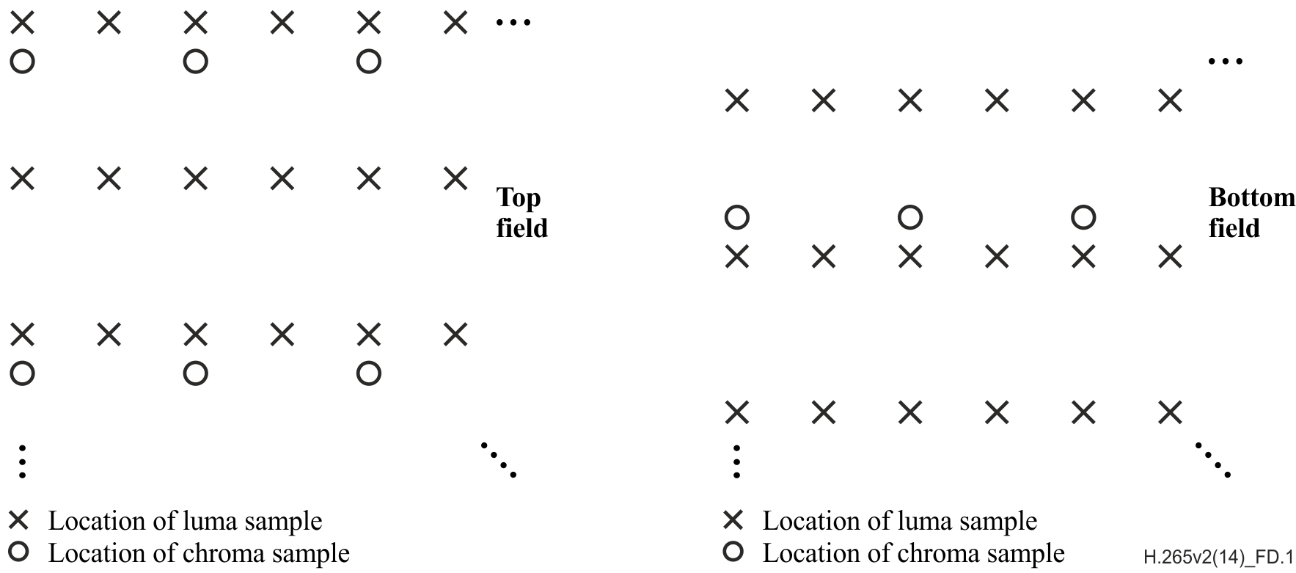


Figure D.1 – Nominal vertical and horizontal sampling locations of 4:2:0 samples in top and bottom fields

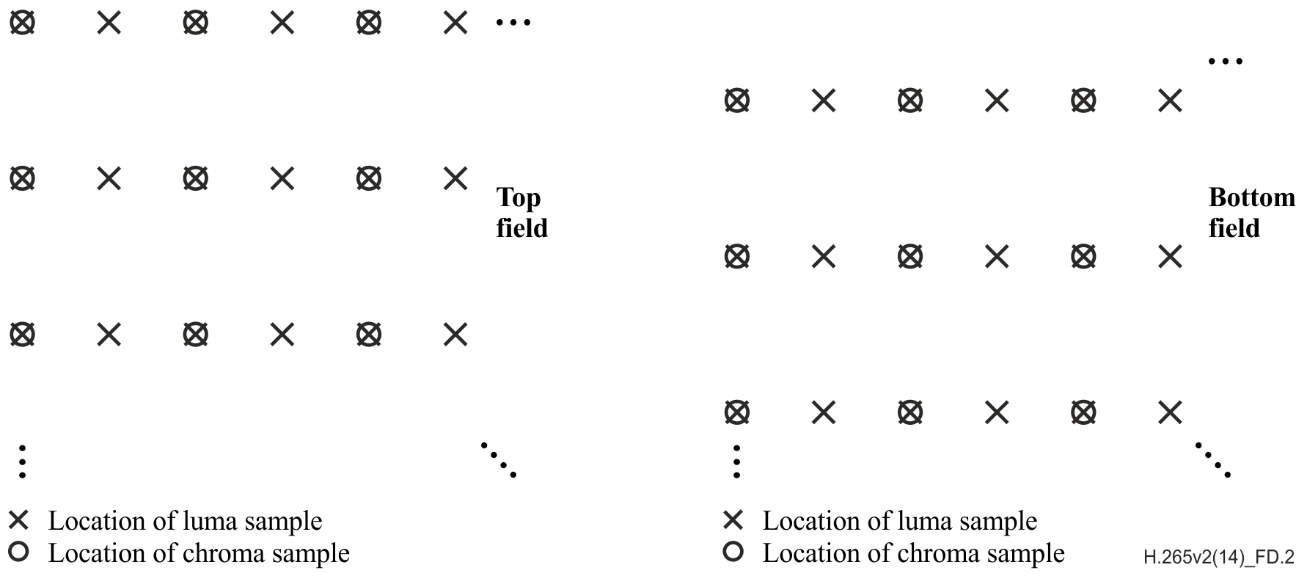


Figure D.2 – Nominal vertical and horizontal sampling locations of 4:2:2 samples in top and bottom fields

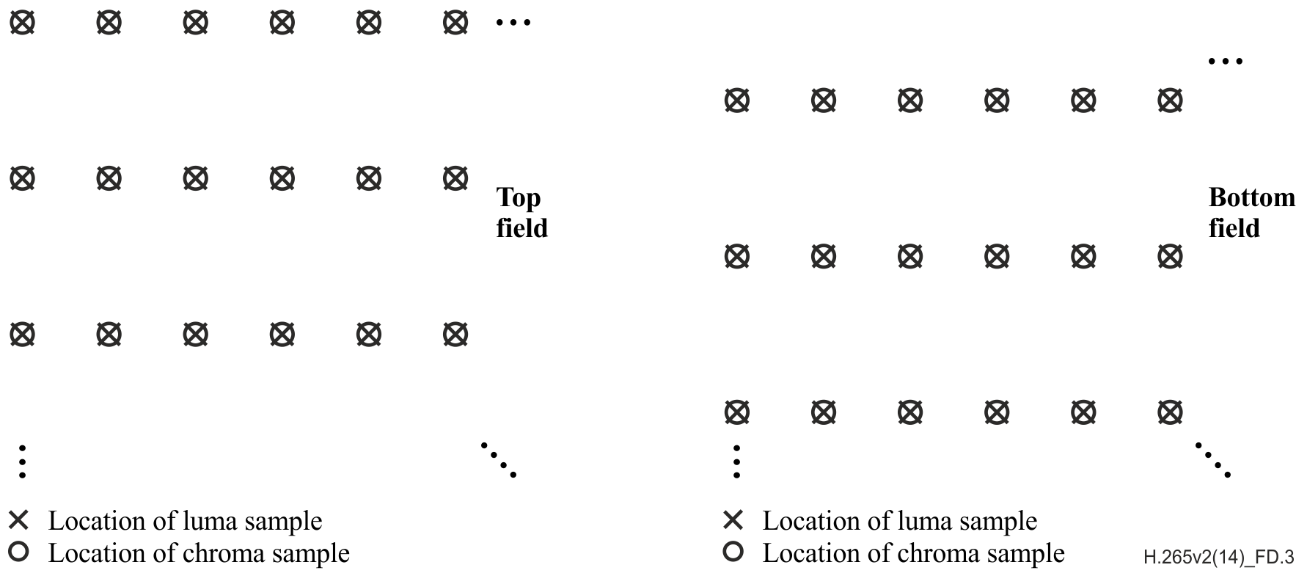


Figure D.3 – Nominal vertical and horizontal sampling locations of 4:4:4 samples in top and bottom fields

[Ed. (YK): Update the above figures by removing "H.265v2(14)\_FD.x."]

**cpb\_removal\_delay\_minus1**[ i ] plus 1 is used to calculate the number of clock ticks between the nominal CPB removal times of the access unit associated with the picture timing SEI message and the preceding access unit in decoding order that contains a buffering period SEI message when HighestTid is equal to i. This value is also used to calculate an earliest possible time of arrival of access unit data into the CPB for the HSS. The length of cpb\_removal\_delay\_minus1[ i ] is cpb\_removal\_delay\_length\_minus1 + 1 bits.

The variable BpResetFlag of the current picture is derived as follows:

* If the current picture is associated with a buffering period SEI message, BpResetFlag is set equal to 1.
* Otherwise, BpResetFlag is set equal to 0.

**sub\_layer\_delays\_present\_flag**[ i ] equal to 1 specifies that cpb\_removal\_delay\_delta\_idx[ i ] or cpb\_removal\_delay\_minus1[ i ] is present for the the sub-layer with TemporalId equal to i. sub\_layer\_delays\_present\_flag[ i ] equal to 0 specifies that neither cpb\_removal\_delay\_delta\_idx[ i ] nor cpb\_removal\_delay\_minus1[ i ] is present for the sub-layer with TemporalId equal to i. When not present, the value of sub\_layer\_delays\_present\_flag[ i ] is infered to be equal to 0.

**cpb\_removal\_delay\_delta\_enabled**[ i ] equal to 1 specifies that cpb\_removal\_delay\_delta\_idx[ i ] is present in the picture timing SEI message. cpb\_removal\_delay\_delta\_enabled[ i ] equal to 0 specifies that cpb\_removal\_delay\_delta\_idx[ i ] is not present in the picture timing SEI message. When not present, the value of cpb\_removal\_delay\_delta\_enabled[ i ] is infered to be equal to 0.

**cpb\_removal\_delay\_delta\_idx**[ i ] specifies the index of the CPB removal delta that applies to HighestTid equal to i in the list of cpb\_removal\_delay\_delta[ j ] for j ranging from 0 to num\_cpb\_removal\_delay\_deltas\_minus1, inclusive.

The variables CpbRemovalDelayMsb[ i ] and CpbRemovalDelayVal[ i ] of the current picture are derived as follows:

* If the current access unit is the access unit that initializes the HRD, CpbRemovalDelayMsb[ i ] and CpbRemovalDelayVal[ i ] are both set equal to 0, and the value of cpbRemovalDelayValTmp[ i ] is set equal to cpb\_removal\_delay\_minus1[ i ] + 1.
* Otherwise, let the picture prevNonDiscardablePic be the previous picture in decoding order that has TemporalId equal to 0 that is not a RASL or RADL, let prevCpbRemovalDelayMinus1[ i ], prevCpbRemovalDelayMsb[ i ], and prevBpResetFlag be set equal to the values of cpbRemovalDelayValTmp[ i ] − 1, CpbRemovalDelayMsb[ i ], and BpResetFlag, respectively, for the picture prevNonDiscardablePic, and the following applies:
* CpbRemovalDelayMsb[ i ] is derived as follows:

cpbRemovalDelayValTmp[ i ] = cpb\_removal\_delay\_delta\_enabled\_flag[ i ] ?  
 cpb\_removal\_delay\_minus1[ sps\_max\_sub\_layers\_minus1 ] + 1 +  
 cpb\_removal\_delay\_delta[ cpb\_removal\_delay\_delta\_idx[ i ] ] : cpb\_removal\_delay\_minus1[ i ] + 1  
if( prevBpResetFlag )  
 CpbRemovalDelayMsb[ i ] = 0  
else if( cpbRemovalDelayValTmp[ i ] < prevCpbRemovalDelayMinus1[ i ] )  
 CpbRemovalDelayMsb[ i ] = prevCpbRemovalDelayMsb[ i ] + 2cpb\_removal\_delay\_length\_minus1 + 1 (D‑1)  
else  
 CpbRemovalDelayMsb[ i ] = prevCpbRemovalDelayMsb[ i ]

* CpbRemovalDelayVal is derived as follows:

CpbRemovalDelayVal[ i ] = CpbRemovalDelayMsb[ i ] + cpbRemovalDelayValTmp[ i ] (D‑2)

The value of CpbRemovalDelayVal[ i ] shall be in the range of 1 to 232, inclusive.

The variable picDpbOutputDelta[ i ] is derived as follows:

* If sub\_layer\_delays\_present\_flag[ i ] is equal to 0, picDpbOutputDelta[ i ] is set equal to 0.
* Otherwise (sub\_layer\_delays\_present\_flag[ i ] is equal to 1), picDpbOutputDelta[ i ] is set equal to CpbRemovalDelayVal[ i ] – ( cpb\_removal\_delay\_minus1[ sps\_max\_sub\_layers\_minus1 ] + 1 ).

**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 an access unit from the CPB before the decoded picture is output from the DPB.

NOTE 6 – 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".

The length of dpb\_output\_delay is dpb\_output\_delay\_length\_minus1 + 1 bits. When sps\_max\_dec\_pic\_buffering\_minus1 is equal to 0, the value of pic\_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 CVS 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 IRAP picture with NoIncorrectPicOutputFlag equal to 1 that has 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 CVS.

1. Annex E  
     
   Video usability information

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

* 1. General

This annex specifies syntax and semantics of the VUI parameters of the SPSs.

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 Specification (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 this 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 in this Specification. When present in the bitstream, VUI parameters shall follow the syntax and semantics specified in 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) |
| } |  |
| } |  |
| **colour\_description\_present\_flag** | u(1) |
| if( colour\_description\_present\_flag ) { |  |
| **colour\_primaries** | u(8) |
| **transfer\_characteristics** | u(8) |
| **matrix\_coeffs** | u(8) |
| } |  |
| **field\_seq\_flag** | u(1) |
| **chroma\_loc\_info\_present\_flag** | u(1) |
| if( chroma\_loc\_info\_present\_flag ) { |  |
| if( !field\_seq\_flag ) { |  |
| **chroma\_sample\_loc\_type** [Ed. (YK): Syntax element name subset issue.] | ue(v) |
| else { |  |
| **chroma\_sample\_loc\_type\_top\_field** | ue(v) |
| **chroma\_sample\_loc\_type\_bottom\_field** | ue(v) |
| } |  |
| } |  |
| **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\_full\_range\_flag** | u(1) |
| } |  |

* + 1. HRD parameters syntax

|  |  |
| --- | --- |
| hrd\_parameters( maxNumSubLayersMinus1 ) { | Descriptor |
| **vui\_nal\_hrd\_parameters\_present\_flag** | u(1) |
| **vui\_vcl\_hrd\_parameters\_present\_flag** | u(1) |
| if( vui\_nal\_hrd\_parameters\_present\_flag | | vui\_vcl\_hrd\_parameters\_present\_flag ){ |  |
| **bit\_rate\_scale** | u(4) |
| **cpb\_size\_scale** | u(4) |
| } |  |
| for( i = 0; i <= maxNumSubLayersMinus1; i++ ) { |  |
| **fixed\_pic\_rate\_general\_flag**[ i ] | u(1) |
| if( !fixed\_pic\_rate\_general\_flag[ i ] ) |  |
| **fixed\_pic\_rate\_within\_cvs\_flag**[ i ] | u(1) |
| if( fixed\_pic\_rate\_within\_cvs\_flag[ i ] ) |  |
| **elemental\_duration\_in\_tc\_minus1**[ i ] | ue(v) |
| else |  |
| **low\_delay\_hrd\_flag**[ i ] | u(1) |
| if( !low\_delay\_hrd\_flag[ i ] ) |  |
| **vui\_cpb\_cnt\_minus1**[ i ] | ue(v) |
| if( vui\_nal\_hrd\_parameters\_present\_flag ) |  |
| sub\_layer\_hrd\_parameters( i ) |  |
| if( vui\_vcl\_hrd\_parameters\_present\_flag ) |  |
| sub\_layer\_hrd\_parameters( i ) |  |
| } |  |
| } |  |

* + 1. Sub-layer HRD parameters syntax

|  |  |
| --- | --- |
| sub\_layer\_hrd\_parameters( subLayerId ) { | Descriptor |
| for( i = 0; i <= CpbCnt; i++ ) { |  |
| **bit\_rate\_value\_minus1[** i ] | ue(v) |
| **cpb\_size\_value\_minus1[** i ] | ue(v) |
| **cbr\_flag[** i ] | u(1) |
| } |  |
| } |  |

* 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. Table E.1 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, the value of aspect\_ratio\_idc is inferred to be equal to 0. Values of aspect\_ratio\_idc in the range of 17 to 254, inclusive, are reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders shall interpret values of aspect\_ratio\_idc in the range of 17 to 254, inclusive, as equivalent to the value 0.

Table E.1 – Interpretation of sample aspect ratio indicator

|  |  |  |
| --- | --- | --- |
| **aspect\_ratio\_idc** | **Sample aspect ratio** | **Examples of use (informative)** |
| 0 | Unspecified |  |
| 1 | 1:1 ("square") | 7680x4320 16:9 frame without horizontal overscan 3840x2160 16:9 frame without horizontal overscan 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 Table E.1, 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 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.

NOTE 2 – For the examples in Table E.1, the frame spatial resolutions shown as examples of use would be the dimensions of the conformance cropping window when field\_seq\_flag is equal to 0 and would have twice the height of the dimensions of the conformance cropping window when field\_seq\_flag is equal to 1.

**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 is unspecified in this Specification.

**colour\_description\_present\_flag** equal to 1 specifies that colour\_primaries, transfer\_characteristics, and matrix\_coeffs are present. colour\_description\_present\_flag equal to 0 specifies that colour\_primaries, transfer\_characteristics, and matrix\_coeffs are not present.

**colour\_primaries** indicates the chromaticity coordinates of the source primaries as specified in Table E.2 in terms of the CIE 1931 definition of x and y as specified in ISO 11664-1.

When the colour\_primaries syntax element is not present, the value of colour\_primaries is inferred to be equal to 2 (the chromaticity is unspecified or is determined by the application). Values of colour\_primaries that are identified as reserved in Table E.2 are reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders shall interpret reserved values of colour\_primaries as equivalent to the value 2.

| Table E.2 – Colour primaries interpretation using the colour\_primaries syntax element | | |
| --- | --- | --- |
| Value | Primaries | Informative remark |
| 0 | Reserved | For future use by ITU-T | ISO/IEC |
| 1 | primary x y  green 0.300 0.600  blue 0.150 0.060  red 0.640 0.330  white D65 0.3127 0.3290 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 conventional colour gamut system and extended colour gamut system (historical)  IEC 61966-2-1 sRGB or sYCC  IEC 61966-2-4  SMPTE 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)  NTSC Recommendation for transmission standards for colour television (1953)  FCC 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.601-7 625  Rec. ITU-R BT.1358-0 625 (historical)  Rec. ITU-R BT.1700-0 625 PAL and 625 SECAM |
| 6 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | Rec. ITU-R BT.601-7 525  Rec. ITU-R BT.1358-1 525 or 625 (historical)  Rec. ITU-R BT.1700-0 NTSC  SMPTE ST 170 (2004)  (functionally the same as the value 7) |
| 7 | primary x y  green 0.310 0.595  blue 0.155 0.070  red 0.630 0.340  white D65 0.3127 0.3290 | SMPTE ST 240 (1999, historical)  (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 | primary x y  green 0.170 0.797  blue 0.131 0.046  red 0.708 0.292  white D65 0.3127 0.3290 | Rec. ITU-R BT.2020-2  Rec. ITU-R BT.2100-1 |
| 10 | primary x y  green (Y) 0.0 1.0  blue (Z) 0.0 0.0  red (X) 1.0 0.0  centre white 1 ÷ 3 1 ÷ 3 | SMPTE ST 428-1 (2006)  (CIE 1931 XYZ) |
| 11 | primary x y  green 0.265 0.690  blue 0.150 0.060  red 0.680 0.320  white 0.314 0.351 | SMPTE RP 431-2 (2011) |
| 12 | primary x y  green 0.265 0.690  blue 0.150 0.060  red 0.680 0.320  white D65 0.3127 0.3290 | SMPTE EG 432-1 (2010) |
| 13..21 | Reserved | For future use by ITU-T | ISO/IEC |
| 22 | primary x y  green 0.295 0.605  blue 0.155 0.077  red 0.630 0.340  white D65 0.3127 0.3290 | EBU Tech. 3213-E (1975) |
| 23..255 | Reserved | For future use by ITU-T | ISO/IEC |

**transfer\_characteristics**, as specified in Table E.3, either indicates the reference opto-electronic transfer characteristic function of the source picture as a function of a source input linear optical intensity Lc with a nominal real-valued range of 0 to 1 or indicates the inverse of the reference electro-optical transfer characteristic function as a function of an output linear optical intensity Lo with a nominal real-valued range of 0 to 1. For interpretation of entries in Table E.3 that are expressed in terms of multiple curve segments parameterized by the variable *α* over a region bounded by the variable *β* or by the variables *β* and *γ*, the values of *α* and *β* are defined to be the positive constants necessary for the curve segments that meet at the value *β* to have continuity of value and continuity of slope at the value *β*, and the value of *γ*, when applicable, is defined to be the positive constant necessary for the associated curve segments to meet at the value *γ*. For example, for transfer\_characteristics equal to 1, 6, 11, 14, or 15, *α* has the value 1 + 5.5 \* *β* = 1.099 296 826 809 442... and *β* has the value 0.018 053 968 510 807....

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). Values of transfer\_characteristics that are identified as reserved in Table E.3 are reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders shall interpret reserved values of transfer\_characteristics as equivalent to the value 2.

NOTE 3 – As indicated in Table E.3, some values of transfer\_characteristics are defined in terms of a reference opto-electronic transfer characteristic function and others are defined in terms of a reference electro-optical transfer characteristic function, according to the convention that has been applied in other Specifications. In the cases of Rec. ITU-R BT.709-6 and Rec. ITU-R BT.2020-2 (which may be indicated by transfer\_characteristics equal to 1, 6, 14, or 15), although the value is defined in terms of a reference opto-electronic transfer characteristic function, a suggested corresponding reference electro-optical transfer characteristic function for flat panel displays used in HDTV studio production has been specified in Rec. ITU-R BT.1886-0.

| Table E.3 – Transfer characteristics interpretation using the transfer\_characteristics syntax element | | |
| --- | --- | --- |
| Value | Transfer characteristic | Informative remark |
| 0 | Reserved | For future use by ITU-T | ISO/IEC |
| 1 | V = *α* \* Lc0.45 − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 4.500 \* Lc for *β* > Lc >= 0 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 conventional colour gamut system (historical)  (functionally the same as the values 6, 14, and 15) |
| 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)  NTSC Recommendation for transmission standards for colour television (1953)  FCC, Title 47 Code of Federal Regulations (2003) 73.682 (a) (20) |
| 5 | Assumed display gamma 2.8 | Rec. ITU-R BT.470-6 System B, G (historical)  Rec. ITU-R BT.1700-0 625 PAL and 625 SECAM |
| 6 | V = *α* \* Lc0.45 − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 4.500 \* Lc for *β* > Lc >= 0 | Rec. ITU-R BT.601-7 525 or 625  Rec. ITU-R BT.1358-1 525 or 625 (historical)  Rec. ITU-R BT.1700-0 NTSC  SMPTE ST 170 (2004)  (functionally the same as the values 1, 14, and 15) |
| 7 | V = *α* \* Lc0.45 − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 4.0 \* Lc for *β* > Lc >= 0 | SMPTE ST 240 (1999, historical) |
| 8 | V = Lc for all values of Lc | 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 = *α* \* Lc0.45 − ( *α* − 1 ) for Lc >= *β*  V = 4.500 \* Lc for *β* > Lc > −*β*  V = −*α* \* ( −Lc )0.45 + ( *α* − 1 ) for −*β* >= Lc | IEC 61966-2-4 |
| 12 | V = *α* \* Lc0.45 − ( *α* − 1 ) for 1.33 > Lc >= *β*  V = 4.500 \* Lc for *β* > Lc >= −*γ*  V = −( *α* \* ( −4 \* Lc )0.45 − ( *α* − 1 ) ) ÷ 4 for −*γ* > Lc >= −0.25 | Rec. ITU-R BT.1361-0 extended colour gamut system (historical) |
| 13 | V = *α* \* Lc( 1 ÷ 2.4 ) − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 12.92 \* Lc for *β* > Lc >= 0 | IEC 61966-2-1 sRGB or sYCC |
| 14 | V =*α* \* Lc0.45 − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 4.500 \* Lc for *β* > Lc >= 0 | Rec. ITU-R BT.2020-2 (functionally the same as the values 1, 6, and 15) |
| 15 | V =*α* \* Lc0.45 − ( *α* − 1 ) for 1 >= Lc >= *β*  V = 4.500 \* Lc for *β* > Lc >= 0 | Rec. ITU-R BT.2020-2 (functionally the same as the values 1, 6, and 14) |
| 16 | V = ( ( c1 + c2 \* Lon ) ÷ ( 1 + c3 \* Lon ) )m for all values of Lo  c1 = c3 − c2 + 1 = 3424 ÷ 4096 = 0.8359375  c2 = 32 \* 2413 ÷ 4096 = 18.8515625  c3 = 32 \* 2392 ÷ 4096 = 18.6875  m = 128 \* 2523 ÷ 4096 = 78.84375  n = 0.25 \* 2610 ÷ 4096 = 0.1593017578125  for which Lo equal to 1 for peak white is ordinarily intended to correspond to a reference output luminance level of 10 000 candelas per square metre | SMPTE ST 2084 (2014) for 10, 12, 14, and 16-bit systems  Rec. ITU-R BT.2100-1 perceptual quantization (PQ) system |
| 17 | V = ( 48 \* Lo ÷ 52.37 )( 1 ÷ 2.6 ) for all values of Lo  for which Lo equal to 1 for peak white is ordinarily intended to correspond to a reference output luminance level of 48 candelas per square metre | SMPTE ST 428-1 (2006) |
| 18 | V = a \* Ln( 12 \* Lc − b ) + c for 1 >= Lc > 1 ÷ 12  V = Sqrt( 3 ) \* Lc0.5 for 1 ÷ 12 >= Lc >= 0  a = 0.17883277, b = 0.28466892, c = 0.55991073 | Association of Radio Industries and Businesses (ARIB) STD-B67  Rec. ITU-R BT.2100-1 hybrid log-gamma (HLG) system |
| 19..255 | Reserved | For future use by ITU-T | ISO/IEC |

NOTE 4 – For transfer\_characteristics equal to 18, the equations given in Table E.3 are normalized for a source input linear optical intensity Lc with a nominal real-valued range of 0 to 1. An alternative scaling that is mathematically equivalent is used in ARIB STD-B67 with the source input linear optical intensity having a nominal real-valued range of 0 to 12.

**matrix\_coeffs** describes the matrix coefficients used in deriving luma and chroma signals from the green, blue, and red, or Y, Z, and X primaries, as specified in Table E.4.

matrix\_coeffs 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 (the 4:4:4 chroma format).

The specification of the use of matrix\_coeffs equal to 0 under all other conditions is reserved for future use by ITU‑T | ISO/IEC.

matrix\_coeffs shall not be equal to 8 unless one of the following conditions is true:

– BitDepthC is equal to BitDepthY,

– BitDepthC is equal to BitDepthY + 1 and chroma\_format\_idc is equal to 3 (the 4:4:4 chroma format).

The specification of the use of matrix\_coeffs equal to 8 under all other conditions is reserved for future use by ITU‑T | ISO/IEC.

When the matrix\_coeffs syntax element is not present, the value of matrix\_coeffs is inferred to be equal to 2 (unspecified).

The interpretation of matrix\_coeffs, together with colour\_primaries and transfer\_characteristics, is specified by the equations below.

NOTE 5 – For purposes of YZX representation when matrix\_coeffs is equal to 0, the symbols R, G, and B are substituted for X, Y, and Z, respectively, in the following descriptions of Equations E‑1 to E‑3, E‑13 to E‑15, E‑19 to E‑21, and E‑31 to E‑33.

ER, EG, and EB are defined as "linear-domain" real-valued signals based on the indicated colour primaries before application of the transfer characteristics function.

Nominal peak white is specified as having ER equal to 1, EG equal to 1, and EB equal to 1.

Nominal black is specified as having ER equal to 0, EG equal to 0, and EB equal to 0.

The application of the transfer characteristics function is denoted by ( x )′ for an argument x.

– If matrix\_coeffs is not equal to 14, the signals E′R, E′G, and E′B are determined by application of the transfer characteristics function as follows:

E′R = ( ER )′ (E‑1)

E′G = ( EG )′ (E‑2)

E′B = ( EB )′ (E‑3)

In this case, 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, inclusive.

– Otherwise (transfer\_characteristics is equal to 11 or 12), E′R, E′G and E′B are real numbers with a larger range not specified in this Specification.

– Otherwise (matrix\_coeffs is equal to 14), the "linear-domain" real-valued signals EL, EM, and ES are determined as follows:

EL = ( 1688 \* ER + 2146 \* EG + 262 \* EB ) ÷ 4096 (E‑4)

EM = ( 683 \* ER + 2951 \* EG + 462 \* EB ) ÷ 4096 (E‑5)

ES = ( 99 \* ER + 309 \* EG + 3688 \* EB ) ÷ 4096 (E‑6)

In this case, the signals E′L, E′M, and E′S are determined by application of the transfer characteristics function as follows:

E′L = ( EL )′ (E‑7)

E′M = ( EM )′ (E‑8)

E′S = ( ES )′ (E‑9)

The interpretation of matrix\_coeffs is specified as follows:

– If video\_full\_range\_flag is equal to 0, the following applies:

– If matrix\_coeffs is equal to 1, 4, 5, 6, 7, 9, 10, 11, 12, 13, or 14, the following equations apply:

Y = Clip1Y( Round( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′Y + 16 ) ) ) (E‑10)

Cb = Clip1C( Round( ( 1 << ( BitDepthC − 8 ) ) \* ( 224 \* E′PB + 128 ) ) ) (E‑11)

Cr = Clip1C( Round( ( 1 << ( BitDepthC − 8 ) ) \* ( 224 \* E′PR + 128 ) ) ) (E‑12)

– Otherwise, if matrix\_coeffs is equal to 0 or 8, the following equations apply:

R = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′R + 16 ) ) (E‑13)

G = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′G + 16 ) ) (E‑14)

B = Clip1Y( ( 1 << ( BitDepthY − 8 ) ) \* ( 219 \* E′B + 16 ) ) (E‑15)

– Otherwise, if matrix\_coeffs is equal to 2, the interpretation of the matrix\_coeffs syntax element is unknown or is determined by the application.

– Otherwise (matrix\_coeffs is not equal to 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, or 14), the interpretation of the matrix\_coeffs 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\_coeffs is equal to 1, 4, 5, 6, 7, 9, 10, 11, 12, 13, or 14, the following applies:

Y = Clip1Y( Round( ( ( 1 << BitDepthY ) − 1 ) \* E′Y ) ) (E‑16)

Cb = Clip1C( Round( ( ( 1 << BitDepthC ) − 1 ) \* E′PB + ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑17)

Cr = Clip1C( Round( ( ( 1 << BitDepthC ) − 1 ) \* E′PR + ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑18)

– Otherwise, if matrix\_coeffs is equal to 0 or 8, the following applies:

R = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′R ) (E‑19)

G = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′G ) (E‑20)

B = Clip1Y( ( ( 1 << BitDepthY ) − 1 ) \* E′B ) (E‑21)

– Otherwise, if matrix\_coeffs is equal to 2, the interpretation of the matrix\_coeffs syntax element is unknown or is determined by the application.

– Otherwise (matrix\_coeffs is not equal to 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, or 14), the interpretation of the matrix\_coeffs syntax element is reserved for future definition by ITU‑T | ISO/IEC. Reserved values for matrix\_coeffs shall not be present in bitstreams conforming to this version of this Specification. Decoders shall interpret reserved values of matrix\_coeffs as equivalent to the value 2.

It is a requirement of bitstream conformance to this version of this Specification that when colour\_primaries is not equal to 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, or 22, matrix\_coeffs shall not be equal to 12 or 13.

When matrix\_coeffs is equal to 1, 4, 5, 6, 7, 9, 10, 11, 12, or 13, the constants KB and KR are specified as follows:

– If matrix\_coeffs is not equal to 12 or 13, the constants KB and KR are specified in Table E.4.

– Otherwise (matrix\_coeffs is equal to 12 or 13), the constants KR and KB are computed as follows, using the chromaticity coordinates (xR, yR), (xG, yG), (xB, yB), and (xW, yW) specified by Table E.2 for the colour\_primaries syntax element for the red, green, blue, and white colour primaries, respectively.

KR =  (E‑22)

KB =  (E‑23)

where the values of zR, zG, zB, and zW, are given by.

zR = 1 − ( xR + yR ) (E‑24)

zG = 1 − ( xG + yG ) (E‑25)

zB = 1 − ( xB + yB ) (E‑26)

zW = 1 − ( xW + yW ) (E‑27)

The variables E′Y, E′PB, and E′PR (for matrix\_coeffs not equal to 0 or 8) or Y, Cb, and Cr (for matrix\_coeffs equal to 0 or 8) are specified as follows:

– If matrix\_coeffs is not equal to 0, 8, 10, 11, 13, or 14, the following equations apply:

E′Y = KR \* E′R + ( 1 − KR − KB ) \* E′G + KB \* E′B (E‑28)

E′PB = 0.5 \* ( E′B − E′Y ) ÷ ( 1 − KB ) (E‑29)

E′PR = 0.5 \* ( E′R − E′Y ) ÷ ( 1 − KR ) (E‑30)

NOTE 6 – 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 inclusive. 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 inclusive. When transfer\_characteristics is equal to 11 or 12, E′Y, E′PB, and E′PR are real numbers with a larger range not specified in this Specification.

– Otherwise, if matrix\_coeffs is equal to 0, the following equations apply:

Y = Round( G ) (E‑31)

Cb = Round( B ) (E‑32)

Cr = Round( R ) (E‑33)

– Otherwise, if matrix\_coeffs 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‑34)

Cb = Round( 0.5 \* G − 0.25 \* ( R + B ) ) + ( 1 << ( BitDepthC − 1 ) ) (E‑35)

Cr = Round( 0.5 \* (R − B ) ) + ( 1 << ( BitDepthC − 1 ) ) (E‑36)

NOTE 7 – In this case, for purposes of the YCgCo nomenclature used in Table E.4, Cb and Cr of Equations E‑35 and E‑36 may be referred to as Cg and Co, respectively. An appropriate inverse conversion for Equations E‑34 to E‑36 is as follows:

t = Y − ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) (E‑37)

G = Clip1Y( Y + ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑38)

B = Clip1Y( t − ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑39)

R = Clip1Y( t + ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑40)

– Otherwise (BitDepthC is not equal to BitDepthY), the following equations apply:

Cr = Round( R ) − Round( B ) + ( 1 << ( BitDepthC − 1 ) ) (E‑41)

t = Round( B ) + ( ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑42)

Cb = Round( G ) − t + ( 1 << ( BitDepthC − 1 ) ) (E‑43)

Y = t + ( ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑44)

NOTE 8 – In this case, for purposes of the YCgCo nomenclature used in Table E.4, Cb and Cr of Equations E‑43 and E‑41 may be referred to as Cg and Co, respectively. An appropriate inverse conversion for Equations E‑41 to E‑44 is as follows:

t = Y − ( ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) (E‑45)

G = Clip1Y( t + ( Cb − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑46)

B = Clip1Y( t − ( ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) >> 1 ) ) (E‑47)

R = Clip1Y( B + ( Cr − ( 1 << ( BitDepthC − 1 ) ) ) ) (E‑48)

– Otherwise, if matrix\_coeffs is equal to 10 or 13, the signal E′Y is determined by application of the transfer characteristics function as follows and Equations E‑51 to E‑54 apply for specification of the signals E′PB and E′PR:

EY = KR \* ER + ( 1 − KR − KB ) \* EG + KB \* EB (E‑49)

E′Y = ( EY )′ (E‑50)

NOTE 9 – In this case, EY is defined from the "linear-domain" signals for ER, EG, and EB, prior to application of the transfer characteristics function, which is then applied to produce the signal E′Y. EY and E′Y are real values with the value 0 associated with nominal black and the value 1 associated with nominal white.

while the signals E′PB and E′PR are determined as follows:

E′PB = ( E′B − E′Y ) ÷ ( 2 \* NB ) for −NB <= E′B − E′Y  <= 0 (E‑51)

E′PB = ( E′B − E′Y ) ÷ ( 2 \* PB ) for 0 < E′B − E′Y <= PB (E‑52)

E′PR = ( E′R − E′Y ) ÷ ( 2 \* NR ) for −NR <= E′R − E′Y  <= 0 (E‑53)

E′PR = ( E′R − E′Y ) ÷ ( 2 \* PR ) for 0 < E′R − E′Y  <= PR (E‑54)

where the constants NB, PB, NR, and PR are determined by application of the transfer characteristics function to expressions involving the constants KB and KR as follows:

NB = ( 1 − KB )′ (E‑55)

PB = 1 − ( KB )′ (E‑56)

NR = ( 1 − KR )′ (E‑57)

PR = 1 − (  KR )′ (E‑58)

– Otherwise, if matrix\_coeffs is equal to 11, the following equations apply:

E′Y = E′G (E‑59)

E′PB = 0.5 \* ( 0.986566 \* E′B − E′Y ) (E‑60)

E′PR = 0.5 \* ( E′R − 0.991902 \* E′Y ) (E‑61)

NOTE 10 – In this case, for purposes of the Y′D′ZD′X nomenclature used in Table E.4, E′PB may be referred to as D′Z and E′PR may be referred to as D′X.

– Otherwise (matrix\_coeffs is equal to 14), the following equations apply:

E′Y = 0.5 \* ( E′L + E′M ) (E‑62)

E′PB = ( 6610 \* E′L − 13613 \* E′M + 7003 \* E′S ) ÷ 4096 (E‑63)

E′PR = ( 17933 \* E′L − 17390 \* E′M − 543 \* E′S ) ÷ 4096 (E‑64)

NOTE 11 – In this case, for purposes of the ICTCP nomenclature used in Table E.4, E′Y, E′PB, and E′PR of Equations E‑62, E‑63, and E‑64 may be referred to as I, CT, and CP, respectively.

Table E.4 – Matrix coefficients interpretation using the matrix\_coeffs syntax element

|  |  |  |
| --- | --- | --- |
| Value | Matrix | Informative remark |
| 0 | Identity | The identity matrix.  Typically used for GBR (often referred to as RGB); however, may also be used for YZX (often referred to as XYZ)  IEC 61966-2-1 sRGB  SMPTE ST 428-1 (2006)  See Equations E‑31 to E‑33 |
| 1 | KR = 0.2126; KB = 0.0722 | Rec. ITU-R BT.709-6  Rec. ITU-R BT.1361-0 conventional colour gamut system and extended colour gamut system (historical)  IEC 61966-2-1 sYCC  IEC 61966-2-4 xvYCC709  SMPTE RP 177 (1993) Annex B  See Equations E‑28 to E‑30 |
| 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 | FCC Title 47 Code of Federal Regulations (2003) 73.682 (a) (20)  See Equations E‑28 to E‑30 |
| 5 | KR = 0.299; KB = 0.114 | Rec. ITU‑R BT.470‑6 System B, G (historical)  Rec. ITU‑R BT.601‑7 625  Rec. ITU‑R BT.1358-0 625 (historical)  Rec. ITU‑R BT.1700-0 625 PAL and 625 SECAM  IEC 61966-2-4 xvYCC601  (functionally the same as the value 6)  See Equations E‑28 to E‑30 |
| 6 | KR = 0.299; KB = 0.114 | Rec. ITU‑R BT.601‑7 525  Rec. ITU‑R BT.1358-1 525 or 625 (historical)  Rec. ITU‑R BT.1700-0 NTSC  SMPTE ST 170 (2004)  (functionally the same as the value 5)  See Equations E‑28 to E‑30 |
| 7 | KR = 0.212; KB = 0.087 | SMPTE ST 240 (1999, historical)  See Equations E‑28 to E‑30 |
| 8 | YCgCo | See Equations E‑34 to E‑48 |
| 9 | KR = 0.2627; KB = 0.0593 | Rec. ITU-R BT.2020-2 non-constant luminance system  Rec. ITU-R BT.2100-1 Y′CbCr  See Equations E‑28 to E‑30 |
| 10 | KR = 0.2627; KB = 0.0593 | Rec. ITU-R BT.2020-2 constant luminance system  See Equations E‑49 to E‑58 |
| 11 | Y′D′ZD′X | SMPTE ST 2085 (2015)  See Equations E‑59 to E‑61 |
| 12 | See Equations E‑22 to E‑27 | Chromaticity-derived non-constant luminance system  See Equations E‑28 to E‑30 |
| 13 | See Equations E‑22 to E‑27 | Chromaticity-derived constant luminance system  See Equations E‑49 to E‑58 |
| 14 | ICTCP | Rec. ITU-R BT.2100-1 ICTCP  See Equations E‑62 to E‑64 |
| 15..255 | Reserved | For future use by ITU‑T | ISO/IEC |

**field\_seq\_flag** equal to 1 indicates that the CVS conveys pictures that represent fields, and specifies that a picture timing SEI message shall be present in every access unit of the current CVS. field\_seq\_flag equal to 0 indicates that the CVS conveys pictures that represent frames and that a picture timing SEI message may or may not be present in any access unit of the current CVS. When field\_seq\_flag is not present, it is inferred to be equal to 0. When general\_frame\_only\_constraint\_flag is equal to 1, the value of field\_seq\_flag shall be equal to 0.

NOTE 12 – 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).

between 50 and 60 Hz), instead of the source frame rate (typically between 25 and 30 Hz).

**chroma\_loc\_info\_present\_flag** equal to 1 specifies that either chroma\_sample\_loc\_type or both 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, 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**, **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) and field\_seq\_flag is equal to 1, 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 Figure E.1.

– Otherwise if chroma\_format\_idc is equal to 1 (4:2:0 chroma format) and field\_seq\_flag is equal to 0, chroma\_sample\_loc\_type specify the location of chroma samples for the whole picture as shown in Figure E.2.

– Otherwise (chroma\_format\_idc is not equal to 1), the values of the syntax elements chroma\_sample\_loc\_type, 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 clause 6.2. When chroma\_format\_idc is equal to 0, there is no chroma sample array.

The value of chroma\_sample\_loc\_type, chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field shall be in the range of 0 to 5, inclusive. When 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 are inferred to be equal to 0. When chroma\_sample\_loc\_type is not present, the value of chroma\_sample\_loc\_type is inferred to be equal to 0.

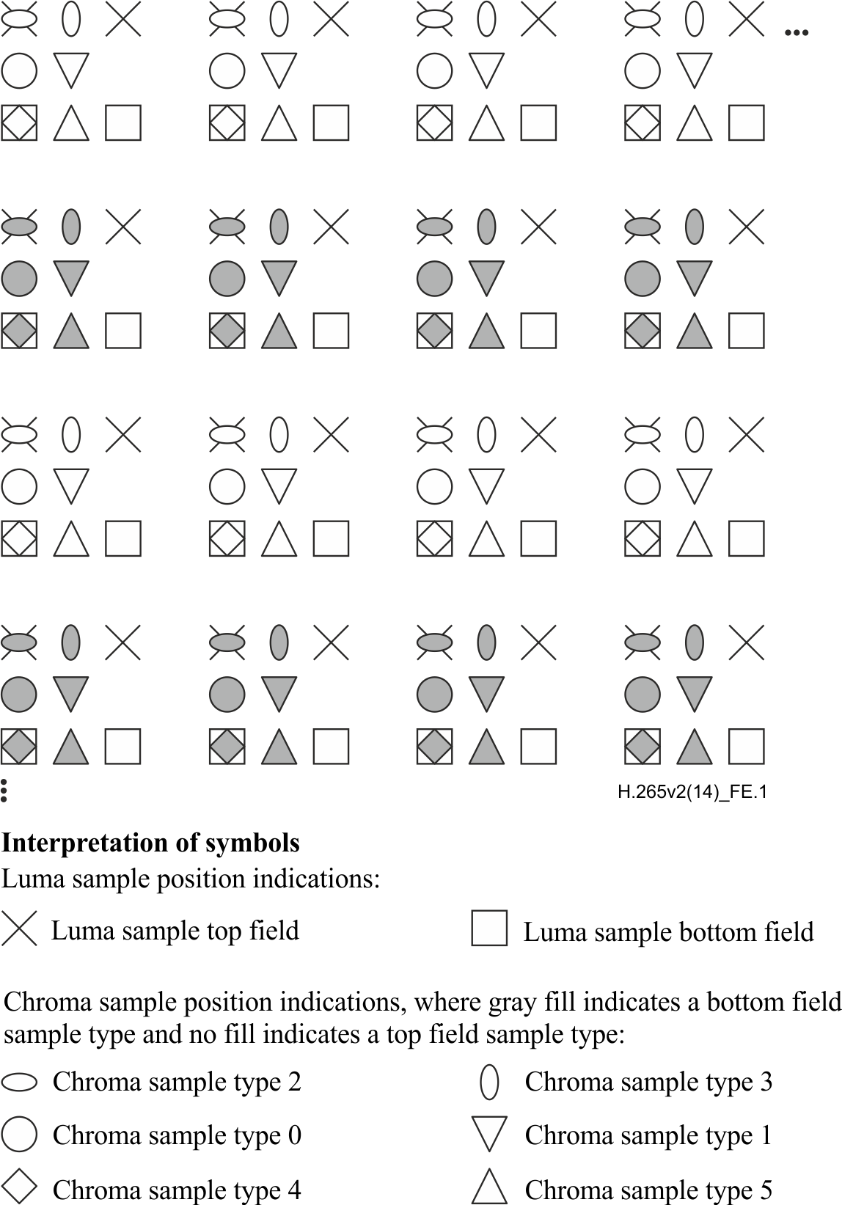


Figure E.1 – Location of chroma samples for top and bottom fields for chroma\_format\_idc equal to 1 (4:2:0 chroma format) as a function of chroma\_sample\_loc\_type\_top\_field and chroma\_sample\_loc\_type\_bottom\_field

Figure E.2 illustrates the indicated relative position of the top-left chroma sample when chroma\_format\_idc is equal to 1 (4:2:0 chroma format), and chroma\_sample\_loc\_type is equal to the value of a variable ChromaLocType. The region represented by the top-left 4:2:0 chroma sample (depicted as a large red square with a large red dot at its centre) is shown relative to the region represented by the top-left luma sample (depicted as a small black square with a small black dot at its centre). The regions represented by neighbouring luma samples are depicted as small grey squares with small grey dots at their centres.

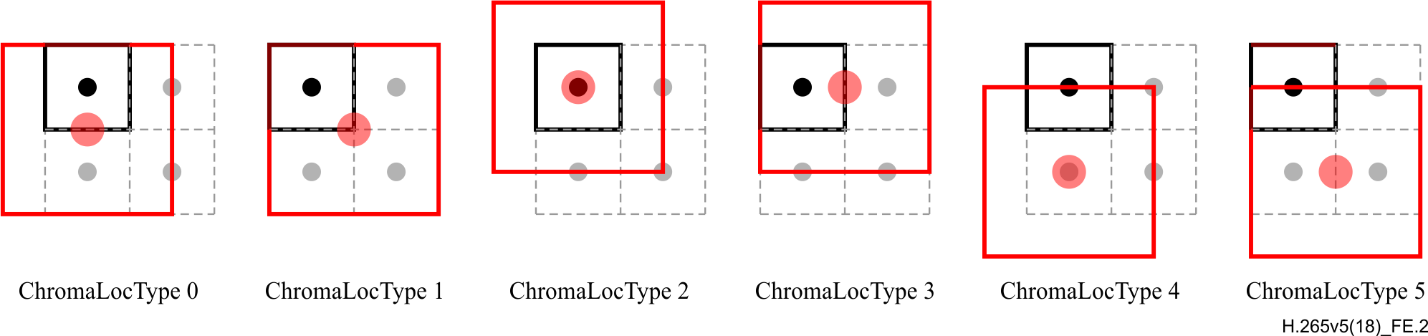


Figure E.2 – Location of the top-left chroma sample when chroma\_format\_idc is equal to 1  
(4:2:0 chroma format) as a function of ChromaLocType

The relative spatial positioning of the chroma samples, as illustrated in Figure E.3, can be expressed by defining two variables HorizontalOffsetC and VerticalOffsetC as a function of chroma\_format\_idc and the variable ChromaLocType as given by Table E.5, where HorizontalOffsetC is the horizontal (x) position of the centre of the top-left chroma sample relative to the centre of the top-left luma sample in units of luma samples and VerticalOffsetC is the vertical (y) position of the centre of the top-left chroma sample relative to the centre of the top-left luma sample in units of luma samples.

In a typical FIR filter design, when chroma\_format\_idc is equal to 1 (4:2:0 chroma format) or 2 (4:2:2 chroma format), HorizontalOffsetC and VerticalOffsetC would serve as the phase offsets for the horizontal and vertical filter operations, respectively, for separable downsampling from 4:4:4 chroma format to the chroma format indicated by chroma\_format\_idc.

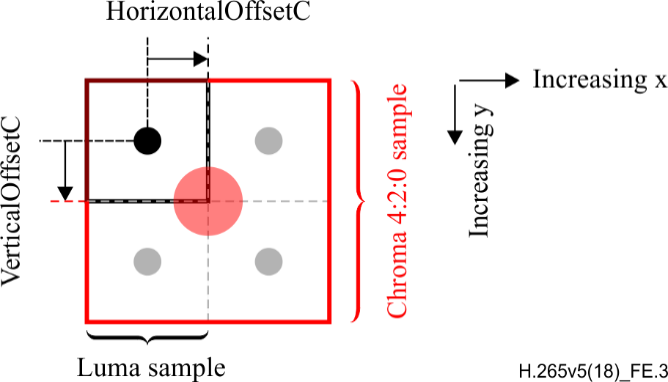


Figure E.3 – Location of the top-left chroma sample when chroma\_format\_idc is equal to 1  
(4:2:0 chroma format) when ChromaLocType is equal to 1

Table E.5 – Definition of HorizontalOffsetC and VerticalOffsetC  
as a function of chroma\_format\_idc and ChromaLocType

|  |  |  |  |
| --- | --- | --- | --- |
| chroma\_format\_idc | ChromaLocType | HorizontalOffsetC | VerticalOffsetC |
| 1 (4:2:0) | 0 | 0 | 0.5 |
| 1 (4:2:0) | 1 | 0.5 | 0.5 |
| 1 (4:2:0) | 2 | 0 | 0 |
| 1 (4:2:0) | 3 | 0.5 | 0 |
| 1 (4:2:0) | 4 | 0 | 1 |
| 1 (4:2:0) | 5 | 0.5 | 1 |
| 2 (4:2:2) | – | 0 | 0 |
| 3 (4:4:4) | – | 0 | 0 |

When chroma\_format\_idc is equal to 1 (4:2:0 chroma format) and the decoded video content is intended for interpretation according to Rec. ITU-R BT.2020-2 or Rec. ITU-R BT.2100-1, chroma\_loc\_info\_present\_flag should be equal to 1, and chroma\_sample\_loc\_type should be equal to 2.

**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 conformance cropping window 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 conformance cropping window, 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 13 – 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.

NOTE 14 – Some of the semantics of video signal type parameters associated with video\_signal\_type\_present\_flag equal to 1 are expressed in terms of the properties of source pictures prior to operation of the encoding process, which is outside the scope of this Specification. This is partly for historical reasons and due to the common general practice of how the indicated data is typically described in industry publications. The actual intent for providing this syntax in the bitstream is to assist decoding systems to properly interpret and make effective use of the decoded video pictures, e.g., for use by the display process (which is also outside the scope of this Specification, but for which having an indication of how the pictures should be interpreted is important).

**video\_full\_range\_flag** indicates the black level and range of the luma and chroma signals as derived from E′Y, E′PB, and E′PR or E′R, E′G, and E′B real-valued component signals. When not present, the value of video\_full\_range\_flag is inferred to be equal to 0.

* + 1. HRD parameters semantics

The hrd\_parameters( ) syntax structure provides HRD parameters used in the HRD operations.

**vui\_nal\_hrd\_parameters\_present\_flag** equal to 1 specifies that NAL HRD parameters (pertaining to Type II bitstream conformance) are present in the hrd\_parameters( ) syntax structure. vui\_nal\_hrd\_parameters\_present\_flag equal to 0 specifies that NAL HRD parameters are not present in the hrd\_parameters( ) syntax structure.

NOTE 1 – When vui\_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 Specification.

The variable NalHrdBpPresentFlag is derived as follows:

– If one or more of the following conditions are true, the value of NalHrdBpPresentFlag is set equal to 1:

– vui\_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 Specification.

– Otherwise, the value of NalHrdBpPresentFlag is set equal to 0.

**vui\_vcl\_hrd\_parameters\_present\_flag** equal to 1 specifies that VCL HRD parameters (pertaining to all bitstream conformance) are present in the hrd\_parameters( ) syntax structure. vui\_vcl\_hrd\_parameters\_present\_flag equal to 0 specifies that VCL HRD parameters are not present in the hrd\_parameters( ) syntax structure.

NOTE 2 – When vui\_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 Specification.

The variable VclHrdBpPresentFlag is derived as follows:

– If one or more of the following conditions are true, the value of VclHrdBpPresentFlag is set equal to 1:

– vui\_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 Specification.

– Otherwise, the value of VclHrdBpPresentFlag is set equal to 0.

The variable CpbDpbDelaysPresentFlag is derived as follows:

– If one or more of the following conditions are true, the value of CpbDpbDelaysPresentFlag is set equal to 1:

– vui\_nal\_hrd\_parameters\_present\_flag is present in the bitstream and is equal to 1.

– vui\_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 Specification.

– Otherwise, the value of CpbDpbDelaysPresentFlag is set equal to 0.

**bit\_rate\_scale** (together with bit\_rate\_value\_minus1[ i ]) specifies the maximum input bit rate of the i-th CPB.

**cpb\_size\_scale** (together with cpb\_size\_value\_minus1[ i ]) specifies the CPB size of the i-th CPB when the CPB operates at the access unit level.

**fixed\_pic\_rate\_general\_flag**[ i ] equal to 1 indicates that, when HighestTid is equal to i, the temporal distance between the HRD output times of consecutive pictures in output order is constrained as specified below. fixed\_pic\_rate\_general\_flag[ i ] equal to 0 indicates that this constraint may not apply.

When fixed\_pic\_rate\_general\_flag[ i ] is not present, it is inferred to be equal to 0.

**fixed\_pic\_rate\_within\_cvs\_flag**[ i ] equal to 1 indicates that, when HighestTid is equal to i, the temporal distance between the HRD output times of consecutive pictures in output order is constrained as specified below. fixed\_pic\_rate\_within\_cvs\_flag[ i ] equal to 0 indicates that this constraint may not apply.

When fixed\_pic\_rate\_general\_flag[ i ] is equal to 1, the value of fixed\_pic\_rate\_within\_cvs\_flag[ i ] is inferred to be equal to 1.

**elemental\_duration\_in\_tc\_minus1**[ i ] plus 1 (when present) specifies, when HighestTid is equal to i, the temporal distance, in clock ticks, between the elemental units that specify the HRD output times of consecutive pictures in output order as specified below. The value of elemental\_duration\_in\_tc\_minus1[ i ] shall be in the range of 0 to 2047, inclusive.

For each picture n that is output and not the last picture in the bitstream (in output order) that is output, the value of the variable DpbOutputElementalInterval[ n ] is specified by:

DpbOutputElementalInterval[ n ] = DpbOutputInterval[ n ]  DeltaToDivisor (E‑65)

where DpbOutputInterval[ n ] is specified in Equation C‑13 and DeltaToDivisor is specified in Table E‑5 based on the value of frame\_field\_info\_present\_flag and pic\_struct for the CVS containing picture n. Entries marked "-" in Table E‑5 indicate a lack of dependence of DeltaToDivisor on the corresponding syntax element.

When HighestTid is equal to i and fixed\_pic\_rate\_general\_flag[ i ] is equal to 1 for a CVS containing picture n, the value computed for DpbOutputElementalInterval[ n ] shall be equal to ClockTick \* ( elemental\_duration\_in\_tc\_minus1[ i ] + 1 ), wherein ClockTick is as specified in Equation C‑1 (using the value of ClockTick for the CVS containing picture n) when one of the following conditions is true for the following picture in output order nextPicInOutputOrder that is specified for use in Equation C‑13:

– picture nextPicInOutputOrder is in the same CVS as picture n.

– picture nextPicInOutputOrder is in a different CVS and fixed\_pic\_rate\_general\_flag[ i ] is equal to 1 in the CVS containing picture nextPicInOutputOrder, the value of ClockTick is the same for both CVSs, and the value of elemental\_duration\_in\_tc\_minus1[ i ] is the same for both CVSs.

When HighestTid is equal to i and fixed\_pic\_rate\_within\_cvs\_flag[ i ] is equal to 1 for a CVS containing picture n, the value computed for DpbOutputElementalInterval[ n ] shall be equal to ClockTick \* ( elemental\_duration\_in\_tc\_minus1[ i ] + 1 ), wherein ClockTick is as specified in Equation C‑1 (using the value of ClockTick for the CVS containing picture n) when the following picture in output order nextPicInOutputOrder that is specified for use in Equation C‑13 is in the same CVS as picture n.

Table E‑5 – Divisor for computation of DpbOutputElementalInterval[ n ]

|  |  |  |
| --- | --- | --- |
| **frame\_field\_info\_present\_flag** | **pic\_struct** | **DeltaToDivisor** |
| 0 | - | 1 |
| 1 | 1 | 1 |
| 1 | 2 | 1 |
| 1 | 0 | 2 |
| 1 | 3 | 2 |
| 1 | 4 | 2 |
| 1 | 5 | 3 |
| 1 | 6 | 3 |
| 1 | 7 | 2 |
| 1 | 8 | 3 |
| 1 | 9 | 1 |
| 1 | 10 | 1 |
| 1 | 11 | 1 |
| 1 | 12 | 1 |

**low\_delay\_hrd\_flag**[ i ] specifies the HRD operational mode, when HighestTid is equal to i, as specified in Annex C. When not present, the value of low\_delay\_hrd\_flag[ i ] is inferred to be equal to 0.

NOTE 3 – When low\_delay\_hrd\_flag[ i ] 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.

**vui\_cpb\_cnt\_minus1**[ i ] plus 1 specifies the number of alternative CPB specifications in the bitstream of the CVS when HighestTid is equal to i. The value of vui\_cpb\_cnt\_minus1[ i ] shall be in the range of 0 to 31, inclusive. When not present, the value of vui\_cpb\_cnt\_minus1[ i ] is inferred to be equal to 0.

* + 1. Sub-layer HRD parameters semantics

The variable CpbCnt is set equal to vui\_cpb\_cnt\_minus1[ subLayerId ].

**bit\_rate\_value\_minus1**[ i ] (together with bit\_rate\_scale) specifies the maximum input bit rate for the i-th CPB when the CPB operates at the access unit level. bit\_rate\_value\_minus1[ i ] shall be in the range of 0 to 232 − 2, inclusive. For any i > 0, bit\_rate\_value\_minus1[ i ] shall be greater than bit\_rate\_value\_minus1[ i − 1 ].

The bit rate in bits per second is given by:

BitRate[ i ] = ( bit\_rate\_value\_minus1[ i ] + 1 ) \* 2( 6 + bit\_rate\_scale ) (E‑66)

When the bit\_rate\_value\_minus1[ i ] syntax element is not present, the value of BitRate[ i ] 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 A.4.]

**cpb\_size\_value\_minus1**[ i ] is used together with cpb\_size\_scale to specify the i-th CPB size when the CPB operates at the access unit level. cpb\_size\_value\_minus1[ i ] shall be in the range of 0 to 232 − 2, inclusive. For any i greater than 0, cpb\_size\_value\_minus1[ i ] shall be less than or equal to cpb\_size\_value\_minus1[ i − 1 ].

The CPB size in bits is given by:

CpbSize[ i ] = ( cpb\_size\_value\_minus1[ i ] + 1 ) \* 2( 4 + cpb\_size\_scale ) (E‑67)

When the cpb\_size\_value\_minus1[ i ] syntax element is not present, the value of CpbSize[ i ] 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 A.4.]

**cbr\_flag**[ i ] equal to 0 specifies that to decode this bitstream by the HRD using the i-th CPB specification, the hypothetical stream scheduler (HSS) operates in an intermittent bit rate mode. cbr\_flag[ i ] equal to 1 specifies that the HSS operates in a constant bit rate (CBR) mode. When not present, the value of cbr\_flag[ i ] is inferred to be equal to 0.

1. Annex F  
     
   Non-HRD related supplemental enhancement information

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

* 1. General

This annex specifies syntax and semantics for SEI message payloads that are non-HRD related. [Ed. (YK): This Annex will be moved to a separate document in the future.]

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 Specification (see Annex C for the specification of conformance). SEI messages specified in Annex D are required to check bitstream conformance and for output timing decoder conformance. SEI messages specified in this Annex are not required to check bitstream conformance and for output timing decoder conformance.

* 1. Non-HRD related SEI payload syntax
     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. Dependent random access point indication SEI message syntax

|  |  |
| --- | --- |
| dependent\_rap\_indication( payloadSize ) { | Descriptor |
| } |  |

* 1. Non-HRD related SEI payload semantics

* + 1. Decoded picture hash SEI message semantics

This message provides a hash for each colour component of the current decoded picture.

NOTE 1 – The decoded picture hash SEI message is a suffix SEI message and cannot be contained in a scalable nesting SEI message.

Prior to computing the hash, the decoded picture data are arranged into one or three strings of bytes called pictureData[ cIdx ] of lengths dataLen[ cIdx ] as follows:

for( cIdx = 0; cIdx < ( chroma\_format\_idc = = 0 ) ? 1 : 3; cIdx++ ) {  
 if( cIdx = = 0 ) {  
 compWidth[ cIdx ] = pic\_width\_in\_luma\_samples  
 compHeight[ cIdx ] = pic\_height\_in\_luma\_samples  
 compDepth[ cIdx ] = BitDepthY  
 } else {  
 compWidth[ cIdx ] = pic\_width\_in\_luma\_samples / SubWidthC  
 compHeight[ cIdx ] = pic\_height\_in\_luma\_samples / SubHeightC  
 compDepth[ cIdx ] = BitDepthC (F‑‑1)  
 }  
 iLen = 0  
 for( i = 0; i < compWidth[ cIdx ] \* compHeight[ cIdx ]; i++ ) {  
 pictureData[ cIdx ][ iLen++ ] = component[ cIdx ][ i ] & 0xFF  
 if( compDepth[ cIdx ] > 8 )  
 pictureData[ cIdx ][ iLen++ ] = component[ cIdx ][ i ] >> 8  
 }  
 dataLen[ cIdx ] = iLen  
}

where component[ cIdx ][ i ] is an array in raster scan of decoded sample values in two's complement representation.

**hash\_type** indicates the method used to calculate the checksum according to Table D.6. Values of hash\_type that are not listed in Table D.6 are reserved for future use by ITU-T | ISO/IEC and shall not be present in bitstreams conforming to this version of this Specification. Decoders shall ignore decoded picture hash SEI messages that contain reserved values of hash\_type.

Table D.6 – Interpretation of hash\_type

|  |  |
| --- | --- |
| hash\_type | Method |
| 0 | MD5 (IETF RFC 1321) |
| 1 | CRC |
| 2 | Checksum |

**picture\_md5**[ cIdx ][ i ] is the 16-byte MD5 hash of the cIdx-th colour component of the decoded picture. The value of picture\_md5[ cIdx ][ i ] shall be equal to the value of digestVal[ cIdx ] obtained as follows, using the MD5 functions defined in IETF RFC 1321:

MD5Init( context )  
MD5Update( context, pictureData[ cIdx ], dataLen[ cIdx ] ) (F‑2)  
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 as follows:

crc = 0xFFFF  
pictureData[ cIdx ][  dataLen[ cIdx ] ] = 0  
pictureData[ cIdx ][  dataLen[ cIdx ] + 1 ] = 0  
for( bitIdx = 0; bitIdx < ( dataLen[ cIdx ]  + 2 ) \* 8; bitIdx++ ) { (F‑3)  
 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 2 – 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 picture\_checksum[ cIdx ] shall be equal to the value of checksumVal[ cIdx ] obtained as follows:

sum = 0  
for( y = 0; y < compHeight[ cIdx ]; y++ )  
 for( x = 0; x < compWidth[ cIdx ]; x++ ) {  
 xorMask = ( x & 0xFF ) ^ ( y & 0xFF ) ^ ( x >> 8 ) ^ ( y >> 8 )  
 sum = ( sum + ( ( component[ cIdx ][ y \* compWidth[ cIdx ] + x ] & 0xFF ) ^ (F‑4)  
 xorMask ) ) & 0xFFFFFFFF  
 if( compDepth[ cIdx ] > 8 )  
 sum = ( sum + ( ( component[ cIdx ][ y \* compWidth[ cIdx ] + x ] >> 8 ) ^  
 xorMask ) ) & 0xFFFFFFFF  
 }  
checksumVal[ cIdx ] = sum

* + 1. Dependent random access point indication SEI message semantics

The picture associated with a dependent random access point (DRAP) indication SEI message is referred to as a DRAP picture.

The presence of the DRAP indication SEI message indicates that the constraints on picture order and picture referencing specified in this subclause apply. These constraints can enable a decoder to properly decode the DRAP picture and the pictures that follow it in both decoding order and output order without needing to decode any other pictures except the associated IRAP picture.

The constraints indicated by the presence of the DRAP indication SEI message are as follows:

– Each VCL NAL unit of the DRAP picture shall have NalUnitType equal to TRAIL\_NUT and TemporalId equal to 0.

– The DRAP picture shall not include any pictures in the active entries of its RefPicList[ 0 ] or RefPicList[ 1 ] except its associated IRAP picture.

– Any picture that follows the DRAP picture in both decoding order and output order shall not include, in the active entries of its RefPicList[ 0 ] or RefPicList[ 1 ], any picture that precedes the DRAP picture in decoding order or output order with the exception of the IRAP picture associated with the DRAP picture.