
**Information technology — High efficiency
coding and media delivery in
heterogeneous environments —**

**Part 2:
High efficiency video coding**

*Technologies de l'information — Codage à haute efficacité et livraison
des medias dans des environnements hétérogènes —*

Partie 2: Codage vidéo à haute efficacité

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2013

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office

Case postale 56 • CH-1211 Geneva 20

Tel. + 41 22 749 01 11

Fax + 41 22 749 09 47

E-mail copyright@iso.org

Web www.iso.org

Published in Switzerland

CONTENTS

	<i>Page</i>
0 Introduction	1
0.1 General.....	1
0.2 Prologue.....	1
0.3 Purpose	1
0.4 Applications.....	1
0.5 Publication and versions of this Specification	1
0.6 Profiles, tiers and levels	2
0.7 Overview of the design characteristics	2
0.8 How to read this Specification	2
1 Scope	3
2 Normative references.....	3
2.1 General.....	3
2.2 Identical Recommendations International Standards.....	3
2.3 Paired Recommendations International Standards equivalent in technical content.....	3
2.4 Additional references.....	3
3 Definitions	3
4 Abbreviations.....	12
5 Conventions	13
5.1 General.....	13
5.2 Arithmetic operators	13
5.3 Logical operators	13
5.4 Relational operators.....	14
5.5 Bit-wise operators.....	14
5.6 Assignment operators	14
5.7 Range notation	14
5.8 Mathematical functions.....	15
5.9 Order of operation precedence.....	15
5.10 Variables, syntax elements, and tables.....	16
5.11 Text description of logical operations.....	17
5.12 Processes.....	18
6 Bitstream and picture formats, partitionings, scanning processes, and neighbouring relationships	18
6.1 Bitstream formats.....	18
6.2 Source, decoded, and output picture formats.....	18
6.3 Partitioning of pictures, slices, slice segments, tiles, coding tree units, and coding tree blocks.....	21
6.3.1 Partitioning of pictures into slices, slice segments, and tiles	21
6.3.2 Block and quadtree structures	22
6.3.3 Spatial or component-wise partitionings.....	23
6.4 Availability processes	23
6.4.1 Derivation process for z-scan order block availability.....	23
6.4.2 Derivation process for prediction block availability.....	24
6.5 Scanning processes	25
6.5.1 Coding tree block raster and tile scanning conversion process.....	25
6.5.2 Z-scan order array initialization process	26
6.5.3 Up-right diagonal scan order array initialization process	26
6.5.4 Horizontal scan order array initialization process.....	27
6.5.5 Vertical scan order array initialization process.....	27
7 Syntax and semantics.....	28
7.1 Method of specifying syntax in tabular form.....	28
7.2 Specification of syntax functions and descriptors.....	29
7.3 Syntax in tabular form	30
7.3.1 NAL unit syntax.....	30
7.3.1.1 General NAL unit syntax.....	30

7.3.1.2	NAL unit header syntax	30
7.3.2	Raw byte sequence payloads, trailing bits, and byte alignment syntax	31
7.3.2.1	Video parameter set RBSP syntax	31
7.3.2.2	Sequence parameter set RBSP syntax	32
7.3.2.3	Picture parameter set RBSP syntax	34
7.3.2.4	Supplemental enhancement information RBSP syntax	35
7.3.2.5	Access unit delimiter RBSP syntax	35
7.3.2.6	End of sequence RBSP syntax	35
7.3.2.7	End of bitstream RBSP syntax	35
7.3.2.8	Filler data RBSP syntax	36
7.3.2.9	Slice segment layer RBSP syntax	36
7.3.2.10	RBSP slice segment trailing bits syntax	36
7.3.2.11	RBSP trailing bits syntax	36
7.3.2.12	Byte alignment syntax	36
7.3.3	Profile, tier and level syntax	37
7.3.4	Scaling list data syntax	38
7.3.5	Supplemental enhancement information message syntax	38
7.3.6	Slice segment header syntax	39
7.3.6.1	General slice segment header syntax	39
7.3.6.2	Reference picture list modification syntax	41
7.3.6.3	Weighted prediction parameters syntax	42
7.3.7	Short-term reference picture set syntax	43
7.3.8	Slice segment data syntax	43
7.3.8.1	General slice segment data syntax	43
7.3.8.2	Coding tree unit syntax	44
7.3.8.3	Sample adaptive offset syntax	45
7.3.8.4	Coding quadtree syntax	46
7.3.8.5	Coding unit syntax	47
7.3.8.6	Prediction unit syntax	49
7.3.8.7	PCM sample syntax	49
7.3.8.8	Transform tree syntax	50
7.3.8.9	Motion vector difference syntax	50
7.3.8.10	Transform unit syntax	51
7.3.8.11	Residual coding syntax	52
7.4	Semantics	54
7.4.1	General	54
7.4.2	NAL unit semantics	54
7.4.2.1	General NAL unit semantics	54
7.4.2.2	NAL unit header semantics	55
7.4.2.3	Encapsulation of an SODB within an RBSP (informative)	58
7.4.2.4	Order of NAL units and association to coded pictures, access units, and coded video sequences	59
7.4.3	Raw byte sequence payloads, trailing bits, and byte alignment semantics	62
7.4.3.1	Video parameter set RBSP semantics	62
7.4.3.2	Sequence parameter set RBSP semantics	64
7.4.3.3	Picture parameter set RBSP semantics	69
7.4.3.4	Supplemental enhancement information RBSP semantics	72
7.4.3.5	Access unit delimiter RBSP semantics	72
7.4.3.6	End of sequence RBSP semantics	72
7.4.3.7	End of bitstream RBSP semantics	72
7.4.3.8	Filler data RBSP semantics	73
7.4.3.9	Slice segment layer RBSP semantics	73
7.4.3.10	RBSP slice segment trailing bits semantics	73
7.4.3.11	RBSP trailing bits semantics	73
7.4.3.12	Byte alignment semantics	73
7.4.4	Profile, tier and level semantics	73
7.4.5	Scaling list data semantics	75
7.4.6	Supplemental enhancement information message semantics	76
7.4.7	Slice segment header semantics	77
7.4.7.1	General slice segment header semantics	77
7.4.7.2	Reference picture list modification semantics	81
7.4.7.3	Weighted prediction parameters semantics	82
7.4.8	Short-term reference picture set semantics	83
7.4.9	Slice segment data semantics	85

7.4.9.1	General slice segment data semantics	85
7.4.9.2	Coding tree unit semantics	85
7.4.9.3	Sample adaptive offset semantics	85
7.4.9.4	Coding quadtree semantics	87
7.4.9.5	Coding unit semantics	87
7.4.9.6	Prediction unit semantics	89
7.4.9.7	PCM sample semantics	90
7.4.9.8	Transform tree semantics	90
7.4.9.9	Motion vector difference semantics	91
7.4.9.10	Transform unit semantics	91
7.4.9.11	Residual coding semantics	92
8	Decoding process	94
8.1	General decoding process	94
8.2	NAL unit decoding process	95
8.3	Slice decoding process	96
8.3.1	Decoding process for picture order count	96
8.3.2	Decoding process for reference picture set	96
8.3.3	Decoding process for generating unavailable reference pictures	100
8.3.3.1	General decoding process for generating unavailable reference pictures	100
8.3.3.2	Generation of one unavailable picture	101
8.3.4	Decoding process for reference picture lists construction	101
8.4	Decoding process for coding units coded in intra prediction mode	102
8.4.1	General decoding process for coding units coded in intra prediction mode	102
8.4.2	Derivation process for luma intra prediction mode	103
8.4.3	Derivation process for chroma intra prediction mode	105
8.4.4	Decoding process for intra blocks	105
8.4.4.1	General decoding process for intra blocks	105
8.4.4.2	Intra sample prediction	106
8.5	Decoding process for coding units coded in inter prediction mode	112
8.5.1	General decoding process for coding units coded in inter prediction mode	112
8.5.2	Inter prediction process	112
8.5.3	Decoding process for prediction units in inter prediction mode	115
8.5.3.1	General	115
8.5.3.2	Derivation process for motion vector components and reference indices	115
8.5.3.3	Decoding process for inter prediction samples	130
8.5.4	Decoding process for the residual signal of coding units coded in inter prediction mode	138
8.5.4.1	General	138
8.5.4.2	Decoding process for luma residual blocks	139
8.5.4.3	Decoding process for chroma residual blocks	140
8.6	Scaling, transformation and array construction process prior to deblocking filter process	141
8.6.1	Derivation process for quantization parameters	141
8.6.2	Scaling and transformation process	142
8.6.3	Scaling process for transform coefficients	143
8.6.4	Transformation process for scaled transform coefficients	144
8.6.4.1	General	144
8.6.4.2	Transformation process	144
8.6.5	Picture construction process prior to in-loop filter process	146
8.7	In-loop filter process	146
8.7.1	General	146
8.7.2	Deblocking filter process	147
8.7.2.1	General	147
8.7.2.2	Derivation process of transform block boundary	148
8.7.2.3	Derivation process of prediction block boundary	149
8.7.2.4	Derivation process of boundary filtering strength	150
8.7.2.5	Edge filtering process	151
8.7.3	Sample adaptive offset process	159
8.7.3.1	General	159
8.7.3.2	Coding tree block modification process	159

9	Parsing process	161
9.1	General	161
9.2	Parsing process for 0-th order Exp-Golomb codes	161
9.2.1	General	161
9.2.2	Mapping process for signed Exp-Golomb codes	163
9.3	CABAC parsing process for slice segment data	163
9.3.1	General	163
9.3.2	Initialization process	165
9.3.2.1	General	165
9.3.2.2	Initialization process for context variables	166
9.3.2.3	Storage process for context variables	174
9.3.2.4	Synchronization process for context variables	174
9.3.2.5	Initialization process for the arithmetic decoding engine	174
9.3.3	Binarization process	175
9.3.3.1	General	175
9.3.3.2	Truncated Rice (TR) binarization process	176
9.3.3.3	k-th order Exp-Golomb (EGk) binarization process	177
9.3.3.4	Fixed-length (FL) binarization process	178
9.3.3.5	Binarization process for part_mode	178
9.3.3.6	Binarization process for intra_chroma_pred_mode	178
9.3.3.7	Binarization process for inter_pred_idc	179
9.3.3.8	Binarization process for cu_qp_delta_abs	179
9.3.3.9	Binarization process for coeff_abs_level_remaining	179
9.3.4	Decoding process flow	180
9.3.4.1	General	180
9.3.4.2	Derivation process for ctxTable, ctxIdx and bypassFlag	180
9.3.4.3	Arithmetic decoding process	186
9.3.5	Arithmetic encoding process (informative)	192
9.3.5.1	General	192
9.3.5.2	Initialization process for the arithmetic encoding engine (informative)	192
9.3.5.3	Encoding process for a binary decision (informative)	193
9.3.5.4	Renormalization process in the arithmetic encoding engine (informative)	194
9.3.5.5	Bypass encoding process for binary decisions (informative)	195
9.3.5.6	Encoding process for a binary decision before termination (informative)	196
9.3.5.7	Byte stuffing process (informative)	197
10	Sub-bitstream extraction process	198
Annex A	Profiles, tiers and levels	199
A.1	Overview of profiles, tiers and levels	199
A.2	Requirements on video decoder capability	199
A.3	Profiles	199
A.3.1	General	199
A.3.2	Main profile	199
A.3.3	Main 10 profile	200
A.3.4	Main Still Picture profile	200
A.4	Tiers and levels	201
A.4.1	General tier and level limits	201
A.4.2	Profile-specific level limits for the Main and Main 10 profiles	202
A.4.3	Effect of level limits on picture rate for the Main and Main 10 profiles (informative)	204
Annex B	Byte stream format	208
B.1	General	208
B.2	Byte stream NAL unit syntax and semantics	208
B.2.1	Byte stream NAL unit syntax	208
B.2.2	Byte stream NAL unit semantics	208
B.3	Byte stream NAL unit decoding process	209
B.4	Decoder byte-alignment recovery (informative)	209
Annex C	Hypothetical reference decoder	210
C.1	General	210
C.2	Operation of coded picture buffer (CPB)	214
C.2.1	General	214
C.2.2	Timing of decoding unit arrival	214

C.2.3	Timing of decoding unit removal and decoding of decoding unit	216
C.3	Operation of the decoded picture buffer (DPB)	218
C.3.1	General	218
C.3.2	Removal of pictures from the DPB	218
C.3.3	Picture output	219
C.3.4	Current decoded picture marking and storage	220
C.4	Bitstream conformance	220
C.5	Decoder conformance	221
C.5.1	General	221
C.5.2	Operation of the output order DPB	222
C.5.2.1	General	222
C.5.2.2	Output and removal of pictures from the DPB	222
C.5.2.3	Picture decoding, marking, additional bumping, and storage	223
C.5.2.4	"Bumping" process	223
Annex D	Supplemental enhancement information	225
D.1	General	225
D.2	SEI payload syntax	226
D.2.1	General SEI message syntax	226
D.2.2	Buffering period SEI message syntax	228
D.2.3	Picture timing SEI message syntax	229
D.2.4	Pan-scan rectangle SEI message syntax	229
D.2.5	Filler payload SEI message syntax	230
D.2.6	User data registered by Rec. ITU-T T.35 SEI message syntax	230
D.2.7	User data unregistered SEI message syntax	230
D.2.8	Recovery point SEI message syntax	230
D.2.9	Scene information SEI message syntax	231
D.2.10	Picture snapshot SEI message syntax	231
D.2.11	Progressive refinement segment start SEI message syntax	231
D.2.12	Progressive refinement segment end SEI message syntax	231
D.2.13	Film grain characteristics SEI message syntax	232
D.2.14	Post-filter hint SEI message syntax	232
D.2.15	Tone mapping information SEI message syntax	233
D.2.16	Frame packing arrangement SEI message syntax	234
D.2.17	Display orientation SEI message syntax	234
D.2.18	Structure of pictures information SEI message syntax	235
D.2.19	Decoded picture hash SEI message syntax	235
D.2.20	Active parameter sets SEI message syntax	235
D.2.21	Decoding unit information SEI message syntax	236
D.2.22	Temporal sub-layer zero index SEI message syntax	236
D.2.23	Scalable nesting SEI message syntax	236
D.2.24	Region refresh information SEI message syntax	237
D.2.25	Reserved SEI message syntax	237
D.3	SEI payload semantics	237
D.3.1	General SEI payload semantics	237
D.3.2	Buffering period SEI message semantics	240
D.3.3	Picture timing SEI message semantics	242
D.3.4	Pan-scan rectangle SEI message semantics	247
D.3.5	Filler payload SEI message semantics	248
D.3.6	User data registered by Rec. ITU-T T.35 SEI message semantics	248
D.3.7	User data unregistered SEI message semantics	248
D.3.8	Recovery point SEI message semantics	248
D.3.9	Scene information SEI message semantics	249
D.3.10	Picture snapshot SEI message semantics	252
D.3.11	Progressive refinement segment start SEI message semantics	252
D.3.12	Progressive refinement segment end SEI message semantics	253
D.3.13	Film grain characteristics SEI message semantics	253
D.3.14	Post-filter hint SEI message semantics	258
D.3.15	Tone mapping information SEI message semantics	259
D.3.16	Frame packing arrangement SEI message semantics	263
D.3.17	Display orientation SEI message semantics	270
D.3.18	Structure of pictures information SEI message semantics	271
D.3.19	Decoded picture hash SEI message semantics	272

D.3.20	Active parameter sets SEI message semantics	273
D.3.21	Decoding unit information SEI message semantics	273
D.3.22	Temporal sub-layer zero index SEI message semantics	275
D.3.23	Scalable nesting SEI message semantics	275
D.3.24	Region refresh information SEI message semantics	276
D.3.25	Reserved SEI message semantics	277
Annex E	Video usability information	278
E.1	General	278
E.1	VUI syntax	279
E.1.1	VUI parameters syntax	279
E.1.2	HRD parameters syntax	281
E.1.3	Sub-layer HRD parameters syntax	282
E.2	VUI semantics	282
E.2.1	VUI parameters semantics	282
E.2.2	HRD parameters semantics	294
E.2.3	Sub-layer HRD parameters semantics	296
Bibliography		298

LIST OF FIGURES

Figure 6-1	– Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture	20
Figure 6-2	– Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture	20
Figure 6-3	– Nominal vertical and horizontal locations of 4:4:4 luma and chroma samples in a picture	21
Figure 6-4	– A picture with 11 by 9 luma coding tree blocks that is partitioned into two slices, the first of which is partitioned into three slice segments (informative)	22
Figure 6-5	– A picture with 11 by 9 luma coding tree blocks that is partitioned into two tiles and one slice (left) or is partitioned into two tiles and three slices (right) (informative)	22
Figure 7-1	– Structure of an access unit not containing any NAL units with nal_unit_type equal to FD_NUT, SUFFIX_SEI_NUT, VPS_NUT, SPS_NUT, PPS_NUT, RSV_VCL_N10, RSV_VCL_R11, RSV_VCL_N12, RSV_VCL_R13, RSV_VCL_N14, RSV_VCL_R15, RSV_IRAP_VCL22, or RSV_IRAP_VCL23, or in the range of RSV_VCL24..RSV_VCL31, RSV_NVCL42..RSV_NVCL47, or UNSPEC48..UNSPEC63	61
Figure 8-1	– Intra prediction mode directions (informative)	103
Figure 8-2	– Intra prediction angle definition (informative)	110
Figure 8-3	– Spatial motion vector neighbours (informative)	125
Figure 8-4	– Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for quarter sample luma interpolation	133
Figure 8-5	– Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for eighth sample chroma interpolation	135
Figure 9-1	– Illustration of CABAC parsing process for a syntax element synEl (informative)	164
Figure 9-2	– Spatial neighbour T that is used to invoke the coding tree block availability derivation process relative to the current coding tree block (informative)	165
Figure 9-3	– Illustration of CABAC initialization process (informative)	166
Figure 9-4	– Illustration of CABAC storage process (informative)	174
Figure 9-5	– Overview of the arithmetic decoding process for a single bin (informative)	187
Figure 9-6	– Flowchart for decoding a decision	188
Figure 9-7	– Flowchart of renormalization	190
Figure 9-8	– Flowchart of bypass decoding process	191
Figure 9-9	– Flowchart of decoding a decision before termination	192
Figure 9-10	– Flowchart for encoding a decision	194

Figure 9-11 – Flowchart of renormalization in the encoder	195
Figure 9-12 – Flowchart of PutBit(B)	195
Figure 9-13 – Flowchart of encoding bypass.....	196
Figure 9-14 – Flowchart of encoding a decision before termination	197
Figure 9-15 – Flowchart of flushing at termination.....	197
Figure C-1 – Structure of byte streams and NAL unit streams for HRD conformance checks	210
Figure C-2 – HRD buffer model.....	213
Figure D-1 – Nominal vertical and horizontal sampling locations of 4:2:0 samples in top and bottom fields.....	244
Figure D-2 – Nominal vertical and horizontal sampling locations of 4:2:2 samples in top and bottom fields.....	245
Figure D-3 – Nominal vertical and horizontal sampling locations of 4:4:4 samples in top and bottom fields.....	245
Figure D-4 – Rearrangement and upconversion of side-by-side packing arrangement with frame_packing_arrangement_type equal to 3, quincunx_sampling_flag equal to 0, and (x, y) equal to (0, 0) or (4, 8) for both constituent frames.....	267
Figure D-5 – Rearrangement and upconversion of side-by-side packing arrangement with frame_packing_arrangement_type equal to 3, quincunx_sampling_flag equal to 0, (x, y) equal to (12, 8) for constituent frame 0, and (x, y) equal to (0, 0) or (4, 8) for constituent frame 1.....	268
Figure D-6 – Rearrangement and upconversion of top-bottom packing arrangement with frame_packing_arrangement_type equal to 4, quincunx_sampling_flag equal to 0, and (x, y) equal to (0, 0) or (8, 4) for both constituent frames.....	268
Figure D-7 – Rearrangement and upconversion of top-bottom packing arrangement with frame_packing_arrangement_type equal to 4, quincunx_sampling_flag equal to 0, (x, y) equal to (8, 12) for constituent frame 0, and (x, y) equal to (0, 0) or (8, 4) for constituent frame 1.....	269
Figure D-8 – Rearrangement and upconversion of side-by-side packing arrangement with quincunx sampling (frame_packing_arrangement_type equal to 3 with quincunx_sampling_flag equal to 1).....	269
Figure D-9 – Rearrangement of a temporal interleaving frame arrangement (frame_packing_arrangement_type equal to 5).....	270
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	290

LIST OF TABLES

Table 5-1 – Operation precedence from highest (at top of table) to lowest (at bottom of table)	16
Table 6-1 – SubWidthC, and SubHeightC values derived from chroma_format_idc and separate_colour_plane_flag.....	19
Table 7-1 – NAL unit type codes and NAL unit type classes.....	56
Table 7-2 – Interpretation of pic_type	72
Table 7-3 – Specification of sizeId	75
Table 7-4 – Specification of matrixId according to sizeId, prediction mode and colour component	75
Table 7-5 – Specification of default values of ScalingList[0][matrixId][i] with i = 0..15	75
Table 7-6 – Specification of default values of ScalingList[1..3][matrixId][i] with i = 0..63	76
Table 7-7 – Name association to slice_type	77
Table 7-8 – Specification of the SAO type.....	86
Table 7-9 – Specification of the SAO edge offset class	87
Table 7-10 – Name association to prediction mode and partitioning type.....	89
Table 7-11 – Name association to inter prediction mode	90
Table 8-1 – Specification of intra prediction mode and associated names	103

Table 8-2 – Specification of IntraPredModeC.....	105
Table 8-3 – Specification of intraHorVerDistThres[nTbS] for various transform block sizes.....	108
Table 8-4 – Specification of intraPredAngle.....	110
Table 8-5 – Specification of invAngle.....	110
Table 8-6 – Specification of l0CandIdx and l1CandIdx.....	123
Table 8-7 – Assignment of the luma prediction sample predSampleLXL[xL, yL].....	134
Table 8-8 – Assignment of the chroma prediction sample predSampleLXC[xC, yC] for (X, Y) being replaced by (1, b), (2, c), (3, d), (4, e), (5, f), (6, g), and (7, h), respectively.....	136
Table 8-9 – Specification of QpC as a function of qPi.....	142
Table 8-10 – Name of association to edgeType.....	147
Table 8-11 – Derivation of threshold variables β' and t_c' from input Q.....	155
Table 8-12 – Specification of hPos and vPos according to the sample adaptive offset class.....	161
Table 9-1 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative).....	162
Table 9-2 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative).....	162
Table 9-3 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v).....	163
Table 9-4 – Association of ctxIdx and syntax elements for each initializationType in the initialization process.....	168
Table 9-5 – Values of initValue for ctxIdx of sao_merge_left_flag and sao_merge_up_flag.....	169
Table 9-6 – Values of initValue for ctxIdx of sao_type_idx_luma and sao_type_idx_chroma.....	169
Table 9-7 – Values of initValue for ctxIdx of split_cu_flag.....	169
Table 9-8 – Values of initValue for ctxIdx of cu_transquant_bypass_flag.....	169
Table 9-9 – Values of initValue for ctxIdx of cu_skip_flag.....	169
Table 9-10 – Values of initValue for ctxIdx of pred_mode_flag.....	169
Table 9-11 – Values of initValue for ctxIdx of part_mode.....	170
Table 9-12 – Values of initValue for ctxIdx of prev_intra_luma_pred_flag.....	170
Table 9-13 – Values of initValue for ctxIdx of intra_chroma_pred_mode.....	170
Table 9-14 – Values of initValue for ctxIdx of rqt_root_cbf.....	170
Table 9-15 – Value of initValue for ctxIdx of merge_flag.....	170
Table 9-16 – Values of initValue for ctxIdx of merge_idx.....	170
Table 9-17 – Values of initValue for ctxIdx of inter_pred_idc.....	171
Table 9-18 – Values of initValue for ctxIdx of ref_idx_l0 and ref_idx_l1.....	171
Table 9-19 – Values of initValue for ctxIdx of.mvp_l0_flag and.mvp_l1_flag.....	171
Table 9-20 – Values of initValue for ctxIdx of split_transform_flag.....	171
Table 9-21 – Values of initValue for ctxIdx of cbf_luma.....	171
Table 9-22 – Values of initValue for ctxIdx of cbf_cb and cbf_cr.....	171
Table 9-23 – Values of initValue for ctxIdx of abs_mvd_greater0_flag and abs_mvd_greater1_flag.....	172
Table 9-24 – Values of initValue for ctxIdx of cu_qp_delta_abs.....	172
Table 9-25 – Values of initValue for ctxIdx of transform_skip_flag.....	172
Table 9-26 – Values of initValue for ctxIdx of last_sig_coeff_x_prefix.....	172
Table 9-27 – Values of initValue for ctxIdx of last_sig_coeff_y_prefix.....	172
Table 9-28 – Values of initValue for ctxIdx of coded_sub_block_flag.....	173

Table 9-29 – Values of initValue for ctxIdx of sig_coeff_flag.....	173
Table 9-30 – Values of initValue for ctxIdx of coeff_abs_level_greater1_flag	173
Table 9-31 – Values of initValue for ctxIdx of coeff_abs_level_greater2_flag	173
Table 9-32 – Syntax elements and associated binarizations	175
Table 9-33 – Bin string of the unary binarization (informative).....	177
Table 9-34 – Binarization for part_mode	178
Table 9-35 – Binarization for intra_chroma_pred_mode	179
Table 9-36 – Binarization for inter_pred_idc	179
Table 9-37 – Assignment of ctxInc to syntax elements with context coded bins	181
Table 9-38 – Specification of ctxInc using left and above syntax elements	183
Table 9-39 – Specification of ctxIdxMap[i]	185
Table 9-40 – Specification of rangeTabLps depending on the values of pStateIdx and qRangeIdx	189
Table 9-41 – State transition table	190
Table A-1 – General tier and level limits	202
Table A-2 – Tier and level limits for the Main and Main 10 profiles.....	204
Table A-3 – Maximum picture rates (pictures per second) at level 1 to 4.3 for some example picture sizes when MinCbSizeY is equal to 64.....	205
Table A-4 – Maximum picture rates (pictures per second) at level 5 to 6.2 for some example picture sizes when MinCbSizeY is equal to 64.....	206
Table D-1 – Persistence scope of SEI messages (informative)	238
Table D-2 – Interpretation of pic_struct	244
Table D-3 – scene_transition_type values	251
Table D-4 – film_grain_model_id values	253
Table D-5 – blending_mode_id values	255
Table D-6 – filter_hint_type values	259
Table D-7 – Interpretation of camera_iso_speed_idc and exposure_index_idc	262
Table D-8 – Definition of frame_packing_arrangement_type.....	264
Table D-9 – Definition of content_interpretation_type	265
Table D-10 – Interpretation of hash_type.....	272
Table E-1 – Interpretation of sample aspect ratio indicator	283
Table E-2 – Meaning of video_format	284
Table E-3 – Colour primaries	285
Table E-4 – Transfer characteristics	286
Table E-5 – Matrix coefficients.....	289
Table E-6 – Divisor for computation of DpbOutputElementalInterval[n].....	296

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 23008-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in collaboration with ITU-T.

This part of ISO/IEC 23008 is technically aligned with Rec. ITU-T H.265 (04/2013) but is not published as identical text.

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

0 Introduction

0.1 General

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

0.2 Prologue

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

0.3 Purpose

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

0.4 Applications

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

- Broadcast (cable TV on optical networks / copper, satellite, terrestrial, etc.)
- Camcorders
- Content production and distribution
- Digital cinema
- Home cinema
- Internet streaming, download and play
- Medical imaging
- Mobile streaming, broadcast and communications
- Real-time conversational services (videoconferencing, videophone, telepresence, etc.)
- Remote video surveillance
- Storage media (optical disks, digital video tape recorder, etc.)
- Wireless display

0.5 Publication and versions of this Specification

This Specification has been jointly developed by ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). It is published as technically-aligned twin text in both ITU-T and ISO/IEC. As the basis text has been drafted to become both an ITU-T Recommendation and an ISO/IEC International Standard, the term "Specification" (with capitalization to indicate that it refers to the whole of the text) is used herein when the text refers to itself.

This is the first version of this Specification. Additional versions are anticipated.

0.6 Profiles, tiers and levels

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

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

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

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

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

0.7 Overview of the design characteristics

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

0.8 How to read this Specification

It is suggested that the reader starts with clause 1 (Scope) and moves on to clause 3 (Definitions). Clause 6 should be read for the geometrical relationship of the source, input, and output of the decoder. Clause 7 (Syntax and semantics) specifies the order to parse syntax elements from the bitstream. See subclauses 7.1–7.3 for syntactical order and see subclause 7.4 for semantics; e.g. the scope, restrictions, and conditions that are imposed on the syntax elements. The actual parsing for most syntax elements is specified in clause 9 (Parsing process). Clause 10 (Sub-bitstream extraction process) specifies the sub-bitstream extraction process. Finally, clause 8 (Decoding process) specifies how the syntax elements are mapped into decoded samples. Throughout reading this Specification, the reader should refer to clauses 2 (Normative references), 4 (Abbreviations), and 5 (Conventions) as needed. Annexes A through E also form an integral part of this Recommendation | International Standard.

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

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

Information technology — High efficiency coding and media delivery in heterogeneous environments —

Part 2: High efficiency video coding

1 Scope

This document specifies High efficiency video coding.

2 Normative references

2.1 General

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

2.2 Identical Recommendations | International Standards

- None.

2.3 Paired Recommendations | International Standards equivalent in technical content

- None

2.4 Additional references

- Rec. ITU-T T.35 (in force), *Procedure for the allocation of ITU-T defined codes for non-standard facilities*.
- ISO/IEC 11578: in force, *Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)*.
- ISO 11664-1: in force, *Colorimetry — Part 1: CIE standard colorimetric observers*.
- ISO 12232: in force, *Photography — Digital still cameras — Determination of exposure index, ISO speed ratings, standard output sensitivity, and recommended exposure index*.
- IETF RFC 1321 (in force), *The MD5 Message-Digest Algorithm*.

3 Definitions

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

- 3.1 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*.

NOTE 1 – In addition to containing the VCL NAL units of the coded picture, an access unit may also contain non-VCL NAL units. The decoding of an access unit always results in a decoded picture.

- 3.2 AC transform coefficient:** Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.

- 3.3 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*.

- 3.4 associated IRAP picture:** The previous *IRAP picture* in *decoding order* (when present).

- 3.5 associated VCL NAL unit:** The preceding *VCL NAL unit* in *decoding order* for a *non-VCL NAL unit* with *nal_unit_type* equal to EOS_NUT, EOB_NUT, FD_NUT, or SUFFIX_SEI_NUT, or in the ranges of

RSV_NVCL45..RSV_NVCL47 or UNSPEC56..UNSPEC63; or otherwise the next *VCL NAL unit* in *decoding order*.

- 3.6 **bin**: One bit of a *bin string*.
- 3.7 **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
- 3.8 **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
- 3.9 **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
- 3.10 **bi-predictive (B) slice**: A *slice* that may be decoded using *intra prediction* or *inter prediction* using at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.
- 3.11 **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 *CVSs*.
- 3.12 **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
- 3.13 **broken link**: A location in a *bitstream* at which it is indicated that some subsequent *pictures* in *decoding order* may contain serious visual artefacts due to unspecified operations performed in the generation of the *bitstream*.
- 3.14 **broken link access (BLA) access unit**: An *access unit* in which the *coded picture* is a *BLA picture*.
- 3.15 **broken link access (BLA) picture**: An *IRAP picture* for which each *VCL NAL unit* has *nal_unit_type* equal to BLA_W_LP, BLA_W_RADL, or BLA_N_LP.
- NOTE 2 – A BLA picture contains only I slices, and may be the first picture in the bitstream in decoding order, or may appear later in the bitstream. Each BLA picture begins a new CVS, and has the same effect on the decoding process as an IDR picture. However, a BLA picture contains syntax elements that specify a non-empty RPS. When a BLA picture for which each VCL NAL unit has *nal_unit_type* equal to BLA_W_LP, it may have associated RASL pictures, which are not output by the decoder and may not be decodable, as they may contain references to pictures that are not present in the bitstream. When a BLA picture for which each VCL NAL unit has *nal_unit_type* equal to BLA_W_LP, it may also have associated RADL pictures, which are specified to be decoded. When a BLA picture for which each VCL NAL unit has *nal_unit_type* equal to BLA_W_RADL, it does not have associated RASL pictures but may have associated RADL pictures. When a BLA picture for which each VCL NAL unit has *nal_unit_type* equal to BLA_N_LP, it does not have any associated leading pictures.
- 3.16 **buffering period**: The set of *access units* starting with an *access unit* that contains a buffering period SEI message and containing all subsequent *access units* in *decoding order* up to but not including the next *access unit* (when present) that contains a buffering period SEI message.
- 3.17 **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.
- 3.18 **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.
- 3.19 **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex B.
- 3.20 **can**: A term used to refer to behaviour that is allowed, but not necessarily required.
- 3.21 **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 3 – 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.
- 3.22 **clean random access (CRA) access unit**: An *access unit* in which the *coded picture* is a *CRA picture*.
- 3.23 **clean random access (CRA) picture**: An *IRAP picture* for which each *VCL NAL unit* has *nal_unit_type* equal to CRA_NUT.
- NOTE 4 – A CRA picture contains only I slices, 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 *NoRaslOutputFlag* 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.
- 3.24 **coded picture**: A *coded representation* of a *picture* containing all *coding tree units* of the *picture*.

- 3.25 coded picture buffer (CPB):** A first-in first-out buffer containing *decoding units* in *decoding order* specified in the *hypothetical reference decoder* in Annex C.
- 3.26 coded representation:** A data element as represented in its coded form.
- 3.27 coded slice segment NAL unit:** A *NAL unit* that has *nal_unit_type* in the range of *TRAIL_N* to *RASL_R*, inclusive, or in the range of *BLA_W_LP* to *RSV_IRAP_VCL23*, inclusive, which indicates that the *NAL unit* contains a coded *slice segment*.
- 3.28 coded video sequence (CVS):** A sequence of *access units* that consists, in decoding order, of an *IRAP access unit* with *NoRaslOutputFlag* equal to 1, followed by zero or more *access units* that are not *IRAP access units* with *NoRaslOutputFlag* equal to 1, including all subsequent *access units* up to but not including any subsequent *access unit* that is an *IRAP access unit* with *NoRaslOutputFlag* equal to 1.
- NOTE 5 – An *IRAP access unit* may be an *IDR access unit*, a *BLA access unit*, or a *CRA access unit*. The value of *NoRaslOutputFlag* is equal to 1 for each *IDR access unit*, each *BLA access unit*, and each *CRA access unit* that is the first *access unit* in the bitstream in decoding order, is the first *access unit* that follows an end of sequence *NAL unit* in decoding order, or has *HandleCraAsBlaFlag* equal to 1.
- 3.29 coding block:** An $N \times N$ *block* of samples for some value of N such that the division of a *coding tree block* into *coding blocks* is a *partitioning*.
- 3.30 coding tree block:** An $N \times N$ *block* of samples for some value of N such that the division of a *component* into *coding tree blocks* is a *partitioning*.
- 3.31 coding tree unit:** A *coding tree block* of *luma* samples, two corresponding *coding tree blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding tree block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.
- 3.32 coding unit:** A *coding block* of *luma* samples, two corresponding *coding blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *coding block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to code the samples.
- 3.33 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.
- 3.34 context variable:** A variable specified for the *adaptive binary arithmetic decoding process* of a *bin* by an equation containing recently decoded *bins*.
- 3.35 cropped decoded picture:** The result of cropping a *decoded picture* based on the conformance cropping window specified in the *SPS* that is referred to by the corresponding *coded picture*.
- 3.36 decoded picture:** A *decoded picture* is derived by decoding a *coded picture*.
- 3.37 decoded picture buffer (DPB):** A buffer holding *decoded pictures* for reference, output reordering, or output delay specified for the *hypothetical reference decoder* in Annex C.
- 3.38 decoder:** An embodiment of a *decoding process*.
- 3.39 decoder under test (DUT):** A *decoder* that is tested for conformance to this Specification by operating the *hypothetical stream scheduler* to deliver a conforming *bitstream* to the *decoder* and to the *hypothetical reference decoder* and comparing the values and timing or order of the output of the two *decoders*.
- 3.40 decoding order:** The order in which *syntax elements* are processed by the *decoding process*.
- 3.41 decoding process:** The process specified in this Specification that reads a *bitstream* and derives *decoded pictures* from it.
- 3.42 decoding unit:** An *access unit* if *SubPicHrdFlag* is equal to 0 or a subset of an *access unit* otherwise, consisting of one or more *VCL NAL units* in an *access unit* and the *associated non-VCL NAL units*.
- 3.43 dependent slice segment:** A *slice segment* for which the values of some *syntax elements* of the *slice segment header* are inferred from the values for the preceding *independent slice segment* in *decoding order*.
- 3.44 display process:** A process not specified in this Specification having, as its input, the *cropped decoded pictures* that are the output of the *decoding process*.
- 3.45 elementary stream:** A sequence of one or more *bitstreams*.
- NOTE 6 – An elementary stream that consists of two or more *bitstreams* would typically have been formed by splicing together two or more *bitstreams* (or parts thereof).

- 3.46 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*.
- 3.47 encoder:** An embodiment of an *encoding process*.
- 3.48 encoding process:** A process not specified in this Specification that produces a *bitstream* conforming to this Specification.
- 3.49 field:** An assembly of alternative rows of samples of a *frame*.
- 3.50 filler data NAL units:** *NAL units* with *nal_unit_type* equal to FD_NUT.
- 3.51 flag:** A variable that can take one of the two possible values 0 and 1.
- 3.52 frame:** The composition of a top *field* and a bottom *field*, where sample rows 0, 2, 4, ... originate from the top *field* and sample rows 1, 3, 5, ... originate from the bottom *field*.
- 3.53 frequency index:** A one-dimensional or two-dimensional index associated with a *transform coefficient* prior to an *inverse transform* part of the *decoding process*.
- 3.54 hypothetical reference decoder (HRD):** A hypothetical *decoder* model that specifies constraints on the variability of conforming *NAL unit streams* or conforming *byte streams* that an encoding process may produce.
- 3.55 hypothetical stream scheduler (HSS):** A hypothetical delivery mechanism used for checking the conformance of a *bitstream* or a *decoder* with regards to the timing and data flow of the input of a *bitstream* into the *hypothetical reference decoder*.
- 3.56 independent slice segment:** A *slice segment* for which the values of the *syntax elements* of the *slice segment header* are not inferred from the values for a preceding *slice segment*.
- 3.57 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.
- 3.58 instantaneous decoding refresh (IDR) access unit:** An *access unit* in which the *coded picture* is an *IDR picture*.
- 3.59 instantaneous decoding refresh (IDR) picture:** An *IRAP picture* for which each *VCL NAL unit* has *nal_unit_type* equal to IDR_W_RADL or IDR_N_LP.
- NOTE 7 – An IDR picture contains only I slices, 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 *nal_unit_type* equal to IDR_W_RADL, it may have associated RADL pictures. When an IDR picture for which each VCL NAL unit has *nal_unit_type* equal to IDR_N_LP, it does not have any associated leading pictures. An IDR picture does not have associated RASL pictures.
- 3.60 inter coding:** Coding of a *coding block*, *slice*, or *picture* that uses *inter prediction*.
- 3.61 inter prediction:** A *prediction* derived in a manner that is dependent on data elements (e.g. sample values or motion vectors) of *pictures* other than the current *picture*.
- 3.62 intra coding:** Coding of a *coding block*, *slice*, or *picture* that uses *intra prediction*.
- 3.63 intra prediction:** A *prediction* derived from only data elements (e.g. sample values) of the same decoded *slice*.
- 3.64 intra random access point (IRAP) access unit:** An *access unit* in which the *coded picture* is an *IRAP picture*.
- 3.65 intra random access point (IRAP) picture:** A *coded picture* for which each *VCL NAL unit* has *nal_unit_type* in the range of BLA_W_LP to RSV_IRAP_VCL23, inclusive.
- NOTE 8 – An IRAP picture contains only I slices, and may be a BLA picture, 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. There may be pictures in a bitstream that contain only I slices that are not IRAP pictures.
- 3.66 intra (I) slice:** A *slice* that is decoded using *intra prediction* only.
- 3.67 inverse transform:** A part of the *decoding process* by which a set of *transform coefficients* are converted into spatial-domain values.
- 3.68 layer:** A set of *VCL NAL units* that all have a particular value of *nuh_layer_id* and the associated non-*VCL NAL units*, or one of a set of syntactical structures having a hierarchical relationship.

NOTE 9 – Depending on the context, either the first layer concept or the second layer concept applies. The first layer concept is also referred to as a scalable layer, wherein a layer may be a spatial scalable layer, a quality scalable layer, a view, etc. A temporal true subset of a scalable layer is not referred to as a layer but referred to as a sub-layer or temporal sub-layer. The second layer concept is also referred to as a coding layer, wherein higher layers contain lower layers, and the coding layers are the CVS, picture, slice, slice segment, and coding tree unit layers.

- 3.69 layer identifier list:** A list of *nuh_layer_id* values that is associated with a *layer set* or an *operation point* and can be used as an input to the *sub-bitstream extraction process*.
- 3.70 layer set:** A set of *layers* represented within a *bitstream* created from another *bitstream* by operation of the *sub-bitstream extraction process* with the another *bitstream*, the target highest TemporalId equal to 6, and the target *layer identifier list* equal to the *layer identifier list* associated with the layer set as inputs.
- 3.71 leading picture:** A *picture* that precedes the *associated IRAP picture* in *output order*.
- 3.72 leaf:** A terminating node of a tree that is a root node of a tree of depth 0.
- 3.73 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 10 – 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.
- 3.74 list 0 (list 1) motion vector:** A *motion vector* associated with a *reference index* pointing into *reference picture list 0 (list 1)*.
- 3.75 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)*.
- 3.76 long-term reference picture:** A *picture* that is marked as "used for long-term reference".
- 3.77 long-term reference picture set:** The two RPS lists that may contain long-term reference pictures.
- 3.78 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 11 – 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.
- 3.79 may:** A term that is used to refer to behaviour that is allowed, but not necessarily required.
- NOTE 12 – 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.
- 3.80 motion vector:** A two-dimensional vector used for *inter prediction* that provides an offset from the coordinates in the *decoded picture* to the coordinates in a *reference picture*.
- 3.81 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).
- 3.82 nested SEI message:** An SEI message that is contained in a scalable nesting SEI message.
- 3.83 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*.
- 3.84 network abstraction layer (NAL) unit stream:** A sequence of *NAL units*.
- 3.85 non-nested SEI message:** An SEI message that is not contained in a scalable nesting SEI message.
- 3.86 non-reference picture:** A *picture* that is marked as "unused for reference".
- NOTE 13 – A non-reference picture contains samples that cannot be used for inter prediction in the decoding process of subsequent pictures in decoding order. In other words, once a picture is marked as "unused for reference", it can never be marked back as "used for reference".
- 3.87 non-VCL NAL unit:** A *NAL unit* that is not a *VCL NAL unit*.
- 3.88 note:** A term that is used to prefix *informative* remarks (used exclusively in an *informative* context).
- 3.89 operation point:** A *bitstream* created from another *bitstream* by operation of the *sub-bitstream extraction process* with the another *bitstream*, a target highest TemporalId, and a target *layer identifier list* as inputs.

NOTE 14 – If the target highest TemporalId of an operation point is equal to the greatest value of TemporalId in the layer set associated with the target layer identification list, the operation point is identical to the layer set. Otherwise it is a subset of the layer set.

- 3.90 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*).
- 3.91 parameter:** A *syntax element* of a *VPS*, *SPS* or *PPS*, or the second word of the defined term *quantization parameter*.
- 3.92 partitioning:** The division of a set into subsets such that each element of the set is in exactly one of the subsets.
- 3.93 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 15 – A picture may be either a frame or a field. However, in one CVS, either all pictures are frames or all pictures are fields.
- 3.94 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 segment header*.
- 3.95 picture order count:** A variable that is associated with each *picture*, uniquely identifies the associated *picture* among all *pictures* in the *CVS*, 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 *CVS* that are to be output from the *decoded picture buffer*.
- 3.96 prediction:** An embodiment of the *prediction process*.
- 3.97 prediction block:** A rectangular *MxN block* of samples on which the same *prediction* is applied.
- 3.98 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.
- 3.99 prediction unit:** A *prediction block* of *luma* samples, two corresponding *prediction blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *prediction block* of samples of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to predict the *prediction block* samples.
- 3.100 predictive (P) slice:** A *slice* that may be decoded using *intra prediction* or *inter prediction* using at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.
- 3.101 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.
- 3.102 prefix SEI message:** An SEI message that is contained in a *prefix SEI NAL unit*.
- 3.103 prefix SEI NAL unit:** An *SEI NAL unit* that has *nal_unit_type* equal to PREFIX_SEI_NUT.
- 3.104 profile:** A specified subset of the syntax of this Specification.
- 3.105 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.
- 3.106 quantization parameter:** A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
- 3.107 random access:** The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
- 3.108 random access decodable leading (RADL) access unit:** An *access unit* in which the *coded picture* is a *RADL picture*.
- 3.109 random access decodable leading (RADL) picture:** A *coded picture* for which each *VCL NAL unit* has *nal_unit_type* equal to RADL_R or RADL_N.
- NOTE 16 – 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.
- 3.110 random access skipped leading (RASL) access unit:** An *access unit* in which the *coded picture* is a *RASL picture*.
- 3.111 random access skipped leading (RASL) picture:** A *coded picture* for which each *VCL NAL unit* has *nal_unit_type* equal to RASL_R or RASL_N.

NOTE 17 – All RASL pictures are leading pictures of an associated BLA or CRA picture. When the associated IRAP picture has NoRaslOutputFlag 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 IRAP picture.

- 3.112 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.
- 3.113 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.
- 3.114 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*.
- 3.115 recovery point:** A point in the *bitstream* at which the recovery of an exact or an approximate representation of the *decoded pictures* represented by the *bitstream* is achieved after a *random access* or *broken link*.
- 3.116 reference index:** An index into a *reference picture list*.
- 3.117 reference picture:** A *picture* that is a *short-term reference picture* or a *long-term reference picture*.
NOTE 18 – A reference picture contains samples that may be used for inter prediction in the decoding process of subsequent pictures in decoding order.
- 3.118 reference picture list:** A list of *reference pictures* that is used for *inter prediction* of a *P* or *B* slice.
NOTE 19 – For the decoding process of a *P* slice, there is one reference picture list – reference picture list 0. For the decoding process of a *B* slice, there are two reference picture lists – reference picture list 0 and reference picture list 1.
- 3.119 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.
- 3.120 reference picture list 1:** The second *reference picture list* used for *inter prediction* of a *B* slice.
- 3.121 reference picture set (RPS):** A set of *reference pictures* associated with a *picture*, consisting of all *reference pictures* that are prior to the associated *picture* in decoding order, that may be used for *inter prediction* of the associated *picture* or any *picture* following the associated *picture* in decoding order.
NOTE 20 – The RPS of a picture consists of five RPS lists, three of which are to contain short-term reference pictures and the other two are to contain long-term reference pictures.
- 3.122 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.
- 3.123 residual:** The decoded difference between a *prediction* of a sample or data element and its decoded value.
- 3.124 sample aspect ratio:** The ratio between the intended horizontal distance between the columns and the intended vertical distance between the rows of the *luma* sample array in a *picture*, which is specified for assisting the *display process* (not specified in this Specification) and expressed as *h:v*, where *h* is the horizontal width and *v* is the vertical height, in arbitrary units of spatial distance.
- 3.125 scaling:** The process of multiplying *transform coefficient levels* by a factor, resulting in *transform coefficients*.
- 3.126 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 segment header*.
- 3.127 shall:** A term used to express mandatory requirements for conformance to this Specification.
NOTE 21 – 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.
- 3.128 short-term reference picture:** A *picture* that is marked as "used for short-term reference".
- 3.129 short-term reference picture set:** The three RPS lists that may contain short-term reference pictures.

- 3.130** **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.
- 3.131** **slice:** An integer number of *coding tree units* contained in one *independent slice segment* and all subsequent *dependent slice segments* (if any) that precede the next *independent slice segment* (if any) within the same *access unit*.
- 3.132** **slice header:** The *slice segment header* of the *independent slice segment* that is a current *slice segment* or the most recent *independent slice segment* that precedes a current *dependent slice segment* in *decoding order*.
- 3.133** **slice segment:** An integer number of *coding tree units* ordered consecutively in the *tile scan* and contained in a single *NAL unit*.
- 3.134** **slice segment header:** A part of a coded *slice segment* containing the data elements pertaining to the first or all *coding tree units* represented in the *slice segment*.
- 3.135** **source:** A term used to describe the video material or some of its attributes before encoding.
- 3.136** **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 22 – 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.
- 3.137** **step-wise temporal sub-layer access (STSA) access unit:** An *access unit* in which the *coded picture* is an *STSA picture*.
- 3.138** **step-wise temporal sub-layer access (STSA) picture:** A *coded picture* for which each *VCL NAL unit* has *nal_unit_type* equal to STSA_R or STSA_N.
NOTE 23 – 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.
- 3.139** **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.
- 3.140** **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 *layer identifier list*, are removed from the *bitstream*, with the output sub-bitstream consisting of the *NAL units* in the *bitstream* that belong to the target set.
- 3.141** **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*.
- 3.142** **sub-layer non-reference picture:** A *picture* that contains samples that cannot be used for *inter prediction* in the *decoding process* of subsequent *pictures* of the same *sub-layer* in *decoding order*.
NOTE 24 – Samples of a sub-layer non-reference picture may be used for inter prediction in the decoding process of subsequent pictures of higher sub-layers in decoding order.
- 3.143** **sub-layer reference picture:** A *picture* that contains samples that may be used for *inter prediction* in the *decoding process* of subsequent *pictures* of the same *sub-layer* in *decoding order*.
NOTE 25 – Samples of a sub-layer reference picture may also be used for inter prediction in the decoding process of subsequent pictures of higher sub-layers in decoding order.
- 3.144** **sub-layer representation:** A subset of the *bitstream* consisting of *NAL units* of a particular *sub-layer* and the lower *sub-layers*.
- 3.145** **suffix SEI message:** An SEI message that is contained in a *suffix SEI NAL unit*.
- 3.146** **suffix SEI NAL unit:** An *SEI NAL unit* that has *nal_unit_type* equal to SUFFIX_SEI_NUT.
- 3.147** **supplemental enhancement information (SEI) NAL unit:** A *NAL unit* that has *nal_unit_type* equal to PREFIX_SEI_NUT or SUFFIX_SEI_NUT.
- 3.148** **syntax element:** An element of data represented in the *bitstream*.
- 3.149** **syntax structure:** Zero or more *syntax elements* present together in the *bitstream* in a specified order.

- 3.150 temporal sub-layer access (TSA) access unit:** An *access unit* in which the *coded picture* is a *TSA picture*.
- 3.151 temporal sub-layer access (TSA) picture:** A *coded picture* for which each *VCL NAL unit* has *nal_unit_type* equal to *TSA_R* or *TSA_N*.
- NOTE 26 – A *TSA picture* and pictures following the *TSA picture* in decoding do not use pictures with *TemporalId* greater than or equal to that of the *TSA picture* for inter prediction reference. A *TSA picture* enables up-switching, at the *TSA picture*, to the sub-layer containing the *TSA picture* or any higher sub-layer, from the immediately lower sub-layer. *TSA pictures* must have *TemporalId* greater than 0.
- 3.152 temporal sub-layer:** A temporal scalable layer of a temporal scalable *bitstream*, consisting of *VCL NAL units* with a particular value of *TemporalId* and the associated *non-VCL NAL units*.
- 3.153 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.
- 3.154 tile:** A rectangular region of *coding tree blocks* within a particular *tile column* and a particular *tile row* in a *picture*.
- 3.155 tile column:** A rectangular region of *coding tree blocks* having a height equal to the height of the *picture* and a width specified by *syntax elements* in the *picture parameter set*.
- 3.156 tile row:** A rectangular region of *coding tree blocks* having a height specified by *syntax elements* in the *picture parameter set* and a width equal to the width of the *picture*.
- 3.157 tile scan:** A specific sequential ordering of *coding tree blocks partitioning a picture* in which the *coding tree blocks* are ordered consecutively in *coding tree block raster scan* in a *tile* whereas *tiles* in a *picture* are ordered consecutively in a *raster scan* of the *tiles* of the *picture*.
- 3.158 trailing picture:** A *picture* that follows the associated *IRAP picture* in *output order*.
- NOTE 27 – 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.
- 3.159 transform block:** A rectangular *MxN block* of samples on which the same *transform* is applied.
- 3.160 transform coefficient:** A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in an *inverse transform* part of the *decoding process*.
- 3.161 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.
- 3.162 transform unit:** A *transform block* of *luma* samples of size 8x8, 16x16, or 32x32 or four *transform blocks* of *luma* samples of size 4x4, two corresponding *transform blocks* of *chroma* samples of a *picture* that has three sample arrays, or a *transform block* of *luma* samples of size 8x8, 16x16, or 32x32 or four *transform blocks* of *luma* samples of size 4x4 of a monochrome *picture* or a *picture* that is coded using three separate colour planes and *syntax structures* used to transform the *transform block* samples.
- 3.163 tree:** A tree is a finite set of nodes with a unique root node.
- 3.164 universal unique identifier (UUID):** An identifier that is unique with respect to the space of all universal unique identifiers.
- 3.165 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.
- 3.166 video coding layer (VCL) NAL unit:** A collective term for *coded slice segment NAL units* and the subset of *NAL units* that have *reserved* values of *nal_unit_type* that are classified as *VCL NAL units* in this Specification.
- 3.167 video parameter set (VPS):** 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 *SPS* referred to by a *syntax element* found in the *PPS* referred to by a *syntax element* found in each *slice segment header*.
- 3.168 z-scan order:** A specified sequential ordering of *blocks partitioning a picture*, where the order is identical to *coding tree block raster scan* of the *picture* when the *blocks* are of the same size as *coding tree blocks*, and, when the *blocks* are of a smaller size than *coding tree blocks*, i.e. *coding tree blocks* are further partitioned into

smaller *coding blocks*, the order traverses from *coding tree block* to *coding tree block* in *coding tree block raster scan* of the *picture*, and inside each *coding tree block*, which may be divided into *quadrees* hierarchically to lower levels, the order traverses from *quadtree* to *quadtree* of a particular level in *quadtree-of-the-particular-level raster scan* of the *quadtree* of the immediately higher level.

4 Abbreviations

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

B	Bi-predictive
BLA	Broken Link Access
CABAC	Context-based Adaptive Binary Arithmetic Coding
CB	Coding Block
CBR	Constant Bit Rate
CRA	Clean Random Access
CPB	Coded Picture Buffer
CTB	Coding Tree Block
CTU	Coding Tree Unit
CU	Coding Unit
CVS	Coded Video Sequence
DPB	Decoded Picture Buffer
DUT	Decoder Under Test
EG	Exponential-Golomb
FIFO	First-In, First-Out
FIR	Finite Impulse Response
FL	Fixed-Length
GDR	Gradual Decoding Refresh
HRD	Hypothetical Reference Decoder
HSS	Hypothetical Stream Scheduler
I	Intra
IDR	Instantaneous Decoding Refresh
IRAP	Intra Random Access Point
LPS	Least Probable Symbol
LSB	Least Significant Bit
MPS	Most Probable Symbol
MSB	Most Significant Bit
NAL	Network Abstraction Layer
P	Predictive
PB	Prediction Block
PPS	Picture Parameter Set
PU	Prediction Unit
RADL	Random Access Decodable Leading (Picture)
RASL	Random Access Skipped Leading (Picture)

RBSP	Raw Byte Sequence Payload
RPS	Reference Picture Set
SEI	Supplemental Enhancement Information
SODB	String Of Data Bits
SPS	Sequence Parameter Set
STSA	Step-wise Temporal Sub-layer Access
TB	Transform Block
TR	Truncated Rice
TSA	Temporal Sub-layer Access
TU	Transform Unit
UUID	Universal Unique Identifier
VBR	Variable Bit Rate
VCL	Video Coding Layer
VPS	Video Parameter Set
VUI	Video Usability Information

5 Conventions

5.1 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.

5.2 Arithmetic operators

The following arithmetic operators are defined as follows:

+	Addition
–	Subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	Multiplication, including matrix multiplication
x^y	Exponentiation. Specifies x to the power of y . In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.
/	Integer division with truncation of the result toward zero. For example, $7 / 4$ and $-7 / -4$ are truncated to 1 and $-7 / 4$ and $7 / -4$ are truncated to -1 .
÷	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\frac{x}{y}$	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\sum_{i=x}^y f(i)$	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 $x \geq 0$ and $y > 0$.

5.3 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 .

5.4 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.

5.5 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.
$\wedge$	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.
$x \gg y$	Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y . Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.
$x \ll y$	Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y . Bits shifted into the LSBs as a result of the left shift have a value equal to 0.

5.6 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 $x = x - (-3)$.

5.7 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 .
------------	---

5.8 Mathematical functions

The following mathematical functions are defined:

$$\text{Abs}(x) = \begin{cases} x & ; \quad x \geq 0 \\ -x & ; \quad x < 0 \end{cases} \quad (5-1)$$

$$\text{Ceil}(x) \text{ the smallest integer greater than or equal to } x. \quad (5-2)$$

$$\text{Clip1}_Y(x) = \text{Clip3}(0, (1 \ll \text{BitDepth}_Y) - 1, x) \quad (5-3)$$

$$\text{Clip1}_C(x) = \text{Clip3}(0, (1 \ll \text{BitDepth}_C) - 1, x) \quad (5-4)$$

$$\text{Clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases} \quad (5-5)$$

$$\text{Floor}(x) \text{ the largest integer less than or equal to } x. \quad (5-6)$$

$$\text{Log2}(x) \text{ the base-2 logarithm of } x. \quad (5-7)$$

$$\text{Log10}(x) \text{ the base-10 logarithm of } x. \quad (5-8)$$

$$\text{Min}(x, y) = \begin{cases} x & ; \quad x \leq y \\ y & ; \quad x > y \end{cases} \quad (5-9)$$

$$\text{Max}(x, y) = \begin{cases} x & ; \quad x \geq y \\ y & ; \quad x < y \end{cases} \quad (5-10)$$

$$\text{Round}(x) = \text{Sign}(x) * \text{Floor}(\text{Abs}(x) + 0.5) \quad (5-11)$$

$$\text{Sign}(x) = \begin{cases} 1 & ; \quad x > 0 \\ 0 & ; \quad x = 0 \\ -1 & ; \quad x < 0 \end{cases} \quad (5-12)$$

$$\text{Sqrt}(x) = \sqrt{x} \quad (5-13)$$

$$\text{Swap}(x, y) = (y, x) \quad (5-14)$$

5.9 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)
x^y
"x * y", "x / y", "x ÷ y", " $\frac{x}{y}$ ", "x % y"
"x + y", "x - y" (as a two-argument operator), " $\sum_{i=x}^y f(i)$ "
"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"

5.10 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 subclause in which they are derived.

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in subclause 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 subclause 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 s_{yx} . 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 $\{ \{ \dots \} \{ \dots \} \}$, 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.

5.11 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
```

5.12 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.

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

6.1 Bitstream formats

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

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

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

6.2 Source, decoded, and output picture formats

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

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

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

- Luma (Y) only (monochrome).

- Luma and two chroma (YCbCr or YCgCo).
- Green, Blue and Red (GBR, also known as RGB).
- Arrays representing other unspecified monochrome or tri-stimulus colour samplings (for example, YZX, also known as XYZ).

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

The variables SubWidthC, and SubHeightC are specified in Table 6-1, depending on the chroma format sampling structure, which is specified through chroma_format_idc and separate_colour_plane_flag. 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.

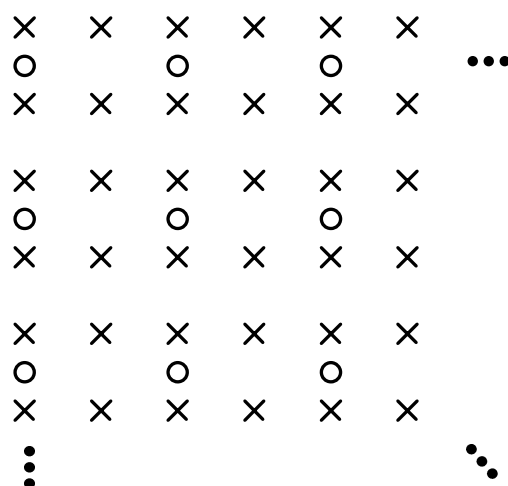
In 4:2:2 sampling, each of the two chroma arrays has the same height and half the width of the luma array.

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 14, 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).

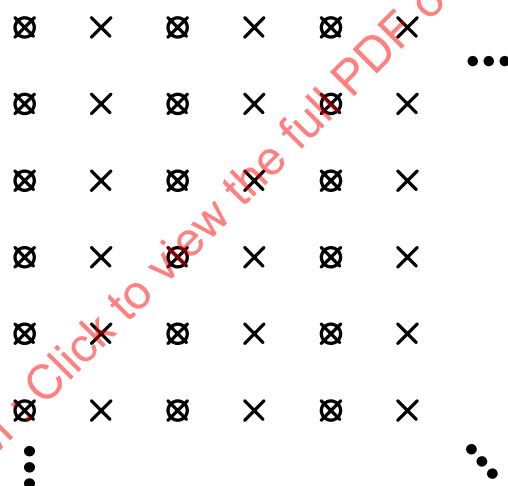


Guide:

- × = Location of luma sample
- = Location of chroma sample

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.



Guide:

- × = Location of luma sample
- = Location of chroma sample

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.

NOTE 2 – When `separate_colour_plane_flag` is equal to 0, each coding tree block of a picture is contained in exactly one slice. When `separate_colour_plane_flag` is equal to 1, each coding tree block of a colour component is contained in exactly one slice (i.e. information for each coding tree block of a picture is present in exactly three slices and these three slices have different values of `colour_plane_id`).

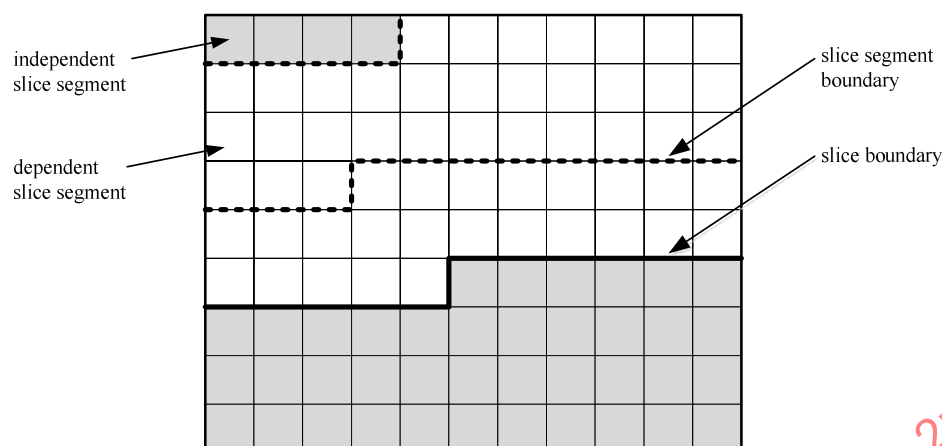


Figure 6-4 – A picture with 11 by 9 luma coding tree blocks that is partitioned into two slices, the first of which is partitioned into three slice segments (informative)

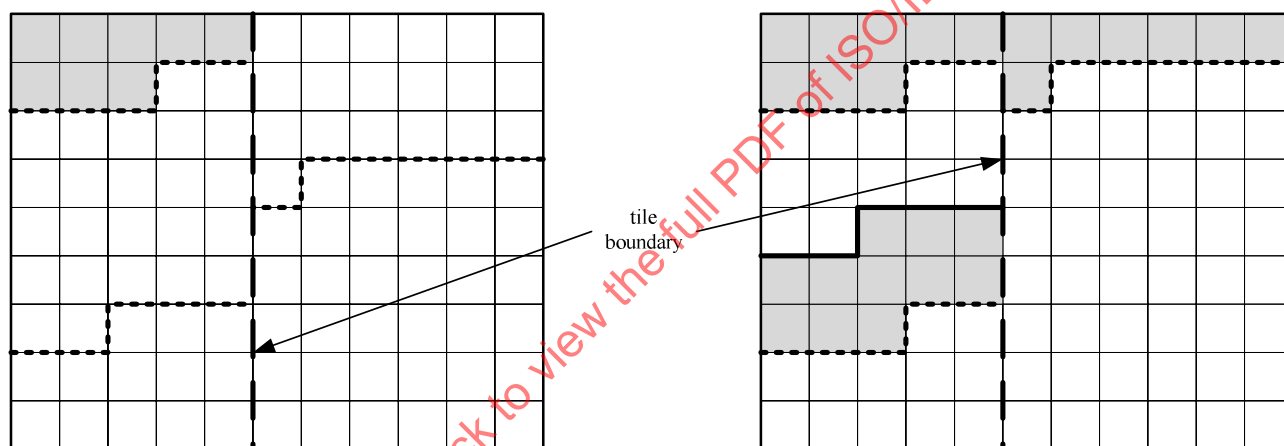


Figure 6-5 – A picture with 11 by 9 luma coding tree blocks that is partitioned into two tiles and one slice (left) or is partitioned into two tiles and three slices (right) (informative)

6.3.2 Block and quadtree structures

The samples are processed in units of coding tree blocks. The array size for each luma coding tree block in both width and height is `CtbSizeY` in units of samples. The width and height of the array for each chroma coding tree block are `CtbWidthC` and `CtbHeightC`, respectively, in units of samples.

Each coding tree block is assigned a partition signalling to identify the block sizes for intra or inter prediction and for transform coding. The partitioning is a recursive quadtree partitioning. The root of the quadtree is associated with the coding tree block. The quadtree is split until a leaf is reached, which is referred to as the coding block. When the component width is not an integer number of the coding tree block size, the coding tree blocks at the right component boundary are incomplete. When the component height is not an integer multiple of the coding tree block size, the coding tree blocks 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 encapsulated in a "unit" as follows:

- One prediction block (monochrome picture or `separate_colour_plane_flag` is equal to 1) or three prediction blocks (luma and chroma) and associated prediction syntax structures units are encapsulated in a prediction unit.
- One transform block (monochrome picture or `separate_colour_plane_flag` is equal to 1) or three transform blocks (luma and chroma) and associated transform syntax structures units are encapsulated in a transform unit.
- One coding block (monochrome picture or `separate_colour_plane_flag` is equal to 1) or three coding blocks (luma and chroma), the associated coding syntax structures and the associated prediction and transform units are encapsulated in a coding unit.
- One coding tree block (monochrome picture or `separate_colour_plane_flag` is equal to 1) or three coding tree blocks (luma and chroma), the associated coding tree syntax structures and the associated coding units are encapsulated in a coding tree unit.

6.3.3 Spatial or component-wise partitionings

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

- The division of each picture into components,
- The division of each component into coding tree blocks,
- 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 coding tree units,
- The division of each picture into slices,
- The division of each slice into slice segments,
- The division of each slice segment into coding tree units,
- The division of each coding tree unit into coding tree blocks,
- The division of each coding tree block into coding blocks, except that the coding tree blocks are incomplete at the right component boundary when the component width is not an integer multiple of the coding tree block size and the coding tree blocks are incomplete at the bottom component boundary when the component height is not an integer multiple of the coding tree block size,
- The division of each coding tree unit into coding units, except that the coding tree units are incomplete at the right picture boundary when the picture width in luma samples is not an integer multiple of the luma coding tree block size and the coding tree units are incomplete at the bottom picture boundary when the picture height in luma samples is not an integer multiple of the luma coding tree block size,
- The division of each coding unit into prediction units,
- 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 prediction blocks,
- The division of each coding block into transform blocks,
- The division of each prediction unit into prediction blocks,
- The division of each transform unit into transform blocks.

6.4 Availability processes

6.4.1 Derivation process for z-scan order 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 (x_{NbY} , y_{NbY}) 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 (x_{NbY} , y_{NbY}), denoted as $availableN$.

The minimum luma block address in z-scan order $minBlockAddrCurr$ of the current block is derived as follows:

$$minBlockAddrCurr = MinTbAddrZs[xCurr \gg \text{Log2MinTrafoSize}][yCurr \gg \text{Log2MinTrafoSize}] \quad (6-1)$$

The minimum luma block address in z-scan order $minBlockAddrN$ of the neighbouring block covering the location (x_{NbY} , y_{NbY}) is derived as follows:

- If one or more of the following conditions are true, $minBlockAddrN$ is set equal to -1 :

- x_{NbY} is less than 0
- y_{NbY} is less than 0
- x_{NbY} is greater than or equal to $pic_width_in_luma_samples$
- y_{NbY} is greater than or equal to $pic_height_in_luma_samples$

- Otherwise (x_{NbY} and y_{NbY} are inside the picture boundaries),

$$minBlockAddrN = MinTbAddrZs[x_{NbY} \gg \text{Log2MinTrafoSize}][y_{NbY} \gg \text{Log2MinTrafoSize}] \quad (6-2)$$

The neighbouring block availability $availableN$ is derived as follows:

- If one or more of the following conditions are true, $availableN$ is set equal to FALSE:
 - $minBlockAddrN$ is less than 0,
 - $minBlockAddrN$ is greater than $minBlockAddrCurr$,
 - the variable $SliceAddrRs$ associated with the slice segment containing the neighbouring block with the minimum luma block address $minBlockAddrN$ differs in value from the variable $SliceAddrRs$ associated with the slice segment containing the current block with the minimum luma block address $minBlockAddrCurr$.
 - the neighbouring block with the minimum luma block address $minBlockAddrN$ is contained in a different tile than the current block with the minimum luma block address $minBlockAddrCurr$.
- Otherwise, $availableN$ is set equal to TRUE.

6.4.2 Derivation process for prediction block availability

Inputs to this process are:

- the luma location (x_{Cb} , y_{Cb}) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a variable $nCbS$ specifying the size of the current luma coding block,
- the luma location (x_{Pb} , y_{Pb}) of the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- two variables $nPbW$ and $nPbH$ specifying the width and the height of the current luma prediction block,
- a variable $partIdx$ specifying the partition index of the current prediction unit within the current coding unit,
- the luma location (x_{NbY} , y_{NbY}) covered by a neighbouring prediction block relative to the top-left luma sample of the current picture.

Output of this process is the availability of the neighbouring prediction block covering the location (x_{NbY} , y_{NbY}), denoted as $availableN$ is derived as follows:

The variable $sameCb$ specifying whether the current luma prediction block and the neighbouring luma prediction block cover the same luma coding block.

- If all of the following conditions are true, $sameCb$ is set equal to TRUE:
 - x_{Cb} is less than or equal than x_{NbY} ,
 - y_{Cb} is less than or equal than y_{NbY} ,
 - ($x_{Cb} + nCbS$) is greater than x_{NbY} ,

- ($y_{Cb} + n_{CbS}$) is greater than y_{NbY} .
- Otherwise, sameCb is set equal to FALSE.

The neighbouring prediction block availability availableN is derived as follows:

- If sameCb is equal to FALSE, the derivation process for z-scan order block availability as specified in subclause 6.4.1 is invoked with (x_{Curr} , y_{Curr}) set equal to (x_{Pb} , y_{Pb}) and the luma location (x_{NbY} , y_{NbY}) as inputs, and the output is assigned to availableN.
- Otherwise, if all of the following conditions are true, availableN is set equal to FALSE:
 - ($n_{PbW} << 1$) is equal to n_{CbS} ,
 - ($n_{PbH} << 1$) is equal to n_{CbS} ,
 - partIdx is equal to 1,
 - ($y_{Cb} + n_{PbH}$) is less than or equal to y_{NbY} ,
 - ($x_{Cb} + n_{PbW}$) is greater than x_{NbY} .
- Otherwise, availableN is set equal to TRUE.

When availableN is equal to TRUE and $CuPredMode[x_{NbY}][y_{NbY}]$ is equal to MODE_INTRA, availableN is set equal to FALSE.

6.5 Scanning processes

6.5.1 Coding tree block raster and tile scanning conversion 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 as follows:

```

if( uniform_spacing_flag )
    for( i = 0; i <= num_tile_columns_minus1; i++ )
        colWidth[ i ] = ( ( i + 1 ) * PicWidthInCtbsY ) / ( num_tile_columns_minus1 + 1 ) -
            ( i * PicWidthInCtbsY ) / ( num_tile_columns_minus1 + 1 )
else {
    colWidth[ num_tile_columns_minus1 ] = PicWidthInCtbsY
    for( i = 0; i < num_tile_columns_minus1; i++ ) {
        colWidth[ i ] = column_width_minus1[ i ] + 1
        colWidth[ num_tile_columns_minus1 ] -= colWidth[ i ]
    }
}

```

(6-3)

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 as follows:

```

if( uniform_spacing_flag )
    for( j = 0; j <= num_tile_rows_minus1; j++ )
        rowHeight[ j ] = ( ( j + 1 ) * PicHeightInCtbsY ) / ( num_tile_rows_minus1 + 1 ) -
            ( j * PicHeightInCtbsY ) / ( num_tile_rows_minus1 + 1 )
else {
    rowHeight[ num_tile_rows_minus1 ] = PicHeightInCtbsY
    for( j = 0; j < num_tile_rows_minus1; j++ ) {
        rowHeight[ j ] = row_height_minus1[ j ] + 1
        rowHeight[ num_tile_rows_minus1 ] -= rowHeight[ j ]
    }
}

```

(6-4)

The list colBd[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 coding tree blocks, is derived as follows:

```

for( colBd[ 0 ] = 0, i = 0; i <= num_tile_columns_minus1; i++ )
    colBd[ i + 1 ] = colBd[ i ] + colWidth[ i ]

```

(6-5)

The list rowBd[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 coding tree blocks, is derived as follows:

```

for( rowBd[ 0 ] = 0; j = 0; j <= num_tile_rows_minus1; j++ )
    rowBd[ j + 1 ] = rowBd[ j ] + rowHeight[ j ]

```

(6-6)

The list CtbAddrRsToTs[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 tile scan, is derived as follows:

```

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ ) {
    tbX = ctbAddrRs % PicWidthInCtbsY
    tbY = ctbAddrRs / PicWidthInCtbsY
    for( i = 0; i <= num_tile_columns_minus1; i++ )
        if( tbX >= colBd[ i ] )
            tileX = i
    for( j = 0; j <= num_tile_rows_minus1; j++ )
        if( tbY >= rowBd[ j ] )
            tileY = j
    CtbAddrRsToTs[ ctbAddrRs ] = 0
    for( i = 0; i < tileX; i++ )
        CtbAddrRsToTs[ ctbAddrRs ] += rowHeight[ tileY ] * colWidth[ i ]
    for( j = 0; j < tileY; j++ )
        CtbAddrRsToTs[ ctbAddrRs ] += PicWidthInCtbsY * rowHeight[ j ]
    CtbAddrRsToTs[ ctbAddrRs ] += ( tbY - rowBd[ tileY ] ) * colWidth[ tileX ] + tbX - colBd[ tileX ]
}

```

(6-7)

The list CtbAddrTsToRs[ctbAddrTs] for ctbAddrTs ranging from 0 to PicSizeInCtbsY – 1, inclusive, specifying the conversion from a CTB address in tile scan to a CTB address in CTB raster scan of a picture, is derived as follows:

```

for( ctbAddrRs = 0; ctbAddrRs < PicSizeInCtbsY; ctbAddrRs++ )
    CtbAddrTsToRs[ CtbAddrRsToTs[ ctbAddrRs ] ] = ctbAddrRs

```

(6-8)

The list TileId[ctbAddrTs] for ctbAddrTs ranging from 0 to PicSizeInCtbsY – 1, inclusive, specifying the conversion from a CTB address in tile scan to a tile ID, is derived as follows:

```

for( j = 0; tileIdx = 0; j <= num_tile_rows_minus1; j++ )
    for( i = 0; i <= num_tile_columns_minus1; i++, tileIdx++ )
        for( y = rowBd[ j ]; y < rowBd[ j + 1 ]; y++ )
            for( x = colBd[ i ]; x < colBd[ i + 1 ]; x++ )
                TileId[ CtbAddrRsToTs[ y * PicWidthInCtbsY + x ] ] = tileIdx

```

(6-9)

The values of ColumnWidthInLumaSamples[i], specifying the width of the i-th tile column in units of luma samples, are set equal to colWidth[i] << CtbLog2SizeY for i ranging from 0 to num_tile_columns_minus1, inclusive.

The values of RowHeightInLumaSamples[j], specifying the height of the j-th tile row in units of luma samples, are set equal to rowHeight[j] << CtbLog2SizeY for j ranging from 0 to num_tile_rows_minus1, inclusive.

6.5.2 Z-scan order array initialization process

The array MinTbAddrZs with elements MinTbAddrZs[x][y] for x ranging from 0 to (PicWidthInCtbsY << (CtbLog2SizeY – Log2MinTrafoSize)) – 1, inclusive, and y ranging from 0 to (PicHeightInCtbsY << (CtbLog2SizeY – Log2MinTrafoSize)) – 1, specifying the conversion from a location (x, y) in units of minimum blocks to a minimum block address in z-scan order, inclusive is derived as follows:

```

for( y = 0; y < ( PicHeightInCtbsY << ( CtbLog2SizeY – Log2MinTrafoSize ) ); y++ )
    for( x = 0; x < ( PicWidthInCtbsY << ( CtbLog2SizeY – Log2MinTrafoSize ) ); x++ ) {
        tbX = ( x << Log2MinTrafoSize ) >> CtbLog2SizeY
        tbY = ( y << Log2MinTrafoSize ) >> CtbLog2SizeY
        ctbAddrRs = PicWidthInCtbsY * tbY + tbX
        MinTbAddrZs[ x ][ y ] = CtbAddrRsToTs[ ctbAddrRs ] <<
            ( ( CtbLog2SizeY – Log2MinTrafoSize ) * 2 )
        for( i = 0, p = 0; i < ( CtbLog2SizeY – Log2MinTrafoSize ); i++ ) {
            m = 1 << i
            p += ( m & x ? m * m : 0 ) + ( m & y ? 2 * m * m : 0 )
        }
        MinTbAddrZs[ x ][ y ] += p
    }
}

```

(6-10)

6.5.3 Up-right diagonal scan order array initialization process

Input to this process is a block size blkSize.

Output of this process is the array `diagScan[ sPos ][ sComp ]`. The array index `sPos` specify the scan position ranging from 0 to $(\text{blkSize} * \text{blkSize}) - 1$. The array index `sComp` equal to 0 specifies the horizontal component and the array index `sComp` equal to 1 specifies the vertical component. Depending on the value of `blkSize`, the array `diagScan` is derived as follows:

```

i = 0
x = 0
y = 0
stopLoop = FALSE
while( !stopLoop ) {
    while( y >= 0 ) {
        if( x < blkSize && y < blkSize ) {
            diagScan[ i ][ 0 ] = x
            diagScan[ i ][ 1 ] = y
            i++
        }
        y--
        x++
    }
    y = x
    x = 0
    if( i >= blkSize * blkSize )
        stopLoop = TRUE
}

```

(6-11)

6.5.4 Horizontal scan order array initialization process

Input to this process is a block size `blkSize`.

Output of this process is the array `horScan[ sPos ][ sComp ]`. The array index `sPos` specifies the scan position ranging from 0 to $(\text{blkSize} * \text{blkSize}) - 1$. The array index `sComp` equal to 0 specifies the horizontal component and the array index `sComp` equal to 1 specifies the vertical component. Depending on the value of `blkSize`, the array `horScan` is derived as follows:

```

i = 0
for( y = 0; y < blkSize; y++ )
    for( x = 0; x < blkSize; x++ ) {
        horScan[ i ][ 0 ] = x
        horScan[ i ][ 1 ] = y
        i++
    }

```

(6-12)

6.5.5 Vertical scan order array initialization process

Input to this process is a block size `blkSize`.

Output of this process is the array `verScan[ sPos ][ sComp ]`. The array index `sPos` specifies the scan position ranging from 0 to $(\text{blkSize} * \text{blkSize}) - 1$. The array index `sComp` equal to 0 specifies the horizontal component and the array index `sComp` equal to 1 specifies the vertical component. Depending on the value of `blkSize`, the array `verScan` is derived as follows:

```

i = 0
for( x = 0; x < blkSize; x++ )
    for( y = 0; y < blkSize; y++ ) {
        verScan[ i ][ 0 ] = x
        verScan[ i ][ 1 ] = y
        i++
    }

```

(6-13)

7 Syntax and semantics

7.1 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(v)
conditioning statement	
/* A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. */	
{	
statement	
statement	
...	
}	
/* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true */	
while(condition)	
statement	
/* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true */	
do	
statement	
while(condition)	
/* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed */	
if(condition)	
primary statement	
else	
alternative statement	
/* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. */	
for(initial statement; condition; subsequent statement)	
primary statement	

7.2 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 $8 * \text{payloadSize}$ 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 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.

`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 $8 * \text{payloadSize}$ bits from the beginning of the syntax structure (i.e. the position of the `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.

`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.

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

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

- `ae(v)`: context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in subclause 9.3.
- `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 )`.
- `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 subclause 9.2.
- `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 subclause 9.2.

7.3 Syntax in tabular form

7.3.1 NAL unit syntax

7.3.1.1 General NAL unit syntax

<code>nal_unit(NumBytesInNalUnit) {</code>	Descriptor
<code>nal_unit_header()</code>	
<code>NumBytesInRbsp = 0</code>	
<code>for(i = 2; i < NumBytesInNalUnit; i++)</code>	
<code>if(i + 2 < NumBytesInNalUnit && next_bits(24) == 0x000003) {</code>	
<code> rbsp_byte[NumBytesInRbsp++]</code>	<code>b(8)</code>
<code> rbsp_byte[NumBytesInRbsp++]</code>	<code>b(8)</code>
<code> i += 2</code>	
<code> emulation_prevention_three_byte /* equal to 0x03 */</code>	<code>f(8)</code>
<code>} else</code>	
<code> rbsp_byte[NumBytesInRbsp++]</code>	<code>b(8)</code>
<code>}</code>	

7.3.1.2 NAL unit header syntax

<code>nal_unit_header() {</code>	Descriptor
<code>forbidden_zero_bit</code>	<code>f(1)</code>
<code>nal_unit_type</code>	<code>u(6)</code>
<code>nuh_layer_id</code>	<code>u(6)</code>
<code>nuh_temporal_id_plus1</code>	<code>u(3)</code>
<code>}</code>	

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

7.3.2.1 Video parameter set RBSP syntax

video_parameter_set_rbsp() {	Descriptor
vps_video_parameter_set_id	u(4)
vps_reserved_three_2bits	u(2)
vps_max_layers_minus1	u(6)
vps_max_sub_layers_minus1	u(3)
vps_temporal_id_nesting_flag	u(1)
vps_reserved_0xffff_16bits	u(16)
profile_tier_level(vps_max_sub_layers_minus1)	
vps_sub_layer_ordering_info_present_flag	u(1)
for(i = (vps_sub_layer_ordering_info_present_flag ? 0 : vps_max_sub_layers_minus1); i <= vps_max_sub_layers_minus1; i++) {	
vps_max_dec_pic_buffering_minus1 [i]	ue(v)
vps_max_num_reorder_pics [i]	ue(v)
vps_max_latency_increase_plus1 [i]	ue(v)
}	
vps_max_layer_id	u(6)
vps_num_layer_sets_minus1	ue(v)
for(i = 1; i <= vps_num_layer_sets_minus1; i++)	
for(j = 0; j <= vps_max_layer_id; j++)	
layer_id_included_flag [i][j]	u(1)
vps_timing_info_present_flag	u(1)
if(vps_timing_info_present_flag) {	
vps_num_units_in_tick	u(32)
vps_time_scale	u(32)
vps_poc_proportional_to_timing_flag	u(1)
if(vps_poc_proportional_to_timing_flag)	
vps_num_ticks_poc_diff_one_minus1	ue(v)
vps_num_hrd_parameters	ue(v)
for(i = 0; i < vps_num_hrd_parameters; i++) {	
hrd_layer_set_idx [i]	ue(v)
if(i > 0)	
cprms_present_flag [i]	u(1)
hrd_parameters(cprms_present_flag[i], vps_max_sub_layers_minus1)	
}	
}	
vps_extension_flag	u(1)
if(vps_extension_flag)	
while(more_rbsp_data())	
vps_extension_data_flag	u(1)
rbsp_trailing_bits()	
}	

7.3.2.2 Sequence parameter set RBSP syntax

seq_parameter_set_rbsp() {	Descriptor
sps_video_parameter_set_id	u(4)
sps_max_sub_layers_minus1	u(3)
sps_temporal_id_nesting_flag	u(1)
profile_tier_level(sps_max_sub_layers_minus1)	
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)
}	
log2_min_luma_coding_block_size_minus3	ue(v)
log2_diff_max_min_luma_coding_block_size	ue(v)
log2_min_transform_block_size_minus2	ue(v)
log2_diff_max_min_transform_block_size	ue(v)
max_transform_hierarchy_depth_inter	ue(v)
max_transform_hierarchy_depth_intra	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()	
}	
amp_enabled_flag	u(1)
sample_adaptive_offset_enabled_flag	u(1)
pcm_enabled_flag	u(1)
if(pcm_enabled_flag) {	
pcm_sample_bit_depth_luma_minus1	u(4)
pcm_sample_bit_depth_chroma_minus1	u(4)
log2_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)
}	
num_short_term_ref_pic_sets	ue(v)
for(i = 0; i < num_short_term_ref_pic_sets; i++)	
short_term_ref_pic_set(i)	
long_term_ref_pics_present_flag	u(1)
if(long_term_ref_pics_present_flag) {	
num_long_term_ref_pics_sps	ue(v)
for(i = 0; i < num_long_term_ref_pics_sps; i++) {	
lt_ref_pic_poc_lsb_sps[i]	u(v)
used_by_curr_pic_lt_sps_flag[i]	u(1)
}	
}	
sps_temporal_mvp_enabled_flag	u(1)
strong_intra_smoothing_enabled_flag	u(1)
vui_parameters_present_flag	u(1)
if(vui_parameters_present_flag)	
vui_parameters()	
sps_extension_flag	u(1)
if(sps_extension_flag)	
while(more_rbsp_data())	
sps_extension_data_flag	u(1)
rbsp_trailing_bits()	
}	

7.3.2.3 Picture parameter set RBSP syntax

pic_parameter_set_rbsp() {	Descriptor
pps_pic_parameter_set_id	ue(v)
pps_seq_parameter_set_id	ue(v)
dependent_slice_segments_enabled_flag	u(1)
output_flag_present_flag	u(1)
num_extra_slice_header_bits	u(3)
sign_data_hiding_enabled_flag	u(1)
cabac_init_present_flag	u(1)
num_ref_idx_l0_default_active_minus1	ue(v)
num_ref_idx_l1_default_active_minus1	ue(v)
init_qp_minus26	se(v)
constrained_intra_pred_flag	u(1)
transform_skip_enabled_flag	u(1)
cu_qp_delta_enabled_flag	u(1)
if(cu_qp_delta_enabled_flag)	
diff_cu_qp_delta_depth	ue(v)
pps_cb_qp_offset	se(v)
pps_cr_qp_offset	se(v)
pps_slice_chroma_qp_offsets_present_flag	u(1)
weighted_pred_flag	u(1)
weighted_bipred_flag	u(1)
transquant_bypass_enabled_flag	u(1)
tiles_enabled_flag	u(1)
entropy_coding_sync_enabled_flag	u(1)
if(tiles_enabled_flag) {	
num_tile_columns_minus1	ue(v)
num_tile_rows_minus1	ue(v)
uniform_spacing_flag	u(1)
if(!uniform_spacing_flag) {	
for(i = 0; i < num_tile_columns_minus1; i++)	
column_width_minus1[i]	ue(v)
for(i = 0; i < num_tile_rows_minus1; i++)	
row_height_minus1[i]	ue(v)
}	
loop_filter_across_tiles_enabled_flag	u(1)
}	
pps_loop_filter_across_slices_enabled_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_scaling_list_data_present_flag	u(1)
if(pps_scaling_list_data_present_flag)	
scaling_list_data()	
lists_modification_present_flag	u(1)
log2_parallel_merge_level_minus2	ue(v)
slice_segment_header_extension_present_flag	u(1)
pps_extension_flag	u(1)
if(pps_extension_flag)	
while(more_rbsp_data())	
pps_extension_data_flag	u(1)
rbsp_trailing_bits()	
}	

7.3.2.4 Supplemental enhancement information RBSP syntax

sei_rbsp() {	Descriptor
do	
sei_message()	
while(more_rbsp_data())	
rbsp_trailing_bits()	
}	

7.3.2.5 Access unit delimiter RBSP syntax

access_unit_delimiter_rbsp() {	Descriptor
pic_type	u(3)
rbsp_trailing_bits()	
}	

7.3.2.6 End of sequence RBSP syntax

end_of_seq_rbsp() {	Descriptor
}	

7.3.2.7 End of bitstream RBSP syntax

end_of_bitstream_rbsp() {	Descriptor
}	

7.3.2.8 Filler data RBSP syntax

filler_data_rbsp() {	Descriptor
while(next_bits(8) == 0xFF)	
ff_byte /* equal to 0xFF */	f(8)
rbsp_trailing_bits()	
}	

7.3.2.9 Slice segment layer RBSP syntax

slice_segment_layer_rbsp() {	Descriptor
slice_segment_header()	
slice_segment_data()	
rbsp_slice_segment_trailing_bits()	
}	

7.3.2.10 RBSP slice segment trailing bits syntax

rbsp_slice_segment_trailing_bits() {	Descriptor
rbsp_trailing_bits()	
while(more_rbsp_trailing_data())	
cabac_zero_word /* equal to 0x0000 */	f(16)
}	

7.3.2.11 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)
}	

7.3.2.12 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)
}	

7.3.3 Profile, tier and level syntax

profile_tier_level(maxNumSubLayersMinus1) {	Descriptor
general_profile_space	u(2)
general_tier_flag	u(1)
general_profile_idc	u(5)
for(j = 0; j < 32; j++)	
general_profile_compatibility_flag[j]	u(1)
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)
general_reserved_zero_44bits	u(44)
general_level_idc	u(8)
for(i = 0; i < maxNumSubLayersMinus1; i++) {	
sub_layer_profile_present_flag[i]	u(1)
sub_layer_level_present_flag[i]	u(1)
}	
if(maxNumSubLayersMinus1 > 0)	
for(i = maxNumSubLayersMinus1; i < 8; i++)	
reserved_zero_2bits[i]	u(2)
for(i = 0; i < maxNumSubLayersMinus1; i++) {	
if(sub_layer_profile_present_flag[i]) {	
sub_layer_profile_space[i]	u(2)
sub_layer_tier_flag[i]	u(1)
sub_layer_profile_idc[i]	u(5)
for(j = 0; j < 32; j++)	
sub_layer_profile_compatibility_flag[i][j]	u(1)
sub_layer_progressive_source_flag[i]	u(1)
sub_layer_interlaced_source_flag[i]	u(1)
sub_layer_non_packed_constraint_flag[i]	u(1)
sub_layer_frame_only_constraint_flag[i]	u(1)
sub_layer_reserved_zero_44bits[i]	u(44)
}	
if(sub_layer_level_present_flag[i])	
sub_layer_level_idc[i]	u(8)
}	
}	

7.3.4 Scaling list data syntax

scaling_list_data() {	Descriptor
for(sizeId = 0; sizeId < 4; sizeId++)	
for(matrixId = 0; matrixId < ((sizeId == 3) ? 2 : 6); matrixId++) {	
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 << (4 + (sizeId << 1))))	
if(sizeId > 1) {	
scaling_list_dc_coef_minus8 [sizeId - 2][matrixId]	se(v)
nextCoef = scaling_list_dc_coef_minus8[sizeId - 2][matrixId] + 8	
}	
for(i = 0; i < coefNum; i++) {	
scaling_list_delta_coef	se(v)
nextCoef = (nextCoef + scaling_list_delta_coef + 256) % 256	
ScalingList[sizeId][matrixId][i] = nextCoef	
}	
}	
}	
}	

7.3.5 Supplemental enhancement information message syntax

sei_message() {	Descriptor
payloadType = 0	
while(next_bits(8) == 0xFF) {	
ff_byte /* equal to 0xFF */	f(8)
payloadType += 255	
}	
last_payload_type_byte	u(8)
payloadType += last_payload_type_byte	
payloadSize = 0	
while(next_bits(8) == 0xFF) {	
ff_byte /* equal to 0xFF */	f(8)
payloadSize += 255	
}	
last_payload_size_byte	u(8)
payloadSize += last_payload_size_byte	
sei_payload(payloadType, payloadSize)	
}	

7.3.6 Slice segment header syntax

7.3.6.1 General slice segment header syntax

slice_segment_header() {	Descriptor
first_slice_segment_in_pic_flag	u(1)
if(nal_unit_type >= BLA_W_LP && nal_unit_type <= RSV_IRAP_VCL23)	
no_output_of_prior_pics_flag	u(1)
slice_pic_parameter_set_id	ue(v)
if(!first_slice_segment_in_pic_flag) {	
if(dependent_slice_segments_enabled_flag)	
dependent_slice_segment_flag	u(1)
slice_segment_address	u(v)
}	
if(!dependent_slice_segment_flag) {	
for(i = 0; i < num_extra_slice_header_bits; i++)	
slice_reserved_flag[i]	u(1)
slice_type	ue(v)
if(output_flag_present_flag)	
pic_output_flag	u(1)
if(separate_colour_plane_flag == 1)	
colour_plane_id	u(2)
if(nal_unit_type != IDR_W_RADL && nal_unit_type != IDR_N_LP) {	
slice_pic_order_cnt_lsb	u(v)
short_term_ref_pic_set_sps_flag	u(1)
if(!short_term_ref_pic_set_sps_flag)	
short_term_ref_pic_set(num_short_term_ref_pic_sets)	
else if(num_short_term_ref_pic_sets > 1)	
short_term_ref_pic_set_idx	u(v)
if(long_term_ref_pics_present_flag) {	
if(num_long_term_ref_pics_sps > 0)	
num_long_term_sps	ue(v)
num_long_term_pics	ue(v)
for(i = 0; i < num_long_term_sps + num_long_term_pics; i++) {	
if(i < num_long_term_sps) {	
if(num_long_term_ref_pics_sps > 1)	
lt_idx_sps[i]	u(v)
} else {	
poc_lsb_lt[i]	u(v)
used_by_curr_pic_lt_flag[i]	u(1)
}	
delta_poc_msb_present_flag[i]	u(1)
if(delta_poc_msb_present_flag[i])	
delta_poc_msb_cycle_lt[i]	ue(v)
}	
}	
if(sps_temporal_mvp_enabled_flag)	
slice_temporal_mvp_enabled_flag	u(1)
}	

if(sample_adaptive_offset_enabled_flag) {	
slice_sao_luma_flag	u(1)
slice_sao_chroma_flag	u(1)
}	
if(slice_type == P slice_type == B) {	
num_ref_idx_active_override_flag	u(1)
if(num_ref_idx_active_override_flag) {	
num_ref_idx_l0_active_minus1	ue(v)
if(slice_type == B)	
num_ref_idx_l1_active_minus1	ue(v)
}	
if(lists_modification_present_flag && NumPocTotalCurr > 1)	
ref_pic_lists_modification()	
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((collocated_from_l0_flag && num_ref_idx_l0_active_minus1 > 0)	
(!collocated_from_l0_flag && num_ref_idx_l1_active_minus1 > 0))	
collocated_ref_idx	ue(v)
}	
if((weighted_pred_flag && slice_type == P)	
(weighted_bipred_flag && slice_type == B))	
pred_weight_table()	
five_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)
}	
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(pps_loop_filter_across_slices_enabled_flag &&	
(slice_sao_luma_flag slice_sao_chroma_flag	
!slice_deblocking_filter_disabled_flag))	
slice_loop_filter_across_slices_enabled_flag	u(1)
}	
if(tiles_enabled_flag entropy_coding_sync_enabled_flag) {	
num_entry_point_offsets	ue(v)

if(num_entry_point_offsets > 0) {	
offset_len_minus1	ue(v)
for(i = 0; i < num_entry_point_offsets; i++)	
entry_point_offset_minus1[i]	u(v)
}	
}	
if(slice_segment_header_extension_present_flag) {	
slice_segment_header_extension_length	ue(v)
for(i = 0; i < slice_segment_header_extension_length; i++)	
slice_segment_header_extension_data_byte[i]	u(8)
}	
byte_alignment()	
}	

7.3.6.2 Reference picture list modification syntax

ref_pic_lists_modification() {	Descriptor
ref_pic_list_modification_flag_l0	u(1)
if(ref_pic_list_modification_flag_l0)	
for(i = 0; i <= num_ref_idx_l0_active_minus1; i++)	
list_entry_l0[i]	u(v)
if(slice_type == B) {	
ref_pic_list_modification_flag_l1	u(1)
if(ref_pic_list_modification_flag_l1)	
for(i = 0; i <= num_ref_idx_l1_active_minus1; i++)	
list_entry_l1[i]	u(v)
}	
}	

7.3.6.3 Weighted prediction parameters syntax

pred_weight_table() {	Descriptor
luma_log2_weight_denom	ue(v)
if(chroma_format_idc != 0)	
delta_chroma_log2_weight_denom	se(v)
for(i = 0; i <= num_ref_idx_l0_active_minus1; i++)	
luma_weight_l0_flag[i]	u(1)
if(chroma_format_idc != 0)	
for(i = 0; i <= num_ref_idx_l0_active_minus1; i++)	
chroma_weight_l0_flag[i]	u(1)
for(i = 0; i <= num_ref_idx_l0_active_minus1; 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 <= num_ref_idx_l1_active_minus1; i++)	
luma_weight_l1_flag[i]	u(1)
if(chroma_format_idc != 0)	
for(i = 0; i <= num_ref_idx_l1_active_minus1; i++)	
chroma_weight_l1_flag[i]	u(1)
for(i = 0; i <= num_ref_idx_l1_active_minus1; 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)
}	
}	
}	
}	

7.3.7 Short-term reference picture set syntax

short_term_ref_pic_set(stRpsIdx) {	Descriptor
if(stRpsIdx != 0)	
inter_ref_pic_set_prediction_flag	u(1)
if(inter_ref_pic_set_prediction_flag) {	
if(stRpsIdx == num_short_term_ref_pic_sets)	
delta_idx_minus1	ue(v)
delta_rps_sign	u(1)
abs_delta_rps_minus1	ue(v)
for(j = 0; j <= NumDeltaPocs[RefRpsIdx]; j++) {	
used_by_curr_pic_flag[j]	u(1)
if(!used_by_curr_pic_flag[j])	
use_delta_flag[j]	u(1)
}	
} else {	
num_negative_pics	ue(v)
num_positive_pics	ue(v)
for(i = 0; i < num_negative_pics; i++) {	
delta_poc_s0_minus1[i]	ue(v)
used_by_curr_pic_s0_flag[i]	u(1)
}	
for(i = 0; i < num_positive_pics; i++) {	
delta_poc_s1_minus1[i]	ue(v)
used_by_curr_pic_s1_flag[i]	u(1)
}	
}	
}	

7.3.8 Slice segment data syntax

7.3.8.1 General slice segment data syntax

slice_segment_data() {	Descriptor
do {	
coding_tree_unit()	
end_of_slice_segment_flag	ae(v)
CtbAddrInTs++	
CtbAddrInRs = CtbAddrTsToRs[CtbAddrInTs]	
if(!end_of_slice_segment_flag && ((tiles_enabled_flag && TileId[CtbAddrInTs] != TileId[CtbAddrInTs - 1]) (entropy_coding_sync_enabled_flag && CtbAddrInTs % PicWidthInCtbsY == 0))) {	
end_of_sub_stream_one_bit /* equal to 1 */	ae(v)
byte_alignment()	
}	
} while(!end_of_slice_segment_flag)	
}	

7.3.8.2 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)	
coding_quadtree(xCtb, yCtb, CtbLog2SizeY, 0)	
}	

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

7.3.8.3 Sample adaptive offset syntax

	Descriptor
sao(rx, ry){	
if(rx > 0) {	
leftCtbInSliceSeg = CtbAddrInRs > SliceAddrRs	
leftCtbInTile = TileId[CtbAddrInTs] == TileId[CtbAddrRsToTs[CtbAddrInRs - 1]]	
if(leftCtbInSliceSeg && leftCtbInTile)	
sao_merge_left_flag	ae(v)
}	
if(ry > 0 && !sao_merge_left_flag) {	
upCtbInSliceSeg = (CtbAddrInRs - PicWidthInCtbsY) >= SliceAddrRs	
upCtbInTile = TileId[CtbAddrInTs] ==	
TileId[CtbAddrRsToTs[CtbAddrInRs - PicWidthInCtbsY]]	
if(upCtbInSliceSeg && upCtbInTile)	
sao_merge_up_flag	ae(v)
}	
if(!sao_merge_up_flag && !sao_merge_left_flag)	
for(cIdx = 0; cIdx < 3; 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)
}	
}	
}	

7.3.8.4 Coding quadtree syntax

coding_quadtree(x0, y0, log2CbSize, cqtDepth) {	Descriptor
if(x0 + (1 << log2CbSize) <= pic_width_in_luma_samples && y0 + (1 << log2CbSize) <= pic_height_in_luma_samples && log2CbSize > MinCbLog2SizeY)	
split_cu_flag [x0][y0]	ae(v)
if(cu_qp_delta_enabled_flag && log2CbSize >= Log2MinCuQpDeltaSize) {	
IsCuQpDeltaCoded = 0	
CuQpDeltaVal = 0	
}	
if(split_cu_flag[x0][y0]) {	
x1 = x0 + (1 << (log2CbSize • 1))	
y1 = y0 + (1 << (log2CbSize • 1))	
coding_quadtree(x0, y0, log2CbSize - 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples)	
coding_quadtree(x1, y0, log2CbSize - 1, cqtDepth + 1)	
if(y1 < pic_height_in_luma_samples)	
coding_quadtree(x0, y1, log2CbSize - 1, cqtDepth + 1)	
if(x1 < pic_width_in_luma_samples && y1 < pic_height_in_luma_samples)	
coding_quadtree(x1, y1, log2CbSize - 1, cqtDepth + 1)	
} else	
coding_unit(x0, y0, log2CbSize)	
}	

7.3.8.5 Coding unit syntax

coding_unit(x0, y0, log2CbSize) {	Descriptor
if(transquant_bypass_enabled_flag)	
cu_transquant_bypass_flag	ae(v)
if(slice_type != I)	
cu_skip_flag [x0][y0]	ae(v)
nCbS = (1 << log2CbSize)	
if(cu_skip_flag[x0][y0])	
prediction_unit(x0, y0, nCbS, nCbS)	
else {	
if(slice_type != I)	
pred_mode_flag	ae(v)
if(CuPredMode[x0][y0] != MODE_INTRA log2CbSize == MinCbLog2SizeY)	
part_mode	ae(v)
if(CuPredMode[x0][y0] == MODE_INTRA) {	
if(PartMode == PART_2Nx2N && pcm_enabled_flag && log2CbSize >= Log2MinIpcmCbSizeY && log2CbSize <= Log2MaxIpcmCbSizeY)	
pcm_flag [x0][y0]	ae(v)
if(pcm_flag[x0][y0]) {	
while(!byte_aligned())	
pcm_alignment_zero_bit	f(1)
pcm_sample(x0, y0, log2CbSize)	
} else {	
pbOffset = (PartMode == PART_NxN) ? (nCbS / 2) : nCbS	
for(j = 0; j < nCbS; j = j + pbOffset)	
for(i = 0; i < nCbS; i = i + pbOffset)	
prev_intra_luma_pred_flag [x0 + i][y0 + j]	ae(v)
for(j = 0; j < nCbS; j = j + pbOffset)	
for(i = 0; i < nCbS; i = i + pbOffset)	
if(prev_intra_luma_pred_flag[x0 + i][y0 + j])	
mpm_idx [x0 + i][y0 + j]	ae(v)
else	
rem_intra_luma_pred_mode [x0 + i][y0 + j]	ae(v)
intra_chroma_pred_mode [x0][y0]	ae(v)
}	
} else {	
if(PartMode == PART_2Nx2N)	
prediction_unit(x0, y0, nCbS, nCbS)	
else if(PartMode == PART_2NxN) {	
prediction_unit(x0, y0, nCbS, nCbS / 2)	
prediction_unit(x0, y0 + (nCbS / 2), nCbS, nCbS / 2)	
} else if(PartMode == PART_Nx2N) {	
prediction_unit(x0, y0, nCbS / 2, nCbS)	
prediction_unit(x0 + (nCbS / 2), y0, nCbS / 2, nCbS)	
} else if(PartMode == PART_2NxN) {	
prediction_unit(x0, y0, nCbS, nCbS / 4)	
prediction_unit(x0, y0 + (nCbS / 4), nCbS, nCbS * 3 / 4)	
} else if(PartMode == PART_2NxND) {	
prediction_unit(x0, y0, nCbS, nCbS * 3 / 4)	

prediction_unit(x0, y0 + (nCbS * 3 / 4), nCbS, nCbS / 4)	
} else if(PartMode == PART_nLx2N) {	
prediction_unit(x0, y0, nCbS / 4, nCbS)	
prediction_unit(x0 + (nCbS / 4), y0, nCbS * 3 / 4, nCbS)	
} else if(PartMode == PART_nRx2N) {	
prediction_unit(x0, y0, nCbS * 3 / 4, nCbS)	
prediction_unit(x0 + (nCbS * 3 / 4), y0, nCbS / 4, nCbS)	
} else { /* PART_NxN */	
prediction_unit(x0, y0, nCbS / 2, nCbS / 2)	
prediction_unit(x0 + (nCbS / 2), y0, nCbS / 2, nCbS / 2)	
prediction_unit(x0, y0 + (nCbS / 2), nCbS / 2, nCbS / 2)	
prediction_unit(x0 + (nCbS / 2), y0 + (nCbS / 2), nCbS / 2, nCbS / 2)	
}	
}	
if(!pcm_flag[x0][y0]) {	
if(CuPredMode[x0][y0] != MODE_INTRA && !(PartMode == PART_2Nx2N && merge_flag[x0][y0]))	
rqt_root_cbf	ae(v)
if(rqt_root_cbf) {	
MaxTrafoDepth = (CuPredMode[x0][y0] == MODE_INTRA ? (max_transform_hierarchy_depth_intra + IntraSplitFlag) : max_transform_hierarchy_depth_inter)	
transform_tree(x0, y0, x0, y0, log2CbSize, 0, 0)	
}	
}	
}	
}	

7.3.8.6 Prediction unit syntax

prediction_unit(x0, y0, nPbW, nPbH) {	Descriptor
if(cu_skip_flag[x0][y0]) {	
if(MaxNumMergeCand > 1)	
merge_idx [x0][y0]	ae(v)
} else { /* MODE_INTER */	
merge_flag [x0][y0]	ae(v)
if(merge_flag[x0][y0]) {	
if(MaxNumMergeCand > 1)	
merge_idx [x0][y0]	ae(v)
} else {	
if(slice_type == B)	
inter_pred_idc [x0][y0]	ae(v)
if(inter_pred_idc[x0][y0] != PRED_L1) {	
if(num_ref_idx_l0_active_minus1 > 0)	
ref_idx_l0 [x0][y0]	ae(v)
mvd_coding(x0, y0, 0)	
mvp_l0_flag [x0][y0]	ae(v)
}	
if(inter_pred_idc[x0][y0] != PRED_L0) {	
if(num_ref_idx_l1_active_minus1 > 0)	
ref_idx_l1 [x0][y0]	ae(v)
if(mvd_l1_zero_flag && inter_pred_idc[x0][y0] == PRED_BI) {	
MvdL1[x0][y0][0] = 0	
MvdL1[x0][y0][1] = 0	
} else	
mvd_coding(x0, y0, 1)	
mvp_l1_flag [x0][y0]	ae(v)
}	
}	
}	
}	

7.3.8.7 PCM sample syntax

pcm_sample(x0, y0, log2CbSize) {	Descriptor
for(i = 0; i < 1 << (log2CbSize << 1); i++)	
pcm_sample_luma [i]	u(v)
for(i = 0; i < (1 << (log2CbSize << 1)) >> 1; i++)	
pcm_sample_chroma [i]	u(v)
}	

7.3.8.8 Transform tree syntax

transform_tree(x0, y0, xBase, yBase, log2TrafoSize, trafoDepth, blkIdx) {	Descriptor
if(log2TrafoSize <= Log2MaxTrafoSize && log2TrafoSize > Log2MinTrafoSize && trafoDepth < MaxTrafoDepth && !(IntraSplitFlag && (trafoDepth == 0)))	
split_transform_flag [x0][y0][trafoDepth]	ae(v)
if(log2TrafoSize > 2) {	
if(trafoDepth == 0 cbf_cb[xBase][yBase][trafoDepth - 1])	
cbf_cb [x0][y0][trafoDepth]	ae(v)
if(trafoDepth == 0 cbf_cr[xBase][yBase][trafoDepth - 1])	
cbf_cr [x0][y0][trafoDepth]	ae(v)
}	
if(split_transform_flag[x0][y0][trafoDepth]) {	
x1 = x0 + (1 << (log2TrafoSize • 1))	
y1 = y0 + (1 << (log2TrafoSize • 1))	
transform_tree(x0, y0, x0, y0, log2TrafoSize - 1, trafoDepth + 1, 0)	
transform_tree(x1, y0, x0, y0, log2TrafoSize - 1, trafoDepth + 1, 1)	
transform_tree(x0, y1, x0, y0, log2TrafoSize - 1, trafoDepth + 1, 2)	
transform_tree(x1, y1, x0, y0, log2TrafoSize - 1, trafoDepth + 1, 3)	
} else {	
if(CuPredMode[x0][y0] == MODE_INTRA trafoDepth != 0 cbf_cb[x0][y0][trafoDepth] cbf_cr[x0][y0][trafoDepth])	
cbf_luma [x0][y0][trafoDepth]	ae(v)
transform_unit(x0, y0, xBase, yBase, log2TrafoSize, trafoDepth, blkIdx)	
}	
}	

7.3.8.9 Motion vector difference syntax

mvd_coding(x0, y0, refList) {	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)
}	
}	

7.3.8.10 Transform unit syntax

transform_unit(x0, y0, xBase, yBase, log2TrafoSize, trafoDepth, blkIdx) {	Descriptor
if(cbf_luma[x0][y0][trafoDepth] cbf_cb[x0][y0][trafoDepth] cbf_cr[x0][y0][trafoDepth]) {	
if(cu_qp_delta_enabled_flag && !IsCuQpDeltaCoded) {	
cu_qp_delta_abs	ae(v)
if(cu_qp_delta_abs)	
cu_qp_delta_sign_flag	ae(v)
}	
if(cbf_luma[x0][y0][trafoDepth])	
residual_coding(x0, y0, log2TrafoSize, 0)	
if(log2TrafoSize > 2) {	
if(cbf_cb[x0][y0][trafoDepth])	
residual_coding(x0, y0, log2TrafoSize - 1, 1)	
if(cbf_cr[x0][y0][trafoDepth])	
residual_coding(x0, y0, log2TrafoSize - 1, 2)	
} else if(blkIdx == 3) {	
if(cbf_cb[xBase][yBase][trafoDepth])	
residual_coding(xBase, yBase, log2TrafoSize, 1)	
if(cbf_cr[xBase][yBase][trafoDepth])	
residual_coding(xBase, yBase, log2TrafoSize, 2)	
}	
}	
}	

7.3.8.11 Residual coding syntax

residual_coding(x0, y0, log2TrafoSize, cIdx) {	Descriptor
if(transform_skip_enabled_flag && !cu_transquant_bypass_flag && (log2TrafoSize == 2))	
transform_skip_flag [x0][y0][cIdx]	ae(v)
last_sig_coeff_x_prefix	ae(v)
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)
lastScanPos = 16	
lastSubBlock = (1 << (log2TrafoSize - 2)) * (1 << (log2TrafoSize - 2)) - 1	
do {	
if(lastScanPos == 0) {	
lastScanPos = 16	
lastSubBlock--	
}	
lastScanPos--	
xS = ScanOrder[log2TrafoSize - 2][scanIdx][lastSubBlock][0]	
yS = ScanOrder[log2TrafoSize - 2][scanIdx][lastSubBlock][1]	
xC = (xS << 2) + ScanOrder[2][scanIdx][lastScanPos][0]	
yC = (yS << 2) + ScanOrder[2][scanIdx][lastScanPos][1]	
} while((xC != LastSignificantCoeffX) (yC != LastSignificantCoeffY))	
for(i = lastSubBlock; i >= 0; i--) {	
xS = ScanOrder[log2TrafoSize - 2][scanIdx][i][0]	
yS = ScanOrder[log2TrafoSize - 2][scanIdx][i][1]	
inferSbDcSigCoeffFlag = 0	
if((i < lastSubBlock) && (i > 0)) {	
coded_sub_block_flag [xS][yS]	ae(v)
inferSbDcSigCoeffFlag = 1	
}	
for(n = (i == lastSubBlock) ? lastScanPos - 1 : 15; n >= 0; n--) {	
xC = (xS << 2) + ScanOrder[2][scanIdx][n][0]	
yC = (yS << 2) + ScanOrder[2][scanIdx][n][1]	
if(coded_sub_block_flag[xS][yS] && (n > 0 !inferSbDcSigCoeffFlag)) {	
sig_coeff_flag [xC][yC]	ae(v)
if(sig_coeff_flag[xC][yC])	
inferSbDcSigCoeffFlag = 0	
}	
}	
firstSigScanPos = 16	
lastSigScanPos = -1	
numGreater1Flag = 0	
lastGreater1ScanPos = -1	
for(n = 15; n >= 0; n--) {	
xC = (xS << 2) + ScanOrder[2][scanIdx][n][0]	
yC = (yS << 2) + ScanOrder[2][scanIdx][n][1]	
if(sig_coeff_flag[xC][yC]) {	

if(numGreater1Flag < 8) {	
coeff_abs_level_greater1_flag [n]	ae(v)
numGreater1Flag++	
if(coeff_abs_level_greater1_flag[n] && lastGreater1ScanPos == -1)	
lastGreater1ScanPos = n	
}	
if(lastSigScanPos == -1)	
lastSigScanPos = n	
firstSigScanPos = n	
}	
}	
signHidden = (lastSigScanPos - firstSigScanPos > 3 && !cu_transquant_bypass_flag)	
if(lastGreater1ScanPos != -1)	
coeff_abs_level_greater2_flag [lastGreater1ScanPos]	ae(v)
for(n = 15; n >= 0; n--) {	
xC = (xS << 2) + ScanOrder[2][scanIdx][n][0]	
yC = (yS << 2) + ScanOrder[2][scanIdx][n][1]	
if(sig_coeff_flag[xC][yC] && (!sign_data_hiding_enabled_flag !signHidden (n != firstSigScanPos)))	
coeff_sign_flag [n]	ae(v)
}	
numSigCoeff = 0	
sumAbsLevel = 0	
for(n = 15; n >= 0; n--) {	
xC = (xS << 2) + ScanOrder[2][scanIdx][n][0]	
yC = (yS << 2) + ScanOrder[2][scanIdx][n][1]	
if(sig_coeff_flag[xC][yC]) {	
baseLevel = 1 + coeff_abs_level_greater1_flag[n] + coeff_abs_level_greater2_flag[n]	
if(baseLevel == ((numSigCoeff < 8) ? (n == lastGreater1ScanPos ? 3 : 2) : 1))	
coeff_abs_level_remaining [n]	ae(v)
TransCoeffLevel[x0][y0][cIdx][xC][yC] = (coeff_abs_level_remaining[n] + baseLevel) * (1 - 2 * coeff_sign_flag[n])	
if(sign_data_hiding_enabled_flag && signHidden) {	
sumAbsLevel += (coeff_abs_level_remaining[n] + baseLevel)	
if((n == firstSigScanPos) && ((sumAbsLevel % 2) == 1))	
TransCoeffLevel[x0][y0][cIdx][xC][yC] = -TransCoeffLevel[x0][y0][cIdx][xC][yC]	
}	
numSigCoeff++	
}	
}	
}	
}	
}	

7.4 Semantics

7.4.1 General

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

7.4.2 NAL unit semantics

7.4.2.1 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 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 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

7.4.2.2 NAL unit header semantics

forbidden_zero_bit shall be equal to 0.

nal_unit_type specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7-1.

NAL units that have **nal_unit_type** in the range of UNSPEC48..UNSPEC63, inclusive, for which semantics are not specified, shall not affect the decoding process specified in this Specification.

NOTE 1 – NAL unit types in the range of UNSPEC48..UNSPEC63 may be used as determined by the application. No decoding process for these values of **nal_unit_type** 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 **nal_unit_type** values, and in the design of decoders that interpret the content of NAL units with these **nal_unit_type** values.

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 **nal_unit_type**.

NOTE 2 – This requirement allows future definition of compatible extensions to this Specification.

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

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

nal_unit_type	Name of nal_unit_type	Content of NAL unit and RBSP syntax structure	NAL unit type class
0 1	TRAIL_N TRAIL_R	Coded slice segment of a non-TSA, non-STSA trailing picture slice_segment_layer_rbsp()	VCL
2 3	TSA_N TSA_R	Coded slice segment of a TSA picture slice_segment_layer_rbsp()	VCL
4 5	STSA_N STSA_R	Coded slice segment of an STSA picture slice_segment_layer_rbsp()	VCL
6 7	RADL_N RADL_R	Coded slice segment of a RADL picture slice_segment_layer_rbsp()	VCL
8 9	RASL_N RASL_R	Coded slice segment of a RASL picture slice_segment_layer_rbsp()	VCL
10 12 14	RSV_VCL_N10 RSV_VCL_N12 RSV_VCL_N14	Reserved non-IRAP sub-layer non-reference VCL NAL unit types	VCL
11 13 15	RSV_VCL_R11 RSV_VCL_R13 RSV_VCL_R15	Reserved non-IRAP sub-layer reference VCL NAL unit types	VCL
16 17 18	BLA_W_LP BLA_W_RADL BLA_N_LP	Coded slice segment of a BLA picture slice_segment_layer_rbsp()	VCL
19 20	IDR_W_RADL IDR_N_LP	Coded slice segment of an IDR picture slice_segment_layer_rbsp()	VCL
21	CRA_NUT	Coded slice segment of a CRA picture slice_segment_layer_rbsp()	VCL
22 23	RSV_IRAP_VCL22 RSV_IRAP_VCL23	Reserved IRAP VCL NAL unit types	VCL
24..31	RSV_VCL24.. RSV_VCL31	Reserved non-IRAP VCL NAL unit types	VCL
32	VPS_NUT	Video parameter set video_parameter_set_rbsp()	non-VCL
33	SPS_NUT	Sequence parameter set seq_parameter_set_rbsp()	non-VCL
34	PPS_NUT	Picture parameter set pic_parameter_set_rbsp()	non-VCL
35	AUD_NUT	Access unit delimiter access_unit_delimiter_rbsp()	non-VCL
36	EOS_NUT	End of sequence end_of_seq_rbsp()	non-VCL
37	EOB_NUT	End of bitstream end_of_bitstream_rbsp()	non-VCL
38	FD_NUT	Filler data filler_data_rbsp()	non-VCL
39 40	PREFIX_SEI_NUT SUFFIX_SEI_NUT	Supplemental enhancement information sei_rbsp()	non-VCL
41..47	RSV_NVCL41.. RSV_NVCL47	Reserved	non-VCL

48..63	UNSPEC48.. UNSPEC63	Unspecified	non-VCL
--------	------------------------	-------------	---------

NOTE 3 – A CRA picture may have associated RASL or RADL pictures present in the bitstream.

NOTE 4 – A BLA picture having `nal_unit_type` equal to `BLA_W_LP` may have associated RASL or RADL pictures present in the bitstream. A BLA picture having `nal_unit_type` equal to `BLA_W_RADL` does not have associated RASL pictures present in the bitstream, but may have associated RADL pictures in the bitstream. A BLA picture having `nal_unit_type` equal to `BLA_N_LP` does not have associated leading pictures present in the bitstream.

NOTE 5 – An IDR picture having `nal_unit_type` equal to `IDR_N_LP` does not have associated leading pictures present in the bitstream. An IDR picture having `nal_unit_type` equal to `IDR_W_RADL` does not have associated RASL pictures present in the bitstream, but may have associated RADL pictures in the bitstream.

NOTE 6 – A sub-layer non-reference picture is not included in any of `RefPicSetStCurrBefore`, `RefPicSetStCurrAfter` and `RefPicSetLtCurr` of any picture with the same value of `TemporalId`, and may be discarded without affecting the decodability of other pictures with the same value of `TemporalId`.

All coded slice segment NAL units of an access unit shall have the same value of `nal_unit_type`. A picture or an access unit is also referred to as having a `nal_unit_type` equal to the `nal_unit_type` of the coded slice segment NAL units of the picture or access unit.

If a picture has `nal_unit_type` equal to `TRAIL_N`, `TSA_N`, `STSA_N`, `RADL_N`, `RASL_N`, `RSV_VCL_N10`, `RSV_VCL_N12`, or `RSV_VCL_N14`, the picture is a sub-layer non-reference picture. Otherwise, the picture is a sub-layer reference picture.

Each picture, other than the first picture in the bitstream in decoding order, is considered to be associated with the previous IRAP picture in decoding order.

When a picture is a leading picture, it shall be a RADL or RASL picture.

When a picture is a trailing picture, it shall not be a RADL or RASL picture.

When a picture is a leading picture, it shall precede, in decoding order, all trailing pictures that are associated with the same IRAP picture.

No RASL pictures shall be present in the bitstream that are associated with a BLA picture having `nal_unit_type` equal to `BLA_W_RADL` or `BLA_N_LP`.

No RASL pictures shall be present in the bitstream that are associated with an IDR picture.

No RADL pictures shall be present in the bitstream that are associated with a BLA picture having `nal_unit_type` equal to `BLA_N_LP` or that are associated with an IDR picture having `nal_unit_type` equal to `IDR_N_LP`.

NOTE 7 – It is possible to perform random access at the position of an IRAP access unit by discarding all access units before the IRAP access unit (and to correctly decode the IRAP picture and all the subsequent non-RASL pictures in decoding order), provided each parameter set is available (either in the bitstream or by external means not specified in this Specification) when it needs to be activated.

Any picture that has `PicOutputFlag` equal to 1 that precedes an IRAP picture in decoding order shall precede the IRAP picture in output order and shall precede any RADL picture associated with the IRAP picture in output order.

Any RASL picture associated with a CRA or BLA picture shall precede any RADL picture associated with the CRA or BLA picture in output order.

Any RASL picture associated with a CRA picture shall follow, in output order, any IRAP picture that precedes the CRA picture in decoding order.

When `sps_temporal_id_nesting_flag` is equal to 1 and `TemporalId` is greater than 0, the `nal_unit_type` shall be equal to `TSA_R`, `TSA_N`, `RADL_R`, `RADL_N`, `RASL_R`, or `RASL_N`.

nuh_layer_id shall be equal to 0. Other values of `nuh_layer_id` 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 (as specified in Annex C), decoders shall ignore (i.e. remove from the bitstream and discard) all NAL units with values of `nuh_layer_id` not equal to 0.

NOTE 8 – It is anticipated that in future scalable or 3D video coding extensions of this specification, this syntax element will be used to identify additional layers that may be present in the CVS, wherein a layer may be, e.g. a spatial scalable layer, a quality scalable layer, a texture view or a depth view.

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.

The variable TemporalId is specified as follows:

$$\text{TemporalId} = \text{nuh_temporal_id_plus1} - 1 \quad (7-1)$$

If nal_unit_type is in the range of BLA_W_LP to RSV_IRAP_VCL23, inclusive, i.e. the coded slice segment belongs to an IRAP picture, TemporalId shall be equal to 0. Otherwise, when nal_unit_type is equal to TSA_R, TSA_N, STSA_R, or STSA_N, TemporalId shall not be equal to 0.

The value of TemporalId shall be the same for all VCL NAL units of an access unit. The value of TemporalId of an access unit is the value of the TemporalId of the VCL NAL units of the access unit.

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

- If nal_unit_type is equal to VPS_NUT or SPS_NUT, TemporalId shall be equal to 0 and the TemporalId of the access unit containing the NAL unit shall be equal to 0.
- Otherwise if nal_unit_type is equal to EOS_NUT or EOB_NUT, TemporalId shall be equal to 0.
- Otherwise, if nal_unit_type is equal to AUD_NUT or FD_NUT, TemporalId shall be equal to the TemporalId of the access unit containing the NAL unit.
- Otherwise, TemporalId shall be greater than or equal to the TemporalId of the access unit containing the NAL unit.

NOTE 9 – 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 access units to which the non-VCL NAL unit applies. When nal_unit_type is equal to PPS_NUT, TemporalId may be greater than or equal to the TemporalId of the containing access unit, as all PPSs may be included in the beginning of a bitstream, wherein the first coded picture has TemporalId equal to 0. When nal_unit_type is equal to PREFIX_SEI_NUT or SUFFIX_SEI_NUT, TemporalId may be greater than or equal to the TemporalId of the containing access unit, as an SEI NAL unit may contain information, e.g. in a buffering period SEI message or a picture timing SEI message, that applies to a bitstream subset that includes access units for which the TemporalId values are greater than the TemporalId of the access unit containing the SEI NAL unit.

7.4.2.3 Encapsulation of an SODB within an RBSP (informative)

This subclause 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.

2. The resulting sequence of bytes is then prefixed with the NAL unit header, within which the nal_unit_type 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.

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

7.4.2.4.1 General

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

Any order of NAL units in the bitstream obeying these constraints is referred to in the text as the decoding order of NAL units. Within a NAL unit, the syntax in subclauses 7.3, D.2, and E.1 specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

7.4.2.4.2 Order of VPS, SPS and PPS RBSPs and their activation

This subclause specifies the activation process of VPSs, SPSs, and PPSs.

NOTE 1 – The VPS, SPS, and PPS mechanism decouples the transmission of infrequently changing information from the transmission of coded block data. VPSs, SPSs, and PPSs may, in some applications, be conveyed "out-of-band".

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

When a PPS RBSP (with a particular value of `pps_pic_parameter_set_id`) is not active and it is referred to by a coded slice segment NAL unit (using a value of `slice_pic_parameter_set_id` equal to the `pps_pic_parameter_set_id` value), it is activated. This PPS RBSP is called the active PPS RBSP until it is deactivated by the activation of another PPS RBSP. A PPS RBSP, with that particular value of `pps_pic_parameter_set_id`, shall be available to the decoding process prior to its activation, included in at least one access unit with `TemporalId` less than or equal to the `TemporalId` of the PPS NAL unit or provided through external means.

Any PPS NAL unit containing the value of `pps_pic_parameter_set_id` for the active PPS RBSP for a coded picture shall have the same content as that of the active PPS RBSP for the coded picture, unless it follows the last VCL NAL unit of the coded picture and precedes the first VCL NAL unit of another coded picture.

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

When an SPS RBSP (with a particular value of `sps_seq_parameter_set_id`) is not already active and it is referred to by activation of a PPS RBSP (in which `pps_seq_parameter_set_id` is equal to the `sps_seq_parameter_set_id` value) or is referred to by an SEI NAL unit containing an active parameter sets SEI message (in which `active_seq_parameter_set_id[0]` is equal to the `sps_seq_parameter_set_id` value), it is activated. This SPS RBSP is called the active SPS RBSP until it is deactivated by the activation of another SPS RBSP. An SPS RBSP, with that particular value of `sps_seq_parameter_set_id`, shall be available to the decoding process prior to its activation, included in at least one access unit with `TemporalId` equal to 0 or provided through external means. An activated SPS RBSP shall remain active for the entire CVS.

NOTE 2 – Because an IRAP access unit with `NoRasOutputFlag` equal to 1 begins a new CVS and an activated SPS RBSP must remain active for the entire CVS, an SPS RBSP can only be activated by an active parameter sets SEI message when the active parameter sets SEI message is part of an IRAP access unit with `NoRasOutputFlag` equal to 1.

Any SPS NAL unit containing the value of `sps_seq_parameter_set_id` for the active SPS RBSP for a CVS shall have the same content as that of the active SPS RBSP for the CVS, unless it follows the last access unit of the CVS and precedes the first VCL NAL unit and the first SEI NAL unit containing an active parameter sets SEI message (when present) of another CVS.

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

When a VPS RBSP (with a particular value of `vps_video_parameter_set_id`) is not already active and it is referred to by activation of an SPS RBSP (in which `sps_video_parameter_set_id` is equal to the `vps_video_parameter_set_id` value), or is referred to by an SEI NAL unit containing an active parameter sets SEI message (in which `active_video_parameter_set_id` is equal to the `vps_video_parameter_set_id` value), it is activated. This VPS RBSP is called the active VPS RBSP until it is deactivated by the activation of another VPS RBSP. A VPS RBSP, with that particular value of `vps_video_parameter_set_id`, shall be available to the decoding process prior to its activation,

included in at least one access unit with TemporalId equal to 0 or provided through external means. An activated VPS RBSP shall remain active for the entire CVS.

NOTE 3 – Because an IRAP access unit with NoRasOutputFlag equal to 1 begins a new CVS and an activated VPS RBSP must remain active for the entire CVS, a VPS RBSP can only be activated by an active parameter sets SEI message when the active parameter sets SEI message is part of an IRAP access unit with NoRasOutputFlag equal to 1.

Any VPS NAL unit containing the value of vps_video_parameter_set_id for the active VPS RBSP for a CVS shall have the same content as that of the active VPS RBSP for the CVS, unless it follows the last access unit of the CVS and precedes the first VCL NAL unit, the first SPS NAL unit, and the first SEI NAL unit containing an active parameter sets SEI message (when present) of another CVS.

NOTE 4 – If VPS RBSP, SPS RBSP, or PPS RBSP are conveyed within the bitstream, these constraints impose an order constraint on the NAL units that contain the VPS RBSP, SPS RBSP, or PPS RBSP, respectively. Otherwise (VPS RBSP, SPS RBSP, or PPS RBSP are conveyed by other means not specified in this Specification), they must be available to the decoding process in a timely fashion such that these constraints are obeyed.

All constraints that are expressed on the relationship between the values of the syntax elements and the values of variables derived from those syntax elements in VPSs, SPSs, and PPSs and other syntax elements are expressions of constraints that apply only to the active VPS, the active SPS, and the active PPS. If any VPS RBSP, SPS RBSP, and PPS RBSP is present that is never activated in the bitstream, its syntax elements shall have values that would conform to the specified constraints if it was activated by reference in an otherwise conforming bitstream.

During operation of the decoding process (see clause 8), the values of parameters of the active VPS, the active SPS, and the active PPS RBSP are considered in effect. For interpretation of SEI messages, the values of the active VPS, the active SPS, and the active PPS RBSP for the operation of the decoding process for the VCL NAL units of the coded picture in the same access unit are considered in effect unless otherwise specified in the SEI message semantics.

7.4.2.4.3 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 subclause 7.4.2.4.4.

The first access unit of a CVS is an IRAP access unit with NoRasOutputFlag equal to 1.

It is a requirement of bitstream conformance that, when present, 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 access unit, which may be an IDR access unit, a BLA access unit, or a CRA access unit.

7.4.2.4.4 Order of NAL units and coded pictures and their association to access units

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

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

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

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

- access unit delimiter NAL unit (when present),
- VPS NAL unit (when present),
- SPS NAL unit (when present),
- PPS NAL unit (when present),
- Prefix SEI NAL unit (when present),
- NAL units with nal_unit_type in the range of RSV_NVCL41..RSV_NVCL44 (when present),
- NAL units with nal_unit_type in the range of UNSPEC48..UNSPEC55 (when present),
- first VCL NAL unit of a coded picture (always present).

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

- When an access unit delimiter NAL unit is present, it shall be the first NAL unit. There shall be at most one access unit delimiter NAL unit in any access unit.
- When any prefix SEI NAL units are present, they shall not follow the last VCL NAL unit of the access unit.
- NAL units having `nal_unit_type` equal to `FD_NUT` or `SUFFIX_SEI_NUT`, or in the range of `RSV_NVCL45..RSV_NVCL47` or `UNSPEC56..UNSPEC63` shall not precede the first VCL NAL unit of the coded picture.
- When an end of sequence NAL unit is present, it shall be the last NAL unit in the access unit other than an end of bitstream NAL unit (when present).
- When an end of bitstream NAL unit is present, it shall be the last NAL unit in the access unit.

NOTE – VPS NAL units, SPS NAL units, PPS NAL units, prefix SEI NAL units, or NAL units with `nal_unit_type` in the range of `RSV_NVCL41..RSV_NVCL44` or `UNSPEC48..UNSPEC55`, may be present in an access unit, but cannot follow the last VCL NAL unit of the coded picture within the access unit, as this condition would specify the start of a new access unit.

The structure of access units not containing any NAL units with `nal_unit_type` equal to `FD_NUT`, `VPS_NUT`, `SPS_NUT`, `PPS_NUT`, `RSV_VCL_N10`, `RSV_VCL_R11`, `RSV_VCL_N12`, `RSV_VCL_R13`, `RSV_VCL_N14`, or `RSV_VCL_R15`, `RSV_IRAP_VCL22`, or `RSV_IRAP_VCL23`, or in the range of `RSV_VCL24..RSV_VCL31`, `RSV_NVCL41..RSV_NVCL47`, or `UNSPEC48..UNSPEC63` is shown in Figure 7-1.

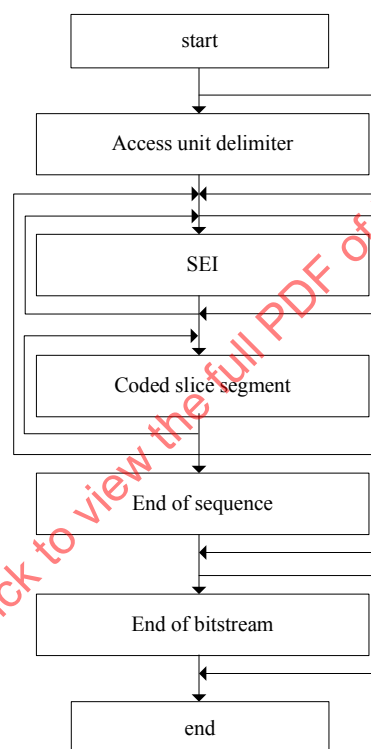


Figure 7-1 – Structure of an access unit not containing any NAL units with `nal_unit_type` equal to `FD_NUT`, `SUFFIX_SEI_NUT`, `VPS_NUT`, `SPS_NUT`, `PPS_NUT`, `RSV_VCL_N10`, `RSV_VCL_R11`, `RSV_VCL_N12`, `RSV_VCL_R13`, `RSV_VCL_N14`, `RSV_VCL_R15`, `RSV_IRAP_VCL22`, or `RSV_IRAP_VCL23`, or in the range of `RSV_VCL24..RSV_VCL31`, `RSV_NVCL41..RSV_NVCL47`, or `UNSPEC48..UNSPEC63`

7.4.2.4.5 Order of VCL NAL units and association to coded pictures

This subclause specifies the order of VCL NAL units and association to coded pictures.

Each VCL NAL unit is part of a coded picture.

The order of the VCL NAL units within a coded picture is constrained as follows:

- The first VCL NAL unit of the coded picture shall have `first_slice_segment_in_pic_flag` equal to 1.
- Let `sliceSegAddrA` and `sliceSegAddrB` be the `slice_segment_address` values of any two coded slice segment NAL units A and B within the same coded picture. When either of the following conditions is true, coded slice segment NAL unit A shall precede the coded slice segment NAL unit B:

- $\text{TileId}[\text{CtbAddrRsToTs}[\text{sliceSegAddrA}]]$ is less than $\text{TileId}[\text{CtbAddrRsToTs}[\text{sliceSegAddrB}]]$.
- $\text{TileId}[\text{CtbAddrRsToTs}[\text{sliceSegAddrA}]]$ is equal to $\text{TileId}[\text{CtbAddrRsToTs}[\text{sliceSegAddrB}]]$ and $\text{CtbAddrRsToTs}[\text{sliceSegAddrA}]$ is less than $\text{CtbAddrRsToTs}[\text{sliceSegAddrB}]$.

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

7.4.3.1 Video parameter set RBSP semantics

NOTE 1 – VPS 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 subclause 7.4.2.4.2. However, the VPS RBSP contains information that is not necessary for operation of the decoding process of this version of this Specification. For purposes other than determining the amount of data in the decoding units of the bitstream (as specified in Annex C), decoders conforming to this version of this Specification may ignore (remove from the bitstream and discard) the content of all VPS NAL units.

Any two instances of the syntax structure `hrd_parameters()` included in a VPS RBSP shall not have the same content.

vps_video_parameter_set_id identifies the VPS for reference by other syntax elements.

vps_reserved_three_2bits shall be equal to 3 in bitstreams conforming to this version of this Specification. Other values for **vps_reserved_three_2bits** are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of **vps_reserved_three_2bits**.

vps_max_layers_minus1 shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for **vps_max_layers_minus1** are reserved for future use by ITU-T | ISO/IEC. Although the value of **vps_max_layers_minus1** is required to be equal to 0 in this version of this Specification, decoders shall allow other values of **vps_max_layers_minus1** to appear in the syntax.

NOTE 2 – It is anticipated that in future scalable or 3D video coding extensions of this Specification, this field will be used to specify the maximum number of layers that may be present in the CVS, wherein a layer may e.g. be a spatial scalable layer, a quality scalable layer, a texture view or a depth view.

vps_max_sub_layers_minus1 plus 1 specifies the maximum number of temporal sub-layers that may be present in the bitstream. The value of **vps_max_sub_layers_minus1** shall be in the range of 0 to 6, inclusive.

vps_temporal_id_nesting_flag, when **vps_max_sub_layers_minus1** is greater than 0, specifies whether inter prediction is additionally restricted for CVSs referring to the VPS. When **vps_max_sub_layers_minus1** is equal to 0, **vps_temporal_id_nesting_flag** shall be equal to 1.

NOTE 3 – The syntax element **vps_temporal_id_nesting_flag** is used to indicate that temporal sub-layer up-switching, i.e. switching from decoding of up to any **TemporalId** **tIdN** to decoding up to any **TemporalId** **tIdM** that is greater than **tIdN**, is always possible.

vps_reserved_0xffff_16bits shall be equal to 0xFFFF in bitstreams conforming to this version of this Specification. Other values for **vps_reserved_0xffff_16bits** are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of **vps_reserved_0xffff_16bits**.

NOTE 4 – It is anticipated that in future scalable or 3D video coding extensions of this Specification, this syntax element will specify a byte offset to the next set of fixed-length coded information in the VPS RBSP, within the data currently specified as **vps_extension_data_flag** syntax elements. The byte offset would then help to locate and access such information in the VPS RBSP without the need for performing entropy decoding.

vps_sub_layer_ordering_info_present_flag equal to 1 specifies that **vps_max_dec_pic_buffering_minus1[i]**, **vps_max_num_reorder_pics[i]**, and **vps_max_latency_increase_plus1[i]** are present for **vps_max_sub_layers_minus1 + 1** sub-layers. **vps_sub_layer_ordering_info_present_flag** equal to 0 specifies that the values of **vps_max_dec_pic_buffering_minus1[vps_max_sub_layers_minus1]**, **vps_max_num_reorder_pics[vps_max_sub_layers_minus1]**, and **vps_max_latency_increase_plus1[vps_max_sub_layers_minus1]** apply to all sub-layers.

vps_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 **vps_max_dec_pic_buffering_minus1[i]** shall be in the range of 0 to **MaxDpbSize - 1** (as specified in subclause A.4), inclusive. When **i** is greater than 0, **vps_max_dec_pic_buffering_minus1[i]** shall be greater than or equal to **vps_max_dec_pic_buffering_minus1[i - 1]**. When **vps_max_dec_pic_buffering_minus1[i]** is not present for **i** in the range of 0 to **vps_max_sub_layers_minus1 - 1**, inclusive, due to **vps_sub_layer_ordering_info_present_flag** being equal to 0, it is inferred to be equal to **vps_max_dec_pic_buffering_minus1[vps_max_sub_layers_minus1]**.

vps_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 **vps_max_num_reorder_pics[i]** shall be in the range of 0 to **vps_max_dec_pic_buffering_minus1[i]**, inclusive. When **i** is greater than 0, **vps_max_num_reorder_pics[i]** shall be greater than or equal to **vps_max_num_reorder_pics[i - 1]**. When **vps_max_num_reorder_pics[i]** is not present for **i** in the range of 0 to **vps_max_sub_layers_minus1 - 1**,

inclusive, due to `vps_sub_layer_ordering_info_present_flag` being equal to 0, it is inferred to be equal to `vps_max_num_reorder_pics[ vps_max_sub_layers_minus1 ]`.

`vps_max_latency_increase_plus1[ i ]` not equal to 0 is used to compute the value of `VpsMaxLatencyPictures[ 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 `vps_max_latency_increase_plus1[ i ]` is not equal to 0, the value of `VpsMaxLatencyPictures[ i ]` is specified as follows:

$$\text{VpsMaxLatencyPictures}[i] = \text{vps_max_num_reorder_pics}[i] + \text{vps_max_latency_increase_plus1}[i] - 1 \quad (7-2)$$

When `vps_max_latency_increase_plus1[ i ]` is equal to 0, no corresponding limit is expressed.

The value of `vps_max_latency_increase_plus1[ i ]` shall be in the range of 0 to $2^{32} - 2$, inclusive. When `vps_max_latency_increase_plus1[ i ]` is not present for `i` in the range of 0 to `vps_max_sub_layers_minus1 - 1`, inclusive, due to `vps_sub_layer_ordering_info_present_flag` being equal to 0, it is inferred to be equal to `vps_max_latency_increase_plus1[ vps_max_sub_layers_minus1 ]`.

`vps_max_layer_id` specifies the maximum allowed value of `nuh_layer_id` of all NAL units in the CVS.

`vps_num_layer_sets_minus1` plus 1 specifies the number of layer sets that are specified by the VPS. In bitstreams conforming to this version of this Specification, the value of `vps_num_layer_sets_minus1` shall be equal to 0. Although the value of `vps_num_layer_sets_minus1` is required to be equal to 0 in this version of this Specification, decoders shall allow other values of `vps_num_layer_sets_minus1` in the range of 0 to 1023, inclusive, to appear in the syntax.

`layer_id_included_flag[ i ][ j ]` equal to 1 specifies that the value of `nuh_layer_id` equal to `j` is included in the layer identifier list `layerSetLayerIdList[ i ]`. **`layer_id_included_flag[ i ][ j ]`** equal to 0 specifies that the value of `nuh_layer_id` equal to `j` is not included in the layer identifier list `layerSetLayerIdList[ i ]`.

The value of `numLayersInIdList[ 0 ]` is set equal to 1 and the value of `layerSetLayerIdList[ 0 ][ 0 ]` is set equal to 0.

For each value of `i` in the range of 1 to `vps_num_layer_sets_minus1`, inclusive, the variable `numLayersInIdList[ i ]` and the layer identifier list `layerSetLayerIdList[ i ]` are derived as follows:

$$\begin{aligned} n &= 0 \\ \text{for}(m = 0; m \leq \text{vps_max_layer_id}; m++) \\ \quad \text{if}(\text{layer_id_included_flag}[i][m]) \\ \quad \quad \text{layerSetLayerIdList}[i][n++] = m \\ \text{numLayersInIdList}[i] &= n \end{aligned} \quad (7-3)$$

For each value of `i` in the range of 1 to `vps_num_layer_sets_minus1`, inclusive, `numLayersInIdList[ i ]` shall be in the range of 1 to `vps_max_layers_minus1 + 1`, inclusive.

When `numLayersInIdList[ iA ]` is equal to `numLayersInIdList[ iB ]` for any `iA` and `iB` in the range of 0 to `vps_num_layer_sets_minus1`, inclusive, with `iA` not equal to `iB`, the value of `layerSetLayerIdList[ iA ][ n ]` shall not be equal to `layerSetLayerIdList[ iB ][ n ]` for at least one value of `n` in the range of 0 to `numLayersInIdList[ iA ]`, inclusive.

A layer set is identified by the associated layer identifier list. The `i`-th layer set specified by the VPS is associated with the layer identifier list `layerSetLayerIdList[ i ]`, for `i` in the range of 0 to `vps_num_layer_sets_minus1`, inclusive.

A layer set consists of all operation points that are associated with the same layer identifier list.

Each operation point is identified by the associated layer identifier list, denoted as `OpLayerIdList`, which consists of the list of `nuh_layer_id` values of all NAL units included in the operation point, in increasing order of `nuh_layer_id` values, and a variable `OpTid`, which is equal to the highest `TemporalId` of all NAL units included in the operation point. The bitstream subset associated with the operation point identified by `OpLayerIdList` and `OpTid` is the output of the sub-bitstream extraction process as specified in clause 10 with the bitstream, the target highest `TemporalId` equal to `OpTid`, and the target layer identifier list equal to `OpLayerIdList` as inputs. The `OpLayerIdList` and `OpTid` that identify an operation point are also referred to as the `OpLayerIdList` and `OpTid` associated with the operation point, respectively.

`vps_timing_info_present_flag` equal to 1 specifies that `vps_num_units_in_tick`, `vps_time_scale`, `vps_poc_proportional_to_timing_flag`, and `vps_num_hrd_parameters` are present in the VPS. **`vps_timing_info_present_flag`** equal to 0 specifies that `vps_num_units_in_tick`, `vps_time_scale`, `vps_poc_proportional_to_timing_flag`, and `vps_num_hrd_parameters` are not present in the VPS.

`vps_num_units_in_tick` is the number of time units of a clock operating at the frequency `vps_time_scale` Hz that corresponds to one increment (called a clock tick) of a clock tick counter. The value of `vps_num_units_in_tick` shall be greater than 0. A clock tick, in units of seconds, is equal to the quotient of `vps_num_units_in_tick` divided by

vps_time_scale. For example, when the picture rate of a video signal is 25 Hz, **vps_time_scale** may be equal to 27 000 000 and **vps_num_units_in_tick** may be equal to 1 080 000, and consequently a clock tick may be 0.04 seconds.

vps_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 **vps_time_scale** of 27 000 000. The value of **vps_time_scale** shall be greater than 0.

vps_poc_proportional_to_timing_flag equal to 1 indicates that the picture order count value for each picture in the CVS that is not the first picture in the CVS, in decoding order, is proportional to the output time of the picture relative to the output time of the first picture in the CVS. **vps_poc_proportional_to_timing_flag** equal to 0 indicates that the picture order count value for each picture in the CVS that is not the first picture in the CVS, in decoding order, may or may not be proportional to the output time of the picture relative to the output time of the first picture in the CVS.

vps_num_ticks_poc_diff_one_minus1 plus 1 specifies the number of clock ticks corresponding to a difference of picture order count values equal to 1. The value of **vps_num_ticks_poc_diff_one_minus1** shall be in the range of 0 to $2^{32} - 2$, inclusive.

vps_num_hrd_parameters specifies the number of **hrd_parameters()** syntax structures present in the VPS RBSP. In bitstreams conforming to this version of this Specification, the value of **vps_num_hrd_parameters** shall be less than or equal to 1. Although the value of **vps_num_hrd_parameters** is required to be less than or equal to 1 in this version of this Specification, decoders shall allow other values of **vps_num_hrd_parameters** in the range of 0 to 1024, inclusive, to appear in the syntax.

hrd_layer_set_idx[i] specifies the index, into the list of layer sets specified by the VPS, of the layer set to which the *i*-th **hrd_parameters()** syntax structure in the VPS applies. In bitstreams conforming to this version of this Specification, the value of **hrd_layer_set_idx[i]** shall be equal to 0. Although the value of **hrd_layer_set_idx[i]** is required to be equal to 0 in this version of this Specification, decoders shall allow other values of **hrd_layer_set_idx[i]** in the range of 0 to 1023, inclusive, to appear in the syntax.

cprms_present_flag[i] equal to 1 specifies that the HRD parameters that are common for all sub-layers are present in the *i*-th **hrd_parameters()** syntax structure in the VPS. **cprms_present_flag[i]** equal to 0 specifies that the HRD parameters that are common for all sub-layers are not present in the *i*-th **hrd_parameters()** syntax structure in the VPS and are derived to be the same as the (*i* - 1)-th **hrd_parameters()** syntax structure in the VPS. **cprms_present_flag[0]** is inferred to be equal to 1.

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** shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for **vps_extension_flag** is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for **vps_extension_flag** in a VPS NAL unit.

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.

7.4.3.2 Sequence parameter set RBSP semantics

sps_video_parameter_set_id specifies the value of the **vps_video_parameter_set_id** of the active VPS.

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_temporal_id_nesting_flag, when **sps_max_sub_layers_minus1** is greater than 0, specifies whether inter prediction is additionally restricted for CVSs referring to the SPS. When **vps_temporal_id_nesting_flag** is equal to 1, **sps_temporal_id_nesting_flag** shall be equal to 1. When **sps_max_sub_layers_minus1** is equal to 0, **sps_temporal_id_nesting_flag** shall be equal to 1.

NOTE 1 – The syntax element **sps_temporal_id_nesting_flag** is used to indicate that temporal up-switching, i.e. switching from decoding up to any TemporalId tIdN to decoding up to any TemporalId tIdM that is greater than tIdN, is always possible in the CVS.

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 subclause 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 2 – 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 3 – 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 BitDepth_Y and the value of the luma quantization parameter range offset QpBdOffset_Y as follows:

$$\text{BitDepth}_Y = 8 + \text{bit_depth_luma_minus8} \quad (7-4)$$

$$\text{QpBdOffset}_Y = 6 * \text{bit_depth_luma_minus8} \quad (7-5)$$

bit_depth_luma_minus8 shall be in the range of 0 to 6, inclusive.

bit_depth_chroma_minus8 specifies the bit depth of the samples of the chroma arrays BitDepth_C and the value of the chroma quantization parameter range offset QpBdOffset_C as follows:

$$\text{BitDepth}_C = 8 + \text{bit_depth_chroma_minus8} \quad (7-6)$$

$$\text{QpBdOffset}_C = 6 * \text{bit_depth_chroma_minus8} \quad (7-7)$$

bit_depth_chroma_minus8 shall be in the range of 0 to 6, 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:

$$\text{MaxPicOrderCntLsb} = 2^{(\text{log2_max_pic_order_cnt_lsb_minus4} + 4)} \quad (7-8)$$

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` (as specified in subclause A.4), inclusive. 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 ]`. The value of `sps_max_dec_pic_buffering_minus1[ i ]` shall be less than or equal to `vps_max_dec_pic_buffering_minus1[ i ]` for each value of `i`. 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 ]`. The value of `sps_max_num_reorder_pics[ i ]` shall be less than or equal to `vps_max_num_reorder_pics[ i ]` for each value of `i`. 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:

$$\text{SpsMaxLatencyPictures}[i] = \text{sps_max_num_reorder_pics}[i] + \text{sps_max_latency_increase_plus1}[i] - 1 \quad (7-9)$$

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 $2^{32} - 2$, inclusive. When `vps_max_latency_increase_plus1[ i ]` is not equal to 0, the value of `sps_max_latency_increase_plus1[ i ]` shall not be equal to 0 and shall be less than or equal to `vps_max_latency_increase_plus1[ i ]` for each value of `i`. 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 ]`.

`log2_min_luma_coding_block_size_minus3` plus 3 specifies the minimum size of a luma coding block.

`log2_diff_max_min_luma_coding_block_size` specifies the difference between the maximum and minimum luma coding block size.

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

$$\text{MinCbLog2SizeY} = \text{log2_min_luma_coding_block_size_minus3} + 3 \quad (7-10)$$

$$\text{CtbLog2SizeY} = \text{MinCbLog2SizeY} + \text{log2_diff_max_min_luma_coding_block_size} \quad (7-11)$$

$$\text{MinCbSizeY} = 1 \ll \text{MinCbLog2SizeY} \quad (7-12)$$

$$\text{CtbSizeY} = 1 \ll \text{CtbLog2SizeY} \quad (7-13)$$

$$\text{PicWidthInMinCbsY} = \text{pic_width_in_luma_samples} / \text{MinCbSizeY} \quad (7-14)$$

$$\text{PicWidthInCtbsY} = \text{Ceil}(\text{pic_width_in_luma_samples} \div \text{CtbSizeY}) \quad (7-15)$$

$$\text{PicHeightInMinCbsY} = \text{pic_height_in_luma_samples} / \text{MinCbSizeY} \quad (7-16)$$

$$\text{PicHeightInCtbsY} = \text{Ceil}(\text{pic_height_in_luma_samples} \div \text{CtbSizeY}) \quad (7-17)$$

$$\text{PicSizeInMinCbsY} = \text{PicWidthInMinCbsY} * \text{PicHeightInMinCbsY} \quad (7-18)$$

$$\text{PicSizeInCtbsY} = \text{PicWidthInCtbsY} * \text{PicHeightInCtbsY} \quad (7-19)$$

$$\text{PicSizeInSamplesY} = \text{pic_width_in_luma_samples} * \text{pic_height_in_luma_samples} \quad (7-20)$$

$$\text{PicWidthInSamplesC} = \text{pic_width_in_luma_samples} / \text{SubWidthC} \quad (7-21)$$

$$\text{PicHeightInSamplesC} = \text{pic_height_in_luma_samples} / \text{SubHeightC} \quad (7-22)$$

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

- If chroma_format_idc is equal to 0 (monochrome) or separate_colour_plane_flag is equal to 1, CtbWidthC and CtbHeightC are both equal to 0.

- Otherwise, CtbWidthC and CtbHeightC are derived as follows:

$$\text{CtbWidthC} = \text{CtbSizeY} / \text{SubWidthC} \quad (7-23)$$

$$\text{CtbHeightC} = \text{CtbSizeY} / \text{SubHeightC} \quad (7-24)$$

log2_min_transform_block_size_minus2 plus 2 specifies the minimum transform block size.

The variable Log2MinTrafoSize is set equal to log2_min_transform_block_size_minus2 + 2. The bitstream shall not contain data that result in Log2MinTrafoSize greater than or equal to MinCbLog2SizeY.

log2_diff_max_min_transform_block_size specifies the difference between the maximum and minimum transform block size.

The variable Log2MaxTrafoSize is set equal to log2_min_transform_block_size_minus2 + 2 + log2_diff_max_min_transform_block_size.

The bitstream shall not contain data that result in Log2MaxTrafoSize greater than Min(CtbLog2SizeY, 5).

The array ScanOrder[log2BlockSize][scanIdx][sPos][sComp] specifies the mapping of the scan position sPos, ranging from 0 to (1 << log2BlockSize) * (1 << log2BlockSize) - 1, inclusive, to horizontal and vertical components of the scan-order matrix. The array index scanIdx equal to 0 specifies an up-right diagonal scan order, scanIdx equal to 1 specifies a horizontal scan order, and scanIdx equal to 2 specifies a vertical scan order. The array index sComp equal to 0 specifies the horizontal component and the array index sComp equal to 1 specifies the vertical component. The array ScanOrder is derived as follows:

For the variable log2BlockSize ranging from 0 to 3, inclusive, the scanning order array ScanOrder is derived as follows:

- The up-right diagonal scan order array initialization process as specified in subclause 6.5.3 is invoked with 1 << log2BlockSize as input, and the output is assigned to ScanOrder[log2BlockSize][0].
- The horizontal scan order array initialization process as specified in subclause 6.5.4 is invoked with 1 << log2BlockSize as input, and the output is assigned to ScanOrder[log2BlockSize][1].
- The vertical scan order array initialization process as specified in subclause 6.5.5 is invoked with 1 << log2BlockSize as input, and the output is assigned to ScanOrder[log2BlockSize][2].

max_transform_hierarchy_depth_inter specifies the maximum hierarchy depth for transform units of coding units coded in inter prediction mode. The value of max_transform_hierarchy_depth_inter shall be in the range of 0 to CtbLog2SizeY - Log2MinTrafoSize, inclusive.

max_transform_hierarchy_depth_intra specifies the maximum hierarchy depth for transform blocks of coding blocks coded in intra prediction mode. The value of max_transform_hierarchy_depth_intra shall be in the range of 0 to CtbLog2SizeY - Log2MinTrafoSize, inclusive.

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 scaling list data are present in the SPS. sps_scaling_list_data_present_flag equal to 0 specifies that scaling list data are not present in the SPS. When not present, the value of sps_scaling_list_data_present_flag is inferred to be equal to 0. When scaling_list_enabled_flag is equal to 1 and sps_scaling_list_data_present_flag is equal to 0, the default scaling list data are used to derive the array ScalingFactor as described in the scaling list data semantics specified in subclause 7.4.5.

amp_enabled_flag equal to 1 specifies that asymmetric motion partitions, i.e. PartMode equal to PART_2NxN, PART_2NxND, PART_nLx2N, or PART_nRx2N, may be used in coding tree blocks. amp_enabled_flag equal to 0 specifies that asymmetric motion partitions cannot be used in coding tree blocks.

sample_adaptive_offset_enabled_flag equal to 1 specifies that the sample adaptive offset process is applied to the reconstructed picture after the deblocking filter process. sample_adaptive_offset_enabled_flag equal to 0 specifies that the sample adaptive offset process is not applied to the reconstructed picture after the deblocking filter process.

pcm_enabled_flag equal to 0 specifies that PCM data are not present in the CVS.

NOTE 4 – When `MinCbLog2SizeY` is equal to 6, PCM data are not present in the CVS even when `pcm_enabled_flag` is equal to 1. The maximum size of coding block with `pcm_enabled_flag` equal to 1 is restricted to be less than or equal to $\text{Min}(\text{CtbLog2SizeY}, 5)$. Encoders are encouraged to use an appropriate combination of `log2_min_luma_coding_block_size_minus3`, `log2_min_pcm_luma_coding_block_size_minus3`, and `log2_diff_max_min_pcm_luma_coding_block_size` values when sending PCM 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:

$$\text{PcmBitDepth}_Y = \text{pcm_sample_bit_depth_luma_minus1} + 1 \quad (7-25)$$

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:

$$\text{PcmBitDepth}_C = \text{pcm_sample_bit_depth_chroma_minus1} + 1 \quad (7-26)$$

The value of `PcmBitDepthC` shall be less than or equal to the value of `BitDepthC`.

`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 `MinCbLog2SizeY` to $\text{Min}(\text{CtbLog2SizeY}, 5)$, 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 $\text{Min}(\text{CtbLog2SizeY}, 5)$.

`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 and sample adaptive offset 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 and sample adaptive offset 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.

`num_short_term_ref_pic_sets` specifies the number of `short_term_ref_pic_set()` syntax structures included in the SPS. The value of `num_short_term_ref_pic_sets` shall be in the range of 0 to 64, inclusive.

NOTE 5 – A decoder should allocate memory for a total number of `num_short_term_ref_pic_sets` + 1 `short_term_ref_pic_set()` syntax structures since there may be a `short_term_ref_pic_set()` syntax structure directly signalled in the slice headers of a current picture. A `short_term_ref_pic_set()` syntax structure directly signalled in the slice headers of a current picture has an index equal to `num_short_term_ref_pic_sets`.

`long_term_ref_pics_present_flag` equal to 0 specifies that no long-term reference picture is used for inter prediction of any coded picture in the CVS. `long_term_ref_pics_present_flag` equal to 1 specifies that long-term reference pictures may be used for inter prediction of one or more coded pictures in the CVS.

`num_long_term_ref_pics_sps` specifies the number of candidate long-term reference pictures that are specified in the SPS. The value of `num_long_term_ref_pics_sps` shall be in the range of 0 to 32, inclusive.

`lt_ref_pic_poc_lsb_sps[i]` specifies the picture order count modulo `MaxPicOrderCntLsb` of the *i*-th candidate long-term reference picture specified in the SPS. The number of bits used to represent `lt_ref_pic_poc_lsb_sps[i]` is equal to `log2_max_pic_order_cnt_lsb_minus4` + 4.

`used_by_curr_pic_lt_sps_flag[i]` equal to 0 specifies that the *i*-th candidate long-term reference picture specified in the SPS is not used for reference by a picture that includes in its long-term RPS the *i*-th candidate long-term reference picture specified in the SPS.

`sps_temporal_mvp_enabled_flag` equal to 1 specifies that `slice_temporal_mvp_enabled_flag` is present in the slice headers of non-IDR pictures 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.

`strong_intra_smoothing_enabled_flag` equal to 1 specifies that bi-linear interpolation is conditionally used in the filtering process in the CVS as specified in subclause 8.4.4.2.3. `strong_intra_smoothing_enabled_flag` equal to 0 specifies that the bi-linear interpolation is not used in the CVS.

vui_parameters_present_flag equal to 1 specifies that the `vui_parameters()` syntax structure as specified in Annex E is present. `vui_parameters_present_flag` equal to 0 specifies that the `vui_parameters()` syntax structure as specified in Annex E is not present.

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` shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for `sps_extension_flag` is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all `sps_extension_data_flag` syntax elements that follow the value 1 for `sps_extension_flag` in an SPS NAL unit.

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.

7.4.3.3 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.

dependent_slice_segments_enabled_flag equal to 1 specifies the presence of the syntax element `dependent_slice_segment_flag` in the slice segment headers for coded pictures referring to the PPS. `dependent_slice_segments_enabled_flag` equal to 0 specifies the absence of the syntax element `dependent_slice_segment_flag` in the slice segment headers for coded pictures referring to the PPS.

output_flag_present_flag equal to 1 indicates that the `pic_output_flag` syntax element is present in the associated slice headers. `output_flag_present_flag` equal to 0 indicates that the `pic_output_flag` syntax element is not present in the associated slice headers.

num_extra_slice_header_bits equal to 0 specifies that no extra slice header bits are present in the slice header RBSP for coded pictures referring to the PPS. `num_extra_slice_header_bits` shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for `num_extra_slice_header_bits` are reserved for future use by ITU-T | ISO/IEC. However, decoders shall allow `num_extra_slice_header_bits` to have any value.

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.

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_l0_default_active_minus1 specifies the inferred value of `num_ref_idx_l0_active_minus1` for P and B slices with `num_ref_idx_active_override_flag` equal to 0. The value of `num_ref_idx_l0_default_active_minus1` shall be in the range of 0 to 14, inclusive.

num_ref_idx_l1_default_active_minus1 specifies the inferred value of `num_ref_idx_l1_active_minus1` with `num_ref_idx_active_override_flag` equal to 0. The value of `num_ref_idx_l1_default_active_minus1` shall be in the range of 0 to 14, inclusive.

init_qp_minus26 specifies the initial value minus 26 of `SliceQpy` for each slice. The initial value is modified at the slice segment layer when a non-zero value of `slice_qp_delta` is decoded, and is modified further when a non-zero value of `cu_qp_delta_abs` is decoded at the coding unit layer. The value of `init_qp_minus26` shall be in the range of $-(26 + QpBdOffset_y)$ to +25, inclusive.

constrained_intra_pred_flag equal to 0 specifies that intra prediction allows usage of residual data and decoded samples of neighbouring coding blocks coded using either intra or inter prediction modes. `constrained_intra_pred_flag` equal to 1 specifies constrained intra prediction, in which case intra prediction only uses residual data and decoded samples from neighbouring coding blocks coded using intra prediction modes.

transform_skip_enabled_flag equal to 1 specifies that `transform_skip_flag` may be present in the residual coding syntax. `transform_skip_enabled_flag` equal to 0 specifies that `transform_skip_flag` is not present in the residual coding syntax.

cu_qp_delta_enabled_flag equal to 1 specifies that the `diff_cu_qp_delta_depth` 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 `diff_cu_qp_delta_depth` syntax element is not present in the PPS and that `cu_qp_delta_abs` is not present in the transform unit syntax.

diff_cu_qp_delta_depth specifies the difference between the luma coding tree block size and the minimum luma coding block size of coding units that convey `cu_qp_delta_abs` and `cu_qp_delta_sign_flag`. The value of `diff_cu_qp_delta_depth`

shall be in the range of 0 to $\log_2_diff_max_min_luma_coding_block_size$, inclusive. When not present, the value of $diff_cu_qp_delta_depth$ is inferred to be equal to 0.

The variable $Log2MinCuQpDeltaSize$ is derived as follows:

$$Log2MinCuQpDeltaSize = CtbLog2SizeY - diff_cu_qp_delta_depth \quad (7-27)$$

pps_cb_qp_offset and **pps_cr_qp_offset** specify 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.

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.

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.

transquant_bypass_enabled_flag equal to 1 specifies that **cu_transquant_bypass_flag** is present. **transquant_bypass_enabled_flag** equal to 0 specifies that **cu_transquant_bypass_flag** is not present.

tiles_enabled_flag equal to 1 specifies that there is more than one tile in each picture referring to the PPS. **tiles_enabled_flag** equal to 0 specifies that there is only one tile in each picture referring to the PPS.

It is a requirement of bitstream conformance that the value of **tiles_enabled_flag** shall be the same for all PPSs that are activated within a CVS.

entropy_coding_sync_enabled_flag equal to 1 specifies that a specific synchronization process for context variables is invoked before decoding the coding tree unit which includes the first coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS, and a specific storage process for context variables is invoked after decoding the coding tree unit which includes the second coding tree block of a row of coding tree blocks in each tile 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 coding tree unit which includes the first coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS, and no specific storage process for context variables is required to be invoked after decoding the coding tree unit which includes the second coding tree block of a row of coding tree blocks in each tile 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.

When **entropy_coding_sync_enabled_flag** is equal to 1 and the first coding tree block in a slice is not the first coding tree block of a row of coding tree blocks in a tile, it is a requirement of bitstream conformance that the last coding tree block in the slice shall belong to the same row of coding tree blocks as the first coding tree block in the slice.

When **entropy_coding_sync_enabled_flag** is equal to 1 and the first coding tree block in a slice segment is not the first coding tree block of a row of coding tree blocks in a tile, it is a requirement of bitstream conformance that the last coding tree block in the slice segment shall belong to the same row of coding tree blocks as the first coding tree block in the slice segment.

num_tile_columns_minus1 plus 1 specifies the number of tile columns partitioning the picture. **num_tile_columns_minus1** shall be in the range of 0 to $PicWidthInCtbsY - 1$, inclusive. When not present, the value of **num_tile_columns_minus1** is inferred to be equal to 0.

num_tile_rows_minus1 plus 1 specifies the number of tile rows partitioning the picture. **num_tile_rows_minus1** shall be in the range of 0 to $PicHeightInCtbsY - 1$, inclusive. When not present, the value of **num_tile_rows_minus1** is inferred to be equal to 0.

When **tiles_enabled_flag** is equal to 1, **num_tile_columns_minus1** and **num_tile_rows_minus1** shall not be both equal to 0.

uniform_spacing_flag equal to 1 specifies that tile column boundaries and likewise tile row boundaries are distributed uniformly across the picture. **uniform_spacing_flag** equal to 0 specifies that tile column boundaries and likewise tile row boundaries are not distributed uniformly across the picture but signalled explicitly using the syntax elements **column_width_minus1[i]** and **row_height_minus1[i]**. When not present, the value of **uniform_spacing_flag** is inferred to be equal to 1.

column_width_minus1[i] plus 1 specifies the width of the i-th tile column in units of coding tree blocks.

row_height_minus1[i] plus 1 specifies the height of the i-th tile row in units of coding tree blocks.

The following variables are derived by invoking the coding tree block raster and tile scanning conversion process as specified in subclause 6.5.1:

- The list **CtbAddrRsToTs[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 tile scan,
- the list **CtbAddrTsToRs[ctbAddrTs]** for **ctbAddrTs** ranging from 0 to **PicSizeInCtbsY – 1**, inclusive, specifying the conversion from a CTB address in tile scan to a CTB address in CTB raster scan of a picture,
- the list **TileId[ctbAddrTs]** for **ctbAddrTs** ranging from 0 to **PicSizeInCtbsY – 1**, inclusive, specifying the conversion from a CTB address in tile scan to a tile ID,
- the list **ColumnWidthInLumaSamples[i]** for **i** ranging from 0 to **num_tile_columns_minus1**, inclusive, specifying the width of the i-th tile column in units of luma samples,
- the list **RowHeightInLumaSamples[j]** for **j** ranging from 0 to **num_tile_rows_minus1**, inclusive, specifying the height of the j-th tile row in units of luma samples.

The values of **ColumnWidthInLumaSamples[i]** for **i** ranging from 0 to **num_tile_columns_minus1**, inclusive, and **RowHeightInLumaSamples[j]** for **j** ranging from 0 to **num_tile_rows_minus1**, inclusive, shall all be greater than 0.

The array **MinTbAddrZs** with elements **MinTbAddrZs[x][y]** for **x** ranging from 0 to $(\text{PicWidthInCtbsY} \ll (\text{CtbLog2SizeY} - \text{Log2MinTrafoSize})) - 1$, inclusive, and **y** ranging from 0 to $(\text{PicHeightInCtbsY} \ll (\text{CtbLog2SizeY} - \text{Log2MinTrafoSize})) - 1$, inclusive, specifying the conversion from a location (x, y) in units of minimum transform blocks to a transform block address in z-scan order, is derived by invoking the z-scan order array initialization process as specified in subclause 6.5.2.

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

pps_loop_filter_across_slices_enabled_flag equal to 1 specifies that in-loop filtering operations may be performed across left and upper boundaries of slices referring to the PPS. **pps_loop_filter_across_slices_enabled_flag** equal to 0 specifies that in-loop filtering operations are not performed across left and upper boundaries of slices referring to the PPS. The in-loop filtering operations include the deblocking filter and sample adaptive offset filter operations.

NOTE 1 – Loop filtering across slice boundaries can be enabled while loop filtering across tile boundaries is disabled and vice versa.

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_scaling_list_data_present_flag equal to 1 specifies that parameters are present in the PPS to modify the scaling lists specified in the active SPS. **pps_scaling_list_data_present_flag** equal to 0 specifies that the scaling lists used for the pictures referring to the PPS is 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 described in the scaling list data semantics 7.4.5.

lists_modification_present_flag equal to 1 specifies that the syntax structure `ref_pic_lists_modification()` is present in the slice segment header. **lists_modification_present_flag** equal to 0 specifies that the syntax structure `ref_pic_lists_modification()` is not present in the slice segment header.

log2_parallel_merge_level_minus2 plus 2 specifies the value of the variable `Log2ParMrgLevel`, which is used in the derivation process for luma motion vectors for merge mode as specified in subclause 8.5.3.2.1 and the derivation process for spatial merging candidates as specified in subclause 8.5.3.2.2. The value of **log2_parallel_merge_level_minus2** shall be in the range of 0 to `CtbLog2SizeY - 2`, inclusive.

The variable `Log2ParMrgLevel` is derived as follows:

$$\text{Log2ParMrgLevel} = \text{log2_parallel_merge_level_minus2} + 2 \quad (7-28)$$

NOTE 2 – The value of `Log2ParMrgLevel` indicates the built-in capability of parallel derivation of the merging candidate lists. For example, when `Log2ParMrgLevel` is equal to 6, the merging candidate lists for all the PUs and CUs contained in a 64x64 block can be derived in parallel.

slice_segment_header_extension_present_flag equal to 0 specifies that no slice segment header extension syntax elements are present in the slice segment headers for coded pictures referring to the PPS. **slice_segment_header_extension_present_flag** shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for **slice_segment_header_extension_present_flag** is reserved for future use by ITU-T | ISO/IEC.

pps_extension_flag equal to 0 specifies that no **pps_extension_data_flag** syntax elements are present in the PPS RBSP syntax structure. **pps_extension_flag** shall be equal to 0 in bitstreams conforming to this version of this Specification. The value of 1 for **pps_extension_flag** is reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore all data that follow the value 1 for **pps_extension_flag** in a PPS NAL unit.

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.

7.4.3.4 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.

7.4.3.5 Access unit delimiter RBSP semantics

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

pic_type indicates that the **slice_type** values for all slices of the coded picture are members of the set listed in Table 7-2 for the given value of **pic_type**.

Table 7-2 – 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

7.4.3.6 End of sequence RBSP semantics

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 with `NoRaslOutputFlag` equal to 1. The syntax content of the SODB and RBSP for the end of sequence RBSP are empty.

7.4.3.7 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.

NOTE – When an elementary stream contains more than one bitstream, the last NAL unit of the last access unit of a bitstream must contain an end of bitstream NAL unit and the first access unit of the subsequent bitstream must be an IRAP access unit. This IRAP access unit may be a CRA, BLA, or IDR access unit.

7.4.3.8 Filler data RBSP semantics

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

ff_byte is a byte equal to 0xFF.

7.4.3.9 Slice segment layer RBSP semantics

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

7.4.3.10 RBSP slice segment trailing bits semantics

cabac_zero_word is a byte-aligned sequence of two bytes equal to 0x0000.

Let **NumBytesInVclNalUnits** be the sum of the values of **NumBytesInNalUnit** for all VCL NAL units of a coded picture.

Let **BinCountsInNalUnits** be the number of times that the parsing process function **DecodeBin()**, specified in subclause 9.3.4.3, is invoked to decode the contents of all VCL NAL units of a coded picture.

Let the variable **RawMinCuBits** be derived as follows:

$$\text{RawMinCuBits} = \text{MinCbSizeY} * \text{MinCbSizeY} * (\text{BitDepth}_Y + \text{BitDepth}_C / 2) \quad (7-29)$$

The value of **BinCountsInNalUnits** shall be less than or equal to $(32 \div 3) * \text{NumBytesInVclNalUnits} + (\text{RawMinCuBits} * \text{PicSizeInMinCbsY}) \div 32$.

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

7.4.3.11 RBSP trailing bits semantics

rbsp_stop_one_bit shall be equal to 1.

rbsp_alignment_zero_bit shall be equal to 0.

7.4.3.12 Byte alignment semantics

alignment_bit_equal_to_one shall be equal to 1.

alignment_bit_equal_to_zero shall be equal to 0.

7.4.4 Profile, tier and level semantics

general_profile_space specifies the context for the interpretation of **general_profile_idc** and **general_profile_compatibility_flag[i]** for all values of *i* in the range of 0 to 31, inclusive. The value of **general_profile_space** shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for **general_profile_space** are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the CVS when **general_profile_space** is not equal to 0.

general_tier_flag specifies the tier context for the interpretation of **general_level_idc** as specified in Annex A.

general_profile_idc, when **general_profile_space** is equal to 0, indicates a profile to which the CVS 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_profile_compatibility_flag[j] equal to 1, when **general_profile_space** is equal to 0, indicates that the CVS conforms to the profile indicated by **general_profile_idc** equal to *i* as specified in Annex A. When **general_profile_space** is equal to 0, **general_profile_compatibility_flag[general_profile_idc]** shall be equal to 1. The value of **general_profile_compatibility_flag[j]** shall be equal to 0 for any value of *j* that is not specified as an allowed value of **general_profile_idc** in Annex A.

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 the CVS 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 the CVS 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 the CVS 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 the CVS is indicated 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` for purposes other than determining the value to be inferred for `frame_field_info_present_flag` when `vui_params_present_flag` is equal to 0, as there are no other decoding process requirements associated with the values of these flags. 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 specifies that there are no frame packing arrangement SEI messages present in the CVS. `general_non_packed_constraint_flag` equal to 0 indicates that there may or may not be one or more frame packing arrangement SEI messages present in the CVS.

NOTE 2 – Decoders may ignore the value of `general_non_packed_constraint_flag`, as there are no decoding process requirements associated with the presence or interpretation of frame packing arrangement SEI messages.

`general_frame_only_constraint_flag` equal to 1 specifies that `field_seq_flag` is equal to 0. `general_frame_only_constraint_flag` equal to 0 indicates that `field_seq_flag` may or may not be equal to 0.

NOTE 3 – Decoders may ignore the value of `general_frame_only_constraint_flag`, as there are no decoding process requirements associated with the value of `field_seq_flag`.

NOTE 4 – When `general_progressive_source_flag` is equal to 1, `general_frame_only_constraint_flag` may or may not be equal to 1.

`general_reserved_zero_44bits` shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for `general_reserved_zero_44bits` are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of `general_reserved_zero_44bits`.

`general_level_idc` indicates a level to which the CVS 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 5 – A greater value of `general_level_idc` indicates a higher level. The maximum level signalled in the VPS for a CVS may be higher than the level signalled in the SPS for the same CVS.

NOTE 6 – When the coded video sequence 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 7 – The `general_reserved_zero_44bits` may be used in future editions of this Specification to indicate further constraints on the bitstream (e.g. that a particular syntax combination that would otherwise be permitted by the indicated values of `general_profile_compatibility_flag[j]`, is not used).

`sub_layer_profile_present_flag[i]` equal to 1 specifies that profile information is present in the `profile_tier_level()` syntax structure for the representation of the sub-layer with `TemporalId` equal to `i`. `sub_layer_profile_present_flag[i]` equal to 0 specifies that profile information is not present in the `profile_tier_level()` syntax structure for the representations of the sub-layer with `TemporalId` equal to `i`.

`sub_layer_level_present_flag[i]` equal to 1 specifies that level information is present in the `profile_tier_level()` syntax structure for the representation of the sub-layer 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 representation of the sub-layer with `TemporalId` equal to `i`.

`reserved_zero_2bits[i]` shall be equal to 0 in bitstreams conforming to this version of this Specification. Other values for `reserved_zero_2bits[i]` are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the value of `reserved_zero_2bits[i]`.

The semantics of the syntax elements **`sub_layer_profile_space[i]`**, **`sub_layer_tier_flag[i]`**, **`sub_layer_profile_idc[i]`**, **`sub_layer_profile_compatibility_flag[i][j]`**, **`sub_layer_progressive_source_flag[i]`**, **`sub_layer_interlaced_source_flag[i]`**, **`sub_layer_non_packed_constraint_flag[i]`**, **`sub_layer_frame_only_constraint_flag[i]`**, **`sub_layer_reserved_zero_44bits[i]`**, and **`sub_layer_level_idc[i]`** are the same as the syntax elements `general_profile_space`, `general_tier_flag`, `general_profile_idc`, `general_profile_compatibility_flag[j]`, `general_progressive_source_flag`, `general_interlaced_source_flag`, `general_non_packed_constraint_flag`, `general_frame_only_constraint_flag`, `general_reserved_zero_44bits`, and `general_level_idc`, respectively, but apply to the representation of the sub-layer with `TemporalId` equal to `i`.

When not present, the value of `sub_layer_tier_flag[i]` is inferred to be equal to 0.

NOTE 8 – It is possible that `sub_layer_tier_flag[i]` is not present and `sub_layer_level_idc[i]` is present. In this case, a default value of `sub_layer_tier_flag[i]` is needed for interpretation of `sub_layer_level_idc[i]`.

7.4.5 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] as follows:

- If **scaling_list_pred_matrix_id_delta** is equal to 0, the scaling list is inferred from the default scaling list **ScalingList**[sizeId][matrixId][i] as specified in Table 7-5 and Table 7-6 for $i = 0..Min(64, (1 \ll (4 + (sizeId \ll 1))))$.
- Otherwise, the scaling list is inferred from the reference scaling list as follows:

$$refMatrixId = matrixId - scaling_list_pred_matrix_id_delta[sizeId][matrixId] \quad (7-30)$$

$$ScalingList[sizeId][matrixId][i] = ScalingList[sizeId][refMatrixId][i] \\ \text{with } i = 0..Min(64, (1 \ll (4 + (sizeId \ll 1)))) \quad (7-31)$$

The value of **scaling_list_pred_matrix_id_delta**[sizeId][matrixId] shall be in the range of 0 to **matrixId**, inclusive.

Table 7-3 – Specification of sizeId

Size of quantization matrix	sizeId
4x4	0
8x8	1
16x16	2
32x32	3

Table 7-4 – Specification of matrixId according to sizeId, prediction mode and colour component

sizeId	CuPredMode	cIdx (colour component)	matrixId
0, 1, 2	MODE_INTRA	0 (Y)	0
0, 1, 2	MODE_INTRA	1 (Cb)	1
0, 1, 2	MODE_INTRA	2 (Cr)	2
0, 1, 2	MODE_INTER	0 (Y)	3
0, 1, 2	MODE_INTER	1 (Cb)	4
0, 1, 2	MODE_INTER	2 (Cr)	5
3	MODE_INTRA	0 (Y)	0
3	MODE_INTER	0 (Y)	1

scaling_list_dc_coef_minus8[sizeId – 2][matrixId] plus 8 specifies the DC value of the scaling list for 16x16 size when sizeId is equal to 2 and specifies the DC value of the scaling list for 32x32 size when sizeId is equal to 3. The value of **scaling_list_dc_coef_minus8**[sizeId – 2][matrixId] shall be in the range of –7 to 247, inclusive. When **scaling_list_dc_coef_minus8** is not present, it is inferred to be equal to 8.

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.

Table 7-5 – Specification of default values of ScalingList[0][matrixId][i] with i = 0..15

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

Table 7-6 – Specification of default values of ScalingList[1..3][matrixId][i] with i = 0..63

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

The four-dimensional array ScalingFactor[sizeId][matrixId][x][y], with $x, y = 0..(1 + \text{sizeId}) - 1$, specifies the array of scaling factors according to the variables sizeId specified in Table 7-3 and matrixId specified in Table 7-4.

The elements of the quantization matrix of size 4x4, ScalingFactor[0][matrixId][][], are derived as follows:

$$\text{ScalingFactor}[0][\text{matrixId}][x][y] = \text{ScalingList}[0][\text{matrixId}][i] \quad (7-32)$$

with $i = 0..15$, $\text{matrixId} = 0..5$, $x = \text{ScanOrder}[2][0][i][0]$, and $y = \text{ScanOrder}[2][0][i][1]$

The elements of the quantization matrix of size 8x8, ScalingFactor[1][matrixId][][], are derived as follows:

$$\text{ScalingFactor}[1][\text{matrixId}][x][y] = \text{ScalingList}[1][\text{matrixId}][i] \quad (7-33)$$

with $i = 0..63$, $\text{matrixId} = 0..5$, $x = \text{ScanOrder}[3][0][i][0]$, and $y = \text{ScanOrder}[3][0][i][1]$

The elements of the quantization matrix of size 16x16, ScalingFactor[2][matrixId][][], are derived as follows:

$$\text{ScalingFactor}[2][\text{matrixId}][x * 2 + k][y * 2 + j] = \text{ScalingList}[2][\text{matrixId}][i] \quad (7-34)$$

with $i = 0..63$, $j = 0..1$, $k = 0..1$, $\text{matrixId} = 0..5$, $x = \text{ScanOrder}[3][0][i][0]$,
and $y = \text{ScanOrder}[3][0][i][1]$

$$\text{ScalingFactor}[2][\text{matrixId}][0][0] = \text{scaling_list_dc_coef_minus8}[0][\text{matrixId}] + 8 \quad (7-35)$$

with $\text{matrixId} = 0..5$

The elements of the quantization matrix of size 32x32, ScalingFactor[3][matrixId][][], are derived as follows:

$$\text{ScalingFactor}[3][\text{matrixId}][x * 4 + k][y * 4 + j] = \text{ScalingList}[3][\text{matrixId}][i] \quad (7-36)$$

with $i = 0..63$, $j = 0..3$, $k = 0..3$, $\text{matrixId} = 0..1$, $x = \text{ScanOrder}[3][0][i][0]$,
and $y = \text{ScanOrder}[3][0][i][1]$

$$\text{ScalingFactor}[3][\text{matrixId}][0][0] = \text{scaling_list_dc_coef_minus8}[1][\text{matrixId}] + 8 \quad (7-37)$$

with $\text{matrixId} = 0..1$

7.4.6 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.

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

last_payload_type_byte is the last byte of the payload type of an SEI message.

last_payload_size_byte is the last byte of the payload size of an SEI message.

7.4.7 Slice segment header semantics

7.4.7.1 General slice segment header semantics

When present, the value of the slice segment header syntax elements `slice_pic_parameter_set_id`, `pic_output_flag`, `no_output_of_prior_pics_flag`, `slice_pic_order_cnt_lsb`, `short_term_ref_pic_set_sps_flag`, `short_term_ref_pic_set_idx`, `num_long_term_sps`, `num_long_term_pics`, and `slice_temporal_mvp_enabled_flag` shall be the same in all slice segment headers of a coded picture. When present, the value of the slice segment header syntax elements `lt_idx_sps[i]`, `poc_lsb_lt[i]`, `used_by_curr_pic_lt_flag[i]`, `delta_poc_msb_present_flag[i]`, and `delta_poc_msb_cycle_lt[i]` shall be the same in all slice segment headers of a coded picture for each possible value of *i*.

first_slice_segment_in_pic_flag equal to 1 specifies that the slice segment is the first slice segment of the picture in decoding order. `first_slice_segment_in_pic_flag` equal to 0 specifies that the slice segment is not the first slice segment of the picture in decoding order.

no_output_of_prior_pics_flag affects the output of previously-decoded pictures in the decoded picture buffer after the decoding of an IDR or a BLA picture that is not the first picture in the bitstream as specified in Annex C.

slice_pic_parameter_set_id specifies the value of `pps_pic_parameter_set` for the PPS in use. The value of `slice_pic_parameter_set_id` shall be in the range of 0 to 63, inclusive.

dependent_slice_segment_flag equal to 1 specifies that the value of each slice segment header syntax element that is not present is inferred to be equal to the value of the corresponding slice segment header syntax element in the slice header. When not present, the value of `dependent_slice_segment_flag` is inferred to be equal to 0.

The variable `SliceAddrRs` is derived as follows:

- If `dependent_slice_segment_flag` is equal to 0, `SliceAddrRs` is set equal to `slice_segment_address`.
- Otherwise, `SliceAddrRs` is set equal to `SliceAddrRs` of the preceding slice segment containing the coding tree block for which the coding tree block address is `CtbAddrTsToRs[ CtbAddrRsToTs[ slice_segment_address ] – 1 ]`.

slice_segment_address specifies the address of the first coding tree block in the slice segment, in coding tree block raster scan of a picture. The length of the `slice_segment_address` syntax element is $\text{Ceil}(\text{Log}_2(\text{PicSizeInCtbsY}))$ bits. The value of `slice_segment_address` shall be in the range of 0 to $\text{PicSizeInCtbsY} - 1$, inclusive and the value of `slice_segment_address` shall not be equal to the value of `slice_segment_address` of any other coded slice segment NAL unit of the same coded picture. When `slice_segment_address` is not present, it is inferred to be equal to 0.

The variable `CtbAddrInRs`, specifying a coding tree block address in coding tree block raster scan of a picture, is set equal to `slice_segment_address`. The variable `CtbAddrInTs`, specifying a coding tree block address in tile scan, is set equal to `CtbAddrRsToTs[ CtbAddrInRs ]`. The variable `CuQpDeltaVal`, specifying the difference between a luma quantization parameter for the coding unit containing `cu_qp_delta_abs` and its prediction, is set equal to 0.

slice_reserved_flag[i] has semantics and values that are reserved for future use by ITU-T | ISO/IEC. Decoders shall ignore the presence and value of `slice_reserved_flag[i]`.

slice_type specifies the coding type of the slice according to Table 7-7.

Table 7-7 – Name association to slice_type

slice_type	Name of slice_type
0	B (B slice)
1	P (P slice)
2	I (I slice)

When `nal_unit_type` has a value in the range of `BLA_W_LP` to `RSV_IRAP_VCL23`, inclusive, i.e. the picture is an IRAP picture, `slice_type` shall be equal to 2.

When `sps_max_dec_pic_buffering_minus1[ TemporalId ]` is equal to 0, `slice_type` shall be equal to 2.

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.

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 1 – There is no dependency between the decoding processes of pictures having different values of **colour_plane_id**.

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 $\log_2 \text{max_pic_order_cnt_lsb_minus4} + 4$ bits. The value of the **slice_pic_order_cnt_lsb** shall be in the range of 0 to **MaxPicOrderCntLsb** – 1, inclusive. When **slice_pic_order_cnt_lsb** is not present, **slice_pic_order_cnt_lsb** is inferred to be equal to 0, except as specified in subclause 8.3.3.1.

short_term_ref_pic_set_sps_flag equal to 1 specifies that the short-term RPS of the current picture is derived based on one of the **short_term_ref_pic_set()** syntax structures in the active SPS that is identified by the syntax element **short_term_ref_pic_set_idx** in the slice header. **short_term_ref_pic_set_sps_flag** equal to 0 specifies that the short-term RPS of the current picture is derived based on the **short_term_ref_pic_set()** syntax structure that is directly included in the slice headers of the current picture. When **num_short_term_ref_pic_sets** is equal to 0, the value of **short_term_ref_pic_set_sps_flag** shall be equal to 0.

short_term_ref_pic_set_idx specifies the index, into the list of the **short_term_ref_pic_set()** syntax structures included in the active SPS, of the **short_term_ref_pic_set()** syntax structure that is used for derivation of the short-term RPS of the current picture. The syntax element **short_term_ref_pic_set_idx** is represented by $\text{Ceil}(\log_2(\text{num_short_term_ref_pic_sets}))$ bits. When not present, the value of **short_term_ref_pic_set_idx** is inferred to be equal to 0. The value of **short_term_ref_pic_set_idx** shall be in the range of 0 to **num_short_term_ref_pic_sets** – 1, inclusive.

The variable **CurrRpsIdx** is derived as follows:

- If **short_term_ref_pic_set_sps_flag** is equal to 1, **CurrRpsIdx** is set equal to **short_term_ref_pic_set_idx**.
- Otherwise, **CurrRpsIdx** is set equal to **num_short_term_ref_pic_sets**.

num_long_term_sps specifies the number of entries in the long-term RPS of the current picture that are derived based on the candidate long-term reference pictures specified in the active SPS. The value of **num_long_term_sps** shall be in the range of 0 to **num_long_term_ref_pics_sps**, inclusive. When not present, the value of **num_long_term_sps** is inferred to be equal to 0.

num_long_term_pics specifies the number of entries in the long-term RPS of the current picture that are directly signalled in the slice header. When not present, the value of **num_long_term_pics** is inferred to be equal to 0.

The sum of **NumNegativePics[CurrRpsIdx]**, **NumPositivePics[CurrRpsIdx]**, **num_long_term_sps**, and **num_long_term_pics** shall be less than or equal to **sps_max_dec_pic_buffering_minus1[sps_max_sub_layers_minus1]**.

lt_idx_sps[i] specifies an index, into the list of candidate long-term reference pictures specified in the active SPS, of the *i*-th entry in the long-term RPS of the current picture. The number of bits used to represent **lt_idx_sps[i]** is equal to $\text{Ceil}(\log_2(\text{num_long_term_ref_pics_sps}))$. When not present, the value of **lt_idx_sps[i]** is inferred to be equal to 0. The value of **lt_idx_sps[i]** shall be in the range of 0 to **num_long_term_ref_pics_sps** – 1, inclusive.

poc_lsb_lt[i] specifies the value of the picture order count modulo **MaxPicOrderCntLsb** of the *i*-th entry in the long-term RPS of the current picture. The length of the **poc_lsb_lt[i]** syntax element is $\log_2 \text{max_pic_order_cnt_lsb_minus4} + 4$ bits.

used_by_curr_pic_lt_flag[i] equal to 0 specifies that the *i*-th entry in the long-term RPS of the current picture is not used for reference by the current picture.

The variables **PocLsbLt[i]** and **UsedByCurrPicLt[i]** are derived as follows:

- If *i* is less than **num_long_term_sps**, **PocLsbLt[i]** is set equal to **lt_ref_pic_poc_lsb_sps[lt_idx_sps[i]]** and **UsedByCurrPicLt[i]** is set equal to **used_by_curr_pic_lt_sps_flag[lt_idx_sps[i]]**.
- Otherwise, **PocLsbLt[i]** is set equal to **poc_lsb_lt[i]** and **UsedByCurrPicLt[i]** is set equal to **used_by_curr_pic_lt_flag[i]**.

delta_poc_msb_present_flag[i] equal to 1 specifies that **delta_poc_msb_cycle_lt[i]** is present. **delta_poc_msb_present_flag[i]** equal to 0 specifies that **delta_poc_msb_cycle_lt[i]** is not present.

Let **prevTid0Pic** be the previous picture in decoding order that has **TemporalId** equal to 0 and is not a RASL picture, a RADL picture, or a sub-layer non-reference picture. Let **setOfPrevPocVals** be a set consisting of the following:

- the **PicOrderCntVal** of **prevTid0Pic**,
- the **PicOrderCntVal** of each picture in the RPS of **prevTid0Pic**,
- the **PicOrderCntVal** of each picture that follows **prevTid0Pic** in decoding order 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 ]`, `delta_poc_msb_present_flag[ i ]` shall be equal to 1.

delta_poc_msb_cycle_lt[i] is used to determine the value of the most significant bits of the picture order count value of the *i*-th entry in the long-term RPS of the current picture. When `delta_poc_msb_cycle_lt[ i ]` is not present, it is inferred to be equal to 0.

The variable `DeltaPocMsbCycleLt[ i ]` is derived as follows:

$$\begin{aligned} &\text{if}(i == 0 \parallel i == \text{num_long_term_sps}) \\ &\quad \text{DeltaPocMsbCycleLt}[i] = \text{delta_poc_msb_cycle_lt}[i] \\ &\text{else} \\ &\quad \text{DeltaPocMsbCycleLt}[i] = \text{delta_poc_msb_cycle_lt}[i] + \text{DeltaPocMsbCycleLt}[i - 1] \end{aligned} \quad (7-38)$$

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.

When both `slice_temporal_mvp_enabled_flag` and `TemporalId` are equal to 0, the syntax elements for all coded pictures that follow the current picture in decoding order shall be constrained such that no temporal motion vector from any picture that precedes the current picture in decoding order is used in decoding of any coded picture that follows the current picture in decoding order.

NOTE 2 – When `slice_temporal_mvp_enabled_flag` is equal to 0 in an I slice, it has no impact on the normative decoding process of the picture but merely expresses a bitstream constraint.

NOTE 3 – When `slice_temporal_mvp_enabled_flag` is equal to 0 in a slice with `TemporalId` are equal to 0, decoders may empty "motion vector storage" for all reference pictures in the decoded picture buffer.

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.

num_ref_idx_active_override_flag equal to 1 specifies that the syntax element `num_ref_idx_l0_active_minus1` is present for P and B slices and that the syntax element `num_ref_idx_l1_active_minus1` is present for B slices. `num_ref_idx_active_override_flag` equal to 0 specifies that the syntax elements `num_ref_idx_l0_active_minus1` and `num_ref_idx_l1_active_minus1` are not present.

num_ref_idx_l0_active_minus1 specifies the maximum reference index for reference picture list 0 that may be used to decode the slice. `num_ref_idx_l0_active_minus1` shall be in the range of 0 to 14, inclusive. When the current slice is a P or B slice and `num_ref_idx_l0_active_minus1` is not present, `num_ref_idx_l0_active_minus1` is inferred to be equal to `num_ref_idx_l0_default_active_minus1`.

num_ref_idx_l1_active_minus1 specifies the maximum reference index for reference picture list 1 that may be used to decode the slice. `num_ref_idx_l1_active_minus1` shall be in the range of 0 to 14, inclusive. When `num_ref_idx_l1_active_minus1` is not present, `num_ref_idx_l1_active_minus1` is inferred to be equal to `num_ref_idx_l1_default_active_minus1`.

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.

collocated_ref_idx specifies the reference index of the collocated picture used for temporal motion vector prediction.

When `slice_type` is equal to P or when `slice_type` is equal to B and `collocated_from_l0` is equal to 1, `collocated_ref_idx` refers to a picture in list 0, and the value of `collocated_ref_idx` shall be in the range of 0 to `num_ref_idx_l0_active_minus1`, inclusive.

When `slice_type` is equal to B and `collocated_from_l0` is equal to 0, `collocated_ref_idx` refers to a picture in list 1, and the value of `collocated_ref_idx` shall be in the range of 0 to `num_ref_idx_l1_active_minus1`, inclusive.

It is a requirement of bitstream conformance that the picture referred to by `collocated_ref_idx` shall be the same for all slices of a coded picture.

five_minus_max_num_merge_cand specifies the maximum number of merging MVP candidates supported in the slice subtracted from 5. The maximum number of merging MVP candidates, `MaxNumMergeCand` is derived as follows:

$$\text{MaxNumMergeCand} = 5 - \text{five_minus_max_num_merge_cand} \quad (7-39)$$

The value of `MaxNumMergeCand` shall be in the range of 1 to 5, inclusive.

slice_qp_delta specifies the initial value of Q_{p_Y} 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 Q_{p_Y} quantization parameter for the slice, `SliceQpY`, is derived as follows:

$$\text{SliceQpY} = 26 + \text{init_qp_minus26} + \text{slice_qp_delta} \quad (7-40)$$

The value of `SliceQpY` shall be in the range of $-Q_{pBdOffset_Y}$ to +51, 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 $Q_{p'_{cb}}$ 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 $Q_{p'_{cr}}$ 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.

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_loop_filter_across_slices_enabled_flag equal to 1 specifies that in-loop filtering operations may be performed across the left and upper boundaries of the current slice. `slice_loop_filter_across_slices_enabled_flag` equal to 0 specifies that in-loop operations are not performed across left and upper boundaries of the current slice. The in-loop filtering operations include the deblocking filter and sample adaptive offset filter. When `slice_loop_filter_across_slices_enabled_flag` is not present, it is inferred to be equal to `pps_loop_filter_across_slices_enabled_flag`.

num_entry_point_offsets specifies the number of `entry_point_offset_minus1[i]` syntax elements in the slice header. When not present, the value of `num_entry_point_offsets` is inferred to be equal to 0.

The value of `num_entry_point_offsets` is constrained as follows:

- If `tiles_enabled_flag` is equal to 0 and `entropy_coding_sync_enabled_flag` is equal to 1, the value of `num_entry_point_offsets` shall be in the range of 0 to `PicHeightInCtbsY` - 1, inclusive.
- Otherwise, if `tiles_enabled_flag` is equal to 1 and `entropy_coding_sync_enabled_flag` is equal to 0, the value of `num_entry_point_offsets` shall be in the range of 0 to $(\text{num_tile_columns_minus1} + 1) * (\text{num_tile_rows_minus1} + 1) - 1$, inclusive.
- Otherwise, when `tiles_enabled_flag` is equal to 1 and `entropy_coding_sync_enabled_flag` is equal to 1, the value of `num_entry_point_offsets` shall be in the range of 0 to $(\text{num_tile_columns_minus1} + 1) * \text{PicHeightInCtbsY} - 1$, inclusive.

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 segment data that follows the slice segment header consists of **num_entry_point_offsets** + 1 subsets, with subset index values ranging from 0 to **num_entry_point_offsets**, inclusive. The first byte of the slice segment data is considered byte 0. When present, emulation prevention bytes that appear in the slice segment data portion of the coded slice segment NAL unit are counted as part of the slice segment data for purposes of subset identification. Subset 0 consists of bytes 0 to **entry_point_offset_minus1**[0], inclusive, of the coded slice segment data, subset *k*, with *k* in the range of 1 to **num_entry_point_offsets** – 1, inclusive, consists of bytes **firstByte**[*k*] to **lastByte**[*k*], inclusive, of the coded slice segment data with **firstByte**[*k*] and **lastByte**[*k*] defined as:

$$\text{firstByte}[k] = \sum_{n=1}^k (\text{entry_point_offset_minus1}[n-1] + 1) \quad (7-41)$$

$$\text{lastByte}[k] = \text{firstByte}[k] + \text{entry_point_offset_minus1}[k] \quad (7-42)$$

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

When **tiles_enabled_flag** is equal to 1 and **entropy_coding_sync_enabled_flag** is equal to 0, each subset shall consist of all coded bits of all coding tree units in the slice segment that are within the same tile, and the number of subsets (i.e. the value of **num_entry_point_offsets** + 1) shall be equal to the number of tiles that contain coding tree units that are in the coded slice segment.

NOTE 4 – When **tiles_enabled_flag** is equal to 1 and **entropy_coding_sync_enabled_flag** is equal to 0, each slice must include either a subset of the coding tree units of one tile (in which case the syntax element **entry_point_offset_minus1**[*i*] is not present) or must include all coding tree units of an integer number of complete tiles.

When **tiles_enabled_flag** is equal to 0 and **entropy_coding_sync_enabled_flag** is equal to 1, each subset *k* with *k* in the range of 0 to **num_entry_point_offsets**, inclusive, shall consist of all coded bits of all coding tree units in the slice segment that include luma coding tree blocks that are in the same luma coding tree block row of the picture, and the number of subsets (i.e. the value of **num_entry_point_offsets** + 1) shall be equal to the number of coding tree block rows of the picture that contain coding tree units that are in the coded slice segment.

NOTE 5 – The last subset (i.e. subset *k* for *k* equal to **num_entry_point_offsets**) may or may not contain all coding tree units that include luma coding tree blocks that are in a luma coding tree block row of the picture.

When **tiles_enabled_flag** is equal to 1 and **entropy_coding_sync_enabled_flag** is equal to 1, each subset *k* with *k* in the range of 0 to **num_entry_point_offsets**, inclusive, shall consist of all coded bits of all coding tree units in the slice segment that include luma coding tree blocks that are in the same luma coding tree block row of a tile, and the number of subsets (i.e. the value of **num_entry_point_offsets** + 1) shall be equal to the number of luma coding tree block rows of a tile that contain coding tree units that are in the coded slice segment.

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

slice_segment_header_extension_data_byte may have any value. Decoders shall ignore the value of **slice_segment_header_extension_data_byte**. Its value does not affect decoder conformance to profiles specified in this version of this Specification.

7.4.7.2 Reference picture list modification semantics

ref_pic_list_modification_flag_l0 equal to 1 indicates that reference picture list 0 is specified explicitly by a list of **list_entry_l0**[*i*] values. **ref_pic_list_modification_flag_l0** equal to 0 indicates that reference picture list 0 is determined implicitly. When **ref_pic_list_modification_flag_l0** is not present in the slice header, it is inferred to be equal to 0.

list_entry_l0[*i*] specifies the index of the reference picture in **RefPicListTemp0** to be placed at the current position of reference picture list 0. The length of the **list_entry_l0**[*i*] syntax element is $\text{Ceil}(\text{Log2}(\text{NumPocTotalCurr}))$ bits. The value of **list_entry_l0**[*i*] shall be in the range of 0 to **NumPocTotalCurr** – 1, inclusive. When the syntax element **list_entry_l0**[*i*] is not present in the slice header, it is inferred to be equal to 0.

The variable **NumPocTotalCurr** is derived as follows:

$$\begin{aligned} &\text{NumPocTotalCurr} = 0 \\ &\text{for}(i = 0; i < \text{NumNegativePics}[\text{CurrRpsIdx}]; i++) \\ &\quad \text{if}(\text{UsedByCurrPicS0}[\text{CurrRpsIdx}][i]) \\ &\quad \quad \text{NumPocTotalCurr}++ \\ &\text{for}(i = 0; i < \text{NumPositivePics}[\text{CurrRpsIdx}]; i++) \\ &\quad \text{if}(\text{UsedByCurrPicS1}[\text{CurrRpsIdx}][i]) \\ &\quad \quad \text{NumPocTotalCurr}++ \end{aligned} \quad (7-43)$$

```

for( i = 0; i < num_long_term_sps + num_long_term_pics; i++ )
    if( UsedByCurrPicLt[ i ] )
        NumPocTotalCurr++

```

ref_pic_list_modification_flag_l1 equal to 1 indicates that reference picture list 1 is specified explicitly by a list of **list_entry_l1[i]** values. **ref_pic_list_modification_flag_l1** equal to 0 indicates that reference picture list 1 is determined implicitly. When **ref_pic_list_modification_flag_l1** is not present in the slice header, it is inferred to be equal to 0.

list_entry_l1[i] specifies the index of the reference picture in **RefPicListTemp1** to be placed at the current position of reference picture list 1. The length of the **list_entry_l1[i]** syntax element is $\text{Ceil}(\log_2(\text{NumPocTotalCurr}))$ bits. The value of **list_entry_l1[i]** shall be in the range of 0 to **NumPocTotalCurr** – 1, inclusive. When the syntax element **list_entry_l1[i]** is not present in the slice header, it is inferred to be equal to 0.

7.4.7.3 Weighted prediction parameters semantics

luma_log2_weight_denom is the base 2 logarithm of the denominator for all luma weighting factors. The value of **luma_log2_weight_denom** shall be in the range of 0 to 7, inclusive.

delta_chroma_log2_weight_denom is the difference of the base 2 logarithm of the denominator for all chroma weighting factors.

The variable **ChromaLog2WeightDenom** is 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 **RefPicList0[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 **RefPicList0[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 **RefPicList0[i]**.

The variable **LumaWeightL0[i]** is derived to be equal to $(1 \ll \text{luma_log2_weight_denom}) + \text{delta_luma_weight_l0}[i]$. 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 $2^{\text{luma_log2_weight_denom}}$.

luma_offset_l0[i] is the additive offset applied to the luma prediction value for list 0 prediction using **RefPicList0[i]**. The value of **luma_offset_l0[i]** shall be in the range of –128 to 127, inclusive. When **luma_weight_l0_flag[i]** is equal to 0, **luma_offset_l0[i]** is inferred as 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 **RefPicList0[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 $(1 \ll \text{ChromaLog2WeightDenom}) + \text{delta_chroma_weight_l0}[i][j]$. 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 $2^{\text{ChromaLog2WeightDenom}}$.

delta_chroma_offset_l0[i][j] is the difference of the additive offset applied to the chroma prediction values for list 0 prediction using **RefPicList0[i]** with **j** equal to 0 for Cb and **j** equal to 1 for Cr.

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

$$\text{ChromaOffsetL0}[i][j] = \text{Clip3}(-128, 127, (\text{delta_chroma_offset_l0}[i][j] - ((128 * \text{ChromaWeightL0}[i][j]) \gg \text{ChromaLog2WeightDenom}) + 128)) \quad (7-44)$$

The value of **delta_chroma_offset_l0[i][j]** shall be in the range of –512 to 511, 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..num_ref_idx_l0_active_minus1.

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 $i = 0..num_ref_idx_l1_active_minus1$.

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.

7.4.8 Short-term reference picture set semantics

A `short_term_ref_pic_set( stRpsIdx )` 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 `short_term_ref_pic_set( stRpsIdx )` syntax structure specifies the short-term RPS of the current picture (the picture containing the slice), and the following applies:
 - The content of the `short_term_ref_pic_set( stRpsIdx )` syntax structure shall be the same in all slice headers of the current picture.
 - The value of `stRpsIdx` shall be equal to the syntax element `num_short_term_ref_pic_sets` in the active SPS.
 - The short-term RPS of the current picture is also referred to as the `num_short_term_ref_pic_sets`-th candidate short-term RPS in the semantics specified in the remainder of this subclause.
- Otherwise (present in an SPS), the `short_term_ref_pic_set( stRpsIdx )` syntax structure specifies a candidate short-term RPS, and the term "the current picture" in the semantics specified in the remainder of this subclause refers to each picture that has `short_term_ref_pic_set_idx` equal to `stRpsIdx` in a CVS that has the SPS as the active SPS.

`inter_ref_pic_set_prediction_flag` equal to 1 specifies that the `stRpsIdx`-th candidate short-term RPS is predicted from another candidate short-term RPS, which is referred to as the source candidate short-term RPS. When `inter_ref_pic_set_prediction_flag` is not present, it is inferred to be equal to 0.

`delta_idx_minus1` plus 1 specifies the difference between the value of `stRpsIdx` and the index, into the list of the candidate short-term RPSs specified in the SPS, of the source candidate short-term RPS. The value of `delta_idx_minus1` shall be in the range of 0 to `stRpsIdx - 1`, inclusive. When `delta_idx_minus1` is not present, it is inferred to be equal to 0.

The variable `RefRpsIdx` is derived as follows:

$$\text{RefRpsIdx} = \text{stRpsIdx} - (\text{delta_idx_minus1} + 1) \quad (7-45)$$

`delta_rps_sign` and `abs_delta_rps_minus1` together specify the value of the variable `deltaRps` as follows:

$$\text{deltaRps} = (1 - 2 * \text{delta_rps_sign}) * (\text{abs_delta_rps_minus1} + 1) \quad (7-46)$$

The variable `deltaRps` represents the value to be added to the picture order count difference values of the source candidate short-term RPS to obtain the picture order count difference values of the `stRpsIdx`-th candidate short-term RPS. The value of `abs_delta_rps_minus1` shall be in the range of 0 to $2^{15} - 1$, inclusive.

`used_by_curr_pic_flag[ j ]` equal to 0 specifies that the j -th entry in the source candidate short-term RPS is not used for reference by the current picture.

`use_delta_flag[ j ]` equal to 1 specifies that the j -th entry in the source candidate short-term RPS is included in the `stRpsIdx`-th candidate short-term RPS. `use_delta_flag[ j ]` equal to 0 specifies that the j -th entry in the source candidate short-term RPS is not included in the `stRpsIdx`-th candidate short-term RPS. When `use_delta_flag[ j ]` is not present, its value is inferred to be equal to 1.

When `inter_ref_pic_set_prediction_flag` is equal to 1, the variables `DeltaPocS0[ stRpsIdx ][ i ]`, `UsedByCurrPicS0[ stRpsIdx ][ i ]`, `NumNegativePics[ stRpsIdx ]`, `DeltaPocS1[ stRpsIdx ][ i ]`, `UsedByCurrPicS1[ stRpsIdx ][ i ]`, and `NumPositivePics[ stRpsIdx ]` are derived as follows:

```

i = 0
for( j = NumPositivePics[ RefRpsIdx ] - 1; j >= 0; j-- ) {
    dPoc = DeltaPocS1[ RefRpsIdx ][ j ] + deltaRps
    if( dPoc < 0 && use_delta_flag[ NumNegativePics[ RefRpsIdx ] + j ] ) {
        DeltaPocS0[ stRpsIdx ][ i ] = dPoc
        UsedByCurrPicS0[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ NumNegativePics[ RefRpsIdx ] + j ]
    }
}
if( deltaRps < 0 && use_delta_flag[ NumDeltaPocs[ RefRpsIdx ] ] ) {
    DeltaPocS0[ stRpsIdx ][ i ] = deltaRps
    UsedByCurrPicS0[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ NumDeltaPocs[ RefRpsIdx ] ]
}

```

(7-47)

```

    }
    for( j = 0; j < NumNegativePics[ RefRpsIdx ]; j++ ) {
        dPoc = DeltaPocS0[ RefRpsIdx ][ j ] + deltaRps
        if( dPoc < 0 && use_delta_flag[ j ] ) {
            DeltaPocS0[ stRpsIdx ][ i ] = dPoc
            UsedByCurrPicS0[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ j ]
        }
    }
    NumNegativePics[ stRpsIdx ] = i

    i = 0
    for( j = NumNegativePics[ RefRpsIdx ] - 1; j >= 0; j-- ) {
        dPoc = DeltaPocS0[ RefRpsIdx ][ j ] + deltaRps
        if( dPoc > 0 && use_delta_flag[ j ] ) {
            DeltaPocS1[ stRpsIdx ][ i ] = dPoc
            UsedByCurrPicS1[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ j ]
        }
    }
    if( deltaRps > 0 && use_delta_flag[ NumDeltaPocs[ RefRpsIdx ] ] ) {
        DeltaPocS1[ stRpsIdx ][ i ] = deltaRps
        UsedByCurrPicS1[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ NumDeltaPocs[ RefRpsIdx ] ]
    }
    for( j = 0; j < NumPositivePics[ RefRpsIdx ]; j++ ) {
        dPoc = DeltaPocS1[ RefRpsIdx ][ j ] + deltaRps
        if( dPoc > 0 && use_delta_flag[ NumNegativePics[ RefRpsIdx ] + j ] ) {
            DeltaPocS1[ stRpsIdx ][ i ] = dPoc
            UsedByCurrPicS1[ stRpsIdx ][ i++ ] = used_by_curr_pic_flag[ NumNegativePics[ RefRpsIdx ] + j ]
        }
    }
    NumPositivePics[ stRpsIdx ] = i

```

(7-48)

num_negative_pics specifies the number of entries in the stRpsIdx-th candidate short-term RPS that have picture order count values less than the picture order count value of the current picture. The value of num_negative_pics shall be in the range of 0 to sps_max_dec_pic_buffering_minus1[sps_max_sub_layers_minus1], inclusive.

num_positive_pics specifies the number of entries in the stRpsIdx-th candidate short-term RPS that have picture order count values greater than the picture order count value of the current picture. The value of num_positive_pics shall be in the range of 0 to sps_max_dec_pic_buffering_minus1[sps_max_sub_layers_minus1] - num_negative_pics, inclusive.

delta_poc_s0_minus1[i] plus 1, when i is equal to 0, specifies the difference between the picture order count values of the current picture and i-th entry in the stRpsIdx-th candidate short-term RPS that has picture order count value less than that of the current picture, or, when i is greater than 0, specifies the difference between the picture order count values of the i-th entry and the (i + 1)-th entry in the stRpsIdx-th candidate short-term RPS that have picture order count values less than the picture order count value of the current picture. The value of delta_poc_s0_minus1[i] shall be in the range of 0 to $2^{15} - 1$, inclusive.

used_by_curr_pic_s0_flag[i] equal to 0 specifies that the i-th entry in the stRpsIdx-th candidate short-term RPS that has picture order count value less than that of the current picture is not used for reference by the current picture.

delta_poc_s1_minus1[i] plus 1, when i is equal to 0, specifies the difference between the picture order count values of the current picture and the i-th entry in the stRpsIdx-th candidate short-term RPS that has picture order count value greater than that of the current picture, or, when i is greater than 0, specifies the difference between the picture order count values of the (i + 1)-th entry and i-th entry in the current candidate short-term RPS that have picture order count values greater than the picture order count value of the current picture. The value of delta_poc_s1_minus1[i] shall be in the range of 0 to $2^{15} - 1$, inclusive.

used_by_curr_pic_s1_flag[i] equal to 0 specifies that the i-th entry in the current candidate short-term RPS that has picture order count value greater than that of the current picture is not used for reference by the current picture.

When inter_ref_pic_set_prediction_flag is equal to 0, the variables NumNegativePics[stRpsIdx], NumPositivePics[stRpsIdx], UsedByCurrPicS0[stRpsIdx][i], UsedByCurrPicS1[stRpsIdx][i], DeltaPocS0[stRpsIdx][i], and DeltaPocS1[stRpsIdx][i] are derived as follows:

$$\text{NumNegativePics[stRpsIdx]} = \text{num_negative_pics} \quad (7-49)$$

$$\text{NumPositivePics[stRpsIdx]} = \text{num_positive_pics} \quad (7-50)$$

$$\text{UsedByCurrPicS0}[\text{stRpsIdx}][i] = \text{used_by_curr_pic_s0_flag}[i] \quad (7-51)$$

$$\text{UsedByCurrPicS1}[\text{stRpsIdx}][i] = \text{used_by_curr_pic_s1_flag}[i] \quad (7-52)$$

- If i is equal to 0, the following applies:

$$\text{DeltaPocS0}[\text{stRpsIdx}][i] = -(\text{delta_poc_s0_minus1}[i] + 1) \quad (7-53)$$

$$\text{DeltaPocS1}[\text{stRpsIdx}][i] = \text{delta_poc_s1_minus1}[i] + 1 \quad (7-54)$$

- Otherwise, the following applies:

$$\text{DeltaPocS0}[\text{stRpsIdx}][i] = \text{DeltaPocS0}[\text{stRpsIdx}][i - 1] - (\text{delta_poc_s0_minus1}[i] + 1) \quad (7-55)$$

$$\text{DeltaPocS1}[\text{stRpsIdx}][i] = \text{DeltaPocS1}[\text{stRpsIdx}][i - 1] + (\text{delta_poc_s1_minus1}[i] + 1) \quad (7-56)$$

The variable $\text{NumDeltaPocs}[\text{stRpsIdx}]$ is derived as follows:

$$\text{NumDeltaPocs}[\text{stRpsIdx}] = \text{NumNegativePics}[\text{stRpsIdx}] + \text{NumPositivePics}[\text{stRpsIdx}] \quad (7-57)$$

7.4.9 Slice segment data semantics

7.4.9.1 General slice segment data semantics

end_of_slice_segment_flag equal to 0 specifies that another coding tree unit is following in the slice. **end_of_slice_segment_flag** equal to 1 specifies the end of the slice segment, i.e. that no further coding tree unit follows in the slice segment.

end_of_sub_stream_one_bit shall be equal to 1.

7.4.9.2 Coding tree unit semantics

The coding tree unit is the root node of the coding quadtree structure.

7.4.9.3 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 coding tree block. **sao_merge_left_flag** equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the left coding tree block. When **sao_merge_left_flag** is not present, it is inferred to be equal to 0.

sao_merge_up_flag equal to 1 specifies that the syntax elements **sao_type_idx_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 coding tree block. **sao_merge_up_flag** equal to 0 specifies that these syntax elements are not derived from the corresponding syntax elements of the above coding tree block. When **sao_merge_up_flag** is not present, it is inferred to be equal to 0.

sao_type_idx_luma specifies the offset type for the luma component. The array $\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ specifies the offset type as specified in Table 7-8 for the coding tree block at the location (rx, ry) for the colour component cIdx . The value of $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is derived as follows:

- If **sao_type_idx_luma** is present, $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is set equal to **sao_type_idx_luma**.
- Otherwise (**sao_type_idx_luma** is not present), $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is derived as follows:
 - If **sao_merge_left_flag** is equal to 1, $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is set equal to $\text{SaoTypeIdx}[0][\text{rx} - 1][\text{ry}]$.
 - Otherwise, if **sao_merge_up_flag** is equal to 1, $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is set equal to $\text{SaoTypeIdx}[0][\text{rx}][\text{ry} - 1]$.
 - Otherwise, $\text{SaoTypeIdx}[0][\text{rx}][\text{ry}]$ is set equal to 0.

sao_type_idx_chroma specifies the offset type for the chroma components. The values of $\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ are derived as follows for cIdx equal to 1..2:

- If **sao_type_idx_chroma** is present, $\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ is set equal to **sao_type_idx_chroma**.
- Otherwise (**sao_type_idx_chroma** is not present), $\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ is derived as follows:
 - If **sao_merge_left_flag** is equal to 1, $\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ is set equal to $\text{SaoTypeIdx}[\text{cIdx}][\text{rx} - 1][\text{ry}]$.

- 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 7-8 – Specification of the SAO type

<code>SaoTypeIdx[cIdx][rx][ry]</code>	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 coding tree block 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 coding tree block at the location (*rx*, *ry*) for the colour component *cIdx* when `SaoTypeIdx[cIdx][rx][ry]` is equal to 1.

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, `sao_offset_sign[cIdx][rx][ry][i]` is inferred to be equal 0.

The variable `offsetSign` is derived as follows:

- If `SaoTypeIdx[cIdx][rx][ry]` is equal to 2 and *i* is equal to 2 or 3, `offsetSign` is set equal to -1.
- Otherwise, if `SaoTypeIdx[cIdx][rx][ry]` is equal to 2 and *i* is equal to 0 or 1, `offsetSign` is set equal to 1.
- Otherwise (`SaoTypeIdx[cIdx][rx][ry]` is equal to 1), the following applies:
 - If `sao_offset_sign[cIdx][rx][ry][i]` is equal to 0, `offsetSign` is set equal to equal to 1.
 - Otherwise, `offsetSign` is set equal to equal to -1.

The variable `bitDepth` is derived as follows:

- If *cIdx* is equal to 0, `bitDepth` is set equal to `BitDepthY`.
- Otherwise (*cIdx* is equal to 1 or 2), `bitDepth` is set equal to `BitDepthC`.

The list `SaoOffsetVal[cIdx][rx][ry][i]` for *i* ranging from 0 to 4, inclusive, is derived as follows:

$$\begin{aligned}
 &\text{SaoOffsetVal}[cIdx][rx][ry][0] = 0 \\
 &\text{for}(i = 0; i < 4; i++) \\
 &\quad \text{SaoOffsetVal}[cIdx][rx][ry][i + 1] = \\
 &\quad \quad \text{offsetSign} * \text{sao_offset_abs}[cIdx][rx][ry][i] << (\text{bitDepth} - \text{Min}(\text{bitDepth}, 10))
 \end{aligned} \tag{7-58}$$

`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 7-9 for the coding tree block 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 7-9 – 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

7.4.9.4 Coding quadtree semantics

split_cu_flag[x0][y0] specifies whether a coding unit is split into coding units with half horizontal and vertical size. The array indices `x0`, `y0` specify the location (`x0`, `y0`) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When `split_cu_flag[ x0 ][ y0 ]` is not present, the following applies:

- If `log2CbSize` is greater than `MinCbLog2SizeY`, the value of `split_cu_flag[ x0 ][ y0 ]` is inferred to be equal to 1.
- Otherwise (`log2CbSize` is equal to `MinCbLog2SizeY`), the value of `split_cu_flag[ x0 ][ y0 ]` is inferred to be equal to 0.

The array `CtDepth[ x ][ y ]` specifies the coding tree depth for a luma coding block covering the location (`x`, `y`). When `split_cu_flag[ x0 ][ y0 ]` is equal to 0, `CtDepth[ x ][ y ]` is inferred to be equal to `cqtDepth` for `x = x0..x0 + nCbS - 1` and `y = y0..y0 + nCbS - 1`.

7.4.9.5 Coding unit semantics

cu_transquant_bypass_flag equal to 1 specifies that the scaling and transform process as specified in subclause 8.6 and the in-loop filter process as specified in subclause 8.7 are bypassed. When `cu_transquant_bypass_flag` is not present, it is inferred to be equal to 0.

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 the merging candidate index `merge_idx[ x0 ][ y0 ]` are parsed after `cu_skip_flag[ x0 ][ y0 ]`.

$\text{cu_skip_flag}[x_0][y_0]$ equal to 0 specifies that the coding unit is not skipped. The array indices x_0, y_0 specify the location (x_0, y_0) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

When $\text{cu_skip_flag}[x_0][y_0]$ is not present, it is inferred to be equal to 0.

pred_mode_flag equal to 0 specifies that the current coding unit is coded in inter prediction mode. **pred_mode_flag** equal to 1 specifies that the current coding unit is coded in intra prediction mode. The variable $\text{CuPredMode}[x][y]$ is derived as follows for $x = x_0..x_0 + \text{nCbS} - 1$ and $y = y_0..y_0 + \text{nCbS} - 1$:

- If **pred_mode_flag** is equal to 0, $\text{CuPredMode}[x][y]$ is set equal to **MODE_INTER**.
- Otherwise (**pred_mode_flag** is equal to 1), $\text{CuPredMode}[x][y]$ is set equal to **MODE_INTRA**.

When **pred_mode_flag** is not present, the variable $\text{CuPredMode}[x][y]$ is derived as follows for $x = x_0..x_0 + \text{nCbS} - 1$ and $y = y_0..y_0 + \text{nCbS} - 1$:

- If **slice_type** is equal to I, $\text{CuPredMode}[x][y]$ is inferred to be equal to **MODE_INTRA**.
- Otherwise (**slice_type** is equal to P or B), when $\text{cu_skip_flag}[x_0][y_0]$ is equal to 1, $\text{CuPredMode}[x][y]$ is inferred to be equal to **MODE_SKIP**.

part_mode specifies partitioning mode of the current coding unit. The semantics of **part_mode** depend on $\text{CuPredMode}[x_0][y_0]$. The variables **PartMode** and **IntraSplitFlag** are derived from the value of **part_mode** as defined in Table 7-10.

The value of **part_mode** is restricted as follows:

- If $\text{CuPredMode}[x_0][y_0]$ is equal to **MODE_INTRA**, **part_mode** shall be equal to 0 or 1.
- Otherwise ($\text{CuPredMode}[x_0][y_0]$ is equal to **MODE_INTER**), the following applies:
 - If $\log_2\text{CbSize}$ is greater than $\text{MinCbLog}_2\text{SizeY}$ and **amp_enabled_flag** is equal to 1, **part_mode** shall be in the range of 0 to 2, inclusive, or in the range of 4 to 7, inclusive.
 - Otherwise, if $\log_2\text{CbSize}$ is greater than $\text{MinCbLog}_2\text{SizeY}$ and **amp_enabled_flag** is equal to 0, or $\log_2\text{CbSize}$ is equal to 3, **part_mode** shall be in the range of 0 to 2, inclusive.
 - Otherwise ($\log_2\text{CbSize}$ is greater than 3 and less than or equal to $\text{MinCbLog}_2\text{SizeY}$), the value of **part_mode** shall be in the range of 0 to 3, inclusive.

When **part_mode** is not present, the variables **PartMode** and **IntraSplitFlag** are derived as follows:

- **PartMode** is set equal to **PART_2Nx2N**.
- **IntraSplitFlag** is set equal to 0.

pcm_flag[x_0][y_0] 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 (x_0, y_0). **pcm_flag**[x_0][y_0] equal to 0 specifies that **pcm_sample()** syntax structure is not present. When **pcm_flag**[x_0][y_0] is not present, it is inferred to be equal to 0.

The value of $\text{pcm_flag}[x_0 + i][y_0 + j]$ with $i = 1.. \text{nCbS} - 1$, $j = 1.. \text{nCbS} - 1$ is inferred to be equal to $\text{pcm_flag}[x_0][y_0]$.

pcm_alignment_zero_bit is a bit equal to 0.

Table 7-10 – Name association to prediction mode and partitioning type

CuPredMode[x0][y0]	part_mode	IntraSplitFlag	PartMode
MODE_INTRA	0	0	PART_2Nx2N
	1	1	PART_NxN
MODE_INTER	0	0	PART_2Nx2N
	1	0	PART_2NxN
	2	0	PART_Nx2N
	3	0	PART_NxN
	4	0	PART_2NxN
	5	0	PART_2NxN
	6	0	PART_nLx2N
	7	0	PART_nRx2N

The syntax elements **prev_intra_luma_pred_flag**[x0 + i][y0 + j], **mpm_idx**[x0 + i][y0 + j] and **rem_intra_luma_pred_mode**[x0 + i][y0 + j] specify the intra prediction mode for luma samples. The array indices x0 + i, y0 + j specify the location (x0 + i, y0 + j) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture. When prev_intra_luma_pred_flag[x0 + i][y0 + j] is equal to 1, the intra prediction mode is inferred from a neighbouring intra-predicted prediction unit according to subclause 8.4.2.

intra_chroma_pred_mode[x0][y0] specifies the intra prediction mode for chroma samples. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

rqt_root_cbf equal to 1 specifies that the transform_tree() syntax structure is present for the current coding unit. rqt_root_cbf equal to 0 specifies that the transform_tree() syntax structure is not present for the current coding unit.

When rqt_root_cbf is not present, its value is inferred to be equal to 1.

7.4.9.6 Prediction unit semantics

mvp_l0_flag[x0][y0] specifies the motion vector predictor index of list 0 where x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When mvp_l0_flag[x0][y0] is not present, it is inferred to be equal to 0.

mvp_l1_flag[x0][y0] has the same semantics as mvp_l0_flag, with l0 and list 0 replaced by l1 and list 1, respectively.

merge_flag[x0][y0] specifies whether the inter prediction parameters for the current prediction unit are inferred from a neighbouring inter-predicted partition. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When merge_flag[x0][y0] is not present, it is inferred as follows:

- If CuPredMode[x0][y0] is equal to MODE_SKIP, merge_flag[x0][y0] is inferred to be equal to 1.
- Otherwise, merge_flag[x0][y0] 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 prediction block relative to the top-left luma sample of the picture.

When merge_idx[x0][y0] is not present, it is inferred to be equal to 0.

inter_pred_idc[x0][y0] specifies whether list0, list1, or bi-prediction is used for the current prediction unit according to Table 7-11. The array indices x0, y0 specify the location (x0, y0) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

Table 7-11 – Name association to inter prediction mode

inter_pred_idc	Name of inter_pred_idc	
	(nPbW + nPbH) != 12	(nPbW + nPbH) == 12
0	PRED_L0	PRED_L0
1	PRED_L1	PRED_L1
2	PRED_BI	na

When `inter_pred_idc[ x0 ][ y0 ]` is not present, it is inferred to be equal to `PRED_L0`.

`ref_idx_l0[ x0 ][ y0 ]` specifies the list 0 reference picture index for the current prediction unit. The array indices `x0`, `y0` specify the location (`x0`, `y0`) of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture.

When `ref_idx_l0[ x0 ][ y0 ]` is not present it is inferred to be equal to 0.

`ref_idx_l1[ x0 ][ y0 ]` has the same semantics as `ref_idx_l0`, with l0 and list 0 replaced by l1 and list 1, respectively.

7.4.9.7 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`.

7.4.9.8 Transform tree semantics

`split_transform_flag[ x0 ][ y0 ][ trafoDepth ]` specifies whether a block is split into four blocks with half horizontal and half vertical size for the purpose of transform coding. The array indices `x0`, `y0` specify the location (`x0`, `y0`) of the top-left luma sample of the considered block relative to the top-left luma sample of the picture. The array index `trafoDepth` specifies the current subdivision level of a coding block into blocks for the purpose of transform coding. `trafoDepth` is equal to 0 for blocks that correspond to coding blocks.

The variable `interSplitFlag` is derived as follows:

- If `max_transform_hierarchy_depth_inter` is equal to 0 and `CuPredMode[ x0 ][ y0 ]` is equal to `MODE_INTER` and `PartMode` is not equal to `PART_2Nx2N` and `trafoDepth` is equal to 0, `interSplitFlag` is set equal to 1.
- Otherwise, `interSplitFlag` is set equal to 0.

When `split_transform_flag[ x0 ][ y0 ][ trafoDepth ]` is not present, it is inferred as follows:

- If one or more of the following conditions are true, the value of `split_transform_flag[ x0 ][ y0 ][ trafoDepth ]` is inferred to be equal to 1:
 - `log2TrafoSize` is greater than `Log2MaxTrafoSize`
 - `IntraSplitFlag` is equal to 1 and `trafoDepth` is equal to 0
 - `interSplitFlag` is equal to 1
- Otherwise, the value of `split_transform_flag[ x0 ][ y0 ][ trafoDepth ]` is inferred to be equal to 0.

`cbf_luma[ x0 ][ y0 ][ trafoDepth ]` equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices `x0`, `y0` specify the location (`x0`, `y0`) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index `trafoDepth` specifies the current subdivision level of a coding block into blocks for the purpose of transform coding. `trafoDepth` is equal to 0 for blocks that correspond to coding blocks.

When `cbf_luma[ x0 ][ y0 ][ trafoDepth ]` is not present, it is inferred to be equal to 1.

`cbf_cb[ x0 ][ y0 ][ trafoDepth ]` equal to 1 specifies that the Cb transform block contains one or more transform coefficient levels not equal to 0. The array indices `x0`, `y0` specify the top-left location (`x0`, `y0`) of the considered transform unit. The array index `trafoDepth` specifies the current subdivision level of a coding block into blocks for the purpose of transform coding. `trafoDepth` is equal to 0 for blocks that correspond to coding blocks.

When $\text{cbf_cb}[x0][y0][\text{trafoDepth}]$ is not present, the value of $\text{cbf_cb}[x0][y0][\text{trafoDepth}]$ is inferred as follows:

- If trafoDepth is greater than 0 and $\log_2\text{TrafoSize}$ is equal to 2, $\text{cbf_cb}[x0][y0][\text{trafoDepth}]$ is inferred to be equal to $\text{cbf_cb}[xBase][yBase][\text{trafoDepth} - 1]$
- Otherwise, $\text{cbf_cb}[x0][y0][\text{trafoDepth}]$ is inferred to be equal to 0.

$\text{cbf_cr}[x0][y0][\text{trafoDepth}]$ equal to 1 specifies that the Cr transform block contains one or more transform coefficient levels not equal to 0. The array indices $x0, y0$ specify the top-left location $(x0, y0)$ of the considered transform unit. The array index trafoDepth specifies the current subdivision level of a coding block into blocks for the purpose of transform coding. trafoDepth is equal to 0 for blocks that correspond to coding blocks.

When $\text{cbf_cr}[x0][y0][\text{trafoDepth}]$ is not present, the value of $\text{cbf_cr}[x0][y0][\text{trafoDepth}]$ is inferred as follows:

- If trafoDepth is greater than 0 and $\log_2\text{TrafoSize}$ is equal to 2, $\text{cbf_cr}[x0][y0][\text{trafoDepth}]$ is inferred to be equal to $\text{cbf_cr}[xBase][yBase][\text{trafoDepth} - 1]$
- Otherwise, $\text{cbf_cr}[x0][y0][\text{trafoDepth}]$ is inferred to be equal to 0.

7.4.9.9 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 $\text{IMvd}[\text{compIdx}]$ for $\text{compIdx} = 0..1$ is derived as follows:

$$\text{IMvd}[\text{compIdx}] = \text{abs_mvd_greater0_flag}[\text{compIdx}] * (\text{abs_mvd_minus2}[\text{compIdx}] + 2) * (1 - 2 * \text{mvd_sign_flag}[\text{compIdx}]) \quad (7-59)$$

The variable $\text{MvdLX}[x0][y0][\text{compIdx}]$, with X being 0 or 1, specifies the difference between a list X vector component to be used and its prediction. The value of $\text{MvdLX}[x0][y0][\text{compIdx}]$ shall be in the range of -2^{15} to $2^{15} - 1$, inclusive. The array indices $x0, y0$ specify the location $(x0, y0)$ of the top-left luma sample of the considered prediction block relative to the top-left luma sample of the picture. The horizontal motion vector component difference is assigned $\text{compIdx} = 0$ and the vertical motion vector component is assigned $\text{compIdx} = 1$.

- If refList is equal to 0, $\text{MvdL0}[x0][y0][\text{compIdx}]$ is set equal to $\text{IMvd}[\text{compIdx}]$ for $\text{compIdx} = 0..1$.
- Otherwise (refList is equal to 1), $\text{MvdL1}[x0][y0][\text{compIdx}]$ is set equal to $\text{IMvd}[\text{compIdx}]$ for $\text{compIdx} = 0..1$.

7.4.9.10 Transform unit semantics

The transform coefficient levels are represented by the arrays $\text{TransCoeffLevel}[x0][y0][cIdx][xC][yC]$, which are either specified in subclause 7.3.8.11 or inferred as follows. 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 $\text{TransCoeffLevel}[x0][y0][cIdx][xC][yC]$ is not specified in subclause 7.3.8.11, it is inferred to be equal to 0.

cu_qp_delta_abs specifies the absolute value of the difference CuQpDeltaVal between the luma 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 ($\text{cu_qp_delta_sign_flag}$ is equal to 1), the corresponding CuQpDeltaVal has a negative value.

When $\text{cu_qp_delta_sign_flag}$ 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:

$$\text{IsCuQpDeltaCoded} = 1 \quad (7-60)$$

$$\text{CuQpDeltaVal} = \text{cu_qp_delta_abs} * (1 - 2 * \text{cu_qp_delta_sign_flag}) \quad (7-61)$$

The value of CuQpDeltaVal shall be in the range of $-(26 + \text{QpBdOffset}_Y / 2)$ to $+(25 + \text{QpBdOffset}_Y / 2)$, inclusive.

7.4.9.11 Residual coding semantics

For intra prediction, different scanning orders are used. The variable scanIdx specifies which scan order is used where scanIdx equal to 0 specifies an up-right diagonal scan order, scanIdx equal to 1 specifies a horizontal scan order, and scanIdx equal to 2 specifies a vertical scan order. The value of scanIdx is derived as follows:

- If $\text{CuPredMode}[x0][y0]$ is equal to MODE_INTRA and one or more of the following conditions are true:

- $\log_2\text{TrafoSize}$ is equal to 2.
- $\log_2\text{TrafoSize}$ is equal to 3 and cIdx is equal to 0.

predModeIntra is derived as follows:

- If cIdx is equal to 0, predModeIntra is set equal to $\text{IntraPredModeY}[x0][y0]$.
- Otherwise, predModeIntra is set equal to IntraPredModeC .

scanIdx is derived as follows:

- If predModeIntra is in the range of 6 to 14, inclusive, scanIdx is set equal to 2.
- Otherwise if predModeIntra is in the range of 22 to 30, inclusive, scanIdx is set equal to 1.
- Otherwise, scanIdx is set equal to 0.

- Otherwise, scanIdx is set equal to 0.

transform_skip_flag $[x0][y0][cIdx]$ specifies whether a transform is applied to the associated transform block or not. The array indices $x0, y0$ specify the location $(x0, y0)$ of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture. The array index cIdx specifies an indicator for the colour component; it is equal to 0 for luma, equal to 1 for Cb, and equal to 2 for Cr. $\text{transform_skip_flag}[x0][y0][cIdx]$ equal to 1 specifies that no transform is applied to the current transform block. $\text{transform_skip_flag}[x0][y0][cIdx]$ equal to 0 specifies that the decision whether transform is applied to the current transform block or not depends on other syntax elements. When $\text{transform_skip_flag}[x0][y0][cIdx]$ is not present, it is inferred to be equal to 0.

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 $\text{last_sig_coeff_x_prefix}$ shall be in the range of 0 to $(\log_2\text{TrafoSize} \ll 1) - 1$, inclusive.

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 $\text{last_sig_coeff_y_prefix}$ shall be in the range of 0 to $(\log_2\text{TrafoSize} \ll 1) - 1$, inclusive.

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 $\text{last_sig_coeff_x_suffix}$ shall be in the range of 0 to $(1 \ll ((\text{last_sig_coeff_x_prefix} \gg 1) - 1)) - 1$, inclusive.

The column position of the last significant coefficient in scanning order within a transform block $\text{LastSignificantCoeffX}$ is derived as follows:

- If $\text{last_sig_coeff_x_suffix}$ is not present, the following applies:

$$\text{LastSignificantCoeffX} = \text{last_sig_coeff_x_prefix} \quad (7-62)$$

- Otherwise ($\text{last_sig_coeff_x_suffix}$ is present), the following applies:

$$\text{LastSignificantCoeffX} = (1 \ll ((\text{last_sig_coeff_x_prefix} \gg 1) - 1)) * (2 + (\text{last_sig_coeff_x_prefix} \& 1)) + \text{last_sig_coeff_x_suffix} \quad (7-63)$$

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 $(1 \ll ((\text{last_sig_coeff_y_prefix} \gg 1) - 1)) - 1$, 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:

$$\text{LastSignificantCoeffY} = \text{last_sig_coeff_y_prefix} \quad (7-64)$$

- Otherwise (**last_sig_coeff_y_suffix** is present), the following applies:

$$\text{LastSignificantCoeffY} = (1 \ll ((\text{last_sig_coeff_y_prefix} \gg 1) - 1)) * (2 + (\text{last_sig_coeff_y_prefix} \& 1)) + \text{last_sig_coeff_y_suffix} \quad (7-65)$$

When **scanIdx** is equal to 2, the coordinates are swapped as follows:

$$(\text{LastSignificantCoeffX}, \text{LastSignificantCoeffY}) = \text{Swap}(\text{LastSignificantCoeffX}, \text{LastSignificantCoeffY}) \quad (7-66)$$

coded_sub_block_flag[xS][yS] specifies the following for the sub-block at location (xS, yS) within the current transform block, where a sub-block 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 sub-block 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 sub-block at location (xS, yS).
 - Otherwise, at least one of the 16 transform coefficient levels of the sub-block at location (xS, yS) has a non zero value.

When **coded_sub_block_flag[xS][yS]** is not present, it is inferred as follows:

- If one or more of the following conditions are true, **coded_sub_block_flag[xS][yS]** is inferred to be equal to 1:
 - (xS, yS) is equal to (0, 0)
 - (xS, yS) is equal to (LastSignificantCoeffX >> 2, LastSignificantCoeffY >> 2)
- Otherwise, **coded_sub_block_flag[xS][yS]** is inferred to be equal to 0.

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 & 3, yC & 3) 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.

coeff_abs_level_greater1_flag[n] specifies for the scanning position n whether there are transform coefficient levels greater than 1.

When **coeff_abs_level_greater1_flag[n]** is not present, it is inferred to be equal to 0.

coeff_abs_level_greater2_flag[n] specifies for the scanning position n whether there are transform coefficient levels greater than 2.

When **coeff_abs_level_greater2_flag[n]** is not present, it is inferred to be equal to 0.

coeff_sign_flag[n] specifies the sign of a transform coefficient level for the scanning position *n* as follows:

- If **coeff_sign_flag[n]** is equal to 0, the corresponding transform coefficient level has a positive value.
- Otherwise (**coeff_sign_flag[n]** is equal to 1), the corresponding transform coefficient level has a negative value.

When **coeff_sign_flag[n]** is not present, it is inferred to be equal to 0.

coeff_abs_level_remaining[n] is the remaining absolute value of a transform coefficient level that is coded with Golomb-Rice code at the scanning position *n*. When **coeff_abs_level_remaining[n]** is not present, it is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of **coeff_abs_level_remaining[n]** shall be constrained such that the corresponding value of **TransCoeffLevel[x0][y0][cIdx][xC][yC]** is in the range of -32768 to 32767 , inclusive.

8 Decoding process

8.1 General decoding process

Input to this process is a bitstream. Output of this process is a list of decoded pictures.

The layer identifier list **TargetDecLayerIdList**, which specifies the list of **nuh_layer_id** values, in increasing order of **nuh_layer_id** values, of the NAL units to be decoded, is specified as follows:

- If some external means, not specified in this Specification, is available to set **TargetDecLayerIdList**, **TargetDecLayerIdList** is set by the external means.
- Otherwise, if the decoding process is invoked in a bitstream conformance test as specified in subclause C.1, **TargetDecLayerIdList** is set as specified in subclause C.1.
- Otherwise, **TargetDecLayerIdList** contains only one **nuh_layer_id** value that is equal to 0.

The variable **HighestTid**, which identifies the highest temporal sub-layer to be decoded, is specified as follows:

- If some external means, not specified in this Specification, is available to set **HighestTid**, **HighestTid** is set by the external means.
- Otherwise, if the decoding process is invoked in a bitstream conformance test as specified in subclause C.1, **HighestTid** is set as specified in subclause C.1.
- Otherwise, **HighestTid** is set equal to **sps_max_sub_layers_minus1**.

The sub-bitstream extraction process as specified in clause 10 is applied with the bitstream, **HighestTid**, and **TargetDecLayerIdList** as inputs, and the output is assigned to a bitstream referred to as **BitstreamToDecode**.

The decoding processes specified in the remainder of this subclause 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 S_L .
- Otherwise (**chroma_format_idc** is not equal to 0), the current picture consists of 3 sample arrays S_L , S_{Cb} , S_{Cr} .

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, the term "the bitstream" (or part thereof, e.g. a CVS of the bitstream) refers to **BitstreamToDecode** (or part thereof).

The decoding process is specified such that all decoders will produce numerically identical cropped decoded pictures. Any decoding process that produces identical cropped decoded 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.

When the current picture is a BLA picture that has **nal_unit_type** equal to **BLA_W_LP** or is a CRA picture, the following applies:

- If some external means not specified in this Specification is available to set the variable **UseAltCpbParamsFlag** to a value, **UseAltCpbParamsFlag** is set equal to the value provided by the external means.
- Otherwise, the value of **UseAltCpbParamsFlag** is set equal to 0.

When the current picture is an IRAP picture, the following applies:

- If the current picture is an IDR picture, a BLA 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 NoRaslOutputFlag is set equal to 1.
- Otherwise, if some external means not specified in this Specification is available to set the variable HandleCraAsBlaFlag to a value for the current picture, the variable HandleCraAsBlaFlag is set equal to the value provided by the external means and the variable NoRaslOutputFlag is set equal to HandleCraAsBlaFlag.
- Otherwise, the variable HandleCraAsBlaFlag is set equal to 0 and the variable NoRaslOutputFlag is set equal to 0.

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 S_L , S_{Cb} , and S_{Cr} , 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 subclause 8.2.
2. The processes in subclause 8.3 specify the following decoding processes using syntax elements in the slice segment layer and above:
 - Variables and functions relating to picture order count are derived in subclause 8.3.1. This needs to be invoked only for the first slice segment of a picture.
 - The decoding process for RPS in subclause 8.3.2 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 segment of a picture.
 - When the current picture is a BLA picture or is a CRA picture with NoRaslOutputFlag equal to 1, the decoding process for generating unavailable reference pictures specified in subclause 8.3.3 is invoked, which needs to be invoked only for the first slice segment of a picture.
 - PicOutputFlag is set as follows:
 - If the current picture is a RASL picture and NoRaslOutputFlag of the associated IRAP picture is equal to 1, PicOutputFlag is set equal to 0.
 - Otherwise, PicOutputFlag is set equal to pic_output_flag.
 - At the beginning of the decoding process for each P or B slice, the decoding process for reference picture lists construction specified in subclause 8.3.4 is invoked for derivation of reference picture list 0 (RefPicList0) and, when decoding a B slice, reference picture list 1 (RefPicList1).
3. The processes in subclauses 8.4, 8.5, 8.6, and 8.7 specify decoding processes using syntax elements in all syntax structure layers. It is a requirement of bitstream conformance that the coded slices of the picture shall contain slice segment data for every coding tree unit of the picture, such that the division of the picture into slices, the division of the slices into slice segments, and the division of the slice segments into coding tree units each forms a partitioning of the picture.
4. After all slices of the current picture have been decoded, the decoded picture is marked as "used for short-term reference".

8.2 NAL unit decoding process

Inputs to this process are NAL units of the access unit containing the current picture.

Outputs of this process are the parsed RBSP syntax structures encapsulated within the NAL units of the access unit containing the current picture.

The decoding process for each NAL unit extracts the RBSP syntax structure from the NAL unit and then parses the RBSP syntax structure.

8.3 Slice decoding process

8.3.1 Decoding process for picture order count

Output of this process is `PicOrderCntVal`, the picture order count of the current picture.

Picture order counts are used to identify pictures, for deriving motion parameters in merge mode and motion vector prediction, and for decoder conformance checking (see subclause C.5).

Each coded picture is associated with a picture order count variable, denoted as `PicOrderCntVal`.

When the current picture is not an IRAP picture with `NoRaslOutputFlag` equal to 1, the variables `prevPicOrderCntLsb` and `prevPicOrderCntMsb` are derived as follows:

- Let `prevTid0Pic` be the previous picture in decoding order that has `TemporalId` equal to 0 and that is not a RASL picture, a RADL picture, or a sub-layer non-reference 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 an IRAP picture with `NoRaslOutputFlag` equal to 1, `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
else if( ( slice_pic_order_cnt_lsb > prevPicOrderCntLsb ) &&
    ( ( slice_pic_order_cnt_lsb - prevPicOrderCntLsb ) > ( MaxPicOrderCntLsb / 2 ) ) )
    PicOrderCntMsb = prevPicOrderCntMsb - MaxPicOrderCntLsb
else
    PicOrderCntMsb = prevPicOrderCntMsb

```

(8-1)

`PicOrderCntVal` is derived as follows:

$$\text{PicOrderCntVal} = \text{PicOrderCntMsb} + \text{slice_pic_order_cnt_lsb} \quad (8-2)$$

NOTE 1 – All IDR pictures will have `PicOrderCntVal` equal to 0 since `slice_pic_order_cnt_lsb` is inferred to be 0 for IDR pictures and `prevPicOrderCntLsb` and `prevPicOrderCntMsb` are both set equal to 0.

The value of `PicOrderCntVal` shall be in the range of -2^{31} to $2^{31} - 1$, inclusive. In one CVS, the `PicOrderCntVal` values for any two coded pictures shall not be the same.

The function `PicOrderCnt( picX )` is specified as follows:

$$\text{PicOrderCnt}(\text{picX}) = \text{PicOrderCntVal of the picture picX} \quad (8-3)$$

The function `DiffPicOrderCnt( picA, picB )` is specified as follows:

$$\text{DiffPicOrderCnt}(\text{picA}, \text{picB}) = \text{PicOrderCnt}(\text{picA}) - \text{PicOrderCnt}(\text{picB}) \quad (8-4)$$

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 -2^{15} to $2^{15} - 1$, inclusive.

NOTE 2 – Let X be the current picture and Y and Z be two other pictures in the same sequence, Y and Z are considered to be in the same output order direction from X when both `DiffPicOrderCnt( X, Y )` and `DiffPicOrderCnt( X, Z )` are positive or both are negative.

8.3.2 Decoding process for reference picture set

This process is invoked once per picture, after decoding of a slice header but prior to the decoding of any coding unit and prior to the decoding process for reference picture list construction for the slice as specified in subclause 8.3.3. 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".

NOTE 1 – The RPS is an absolute description of the reference pictures used in the decoding process of the current and future coded pictures. The RPS signalling is explicit in the sense that all reference pictures included in the RPS are listed explicitly.

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).

When the current picture is an IRAP picture with NoRaslOutputFlag equal to 1, all reference pictures currently in the DPB (if any) are marked as "unused for reference".

Short-term reference pictures are identified by their PicOrderCntVal values. Long-term reference pictures are identified either by their PicOrderCntVal values or their slice_pic_order_cnt_lsb values.

Five lists of picture order count values are constructed to derive the RPS. These five lists are PocStCurrBefore, PocStCurrAfter, PocStFoll, PocLtCurr, and PocLtFoll, with NumPocStCurrBefore, NumPocStCurrAfter, NumPocStFoll, NumPocLtCurr, and NumPocLtFoll number of elements, respectively. The five lists and the five variables are derived as follows:

- If the current picture is an IDR picture, PocStCurrBefore, PocStCurrAfter, PocStFoll, PocLtCurr, and PocLtFoll are all set to be empty, and NumPocStCurrBefore, NumPocStCurrAfter, NumPocStFoll, NumPocLtCurr, and NumPocLtFoll are all set equal to 0.
- Otherwise, the following applies:

```

for( i = 0, j = 0, k = 0; i < NumNegativePics[ CurrRpsIdx ]; i++ )
    if( UsedByCurrPicS0[ CurrRpsIdx ][ i ] )
        PocStCurrBefore[ j++ ] = PicOrderCntVal + DeltaPocS0[ CurrRpsIdx ][ i ]
    else
        PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS0[ CurrRpsIdx ][ i ]
NumPocStCurrBefore = j

for( i = 0, j = 0; i < NumPositivePics[ CurrRpsIdx ]; i++ )
    if( UsedByCurrPicS1[ CurrRpsIdx ][ i ] )
        PocStCurrAfter[ j++ ] = PicOrderCntVal + DeltaPocS1[ CurrRpsIdx ][ i ]
    else
        PocStFoll[ k++ ] = PicOrderCntVal + DeltaPocS1[ CurrRpsIdx ][ i ]
NumPocStCurrAfter = j
NumPocStFoll = k
for( i = 0, j = 0, k = 0; i < num_long_term_sps + num_long_term_pics; i++ ) {
    pocLt = PocLsbLt[ i ]
    if( delta_poc_msb_present_flag[ i ] )
        pocLt += PicOrderCntVal - DeltaPocMsbCycleLt[ i ] * MaxPicOrderCntLsb - slice_pic_order_cnt_lsb
    if( UsedByCurrPicLt[ i ] ) {
        PocLtCurr[ j ] = pocLt
        CurrDeltaPocMsbPresentFlag[ j++ ] = delta_poc_msb_present_flag[ i ]
    } else {
        PocLtFoll[ k ] = pocLt
        FollDeltaPocMsbPresentFlag[ k++ ] = delta_poc_msb_present_flag[ i ]
    }
}
NumPocLtCurr = j
NumPocLtFoll = k

```

(8-5)

where PicOrderCntVal is the picture order count of the current picture as specified in subclause 8.3.1.

NOTE 2 – A value of CurrRpsIdx in the range of 0 to num_short_term_ref_pic_sets – 1, inclusive, indicates that a candidate short-term RPS from the active SPS is being used, where CurrRpsIdx is the index of the candidate short-term RPS into the list of candidate short-term RPSs signalled in the active SPS. CurrRpsIdx equal to num_short_term_ref_pic_sets indicates that the short-term RPS of the current picture is directly signalled in the slice header.

For each i in the range of 0 to NumPocLtCurr – 1, inclusive, when CurrDeltaPocMsbPresentFlag[i] is equal to 1, it is a requirement of bitstream conformance that the following conditions apply:

- There shall be no j in the range of 0 to NumPocStCurrBefore – 1, inclusive, for which PocLtCurr[i] is equal to PocStCurrBefore[j].
- There shall be no j in the range of 0 to NumPocStCurrAfter – 1, inclusive, for which PocLtCurr[i] is equal to PocStCurrAfter[j].

- There shall be no j in the range of 0 to $\text{NumPocStFoll} - 1$, inclusive, for which $\text{PocLtCurr}[i]$ is equal to $\text{PocStFoll}[j]$.
- There shall be no j in the range of 0 to $\text{NumPocLtCurr} - 1$, inclusive, where j is not equal to i , for which $\text{PocLtCurr}[i]$ is equal to $\text{PocLtCurr}[j]$.

For each i in the range of 0 to $\text{NumPocLtFoll} - 1$, inclusive, when $\text{FollDeltaPocMsbPresentFlag}[i]$ is equal to 1, it is a requirement of bitstream conformance that the following conditions apply:

- There shall be no j in the range of 0 to $\text{NumPocStCurrBefore} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $\text{PocStCurrBefore}[j]$.
- There shall be no j in the range of 0 to $\text{NumPocStCurrAfter} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $\text{PocStCurrAfter}[j]$.
- There shall be no j in the range of 0 to $\text{NumPocStFoll} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $\text{PocStFoll}[j]$.
- There shall be no j in the range of 0 to $\text{NumPocLtFoll} - 1$, inclusive, where j is not equal to i , for which $\text{PocLtFoll}[i]$ is equal to $\text{PocLtFoll}[j]$.
- There shall be no j in the range of 0 to $\text{NumPocLtCurr} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $\text{PocLtCurr}[j]$.

For each i in the range of 0 to $\text{NumPocLtCurr} - 1$, inclusive, when $\text{CurrDeltaPocMsbPresentFlag}[i]$ is equal to 0, it is a requirement of bitstream conformance that the following conditions apply:

- There shall be no j in the range of 0 to $\text{NumPocStCurrBefore} - 1$, inclusive, for which $\text{PocLtCurr}[i]$ is equal to $(\text{PocStCurrBefore}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocStCurrAfter} - 1$, inclusive, for which $\text{PocLtCurr}[i]$ is equal to $(\text{PocStCurrAfter}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocStFoll} - 1$, inclusive, for which $\text{PocLtCurr}[i]$ is equal to $(\text{PocStFoll}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocLtCurr} - 1$, inclusive, where j is not equal to i , for which $\text{PocLtCurr}[i]$ is equal to $(\text{PocLtCurr}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.

For each i in the range of 0 to $\text{NumPocLtFoll} - 1$, inclusive, when $\text{FollDeltaPocMsbPresentFlag}[i]$ is equal to 0, it is a requirement of bitstream conformance that the following conditions apply:

- There shall be no j in the range of 0 to $\text{NumPocStCurrBefore} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $(\text{PocStCurrBefore}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocStCurrAfter} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $(\text{PocStCurrAfter}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocStFoll} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $(\text{PocStFoll}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocLtFoll} - 1$, inclusive, where j is not equal to i , for which $\text{PocLtFoll}[i]$ is equal to $(\text{PocLtFoll}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.
- There shall be no j in the range of 0 to $\text{NumPocLtCurr} - 1$, inclusive, for which $\text{PocLtFoll}[i]$ is equal to $(\text{PocLtCurr}[j] \& (\text{MaxPicOrderCntLsb} - 1))$.

The variable NumPocTotalCurr is derived as specified in subclause 7.4.7.2. It is a requirement of bitstream conformance that the following applies to the value of NumPocTotalCurr :

- If the current picture is a BLA or CRA picture, the value of NumPocTotalCurr shall be equal to 0.
- Otherwise, when the current picture contains a P or B slice, the value of NumPocTotalCurr shall not be equal to 0.

The RPS of the current picture consists of five RPS lists; $\text{RefPicSetStCurrBefore}$, $\text{RefPicSetStCurrAfter}$, RefPicSetStFoll , RefPicSetLtCurr and RefPicSetLtFoll . $\text{RefPicSetStCurrBefore}$, $\text{RefPicSetStCurrAfter}$, and RefPicSetStFoll are collectively referred to as the short-term RPS. RefPicSetLtCurr and RefPicSetLtFoll are collectively referred to as the long-term RPS.

NOTE 3 – $\text{RefPicSetStCurrBefore}$, $\text{RefPicSetStCurrAfter}$, and RefPicSetLtCurr contain 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. RefPicSetStFoll and RefPicSetLtFoll consist of 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.

The derivation process for the RPS and picture marking are performed according to the following ordered steps:

1. The following applies:

```

for( i = 0; i < NumPocLtCurr; i++ )
    if( !CurrDeltaPocMsbPresentFlag[ i ] )
        if( there is a reference picture picX in the DPB with slice_pic_order_cnt_lsb equal to PocLtCurr[ i ] )
            RefPicSetLtCurr[ i ] = picX
        else
            RefPicSetLtCurr[ i ] = "no reference picture"
    else
        if( there is a reference picture picX in the DPB with PicOrderCntVal equal to PocLtCurr[ i ] )
            RefPicSetLtCurr[ i ] = picX
        else
            RefPicSetLtCurr[ i ] = "no reference picture"
for( i = 0; i < NumPocLtFoll; i++ )
    if( !FollDeltaPocMsbPresentFlag[ i ] )
        if( there is a reference picture picX in the DPB with slice_pic_order_cnt_lsb equal to PocLtFoll[ i ] )
            RefPicSetLtFoll[ i ] = picX
        else
            RefPicSetLtFoll[ i ] = "no reference picture"
    else
        if( there is a reference picture picX in the DPB with PicOrderCntVal equal to PocLtFoll[ i ] )
            RefPicSetLtFoll[ i ] = picX
        else
            RefPicSetLtFoll[ i ] = "no reference picture"

```

(8-6)

2. All reference pictures that are included in RefPicSetLtCurr and RefPicSetLtFoll are marked as "used for long-term reference".
3. The following applies:

```

for( i = 0; i < NumPocStCurrBefore; i++ )
    if( there is a short-term reference picture picX in the DPB
        with PicOrderCntVal equal to PocStCurrBefore[ i ] )
        RefPicSetStCurrBefore[ i ] = picX
    else
        RefPicSetStCurrBefore[ i ] = "no reference picture"
for( i = 0; i < NumPocStCurrAfter; i++ )
    if( there is a short-term reference picture picX in the DPB
        with PicOrderCntVal equal to PocStCurrAfter[ i ] )
        RefPicSetStCurrAfter[ i ] = picX
    else
        RefPicSetStCurrAfter[ i ] = "no reference picture"
for( i = 0; i < NumPocStFoll; i++ )
    if( there is a short-term reference picture picX in the DPB
        with PicOrderCntVal equal to PocStFoll[ i ] )
        RefPicSetStFoll[ i ] = picX
    else
        RefPicSetStFoll[ i ] = "no reference picture"

```

(8-7)

4. All reference pictures in the DPB that are not included in RefPicSetLtCurr, RefPicSetLtFoll, RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetStFoll are marked as "unused for reference".

NOTE 4 – There may be one or more entries in the RPS lists that are equal to "no reference picture" because the corresponding pictures are not present in the DPB. Entries in RefPicSetStFoll or RefPicSetLtFoll that are equal to "no reference picture" should be ignored. An unintentional picture loss should be inferred for each entry in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr that is equal to "no reference picture".

NOTE 5 – A picture cannot be included in more than one of the five RPS lists.

It is a requirement of bitstream conformance that the RPS is restricted as follows:

- There shall be no entry in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr for which one or more of the following are true:
 - The entry is equal to "no reference picture".

- The entry is a sub-layer non-reference picture and has TemporalId equal to that of the current picture.
- The entry is a picture that has TemporalId greater than that of the current picture.
- There shall be no entry in RefPicSetLtCurr or RefPicSetLtFoll for which the difference between the picture order count value of the current picture and the picture order count value of the entry is greater than or equal to 2^{24} .
- When the current picture is a TSA picture, there shall be no picture included in the RPS with TemporalId greater than or equal to the TemporalId of the current picture.
- When the current picture is an STSA picture, there shall be no picture included in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr that has TemporalId equal to that of the current picture.
- When the current picture is a picture that follows, in decoding order, an STSA picture that has TemporalId equal to that of the current picture, there shall be no picture that has TemporalId equal to that of the current picture included in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr that precedes the STSA picture in decoding order.
- When the current picture is a CRA picture, there shall be no picture included in the RPS that precedes, in decoding order, any preceding IRAP picture in decoding order (when present).
- When the current picture is a trailing picture, there shall be no picture in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr that was generated by the decoding process for generating unavailable reference pictures as specified in subclause 8.3.3.
- When the current picture is a trailing picture, there shall be no picture in the RPS that precedes the associated IRAP picture in output order or decoding order.
- When the current picture is a RADL picture, there shall be no picture included in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr that is any of the following:
 - A RASL picture
 - A picture that was generated by the decoding process for generating unavailable reference pictures as specified in subclause 8.3.3
 - A picture that precedes the associated IRAP picture in decoding order
- When the sps_temporal_id_nesting_flag is equal to 1, the following applies:
 - Let tIdA be the value of TemporalId of the current picture picA.
 - Any picture picB with TemporalId equal to tIdB that is less than or equal to tIdA shall not be included in RefPicSetStCurrBefore, RefPicSetStCurrAfter, or RefPicSetLtCurr of picA when there exists a picture picC that has TemporalId less than tIdB, follows picB in decoding order, and precedes picA in decoding order.

8.3.3 Decoding process for generating unavailable reference pictures

8.3.3.1 General decoding process for generating unavailable reference pictures

This process is invoked once per coded picture when the current picture is a BLA picture or is a CRA picture with NoRasOutputFlag equal to 1.

NOTE – This process is primarily specified only for the specification of syntax constraints for RASL pictures. The entire specification of the decoding process for RASL pictures associated with an IRAP picture that has NoRasOutputFlag equal to 1 is included herein only for purposes of specifying constraints on the allowed syntax content of such RASL pictures. During the decoding process, any RASL pictures associated with an IRAP picture that has NoRasOutputFlag equal to 1 may be ignored, as these pictures are not specified for output and have no effect on the decoding process of any other pictures that are specified for output. However, in HRD operations as specified in Annex C, RASL access units may need to be taken into consideration in derivation of CPB arrival and removal times.

When this process is invoked, the following applies:

- For each RefPicSetStFoll[i], with i in the range of 0 to NumPocStFoll – 1, inclusive, that is equal to "no reference picture", a picture is generated as specified in subclause 8.3.3.2, and the following applies:
 - The value of PicOrderCntVal for the generated picture is set equal to PocStFoll[i].
 - The value of PicOutputFlag for the generated picture is set equal to 0.
 - The generated picture is marked as "used for short-term reference".
 - RefPicSetStFoll[i] is set to be the generated reference picture.

- For each $\text{RefPicSetLtFoll}[i]$, with i in the range of 0 to $\text{NumPocLtFoll} - 1$, inclusive, that is equal to "no reference picture", a picture is generated as specified in subclause 8.3.3.2, and the following applies:
 - The value of PicOrderCntVal for the generated picture is set equal to $\text{PocLtFoll}[i]$.
 - The value of $\text{slice_pic_order_cnt_lsb}$ for the generated picture is inferred to be equal to $(\text{PocLtFoll}[i] \& (\text{MaxPicOrderCntLsb} - 1))$.
 - The value of PicOutputFlag for the generated picture is set equal to 0.
 - The generated picture is marked as "used for long-term reference".
 - $\text{RefPicSetLtFoll}[i]$ is set to be the generated reference picture.

8.3.3.2 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 S_L for the picture is set equal to $1 \ll (\text{BitDepth}_Y - 1)$.
- The value of each element in the sample arrays S_{Cb} and S_{Cr} for the picture is set equal to $1 \ll (\text{BitDepth}_C - 1)$.
- The prediction mode $\text{CuPredMode}[x][y]$ is set equal to MODE_INTRA for $x = 0..\text{pic_width_in_luma_samples} - 1, y = 0..\text{pic_height_in_luma_samples} - 1$.

8.3.4 Decoding process for reference picture lists construction

This process is invoked at the beginning of the decoding process for each P or B slice.

Reference pictures are addressed through reference indices as specified in subclause 8.5.3.3.2. A reference index is an index into a reference picture list. When decoding a P slice, there is a single reference picture list RefPicList0 . When decoding a B slice, there is a second independent reference picture list RefPicList1 in addition to RefPicList0 .

At the beginning of the decoding process for each slice, the reference picture lists RefPicList0 and, for B slices, RefPicList1 are derived as follows:

The variable $\text{NumRpsCurrTempList0}$ is set equal to $\text{Max}(\text{num_ref_idx_l0_active_minus1} + 1, \text{NumPocTotalCurr})$ and the list RefPicListTemp0 is constructed as follows:

```

rIdx = 0
while( rIdx < NumRpsCurrTempList0 ) {
    for( i = 0; i < NumPocStCurrBefore && rIdx < NumRpsCurrTempList0; rIdx++, i++ )
        RefPicListTemp0[ rIdx ] = RefPicSetStCurrBefore[ i ]
    for( i = 0; i < NumPocStCurrAfter && rIdx < NumRpsCurrTempList0; rIdx++, i++ )
        RefPicListTemp0[ rIdx ] = RefPicSetStCurrAfter[ i ]
    for( i = 0; i < NumPocLtCurr && rIdx < NumRpsCurrTempList0; rIdx++, i++ )
        RefPicListTemp0[ rIdx ] = RefPicSetLtCurr[ i ]
}

```

(8-8)

The list RefPicList0 is constructed as follows:

```

for( rIdx = 0; rIdx <= num_ref_idx_l0_active_minus1; rIdx++ )
    RefPicList0[ rIdx ] = ref_pic_list_modification_flag_l0 ? RefPicListTemp0[ list_entry_l0[ rIdx ] ] :
                                                                RefPicListTemp0[ rIdx ]

```

(8-9)

When the slice is a B slice, the variable $\text{NumRpsCurrTempList1}$ is set equal to $\text{Max}(\text{num_ref_idx_l1_active_minus1} + 1, \text{NumPocTotalCurr})$ and the list RefPicListTemp1 is constructed as follows:

```

rIdx = 0
while( rIdx < NumRpsCurrTempList1 ) {
    for( i = 0; i < NumPocStCurrAfter && rIdx < NumRpsCurrTempList1; rIdx++, i++ )
        RefPicListTemp1[ rIdx ] = RefPicSetStCurrAfter[ i ]
    for( i = 0; i < NumPocStCurrBefore && rIdx < NumRpsCurrTempList1; rIdx++, i++ )
        RefPicListTemp1[ rIdx ] = RefPicSetStCurrBefore[ i ]
    for( i = 0; i < NumPocLtCurr && rIdx < NumRpsCurrTempList1; rIdx++, i++ )
        RefPicListTemp1[ rIdx ] = RefPicSetLtCurr[ i ]
}

```

(8-10)

When the slice is a B slice, the list RefPicList1 is constructed as follows:

```
for( rIdx = 0; rIdx <= num_ref_idx_l1_active_minus1; rIdx++)
    RefPicList1[ rIdx ] = ref_pic_list_modification_flag_l1 ? RefPicListTemp1[ list_entry_l1[ rIdx ] ] :
                                                                RefPicListTemp1[ rIdx ]
```

(8-11)

8.4 Decoding process for coding units coded in intra prediction mode

8.4.1 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 luma coding block relative to the top-left luma sample of the current picture,
- a variable log2CbSize specifying the size of the current luma coding block.

Output of this process is a modified reconstructed picture before deblocking filtering.

The derivation process for quantization parameters as specified in subclause 8.6.1 is invoked with the luma location (xCb, yCb) as input.

A variable nCbS is set equal to $1 \ll \log_2 \text{CbSize}$.

Depending on the values of pcm_flag[xCb][yCb] and IntraSplitFlag, 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:

$$S_L[xCb + i][yCb + j] = \text{pcm_sample_luma}[(nCbS * j) + i] \ll (\text{BitDepth}_Y - \text{PcmBitDepth}_Y), \text{ with } i, j = 0..nCbS - 1 \quad (8-12)$$

- Otherwise (pcm_flag[xCb][yCb] is equal to 0), if IntraSplitFlag is equal to 0, the following ordered steps apply:
 1. The derivation process for the intra prediction mode as specified in subclause 8.4.2 is invoked with the luma location (xCb, yCb) as input.
 2. The general decoding process for intra blocks as specified in subclause 8.4.4.1 is invoked with the luma location (xCb, yCb), the variable log2TrafoSize set equal to log2CbSize, the variable trafoDepth set equal to 0, the variable predModeIntra set equal to IntraPredModeY[xCb][yCb], and the variable cIdx set equal to 0 as inputs, and the output is a modified reconstructed picture before deblocking filtering.
- Otherwise (pcm_flag[xCb][yCb] is equal to 0 and IntraSplitFlag is equal to 1), for the variable blkIdx proceeding over the values 0..3, the following ordered steps apply:
 1. The variable xPb is set equal to $xCb + (nCbS \gg 1) * (\text{blkIdx} \% 2)$.
 2. The variable yPb is set equal to $yCb + (nCbS \gg 1) * (\text{blkIdx} / 2)$.
 3. The derivation process for the intra prediction mode as specified in subclause 8.4.2 is invoked with the luma location (xPb, yPb) as input.
 4. The general decoding process for intra blocks as specified in subclause 8.4.4.1 is invoked with the luma location (xPb, yPb), the variable log2TrafoSize set equal to log2CbSize – 1, the variable trafoDepth set equal to 1, the variable predModeIntra set equal to IntraPredModeY[xPb][yPb], and the variable cIdx set equal to 0 as inputs, and the output is a modified reconstructed picture before deblocking filtering.

Depending on the value of pcm_flag[xCb][yCb], 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:

$$S_{Cb}[xCb / 2 + i][yCb / 2 + j] = \text{pcm_sample_chroma}[(nCbS / 2 * j) + i] \ll (\text{BitDepth}_C - \text{PcmBitDepth}_C), \text{ with } i, j = 0..nCbS / 2 - 1 \quad (8-13)$$

$$S_{Cr}[xCb / 2 + i][yCb / 2 + j] = \text{pcm_sample_chroma}[(nCbS / 2 * (j + nCbS / 2)) + i] \ll (\text{BitDepth}_C - \text{PcmBitDepth}_C), \text{ with } i, j = 0..nCbS / 2 - 1 \quad (8-14)$$

- Otherwise (pcm_flag[xCb][yCb] is equal to 0), the following ordered steps apply:

1. The derivation process for the chroma intra prediction mode as specified in 8.4.3 is invoked with the luma location (xCb, yCb) as input, and the output is the variable IntraPredModeC.
2. The general decoding process for intra blocks as specified in subclause 8.4.4.1 is invoked with the chroma location (xCb / 2, yCb / 2), the variable log2TrafoSize set equal to log2CbSize – 1, the variable trafoDepth set

equal to 0, the variable `predModeIntra` set equal to `IntraPredModeC`, and the variable `cIdx` set equal to 1 as inputs, and the output is a modified reconstructed picture before deblocking filtering.

3. The general decoding process for intra blocks as specified in subclause 8.4.4.1 is invoked with the chroma location ($x_{Cb} / 2, y_{Cb} / 2$), the variable `log2TrafoSize` set equal to `log2CbSize - 1`, the variable `trafoDepth` set equal to 0, the variable `predModeIntra` set equal to `IntraPredModeC`, and the variable `cIdx` set equal to 2 as inputs, and the output is a modified reconstructed picture before deblocking filtering.

8.4.2 Derivation process for luma intra prediction mode

Input to this process is a luma location (x_{Pb}, y_{Pb}) specifying the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture.

In this process, the luma intra prediction mode `IntraPredModeY[  $x_{Pb}$  ][  $y_{Pb}$  ]` is derived.

Table 8-1 specifies the value for the intra prediction mode and the associated names.

Table 8-1 – Specification of intra prediction mode and associated names

Intra prediction mode	Associated name
0	INTRA_PLANAR
1	INTRA_DC
2..34	INTRA_ANGULAR2..INTRA_ANGULAR34

`IntraPredModeY[  $x_{Pb}$  ][  $y_{Pb}$  ]` labelled 0..34 represents directions of predictions as illustrated in Figure 8-1.

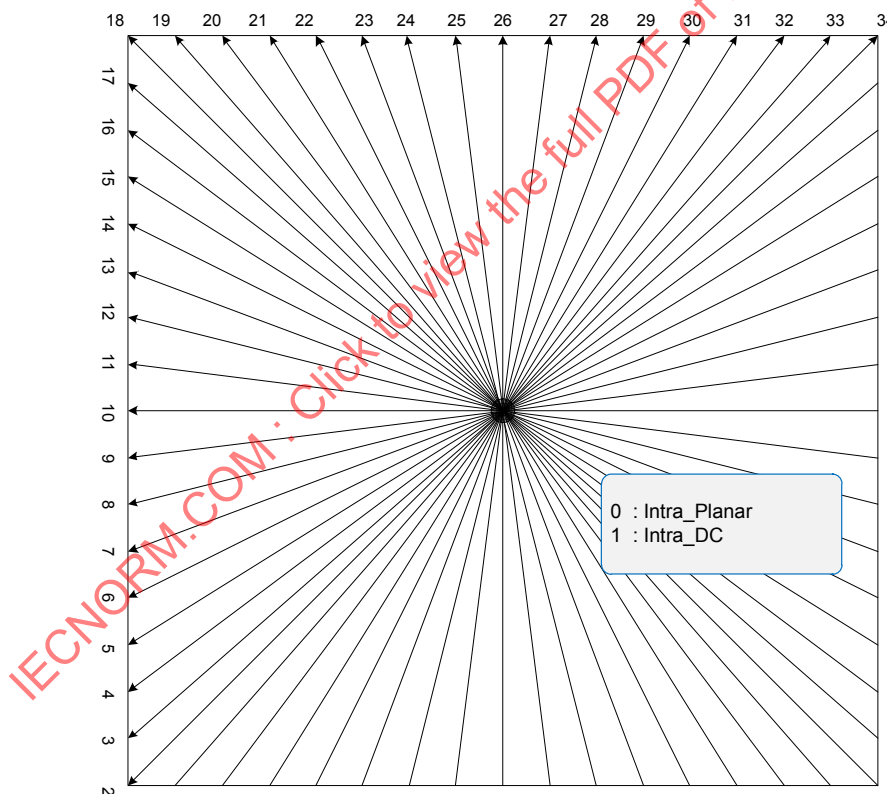


Figure 8-1 – Intra prediction mode directions (informative)

`IntraPredModeY[  $x_{Pb}$  ][  $y_{Pb}$  ]` is derived by the following ordered steps:

1. The neighbouring locations (x_{NbA}, y_{NbA}) and (x_{NbB}, y_{NbB}) are set equal to ($x_{Pb} - 1, y_{Pb}$) and ($x_{Pb}, y_{Pb} - 1$), 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 in z-scan order as specified in subclause 6.4.1 is invoked with the location (xCurr, yCurr) set equal to (xPb, yPb) 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 availableX is equal to FALSE, candIntraPredModeX is set equal to INTRA_DC.
 - Otherwise, if CuPredMode[xNbX][yNbX] is not equal to MODE_INTRA or pcm_flag[xNbX][yNbX] is equal to 1, candIntraPredModeX is set equal to INTRA_DC,
 - Otherwise, if X is equal to B and yPb – 1 is less than ((yPb >> CtbLog2SizeY) << CtbLog2SizeY), candIntraPredModeB is set equal to INTRA_DC.
 - Otherwise, candIntraPredModeX is set equal to IntraPredModeY[xNbX][yNbX].
3. The candModeList[x] with x = 0..2 is derived as follows:
- If candIntraPredModeB is equal to candIntraPredModeA, the following applies:
 - If candIntraPredModeA is less than 2 (i.e. equal to INTRA_PLANAR or INTRA_DC), candModeList[x] with x = 0..2 is derived as follows:

$$\text{candModeList}[0] = \text{INTRA_PLANAR} \quad (8-15)$$

$$\text{candModeList}[1] = \text{INTRA_DC} \quad (8-16)$$

$$\text{candModeList}[2] = \text{INTRA_ANGULAR26} \quad (8-17)$$
 - Otherwise, candModeList[x] with x = 0..2 is derived as follows:

$$\text{candModeList}[0] = \text{candIntraPredModeA} \quad (8-18)$$

$$\text{candModeList}[1] = 2 + ((\text{candIntraPredModeA} + 29) \% 32) \quad (8-19)$$

$$\text{candModeList}[2] = 2 + ((\text{candIntraPredModeA} - 2 + 1) \% 32) \quad (8-20)$$
 - Otherwise (candIntraPredModeB is not equal to candIntraPredModeA), the following applies:
 - candModeList[0] and candModeList[1] are derived as follows:

$$\text{candModeList}[0] = \text{candIntraPredModeA} \quad (8-21)$$

$$\text{candModeList}[1] = \text{candIntraPredModeB} \quad (8-22)$$
 - If neither of candModeList[0] and candModeList[1] is equal to INTRA_PLANAR, candModeList[2] is set equal to INTRA_PLANAR,
 - Otherwise, if neither of candModeList[0] and candModeList[1] is equal to INTRA_DC, candModeList[2] is set equal to INTRA_DC,
 - Otherwise, candModeList[2] is set equal to INTRA_ANGULAR26.
4. IntraPredModeY[xPb][yPb] is derived by applying the following procedure:
- If prev_intra_luma_pred_flag[xPb][yPb] is equal to 1, the IntraPredModeY[xPb][yPb] is set equal to candModeList[mpm_idx].
 - Otherwise, IntraPredModeY[xPb][yPb] is derived by applying the following ordered steps:
 - 1) The array candModeList[x], x = 0..2 is modified as the following ordered steps:
 - i. When candModeList[0] is greater than candModeList[1], both values are swapped as follows:

$$(\text{candModeList}[0], \text{candModeList}[1]) = \text{Swap}(\text{candModeList}[0], \text{candModeList}[1]) \quad (8-23)$$
 - ii. When candModeList[0] is greater than candModeList[2], both values are swapped as follows:

$$(\text{candModeList}[0], \text{candModeList}[2]) = \text{Swap}(\text{candModeList}[0], \text{candModeList}[2]) \quad (8-24)$$
 - iii. When candModeList[1] is greater than candModeList[2], both values are swapped as follows:

$$(\text{candModeList}[1], \text{candModeList}[2]) = \text{Swap}(\text{candModeList}[1], \text{candModeList}[2]) \quad (8-25)$$
 - 2) IntraPredModeY[xPb][yPb] is derived by the following ordered steps:
 - i. IntraPredModeY[xPb][yPb] is set equal to rem_intra_luma_pred_mode[xPb][yPb].

- ii. For i equal to 0 to 2, inclusive, when $\text{IntraPredModeY}[xPb][yPb]$ is greater than or equal to $\text{candModeList}[i]$, the value of $\text{IntraPredModeY}[xPb][yPb]$ is incremented by one.

8.4.3 Derivation process for chroma intra prediction mode

Input to this process is 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.

Output of this process is the variable IntraPredModeC .

The chroma intra prediction mode IntraPredModeC is derived using $\text{intra_chroma_pred_mode}[xCb][yCb]$ and $\text{IntraPredModeY}[xCb][yCb]$ as specified in Table 8-2.

Table 8-2 – Specification of IntraPredModeC

$\text{intra_chroma_pred_mode}[xCb][yCb]$	$\text{IntraPredModeY}[xCb][yCb]$				
	0	26	10	1	$X (0 \leq X \leq 34)$
0	34	0	0	0	0
1	26	34	26	26	26
2	10	10	34	10	10
3	1	1	1	34	1
4	0	26	10	1	X

8.4.4 Decoding process for intra blocks

8.4.4.1 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 $\log2\text{TrafoSize}$ specifying the size of the current transform block,
- a variable trafoDepth specifying the hierarchy depth of the current block relative to the coding unit,
- 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 deblocking filtering.

The luma sample 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 is derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTb0, yTb0) : (xTb0 \ll 1, yTb0 \ll 1) \quad (8-26)$$

The variable splitFlag is derived as follows:

- If $cIdx$ is equal to 0, splitFlag is set equal to $\text{split_transform_flag}[xTbY][yTbY][\text{trafoDepth}]$.
- Otherwise, if all of the following conditions are true, splitFlag is set equal to 1.
 - $cIdx$ is greater than 0
 - $\text{split_transform_flag}[xTbY][yTbY][\text{trafoDepth}]$ is equal to 1
 - $\log2\text{TrafoSize}$ is greater than 2
- Otherwise, splitFlag is set equal to 0.

Depending on the value of splitFlag , the following applies:

- If splitFlag is equal to 1, the following ordered steps apply:

1. The variables $xTb1$ and $yTb1$ are derived as follows:

- The variable $xTb1$ is set equal to $xTb0 + (1 \ll (\log2\text{TrafoSize} - 1))$.

- The variable $yTb1$ is set equal to $yTb0 + (1 \ll (\log2TrafoSize - 1))$.
- 2. The general decoding process for intra blocks as specified in this subclause is invoked with the location $(xTb0, yTb0)$, the variable $\log2TrafoSize$ set equal to $\log2TrafoSize - 1$, the variable $trafoDepth$ set equal to $trafoDepth + 1$, the intra prediction mode $predModeIntra$, and the variable $cIdx$ as inputs, and the output is a modified reconstructed picture before deblocking filtering.
- 3. The general decoding process for intra blocks as specified in this subclause is invoked with the location $(xTb1, yTb0)$, the variable $\log2TrafoSize$ set equal to $\log2TrafoSize - 1$, the variable $trafoDepth$ set equal to $trafoDepth + 1$, the intra prediction mode $predModeIntra$, and the variable $cIdx$ as inputs, and the output is a modified reconstructed picture before deblocking filtering.
- 4. The general decoding process for intra blocks as specified in this subclause is invoked with the location $(xTb0, yTb1)$, the variable $\log2TrafoSize$ set equal to $\log2TrafoSize - 1$, the variable $trafoDepth$ set equal to $trafoDepth + 1$, the intra prediction mode $predModeIntra$, and the variable $cIdx$ as inputs, and the output is a modified reconstructed picture before deblocking filtering.
- 5. The general decoding process for intra blocks as specified in this subclause is invoked with the location $(xTb1, yTb1)$, the variable $\log2TrafoSize$ set equal to $\log2TrafoSize - 1$, the variable $trafoDepth$ set equal to $trafoDepth + 1$, the intra prediction mode $predModeIntra$, and the variable $cIdx$ as inputs, and the output is a modified reconstructed picture before deblocking filtering.
- Otherwise ($splitFlag$ is equal to 0), the following ordered steps apply:
 1. The variable $nTbS$ is set equal to $1 \ll \log2TrafoSize$.
 2. The general intra sample prediction process as specified in subclause 8.4.4.2.1 is invoked with the transform block location $(xTb0, yTb0)$, the intra prediction mode $predModeIntra$, the transform block size $nTbS$, and the variable $cIdx$ as inputs, and the output is an $(nTbS) \times (nTbS)$ array $predSamples$.
 3. The scaling and transformation process as specified in subclause 8.6.2 is invoked with the luma location $(xTbY, yTbY)$, the variable $trafoDepth$, the variable $cIdx$, and the transform size $trafoSize$ set equal to $nTbS$ as inputs, and the output is an $(nTbS) \times (nTbS)$ array $resSamples$.
 4. The picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 8.6.5 is invoked with the transform block location $(xTb0, yTb0)$, the transform block size $nTbS$, the variable $cIdx$, the $(nTbS) \times (nTbS)$ array $predSamples$, and the $(nTbS) \times (nTbS)$ array $resSamples$ as inputs.

8.4.4.2 Intra sample prediction

8.4.4.2.1 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 $nTbS$ specifying the transform block size,
- a variable $cIdx$ specifying the colour component of the current block.

Output of this process is the predicted samples $predSamples[x][y]$, with $x, y = 0..nTbS - 1$.

The $nTbS * 4 + 1$ neighbouring samples $p[x][y]$ that are constructed samples prior to the deblocking filter process, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$, are derived as follows:

- The neighbouring location $(xNbCmp, yNbCmp)$ is specified by:

$$(xNbCmp, yNbCmp) = (xTbCmp + x, yTbCmp + y) \quad (8-27)$$

- The current luma location $(xTbY, yTbY)$ and the neighbouring luma location $(xNbY, yNbY)$ are derived as follows:

$$(xTbY, yTbY) = (cIdx == 0) ? (xTbCmp, yTbCmp) : (xTbCmp \ll 1, yTbCmp \ll 1) \quad (8-28)$$

$$(xNbY, yNbY) = (cIdx == 0) ? (xNbCmp, yNbCmp) : (xNbCmp \ll 1, yNbCmp \ll 1) \quad (8-29)$$

- The availability derivation process for a block in z-scan order as specified in subclause 6.4.1 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 $p[x][y]$ is derived as follows:
 - If one or more of the following conditions are true, the sample $p[x][y]$ is marked as "not available for intra prediction":
 - The variable availableN is equal to FALSE.
 - $\text{CuPredMode}[xNbY][yNbY]$ is not equal to MODE_INTRA and constrained_intra_pred_flag is equal to 1.
 - Otherwise, the sample $p[x][y]$ is marked as "available for intra prediction" and the sample at the location $(xNbCmp, yNbCmp)$ is assigned to $p[x][y]$.

When at least one sample $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ is marked as "not available for intra prediction", the reference sample substitution process for intra sample prediction in subclause 8.4.4.2.2 is invoked with the samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1, nTbS$, and cIdx as inputs, and the modified samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ as output.

Depending on the value of predModeIntra, the following ordered steps apply:

1. When cIdx is equal to 0, the filtering process of neighbouring samples specified in subclause 8.4.4.2.3 is invoked with the sample array p and the transform block size nTbS as inputs, and the output is reassigned to the sample array p.
2. The intra sample prediction process according to predModeIntra applies as follows:
 - If predModeIntra is equal to INTRA_PLANAR, the corresponding intra prediction mode specified in subclause 8.4.4.2.4 is invoked with the sample array p and the transform block size nTbS as inputs, and the output is the predicted sample array predSamples.
 - Otherwise, if predModeIntra is equal to INTRA_DC, the corresponding intra prediction mode specified in subclause 8.4.4.2.5 is invoked with the sample array p, the transform block size nTbS, and the colour component index cIdx as inputs, and the output is the predicted sample array predSamples.
 - Otherwise (predModeIntra is in the range of INTRA_ANGULAR2..INTRA_ANGULAR34), the corresponding intra prediction mode specified in subclause 8.4.4.2.6 is invoked with the intra prediction mode predModeIntra, the sample array p, the transform block size nTbS, and the colour component index cIdx as inputs, and the output is the predicted sample array predSamples.

8.4.4.2.2 Reference sample substitution process for intra sample prediction

Inputs to this process are:

- reference samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ for intra sample prediction,
- a transform block size nTbS,
- a variable cIdx specifying the colour component of the current block.

Outputs of this process are the modified reference samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ for intra sample prediction.

The variable bitDepth is derived as follows:

- If cIdx is equal to 0, bitDepth is set equal to BitDepth_Y.
- Otherwise, bitDepth is set equal to BitDepth_C.

The values of the samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ are modified as follows:

- If all samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ are marked as "not available for intra prediction", the value $1 \ll (\text{bitDepth} - 1)$ is substituted for the values of all samples $p[x][y]$.
- Otherwise (at least one but not all samples $p[x][y]$ are marked as "not available for intra prediction"), the following ordered steps are performed:
 1. When $p[-1][nTbS * 2 - 1]$ is marked as "not available for intra prediction", search sequentially starting from $x = -1, y = nTbS * 2 - 1$ to $x = -1, y = -1$, then from $x = 0, y = -1$ to $x = nTbS * 2 - 1, y = -1$. Once a sample $p[x][y]$ marked as "available for intra prediction" is found, the search is terminated and the value of $p[x][y]$ is assigned to $p[-1][nTbS * 2 - 1]$.

2. Search sequentially starting from $x = -1, y = nTbS * 2 - 2$ to $x = -1, y = -1$, when $p[x][y]$ is marked as "not available for intra prediction", the value of $p[x][y + 1]$ is substituted for the value of $p[x][y]$.
3. For $x = 0..nTbS * 2 - 1, y = -1$, when $p[x][y]$ is marked as "not available for intra prediction", the value of $p[x - 1][y]$ is substituted for the value of $p[x][y]$.

All samples $p[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ are marked as "available for intra prediction".

8.4.4.2.3 Filtering process of neighbouring samples

Inputs to this process are:

- the neighbouring samples $p[x][y]$, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$,
- a variable $nTbS$ specifying the transform block size.

Outputs of this process are the filtered samples $pF[x][y]$, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$.

The variable $filterFlag$ is derived as follows:

- If one or more of the following conditions are true, $filterFlag$ is set equal to 0:
 - $predModeIntra$ is equal to `INTRA_DC`.
 - $nTbS$ is equal 4.
- Otherwise, the following applies:
 - The variable $minDistVerHor$ is set equal to $\text{Min}(\text{Abs}(\text{predModeIntra} - 26), \text{Abs}(\text{predModeIntra} - 10))$.
 - The variable $intraHorVerDistThres[nTbS]$ is specified in Table 8-3.
 - The variable $filterFlag$ is derived as follows:
 - If $minDistVerHor$ is greater than $intraHorVerDistThres[nTbS]$, $filterFlag$ is set equal to 1.
 - Otherwise, $filterFlag$ is set equal to 0.

Table 8-3 – Specification of $intraHorVerDistThres[nTbS]$ for various transform block sizes

	$nTbS = 8$	$nTbS = 16$	$nTbS = 32$
$intraHorVerDistThres[nTbS]$	7	1	0

When $filterFlag$ is equal to 1, the following applies:

- The variable $biIntFlag$ is derived as follows:
 - If all of the following conditions are true, $biIntFlag$ is set equal to 1:
 - $strong_intra_smoothing_enabled_flag$ is equal to 1
 - $nTbS$ is equal to 32
 - $\text{Abs}(p[-1][-1] + p[nTbS * 2 - 1][-1] - 2 * p[nTbS - 1][-1]) < (1 << (\text{BitDepth}_Y - 5))$
 - $\text{Abs}(p[-1][-1] + p[-1][nTbS * 2 - 1] - 2 * p[-1][nTbS - 1]) < (1 << (\text{BitDepth}_Y - 5))$
 - Otherwise, $biIntFlag$ is set equal to 0.
- The filtering is performed as follows:
 - If $biIntFlag$ is equal to 1, the filtered sample values $pF[x][y]$ with $x = -1, y = -1..63$ and $x = 0..63, y = -1$ are derived as follows:

$$pF[-1][-1] = p[-1][-1] \quad (8-30)$$

$$pF[-1][y] = ((63 - y) * p[-1][-1] + (y + 1) * p[-1][63] + 32) >> 6 \text{ for } y = 0..62 \quad (8-31)$$

$$pF[-1][63] = p[-1][63] \quad (8-32)$$

$$pF[x][-1] = ((63 - x) * p[-1][-1] + (x + 1) * p[63][-1] + 32) >> 6 \text{ for } x = 0..62 \quad (8-33)$$

$$pF[63][y-1] = p[63][y-1] \quad (8-34)$$

- Otherwise (biIntFlag is equal to 0), the filtered sample values $pF[x][y]$ with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$ are derived as follows:

$$pF[-1][y-1] = (p[-1][0] + 2 * p[-1][y-1] + p[0][y-1] + 2) >> 2 \quad (8-35)$$

$$pF[-1][y] = (p[-1][y+1] + 2 * p[-1][y] + p[-1][y-1] + 2) >> 2 \text{ for } y = 0..nTbS * 2 - 2 \quad (8-36)$$

$$pF[-1][nTbS * 2 - 1] = p[-1][nTbS * 2 - 1] \quad (8-37)$$

$$pF[x][y-1] = (p[x-1][y-1] + 2 * p[x][y-1] + p[x+1][y-1] + 2) >> 2 \text{ for } x = 0..nTbS * 2 - 2 \quad (8-38)$$

$$pF[nTbS * 2 - 1][y-1] = p[nTbS * 2 - 1][y-1] \quad (8-39)$$

8.4.4.2.4 Specification of intra prediction mode INTRA_PLANAR

Inputs to this process are:

- the neighbouring samples $p[x][y]$, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$,
- a variable $nTbS$ specifying the transform block size.

Outputs of this process are the predicted samples $predSamples[x][y]$, with $x, y = 0..nTbS - 1$.

The values of the prediction samples $predSamples[x][y]$, with $x, y = 0..nTbS - 1$, are derived as follows:

$$predSamples[x][y] = ((nTbS - 1 - x) * p[-1][y] + (x + 1) * p[nTbS][y-1] + (nTbS - 1 - y) * p[x][y-1] + (y + 1) * p[-1][nTbS] + nTbS) >> (Log2(nTbS) + 1) \quad (8-40)$$

8.4.4.2.5 Specification of intra prediction mode INTRA_DC

Inputs to this process are:

- the neighbouring samples $p[x][y]$, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$,
- a variable $nTbS$ specifying the transform block size,
- a variable $cIdx$ specifying the colour component of the current block.

Outputs of this process are the predicted samples $predSamples[x][y]$, with $x, y = 0..nTbS - 1$.

The values of the prediction samples $predSamples[x][y]$, with $x, y = 0..nTbS - 1$, are derived by the following ordered steps:

1. A variable $dcVal$ is derived as follows:

$$dcVal = \left(\sum_{x'=-1}^{nTbS-1} p[x'][y-1] + \sum_{y'=-1}^{nTbS-1} p[-1][y'] + nTbS \right) >> (k + 1) \quad (8-41)$$

where $k = Log2(nTbS)$.

2. Depending on the value of the colour component index $cIdx$, the following applies:

- If $cIdx$ is equal to 0 and $nTbS$ is less than 32, the following applies:

$$predSamples[0][0] = (p[-1][0] + 2 * dcVal + p[0][y-1] + 2) >> 2 \quad (8-42)$$

$$predSamples[x][0] = (p[x][y-1] + 3 * dcVal + 2) >> 2, \text{ with } x = 1..nTbS - 1 \quad (8-43)$$

$$predSamples[0][y] = (p[-1][y] + 3 * dcVal + 2) >> 2, \text{ with } y = 1..nTbS - 1 \quad (8-44)$$

$$predSamples[x][y] = dcVal, \text{ with } x, y = 1..nTbS - 1 \quad (8-45)$$

- Otherwise, the prediction samples $predSamples[x][y]$ are derived as follows:

$$predSamples[x][y] = dcVal, \text{ with } x, y = 0..nTbS - 1 \quad (8-46)$$

8.4.4.2.6 Specification of intra prediction mode in the range of INTRA_ANGULAR2.. INTRA_ANGULAR34

Inputs to this process are:

- the intra prediction mode $predModeIntra$,
- the neighbouring samples $p[x][y]$, with $x = -1, y = -1..nTbS * 2 - 1$ and $x = 0..nTbS * 2 - 1, y = -1$,

- a variable nTbS specifying the transform block size,
- a variable cIdx specifying the colour component of the current block.

Outputs of this process are the predicted samples $\text{predSamples}[x][y]$, with $x, y = 0..nTbS - 1$.

Figure 8-2 illustrates the total 33 intra angles and Table 8-4 specifies the mapping table between predModeIntra and the angle parameter intraPredAngle .

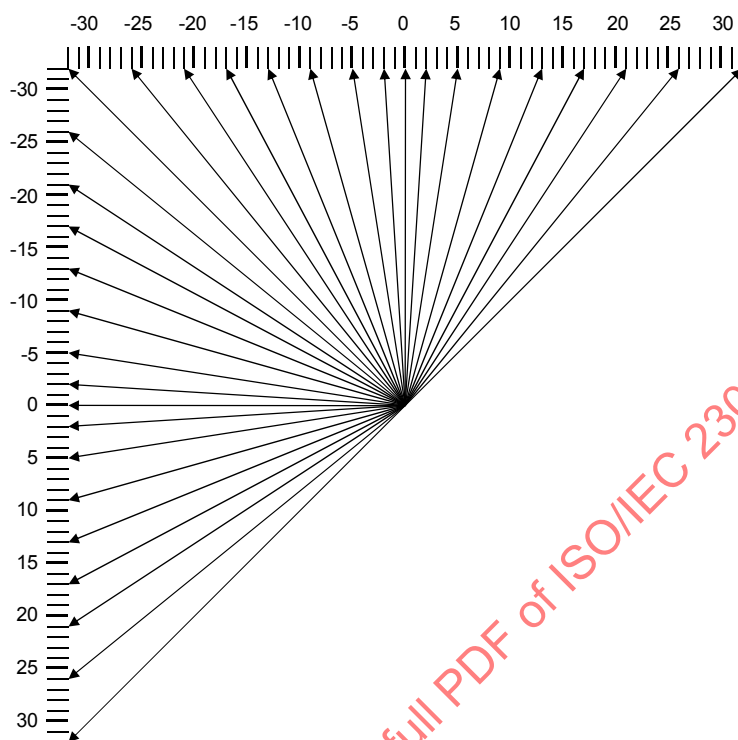


Figure 8-2 – Intra prediction angle definition (informative)

Table 8-4 – Specification of intraPredAngle

predModeIntra	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
intraPredAngle	-	32	26	21	17	13	9	5	2	0	-2	-5	-9	-13	-17	-21	-26
predModeIntra	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
intraPredAngle	-32	-26	-21	-17	-13	-9	-5	-2	0	2	5	9	13	17	21	26	32

Table 8-5 further specifies the mapping table between predModeIntra and the inverse angle parameter invAngle .

Table 8-5 – Specification of invAngle

predModeIntra	11	12	13	14	15	16	17	18
invAngle	-4096	-1638	-910	-630	-482	-390	-315	-256
predModeIntra	19	20	21	22	23	24	25	26
invAngle	-315	-390	-482	-630	-910	-1638	-4096	-

The values of the prediction samples $\text{predSamples}[x][y]$, with $x, y = 0..nTbS - 1$ are derived as follows:

- If predModeIntra is equal or greater than 18, the following ordered steps apply:
 1. The reference sample array $\text{ref}[x]$ is specified as follows:

- The following applies:

$$\text{ref}[x] = p[-1 + x][-1], \text{ with } x = 0..nTbS \quad (8-47)$$

- If intraPredAngle is less than 0, the main reference sample array is extended as follows:

- When $(nTbS * \text{intraPredAngle}) \gg 5$ is less than -1 ,

$$\text{ref}[x] = p[-1][-1 + ((x * \text{invAngle} + 128) \gg 8)], \\ \text{ with } x = -1..(nTbS * \text{intraPredAngle}) \gg 5 \quad (8-48)$$

- Otherwise,

$$\text{ref}[x] = p[-1 + x][-1], \text{ with } x = nTbS + 1..2 * nTbS \quad (8-49)$$

- The values of the prediction samples $\text{predSamples}[x][y]$, with $x, y = 0..nTbS - 1$ are derived as follows:

- The index variable $iIdx$ and the multiplication factor $iFact$ are derived as follows:

$$iIdx = ((y + 1) * \text{intraPredAngle}) \gg 5 \quad (8-50)$$

$$iFact = ((y + 1) * \text{intraPredAngle}) \& 31 \quad (8-51)$$

- Depending on the value of $iFact$, the following applies:

- If $iFact$ is not equal to 0, the value of the prediction samples $\text{predSamples}[x][y]$ is derived as follows:

$$\text{predSamples}[x][y] = \\ ((32 - iFact) * \text{ref}[x + iIdx + 1] + iFact * \text{ref}[x + iIdx + 2] + 16) \gg 5 \quad (8-52)$$

- Otherwise, the value of the prediction samples $\text{predSamples}[x][y]$ is derived as follows:

$$\text{predSamples}[x][y] = \text{ref}[x + iIdx + 1] \quad (8-53)$$

- When predModeIntra is equal to 26 (vertical), $cIdx$ is equal to 0 and $nTbS$ is less than 32, the following filtering applies with $x = 0, y = 0..nTbS - 1$:

$$\text{predSamples}[x][y] = \text{Clip1}_V(p[x][-1] + ((p[-1][y] - p[-1][-1]) \gg 1)) \quad (8-54)$$

- Otherwise (predModeIntra is less than 18), the following ordered steps apply:

- The reference sample array $\text{ref}[x]$ is specified as follows:

- The following applies:

$$\text{ref}[x] = p[-1][-1 + x], \text{ with } x = 0..nTbS \quad (8-55)$$

- If intraPredAngle is less than 0, the main reference sample array is extended as follows:

- When $(nTbS * \text{intraPredAngle}) \gg 5$ is less than -1 ,

$$\text{ref}[x] = p[-1 + ((x * \text{invAngle} + 128) \gg 8)][-1], \\ \text{ with } x = -1..(nTbS * \text{intraPredAngle}) \gg 5 \quad (8-56)$$

- Otherwise,

$$\text{ref}[x] = p[-1][-1 + x], \text{ with } x = nTbS + 1..2 * nTbS \quad (8-57)$$

- The values of the prediction samples $\text{predSamples}[x][y]$, with $x, y = 0..nTbS - 1$ are derived as follows:

- The index variable $iIdx$ and the multiplication factor $iFact$ are derived as follows:

$$iIdx = ((x + 1) * \text{intraPredAngle}) \gg 5 \quad (8-58)$$

$$iFact = ((x + 1) * \text{intraPredAngle}) \& 31 \quad (8-59)$$

- Depending on the value of $iFact$, the following applies:

- If $iFact$ is not equal to 0, the value of the prediction samples $\text{predSamples}[x][y]$ is derived as follows:

$$\text{predSamples}[x][y] = \\ ((32 - iFact) * \text{ref}[y + iIdx + 1] + iFact * \text{ref}[y + iIdx + 2] + 16) \gg 5 \quad (8-60)$$

- Otherwise, the value of the prediction samples $\text{predSamples}[x][y]$ is derived as follows:

$$\text{predSamples}[x][y] = \text{ref}[y + iIdx + 1] \quad (8-61)$$

- c. When predModeIntra is equal to 10 (horizontal), cIdx is equal to 0 and nTbS is less than 32, the following filtering applies with $x = 0..nTbS - 1$, $y = 0$:

$$\text{predSamples}[x][y] = \text{Clip1}_Y(p[-1][y] + ((p[x][-1] - p[-1][-1]) >> 1)) \quad (8-62)$$

8.5 Decoding process for coding units coded in inter prediction mode

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

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a variable $\log2CbSize$ specifying the size of the current coding block.

Output of this process is a modified reconstructed picture before deblocking filtering.

The derivation process for quantization parameters as specified in subclause 8.6.1 is invoked with the luma location (x_{Cb} , y_{Cb}) as input.

The variable $nCbS_L$ is set equal to $1 \ll \log2CbSize$ and the variable $nCbS_C$ is set equal to $1 \ll (\log2CbSize - 1)$.

The decoding process for coding units coded in inter prediction mode consists of following ordered steps:

1. The inter prediction process as specified in subclause 8.5.2 is invoked with the luma location (x_{Cb} , y_{Cb}) and the luma coding block size $\log2CbSize$ as inputs, and the outputs are three arrays predSamples_L , predSamples_{Cb} , and predSamples_{Cr} .
2. The decoding process for the residual signal of coding units coded in inter prediction mode specified in subclause 8.5.4 is invoked with the luma location (x_{Cb} , y_{Cb}) and the luma coding block size $\log2CbSize$ as inputs, and the outputs are three arrays resSamples_L , resSamples_{Cb} , and resSamples_{Cr} .
3. The reconstructed samples of the current coding unit are derived as follows:
 - The picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 8.6.5 is invoked with the luma coding block location (x_{Cb} , y_{Cb}), the variable $nCurrS$ set equal to $nCbS_L$, the variable $cIdx$ set equal to 0, the $(nCbS_L) \times (nCbS_L)$ array predSamples set equal to predSamples_L , and the $(nCbS_L) \times (nCbS_L)$ array resSamples set equal to resSamples_L as inputs.
 - The picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 8.6.5 is invoked with the chroma coding block location ($x_{Cb} / 2$, $y_{Cb} / 2$), the variable $nCurrS$ set equal to $nCbS_C$, the variable $cIdx$ set equal to 1, the $(nCbS_C) \times (nCbS_C)$ array predSamples set equal to predSamples_{Cb} , and the $(nCbS_C) \times (nCbS_C)$ array resSamples set equal to resSamples_{Cb} as inputs.
 - The picture reconstruction process prior to in-loop filtering for a colour component as specified in subclause 8.6.5 is invoked with the chroma coding block location ($x_{Cb} / 2$, $y_{Cb} / 2$), the variable $nCurrS$ set equal to $nCbS_C$, the variable $cIdx$ set equal to 2, the $(nCbS_C) \times (nCbS_C)$ array predSamples set equal to predSamples_{Cr} , and the $(nCbS_C) \times (nCbS_C)$ array resSamples set equal to resSamples_{Cr} as inputs.

8.5.2 Inter prediction process

This process is invoked when decoding coding unit whose $\text{CuPredMode}[x_{Cb}][y_{Cb}]$ is not equal to MODE_INTRA .

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a variable $\log2CbSize$ specifying the size of the current luma coding block.

Outputs of this process are:

- an $(nCbS_L) \times (nCbS_L)$ array predSamples_L of luma prediction samples, where $nCbS_L$ is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array predSamples_{Cb} of chroma prediction samples for the component Cb, where $nCbS_C$ is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array predSamples_{Cr} of chroma prediction samples for the component Cr, where $nCbS_C$ is derived as specified below.

The variable $nCbS_L$ is set equal to $1 \ll \log2CbSize$ and the variable $nCbS_C$ is set equal to $nCbS_L >> 1$.

The variable $nCbS_{1L}$ is set equal to $nCbS_L \gg 1$.

Depending on the value of PartMode, the following applies:

- If PartMode is equal to PART_2Nx2N, the decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L$, and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if PartMode is equal to PART_2NxN, the following ordered steps apply:
 1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L \gg 1$, and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, $nCbS_L \gg 1$), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L \gg 1$, and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and the two modified $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if PartMode is equal to PART_Nx2N, the following ordered steps apply:
 1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L \gg 1$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L$, and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to ($nCbS_L \gg 1$, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L \gg 1$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L$, and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and the two modified $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if PartMode is equal to PART_2NxN_U, the following ordered steps apply:
 1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to $nCbS_L \gg 2$, and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, $nCbS_L \gg 2$), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to ($nCbS_L \gg 1$) + ($nCbS_L \gg 2$), and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and the two modified $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if PartMode is equal to PART_2NxN_D, the following ordered steps apply:
 1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (xCb, yCb), the luma location (xBI, yBI) set equal to (0, 0), the size of the luma coding block $nCbS_L$, the width of the luma prediction block $nPbW$ set equal to $nCbS_L$, the height of the luma prediction block $nPbH$ set equal to ($nCbS_L \gg 1$) + ($nCbS_L \gg 2$), and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.

2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to (0, ($n_{CbS_L} \gg 1$) + ($n_{CbS_L} \gg 2$)), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to n_{CbS_L} , the height of the luma prediction block n_{PbH} set equal to $n_{CbS_L} \gg 2$, and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and the two modified (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if $PartMode$ is equal to $PART_nLx2N$, the following ordered steps apply:
1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to (0, 0), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 2$, the height of the luma prediction block n_{PbH} set equal to n_{CbS_L} , and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and two (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to ($n_{CbS_L} \gg 2$, 0), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to ($n_{CbS_L} \gg 1$) + ($n_{CbS_L} \gg 2$), the height of the luma prediction block n_{PbH} set equal to n_{CbS_L} , and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and the two modified (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise, if $PartMode$ is equal to $PART_nRx2N$, the following ordered steps apply:
1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to (0, 0), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to ($n_{CbS_L} \gg 1$) + ($n_{CbS_L} \gg 2$), the height of the luma prediction block n_{PbH} set equal to n_{CbS_L} , and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and two (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to ($n_{CbS_L} + (n_{CbS_L} \gg 2), 0$), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 2$, the height of the luma prediction block n_{PbH} set equal to n_{CbS_L} , and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and the two modified (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
- Otherwise ($PartMode$ is equal to $PART_NxN$), the following ordered steps apply:
1. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to (0, 0), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 1$, the height of the luma prediction block n_{PbH} set equal to $n_{CbS_L} \gg 1$, and a partition index $partIdx$ set equal to 0 as inputs, and the outputs are an (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and two (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 2. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to ($n_{CbS_L} \gg 1$, 0), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 1$, the height of the luma prediction block n_{PbH} set equal to $n_{CbS_L} \gg 1$, and a partition index $partIdx$ set equal to 1 as inputs, and the outputs are the modified (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and the two modified (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 3. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to (0, $n_{CbS_L} \gg 1$), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 1$, the height of the luma prediction block n_{PbH} set equal to $n_{CbS_L} \gg 1$, and a partition index $partIdx$ set equal to 2 as inputs, and the outputs are the modified (n_{CbS_L}) $\times$ (n_{CbS_L}) array $predSamples_L$ and the two modified (n_{CbS_C}) $\times$ (n_{CbS_C}) arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.
 4. The decoding process for prediction units in inter prediction mode as specified in subclause 8.5.3 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Bl} , y_{Bl}) set equal to ($n_{CbS_L} \gg 1$, $n_{CbS_L} \gg 1$), the size of the luma coding block n_{CbS_L} , the width of the luma prediction block n_{PbW} set equal to $n_{CbS_L} \gg 1$, the height of the luma prediction block n_{PbH} set equal to $n_{CbS_L} \gg 1$, and a partition index

partIdx set equal to 3 as inputs, and the outputs are the modified $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ and the two modified $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cb}$ and $predSamples_{Cr}$.

8.5.3 Decoding process for prediction units in inter prediction mode

8.5.3.1 General

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{Bl} , y_{Bl}) specifying the top-left sample of the current luma prediction block relative to the top-left sample of the current luma coding block,
- a variable $nCbS$ specifying the size of the current luma coding block,
- a variable $nPbW$ specifying the width of the current luma prediction block,
- a variable $nPbH$ specifying the height of the current luma prediction block,
- a variable $partIdx$ specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are:

- an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ of luma prediction samples, where nCS_L is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array $predSamples_{Cb}$ of chroma prediction samples for the component Cb, where nCS_C is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array $predSamples_{Cr}$ of chroma prediction samples for the component Cr, where nCS_C is derived as specified below.

The variable $nCbS_L$ is set equal to $nCbS$ and the variable $nCbS_C$ is set equal to $nCbS \gg 1$.

The decoding process for prediction units in inter prediction mode consists of the following ordered steps:

1. The derivation process for motion vector components and reference indices as specified in subclause 8.5.3.2 is invoked with the luma coding block location (x_{Cb} , y_{Cb}), the luma prediction block location (x_{Bl} , y_{Bl}), the luma coding block size block $nCbS$, the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, and the prediction unit index $partIdx$ as inputs, and the luma motion vectors $mvL0$ and $mvL1$, the chroma motion vectors $mvCL0$ and $mvCL1$, the reference indices $refIdxL0$ and $refIdxL1$, and the prediction list utilization flags $predFlagL0$ and $predFlagL1$ as outputs.
2. The decoding process for inter sample prediction as specified in subclause 8.5.3.3 is invoked with the luma coding block location (x_{Cb} , y_{Cb}), the luma prediction block location (x_{Bl} , y_{Bl}), the luma coding block size block $nCbS$, the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, the luma motion vectors $mvL0$ and $mvL1$, the chroma motion vectors $mvCL0$ and $mvCL1$, the reference indices $refIdxL0$ and $refIdxL1$, and the prediction list utilization flags $predFlagL0$ and $predFlagL1$ as inputs, and the inter prediction samples ($predSamples$) that are an $(nCbS_L) \times (nCbS_L)$ array $predSamples_L$ of prediction luma samples and two $(nCbS_C) \times (nCbS_C)$ arrays $predSamples_{Cr}$ and $predSamples_{Cb}$ of prediction chroma samples, one for each of the chroma components Cb and Cr, as outputs.

For use in derivation processes of variables invoked later in the decoding process, the following assignments are made for $x = x_{Bl}..x_{Bl} + nPbW - 1$ and $y = y_{Bl}..y_{Bl} + nPbH - 1$:

$$MvL0[x_{Cb} + x][y_{Cb} + y] = mvL0 \quad (8-63)$$

$$MvL1[x_{Cb} + x][y_{Cb} + y] = mvL1 \quad (8-64)$$

$$RefIdxL0[x_{Cb} + x][y_{Cb} + y] = refIdxL0 \quad (8-65)$$

$$RefIdxL1[x_{Cb} + x][y_{Cb} + y] = refIdxL1 \quad (8-66)$$

$$PredFlagL0[x_{Cb} + x][y_{Cb} + y] = predFlagL0 \quad (8-67)$$

$$PredFlagL1[x_{Cb} + x][y_{Cb} + y] = predFlagL1 \quad (8-68)$$

8.5.3.2 Derivation process for motion vector components and reference indices

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) of the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{Bl} , y_{Bl}) of the top-left sample of the current luma prediction block relative to the top-left sample of the current luma coding block,
- a variable n_{CbS} specifying the size of the current luma coding block,
- two variables n_{PbW} and n_{PbH} specifying the width and the height of the luma prediction block,
- a variable $partIdx$ specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are:

- the luma motion vectors $mvL0$ and $mvL1$,
- the chroma motion vectors $mvCL0$ and $mvCL1$,
- the reference indices $refIdxL0$ and $refIdxL1$,
- the prediction list utilization flags $predFlagL0$ and $predFlagL1$.

Let (x_{Pb} , y_{Pb}) specify the top-left sample location of the current luma prediction block relative to the top-left luma sample of the current picture where $x_{Pb} = x_{Cb} + x_{Bl}$ and $y_{Pb} = y_{Cb} + y_{Bl}$.

Let the variable $currPic$ and $ListX$ be the current picture and $RefPicListX$, with X being 0 or 1, of the current picture, respectively.

The function $LongTermRefPic(aPic, aPb, refIdx, LX)$, with X being 0 or 1, is defined as follows:

- If the picture with index $refIdx$ from reference picture list LX of the slice containing prediction block aPb in the picture $aPic$ was marked as "used for long term reference" at the time when $aPic$ was the current picture, $LongTermRefPic(aPic, aPb, refIdx, LX)$ is equal to 1.
- Otherwise, $LongTermRefPic(aPic, aPb, refIdx, LX)$ is equal to 0.

For the derivation of the variables $mvL0$ and $mvL1$, $refIdxL0$ and $refIdxL1$, as well as $predFlagL0$ and $predFlagL1$, the following applies:

- If $merge_flag[x_{Pb}][y_{Pb}]$ is equal to 1, the derivation process for luma motion vectors for merge mode as specified in subclause 8.5.3.2.1 is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{Pb} , y_{Pb}), the variables n_{CbS} , n_{PbW} , n_{PbH} , and the partition index $partIdx$ as inputs, and the output being the luma motion vectors $mvL0$, $mvL1$, the reference indices $refIdxL0$, $refIdxL1$, and the prediction list utilization flags $predFlagL0$ and $predFlagL1$.
- Otherwise, for X being replaced by either 0 or 1 in the variables $predFlagLX$, $mvLX$, and $refIdxLX$, in $PRED_LX$, and in the syntax elements ref_idx_IX and $MvdLX$, the following applies:

1. The variables $refIdxLX$ and $predFlagLX$ are derived as follows:

- If $inter_pred_idc[x_{Pb}][y_{Pb}]$ is equal to $PRED_LX$ or $PRED_BI$,

$$refIdxLX = ref_idx_IX[x_{Pb}][y_{Pb}] \quad (8-69)$$

$$predFlagLX = 1 \quad (8-70)$$

Otherwise, the variables $refIdxLX$ and $predFlagLX$ are specified by:

$$refIdxLX = -1 \quad (8-71)$$

$$predFlagLX = 0 \quad (8-72)$$

2. The variable $mvdLX$ is derived as follows:

$$mvdLX[0] = MvdLX[x_{Pb}][y_{Pb}][0] \quad (8-73)$$

$$mvdLX[1] = MvdLX[x_{Pb}][y_{Pb}][1] \quad (8-74)$$

3. When $predFlagLX$ is equal to 1, the derivation process for luma motion vector prediction in subclause 8.5.3.2.5 is invoked with the luma coding block location (x_{Cb} , y_{Cb}), the coding block size n_{CbS} , the luma prediction block location (x_{Pb} , y_{Pb}), the variables n_{PbW} , n_{PbH} , $refIdxLX$, and the partition index $partIdx$ as inputs, and the output being $mvpLX$.
4. When $predFlagLX$ is equal to 1, the luma motion vector $mvLX$ is derived as follows:

$$uLX[0] = (mvLX[0] + mvdLX[0] + 2^{16}) \% 2^{16} \quad (8-75)$$

$$mvLX[0] = (uLX[0] \geq 2^{15}) ? (uLX[0] - 2^{16}) : uLX[0] \quad (8-76)$$

$$uLX[1] = (mvLX[1] + mvdLX[1] + 2^{16}) \% 2^{16} \quad (8-77)$$

$$mvLX[1] = (uLX[1] \geq 2^{15}) ? (uLX[1] - 2^{16}) : uLX[1] \quad (8-78)$$

NOTE – The resulting values of $mvLX[0]$ and $mvLX[1]$ as specified above will always be in the range of -2^{15} to $2^{15} - 1$, inclusive.

When $ChromaArrayType$ is not equal to 0 and $predFlagLX$, with X being 0 or 1, is equal to 1, the derivation process for chroma motion vectors in subclause 8.5.3.2.9 is invoked with $mvLX$ as input, and the output being $mvCLX$.

8.5.3.2.1 Derivation process for luma motion vectors for merge mode

This process is only invoked when $merge_flag[xPb][yPb]$ is equal to 1, where (xPb, yPb) specify the top-left sample of the current luma prediction 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 luma location (xPb, yPb) of the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- a variable $nCbS$ specifying the size of the current luma coding block,
- two variables $nPbW$ and $nPbH$ specifying the width and the height of the luma prediction block,
- a variable $partIdx$ specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are:

- the luma motion vectors $mvL0$ and $mvL1$,
- the reference indices $refIdxL0$ and $refIdxL1$,
- the prediction list utilization flags $predFlagL0$ and $predFlagL1$.

The location $(xOrigP, yOrigP)$ and the variables $nOrigPbW$ and $nOrigPbH$ are derived to store the values of (xPb, yPb) , $nPbW$, and $nPbH$ as follows:

$$(xOrigP, yOrigP) \text{ is set equal to } (xPb, yPb) \quad (8-79)$$

$$nOrigPbW = nPbW \quad (8-80)$$

$$nOrigPbH = nPbH \quad (8-81)$$

When $Log2ParMrgLevel$ is greater than 2 and $nCbS$ is equal to 8, (xPb, yPb) , $nPbW$, $nPbH$, and $partIdx$ are modified as follows:

$$(xPb, yPb) = (xCb, yCb) \quad (8-82)$$

$$nPbW = nCbS \quad (8-83)$$

$$nPbH = nCbS \quad (8-84)$$

$$partIdx = 0 \quad (8-85)$$

NOTE – When $Log2ParMrgLevel$ is greater than 2 and $nCbS$ is equal to 8, all the prediction units of the current coding unit share a single merge candidate list, which is identical to the merge candidate list of the $2N \times 2N$ prediction unit.

The motion vectors $mvL0$ and $mvL1$, the reference indices $refIdxL0$ and $refIdxL1$, and the prediction utilization flags $predFlagL0$ and $predFlagL1$ are derived by the following ordered steps:

1. The derivation process for merging candidates from neighbouring prediction unit partitions in subclause 8.5.3.2.2 is invoked with the luma coding block location (xCb, yCb) , the coding block size $nCbS$, the luma prediction block location (xPb, yPb) , the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, and the partition index $partIdx$ as inputs, and the output being the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$, and $availableFlagB_2$, the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$, and $refIdxLXB_2$, the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$, and $predFlagLXB_2$, and the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$, and $mvLXB_2$, with X being 0 or 1.

2. The reference indices for the temporal merging candidate, refIdxLXCol , with X being 0 or 1, are set equal to 0.
3. The derivation process for temporal luma motion vector prediction in subclause 8.5.3.2.7 is invoked with the luma location (x_{Pb} , y_{Pb}), the luma prediction block width n_{PbW} , the luma prediction block height n_{PbH} , and the variable refIdxL0Col as inputs, and the output being the availability flag $\text{availableFlagL0Col}$ and the temporal motion vector mvL0Col . The variables availableFlagCol , predFlagL0Col and predFlagL1Col are derived as follows:

$$\text{availableFlagCol} = \text{availableFlagL0Col} \quad (8-86)$$

$$\text{predFlagL0Col} = \text{availableFlagL0Col} \quad (8-87)$$

$$\text{predFlagL1Col} = 0 \quad (8-88)$$

4. When slice_type is equal to B, the derivation process for temporal luma motion vector prediction in subclause 8.5.3.2.7 is invoked with the luma location (x_{Pb} , y_{Pb}), the luma prediction block width n_{PbW} , the luma prediction block height n_{PbH} , and the variable refIdxL1Col as inputs, and the output being the availability flag $\text{availableFlagL1Col}$ and the temporal motion vector mvL1Col . The variables availableFlagCol and predFlagL1Col are derived as follows:

$$\text{availableFlagCol} = \text{availableFlagL0Col} \mid \mid \text{availableFlagL1Col} \quad (8-89)$$

$$\text{predFlagL1Col} = \text{availableFlagL1Col} \quad (8-90)$$

5. The merging candidate list, mergeCandList , is constructed as follows:

```

i = 0
if( availableFlagA1 )
    mergeCandList[ i++ ] = A1
if( availableFlagB1 )
    mergeCandList[ i++ ] = B1
if( availableFlagB0 )
    mergeCandList[ i++ ] = B0
if( availableFlagA0 )
    mergeCandList[ i++ ] = A0
if( availableFlagB2 )
    mergeCandList[ i++ ] = B2
if( availableFlagCol )
    mergeCandList[ i++ ] = Col

```

(8-91)

6. The variable numCurrMergeCand and numOrigMergeCand are set equal to the number of merging candidates in the mergeCandList .
7. When slice_type is equal to B, the derivation process for combined bi-predictive merging candidates specified in subclause 8.5.3.2.3 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 , numCurrMergeCand , and numOrigMergeCand as inputs, and the output is assigned to mergeCandList , numCurrMergeCand , the reference indices $\text{refIdxL0combCand}_k$ and $\text{refIdxL1combCand}_k$, the prediction list utilization flags $\text{predFlagL0combCand}_k$ and $\text{predFlagL1combCand}_k$, and the motion vectors mvL0combCand_k and mvL1combCand_k of every new candidate combCand_k being added into mergeCandList . The number of candidates being added, numCombMergeCand , is set equal to ($\text{numCurrMergeCand} - \text{numOrigMergeCand}$). When numCombMergeCand is greater than 0, k ranges from 0 to $\text{numCombMergeCand} - 1$, inclusive.
8. The derivation process for zero motion vector merging candidates specified in subclause 8.5.3.2.4 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 $\text{refIdxL0zeroCand}_m$ and $\text{refIdxL1zeroCand}_m$, the prediction list utilization flags $\text{predFlagL0zeroCand}_m$ and $\text{predFlagL1zeroCand}_m$, and the motion vectors mvL0zeroCand_m and mvL1zeroCand_m of every new candidate zeroCand_m being added into mergeCandList . The number of candidates being added, numZeroMergeCand , is set equal to ($\text{numCurrMergeCand} - \text{numOrigMergeCand} - \text{numCombMergeCand}$). When numZeroMergeCand is greater than 0, m ranges from 0 to $\text{numZeroMergeCand} - 1$, inclusive.
9. The following assignments are made with N being the candidate at position $\text{merge_idx}[\text{xOrigP}][\text{yOrigP}]$ in the merging candidate list mergeCandList ($N = \text{mergeCandList}[\text{merge_idx}[\text{xOrigP}][\text{yOrigP}]]$) and X being replaced by 0 or 1:

$$mvLX[0] = mvLXN[0] \quad (8-92)$$

$$mvLX[1] = mvLXN[1] \quad (8-93)$$

$$refIdxLX = refIdxLXN \quad (8-94)$$

$$predFlagLX = predFlagLXN \quad (8-95)$$

10. When $predFlagL0$ is equal to 1 and $predFlagL1$ is equal to 1, and $(nOrigPbW + nOrigPbH)$ is equal to 12, the following applies:

$$refIdxL1 = -1 \quad (8-96)$$

$$predFlagL1 = 0 \quad (8-97)$$

8.5.3.2.2 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 $nCbS$ specifying the size of the current luma coding block,
- a luma location (xPb, yPb) specifying the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- two variables $nPbW$ and $nPbH$ specifying the width and the height of the luma prediction block,
- a variable $partIdx$ specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are as follows, with X being 0 or 1:

- the availability flags $availableFlagA_0$, $availableFlagA_1$, $availableFlagB_0$, $availableFlagB_1$, and $availableFlagB_2$ of the neighbouring prediction units,
- the reference indices $refIdxLXA_0$, $refIdxLXA_1$, $refIdxLXB_0$, $refIdxLXB_1$, and $refIdxLXB_2$ of the neighbouring prediction units,
- the prediction list utilization flags $predFlagLXA_0$, $predFlagLXA_1$, $predFlagLXB_0$, $predFlagLXB_1$, and $predFlagLXB_2$ of the neighbouring prediction units,
- the motion vectors $mvLXA_0$, $mvLXA_1$, $mvLXB_0$, $mvLXB_1$, and $mvLXB_2$ of the neighbouring prediction units.

For the derivation of $availableFlagA_1$, $refIdxLXA_1$, $predFlagLXA_1$, and $mvLXA_1$ the following applies:

- The luma location $(xNbA_1, yNbA_1)$ inside the neighbouring luma coding block is set equal to $(xPb - 1, yPb + nPbH - 1)$.
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb) , the current luma coding block size $nCbS$, the luma prediction block location (xPb, yPb) , the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, the luma location $(xNbA_1, yNbA_1)$, and the partition index $partIdx$ as inputs, and the output is assigned to the prediction block availability flag $availableA_1$.
- When one or more of the following conditions are true, $availableA_1$ is set equal to FALSE:
 - $xPb \gg \text{Log2ParMrgLevel}$ is equal to $xNbA_1 \gg \text{Log2ParMrgLevel}$ and $yPb \gg \text{Log2ParMrgLevel}$ is equal to $yNbA_1 \gg \text{Log2ParMrgLevel}$.
 - $PartMode$ of the current prediction unit is equal to $PART_Nx2N$, $PART_nLx2N$, or $PART_nRx2N$, and $partIdx$ is equal to 1.
- The variables $availableFlagA_1$, $refIdxLXA_1$, $predFlagLXA_1$, and $mvLXA_1$ are derived as follows:
 - If $availableA_1$ is equal to FALSE, $availableFlagA_1$ is set equal to 0, both components of $mvLXA_1$ are set equal to 0, $refIdxLXA_1$ is set equal to -1 and $predFlagLXA_1$ is set equal to 0, with X being 0 or 1.
 - Otherwise, $availableFlagA_1$ is set equal to 1 and the following assignments are made:

$$mvLXA_1 = MvLX[xNbA_1][yNbA_1] \quad (8-98)$$

$$refIdxLXA_1 = RefIdxLX[xNbA_1][yNbA_1] \quad (8-99)$$

$$predFlagLXA_1 = PredFlagLX[xNbA_1][yNbA_1] \quad (8-100)$$

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 (xPb + nPbW – 1, yPb – 1).
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb), the current luma coding block size nCbS, the luma prediction block location (xPb, yPb), the luma prediction block width nPbW, the luma prediction block height nPbH, the luma location (xNbB₁, yNbB₁), and the partition index partIdx as inputs, and the output is assigned to the prediction block availability flag availableB₁.
- When one or more of the following conditions are true, availableB₁ is set equal to FALSE:
 - xPb >> Log2ParMrgLevel is equal to xNbB₁ >> Log2ParMrgLevel and yPb >> Log2ParMrgLevel is equal to yNbB₁ >> Log2ParMrgLevel.
 - PartMode of the current prediction unit is equal to PART_2NxN, PART_2NxN_U, or PART_2NxN_D, and partIdx is equal to 1.
- 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:
 - availableB₁ is equal to FALSE.
 - availableA₁ is equal to TRUE and the prediction units covering 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:

$$mvLXB_1 = MvLX[xNbB_1][yNbB_1] \quad (8-101)$$

$$refIdxLXB_1 = RefIdxLX[xNbB_1][yNbB_1] \quad (8-102)$$

$$predFlagLXB_1 = PredFlagLX[xNbB_1][yNbB_1] \quad (8-103)$$

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 (xPb + nPbW, yPb – 1).
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb), the current luma coding block size nCbS, the luma prediction block location (xPb, yPb), the luma prediction block width nPbW, the luma prediction block height nPbH, the luma location (xNbB₀, yNbB₀), and the partition index partIdx as inputs, and the output is assigned to the prediction block availability flag availableB₀.
- When xPb >> Log2ParMrgLevel is equal to xNbB₀ >> Log2ParMrgLevel and yPb >> Log2ParMrgLevel is equal to yNbB₀ >> Log2ParMrgLevel, availableB₀ is set equal to FALSE.
- 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:
 - availableB₀ is equal to FALSE.
 - availableB₁ is equal to TRUE and the prediction units covering the luma locations (xNbB₁, yNbB₁) and (xNbB₀, yNbB₀) have the same motion vectors and the same reference indices.
 - Otherwise, availableFlagB₀ is set equal to 1 and the following assignments are made:

$$mvLXB_0 = MvLX[xNbB_0][yNbB_0] \quad (8-104)$$

$$refIdxLXB_0 = RefIdxLX[xNbB_0][yNbB_0] \quad (8-105)$$

$$predFlagLXB_0 = PredFlagLX[xNbB_0][yNbB_0] \quad (8-106)$$

For the derivation of availableFlagA₀, refIdxLXA₀, predFlagLXA₀, and mvLXA₀ the following applies:

- The luma location ($xNbA_0, yNbA_0$) inside the neighbouring luma coding block is set equal to ($xPb - 1, yPb + nPbH$).
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb), the current luma coding block size $nCbS$, the luma prediction block location (xPb, yPb), the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, the luma location ($xNbA_0, yNbA_0$), and the partition index $partIdx$ as inputs, and the output is assigned to the prediction block availability flag $availableA_0$.
- When $xPb \gg \text{Log2ParMrgLevel}$ is equal to $xNbA_0 \gg \text{Log2ParMrgLevel}$ and $yPb \gg \text{Log2ParMrgLevel}$ is equal to $yA_0 \gg \text{Log2ParMrgLevel}$, $availableA_0$ is set equal to FALSE.
- The variables $availableFlagA_0$, $refIdxLXA_0$, $predFlagLXA_0$, and $mvLXA_0$ are derived as follows:
 - If one or more of the following conditions are true, $availableFlagA_0$ is set equal to 0, both components of $mvLXA_0$ are set equal to 0, $refIdxLXA_0$ is set equal to -1 , and $predFlagLXA_0$ is set equal to 0, with X being 0 or 1:
 - $availableA_0$ is equal to FALSE.
 - $availableA_1$ is equal to TRUE and the prediction units covering the luma locations ($xNbA_1, yNbA_1$) and ($xNbA_0, yNbA_0$) have the same motion vectors and the same reference indices.
 - Otherwise, $availableFlagA_0$ is set equal to 1 and the following assignments are made:

$$mvLXA_0 = MvLX[xNbA_0][yNbA_0] \quad (8-107)$$

$$refIdxLXA_0 = RefIdxLX[xNbA_0][yNbA_0] \quad (8-108)$$

$$predFlagLXA_0 = PredFlagLX[xNbA_0][yNbA_0] \quad (8-109)$$

For the derivation of $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$, and $mvLXB_2$ the following applies:

- The luma location ($xNbB_2, yNbB_2$) inside the neighbouring luma coding block is set equal to ($xPb - 1, yPb - 1$).
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb), the current luma coding block size $nCbS$, the luma prediction block location (xPb, yPb), the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, the luma location ($xNbB_2, yNbB_2$), and the partition index $partIdx$ as inputs, and the output is assigned to the prediction block availability flag $availableB_2$.
- When $xPb \gg \text{Log2ParMrgLevel}$ is equal to $xNbB_2 \gg \text{Log2ParMrgLevel}$ and $yPb \gg \text{Log2ParMrgLevel}$ is equal to $yNbB_2 \gg \text{Log2ParMrgLevel}$, $availableB_2$ is set equal to FALSE.
- The variables $availableFlagB_2$, $refIdxLXB_2$, $predFlagLXB_2$, and $mvLXB_2$ are derived as follows:
 - If one or more of the following conditions are true, $availableFlagB_2$ is set equal to 0, both components of $mvLXB_2$ are set equal to 0, $refIdxLXB_2$ is set equal to -1 , and $predFlagLXB_2$ is set equal to 0, with X being 0 or 1:
 - $availableB_2$ is equal to FALSE.
 - $availableA_1$ is equal to TRUE and prediction units covering the luma locations ($xNbA_1, yNbA_1$) and ($xNbB_2, yNbB_2$) have the same motion vectors and the same reference indices.
 - $availableB_1$ is equal to TRUE and the prediction units covering the luma locations ($xNbB_1, yNbB_1$) and ($xNbB_2, yNbB_2$) have the same motion vectors and the same reference indices.
 - $availableFlagA_0 + availableFlagA_1 + availableFlagB_0 + availableFlagB_1$ is equal to 4.
 - Otherwise, $availableFlagB_2$ is set equal to 1, and the following assignments are made:

$$mvLXB_2 = MvLX[xNbB_2][yNbB_2] \quad (8-110)$$

$$refIdxLXB_2 = RefIdxLX[xNbB_2][yNbB_2] \quad (8-111)$$

$$predFlagLXB_2 = PredFlagLX[xNbB_2][yNbB_2] \quad (8-112)$$

8.5.3.2.3 Derivation process for combined bi-predictive 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,
- the number of elements numOrigMergeCand within the mergeCandList after the spatial and temporal merge candidate derivation process.

Outputs of this process are:

- the merging candidate list mergeCandList,
- the number of elements numCurrMergeCand within mergeCandList,
- the reference indices refIdxL0combCand_k and refIdxL1combCand_k of every new candidate combCand_k added into mergeCandList during the invocation of this process,
- the prediction list utilization flags predFlagL0combCand_k and predFlagL1combCand_k of every new candidate combCand_k added into mergeCandList during the invocation of this process,
- the motion vectors mvL0combCand_k and mvL1combCand_k of every new candidate combCand_k added into mergeCandList during the invocation of this process.

When numOrigMergeCand is greater than 1 and less than MaxNumMergeCand, the variable numInputMergeCand is set equal to numCurrMergeCand, the variable combIdx is set equal to 0, the variable combStop is set equal to FALSE, and the following steps are repeated until combStop is equal to TRUE:

1. The variables l0CandIdx and l1CandIdx are derived using combIdx as specified in Table 8-6.
2. The following assignments are made, with l0Cand being the candidate at position l0CandIdx and l1Cand being the candidate at position l1CandIdx in the merging candidate list mergeCandList:
 - l0Cand = mergeCandList[l0CandIdx]
 - l1Cand = mergeCandList[l1CandIdx]
3. When all of the following conditions are true:
 - predFlagL0l0Cand == 1
 - predFlagL1l1Cand == 1
 - (DiffPicOrderCnt(RefPicList0[refIdxL0l0Cand], RefPicList1[refIdxL1l1Cand]) != 0) || (mvL0l0Cand != mvL1l1Cand)

the candidate combCand_k with k equal to (numCurrMergeCand – numInputMergeCand) is added at the end of mergeCandList, i.e. mergeCandList[numCurrMergeCand] is set equal to combCand_k, and the reference indices, the prediction list utilization flags, and the motion vectors of combCand_k are derived as follows and numCurrMergeCand is incremented by 1:

$$\text{refIdxL0combCand}_k = \text{refIdxL0l0Cand} \quad (8-113)$$

$$\text{refIdxL1combCand}_k = \text{refIdxL1l1Cand} \quad (8-114)$$

$$\text{predFlagL0combCand}_k = 1 \quad (8-115)$$

$$\text{predFlagL1combCand}_k = 1 \quad (8-116)$$

$$\text{mvL0combCand}_k[0] = \text{mvL0l0Cand}[0] \quad (8-117)$$

$$\text{mvL0combCand}_k[1] = \text{mvL0l0Cand}[1] \quad (8-118)$$

$$\text{mvL1combCand}_k[0] = \text{mvL1l1Cand}[0] \quad (8-119)$$

$$\text{mvL1combCand}_k[1] = \text{mvL1l1Cand}[1] \quad (8-120)$$

$$\text{numCurrMergeCand} = \text{numCurrMergeCand} + 1 \quad (8-121)$$

4. The variable combIdx is incremented by 1.
5. When combIdx is equal to (numOrigMergeCand * (numOrigMergeCand – 1)) or numCurrMergeCand is equal to MaxNumMergeCand, combStop is set equal to TRUE.

Table 8-6 – Specification of l0CandIdx and l1CandIdx

combIdx	0	1	2	3	4	5	6	7	8	9	10	11
l0CandIdx	0	1	0	2	1	2	0	3	1	3	2	3
l1CandIdx	1	0	2	0	2	1	3	0	3	1	3	2

8.5.3.2.4 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 refIdxL0zeroCand_m and refIdxL1zeroCand_m of every new candidate zeroCand_m added into mergeCandList during the invocation of this process,
- the prediction list utilization flags predFlagL0zeroCand_m and predFlagL1zeroCand_m of every new candidate zeroCand_m added into mergeCandList during the invocation of this process,
- the motion vectors mvL0zeroCand_m and mvL1zeroCand_m of every new candidate zeroCand_m 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 num_ref_idx_l0_active_minus1 + 1.
- Otherwise (slice_type is equal to B), numRefIdx is set equal to Min(num_ref_idx_l0_active_minus1 + 1, num_ref_idx_l1_active_minus1 + 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 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 zeroCand_m with m equal to (numCurrMergeCand – numInputMergeCand) is added at the end of mergeCandList, i.e. mergeCandList[numCurrMergeCand] is set equal to zeroCand_m, and the reference indices, the prediction list utilization flags, and the motion vectors of zeroCand_m are derived as follows and numCurrMergeCand is incremented by 1:

$$\text{refIdxL0zeroCand}_m = (\text{zeroIdx} < \text{numRefIdx}) ? \text{zeroIdx} : 0 \quad (8-122)$$

$$\text{refIdxL1zeroCand}_m = -1 \quad (8-123)$$

$$\text{predFlagL0zeroCand}_m = 1 \quad (8-124)$$

$$\text{predFlagL1zeroCand}_m = 0 \quad (8-125)$$

$$\text{mvL0zeroCand}_m[0] = 0 \quad (8-126)$$

$$\text{mvL0zeroCand}_m[1] = 0 \quad (8-127)$$

$$\text{mvL1zeroCand}_m[0] = 0 \quad (8-128)$$

$$\text{mvL1zeroCand}_m[1] = 0 \quad (8-129)$$

$$\text{numCurrMergeCand} = \text{numCurrMergeCand} + 1 \quad (8-130)$$

- Otherwise (slice_type is equal to B), the candidate zeroCand_m with m equal to (numCurrMergeCand – numInputMergeCand) is added at the end of mergeCandList, i.e. mergeCandList[numCurrMergeCand] is set equal to zeroCand_m, and the reference indices, the prediction list utilization flags, and the motion vectors of zeroCand_m are derived as follows and numCurrMergeCand is incremented by 1:

$$\text{refIdxL0zeroCand}_m = (\text{zeroIdx} < \text{numRefIdx}) ? \text{zeroIdx} : 0 \quad (8-131)$$

$$\text{refIdxL1zeroCand}_m = (\text{zeroIdx} < \text{numRefIdx}) ? \text{zeroIdx} : 0 \quad (8-132)$$

$$\text{predFlagL0zeroCand}_m = 1 \quad (8-133)$$

$$\text{predFlagL1zeroCand}_m = 1 \quad (8-134)$$

$$\text{mvL0zeroCand}_m[0] = 0 \quad (8-135)$$

$$\text{mvL0zeroCand}_m[1] = 0 \quad (8-136)$$

$$\text{mvL1zeroCand}_m[0] = 0 \quad (8-137)$$

$$\text{mvL1zeroCand}_m[1] = 0 \quad (8-138)$$

$$\text{numCurrMergeCand} = \text{numCurrMergeCand} + 1 \quad (8-139)$$

2. The variable zeroIdx is incremented by 1.

8.5.3.2.5 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 nCbS specifying the size of the current luma coding block,
- a luma location (xPb, yPb) specifying the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- two variables nPbW and nPbH specifying the width and the height of the luma prediction block,
- the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,
- a variable partIdx specifying the index of the current prediction unit within the current coding unit.

Output of this process is the prediction mvpLX of the motion vector mvLX, with X being 0 or 1.

The motion vector predictor mvpLX is derived in the following ordered steps:

1. The derivation process for motion vector predictor candidates from neighbouring prediction unit partitions in subclause 8.5.3.2.6 is invoked with the luma coding block location (xCb, yCb), the coding block size nCbS, the luma prediction block location (xPb, yPb), the luma prediction block width nPbW, the luma prediction block height nPbH, refIdxLX, with X being 0 or 1, and the partition index partIdx as inputs, and the availability flags availableFlagLXN and the motion vectors mvLXN, with N being replaced by A or B, as output.
2. If both availableFlagLXA and availableFlagLXB are equal to 1 and mvLXA is not equal to mvLXB, availableFlagLXCol is set equal to 0. Otherwise, the derivation process for temporal luma motion vector prediction in subclause 8.5.3.2.7 is invoked with luma prediction block location (xPb, yPb), the luma prediction block width nPbW, the luma prediction block height nPbH, 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.
3. The motion vector predictor candidate list, mvpListLX, is constructed as follows:

$$\begin{aligned}
 &i = 0 \\
 &\text{if(availableFlagLXA)} \\
 &\quad \text{mvpListLX}[i++] = \text{mvLXA} \\
 &\text{if(availableFlagLXB)} \\
 &\quad \text{mvpListLX}[i++] = \text{mvLXB} \\
 &\text{if(availableFlagLXCol)} \\
 &\quad \text{mvpListLX}[i++] = \text{mvLXCol}
 \end{aligned} \quad (8-140)$$

4. The motion vector predictor list is modified as follows:

- When mvLXA and mvLXB have the same value, mvLXB is removed from the list and the variable numMvpCandLX is set equal to the number of elements within the mvpListLX.
- When numMvpCandLX is less than 2, the following applies repeatedly until numMvpCandLX is equal to 2:

$$\text{mvpListLX}[\text{numMvpCandLX}][0] = 0 \quad (8-141)$$

$$\text{mvpListLX}[\text{numMvpCandLX}][1] = 0 \quad (8-142)$$

$$\text{numMvpCandLX} = \text{numMvpCandLX} + 1 \quad (8-143)$$

- When numMvpCandLX is greater than 2, all motion vector predictor candidates mvpListLX[idx] with idx greater than 1 are removed from the list.

5. The motion vector of mvpListLX[mvp_lx_flag[xPb][yPb]] is assigned to mvPLX.

8.5.3.2.6 Derivation process for 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 nCbS specifying the size of the current luma coding block,
- a luma location (xPb, yPb) specifying the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- two variables nPbW and nPbH specifying the width and the height of the luma prediction block,
- the reference index of the current prediction unit partition refIdxLX, with X being 0 or 1,
- a variable partIdx specifying the index of the current prediction unit within the current coding unit.

Outputs of this process are (with N being replaced by A or B):

- the motion vectors mvLXN of the neighbouring prediction units,
- the availability flags availableFlagLXN of the neighbouring prediction units.

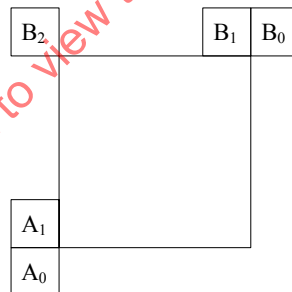


Figure 8-3 – Spatial motion vector neighbours (informative)

The variable currPb specifies the current luma prediction block at luma location (xPb, yPb) 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 (xPb – 1, yPb + nPbH) 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 prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb), the current luma coding block size nCbS, the luma prediction block location (xPb, yPb), the luma prediction block width nPbW, the luma prediction block height nPbH, the luma location (xNbY, yNbY) set equal to (xNbA₀, yNbA₀), and the partition index partIdx as inputs, and the output is assigned to the prediction block availability flag availableA₀.

4. The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (x_{Cb} , y_{Cb}), the current luma coding block size n_{CbS} , the luma prediction block location (x_{Pb} , y_{Pb}), the luma prediction block width n_{PbW} , the luma prediction block height n_{PbH} , the luma location (x_{NbY} , y_{NbY}) set equal to (x_{NbA_1} , y_{NbA_1}), and the partition index $partIdx$ as inputs, and the output is assigned to the prediction block availability flag $availableA_1$.
5. When $availableA_0$ or $availableA_1$ is equal to TRUE, the variable $isScaledFlagLX$ is set equal to 1.
6. The following applies for (x_{NbA_k} , y_{NbA_k}) from (x_{NbA_0} , y_{NbA_0}) to (x_{NbA_1} , y_{NbA_1}):

– When $availableA_k$ is equal to TRUE and $availableFlagLXA$ is equal to 0, the following applies:

- If $PredFlagLX[x_{NbA_k}][y_{NbA_k}]$ is equal to 1 and $DiffPicOrderCnt(RefPicListX[RefIdxLX[x_{NbA_k}][y_{NbA_k}]], RefPicListX[refIdxLX])$ is equal to 0, $availableFlagLXA$ is set equal to 1 and the following applies:

$$mvLXA = MvLX[x_{NbA_k}][y_{NbA_k}] \quad (8-144)$$

- Otherwise, when $PredFlagLY[x_{NbA_k}][y_{NbA_k}]$ (with $Y = !X$) is equal to 1 and $DiffPicOrderCnt(RefPicListY[RefIdxLY[x_{NbA_k}][y_{NbA_k}]], RefPicListX[refIdxLX])$ is equal to 0, $availableFlagLXA$ is set equal to 1 and the following applies:

$$mvLXA = MvLY[x_{NbA_k}][y_{NbA_k}] \quad (8-145)$$

7. When $availableFlagLXA$ is equal to 0, the following applies for (x_{NbA_k} , y_{NbA_k}) from (x_{NbA_0} , y_{NbA_0}) to (x_{NbA_1} , y_{NbA_1}) or until $availableFlagLXA$ is equal to 1:

– When $availableA_k$ is equal to TRUE and $availableFlagLXA$ is equal to 0, the following applies:

- If $PredFlagLX[x_{NbA_k}][y_{NbA_k}]$ is equal to 1 and $LongTermRefPic(currPic, currPb, refIdxLX, RefPicListX)$ is equal to $LongTermRefPic(currPic, currPb, RefIdxLX[x_{NbA_k}][y_{NbA_k}], RefPicListX)$, $availableFlagLXA$ is set equal to 1 and the following assignments are made:

$$mvLXA = MvLX[x_{NbA_k}][y_{NbA_k}] \quad (8-146)$$

$$refIdxA = RefIdxLX[x_{NbA_k}][y_{NbA_k}] \quad (8-147)$$

$$refPicListA = RefPicListX \quad (8-148)$$

- Otherwise, when $PredFlagLY[x_{NbA_k}][y_{NbA_k}]$ (with $Y = !X$) is equal to 1 and $LongTermRefPic(currPic, currPb, refIdxLX, RefPicListX)$ is equal to $LongTermRefPic(currPic, currPb, RefIdxLY[x_{NbA_k}][y_{NbA_k}], RefPicListY)$, $availableFlagLXA$ is set equal to 1 and the following assignments are made:

$$mvLXA = MvLY[x_{NbA_k}][y_{NbA_k}] \quad (8-149)$$

$$refIdxA = RefIdxLY[x_{NbA_k}][y_{NbA_k}] \quad (8-150)$$

$$refPicListA = RefPicListY \quad (8-151)$$

- When $availableFlagLXA$ is equal to 1, $DiffPicOrderCnt(refPicListA[refIdxA], RefPicListX[refIdxLX])$ is not equal to 0, and both $refPicListA[refIdxA]$ and $RefPicListX[refIdxLX]$ are short-term reference pictures, $mvLXA$ is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8-152)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8-153)$$

$$mvLXA = Clip3(-32768, 32767, Sign(distScaleFactor * mvLXA) * ((Abs(distScaleFactor * mvLXA) + 127) >> 8)) \quad (8-154)$$

where td and tb are derived as follows:

$$td = Clip3(-128, 127, DiffPicOrderCnt(currPic, refPicListA[refIdxA])) \quad (8-155)$$

$$tb = Clip3(-128, 127, DiffPicOrderCnt(currPic, RefPicListX[refIdxLX])) \quad (8-156)$$

The motion vector $mvLXB$ and the availability flag $availableFlagLXB$ are derived in the following ordered steps:

1. The sample locations (x_{NbB_0} , y_{NbB_0}), (x_{NbB_1} , y_{NbB_1}), and (x_{NbB_2} , y_{NbB_2}) are set equal to ($x_{Pb} + n_{PbW}$, $y_{Pb} - 1$), ($x_{Pb} + n_{PbW} - 1$, $y_{Pb} - 1$), and ($x_{Pb} - 1$, $y_{Pb} - 1$), 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 $(xNbB_k, yNbB_k)$ from $(xNbB_0, yNbB_0)$ to $(xNbB_2, yNbB_2)$:
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb) , the current luma coding block size `nCbS`, the luma prediction block location (xPb, yPb) , the luma prediction block width `nPbW`, the luma prediction block height `nPbH`, the luma location $(xNbY, yNbY)$ set equal to $(xNbB_k, yNbB_k)$, and the partition index `partIdx` as inputs, and the output is assigned to the prediction 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( RefPicListX[ RefIdxLX[ xNbBk ][ yNbBk ] ], RefPicListX[ refIdxLX ] )` is equal to 0, `availableFlagLXB` is set equal to 1 and the following assignments are made:

$$mvLXB = MvLX[xNbB_k][yNbB_k] \quad (8-157)$$

$$refIdxB = RefIdxLX[xNbB_k][yNbB_k] \quad (8-158)$$
 - Otherwise, when `PredFlagLY[ xNbBk ][ yNbBk ]` (with $Y = !X$) is equal to 1 and `DiffPicOrderCnt( RefPicListY[ RefIdxLY[ xNbBk ][ yNbBk ] ], RefPicListX[ refIdxLX ] )` is equal to 0, `availableFlagLXB` is set equal to 1 and the following assignments are made:

$$mvLXB = MvLY[xNbB_k][yNbB_k] \quad (8-159)$$

$$refIdxB = RefIdxLY[xNbB_k][yNbB_k] \quad (8-160)$$
4. When `isScaledFlagLX` is equal to 0 and `availableFlagLXB` is equal to 1, `availableFlagLXA` is set equal to 1 and the following applies:
- $$mvLXA = mvLXB \quad (8-161)$$
5. When `isScaledFlagLX` is equal to 0, `availableFlagLXB` is set equal to 0 and the following applies for $(xNbB_k, yNbB_k)$ from $(xNbB_0, yNbB_0)$ to $(xNbB_2, yNbB_2)$ or until `availableFlagLXB` is equal to 1:
- The availability derivation process for a prediction block as specified in subclause 6.4.2 is invoked with the luma location (xCb, yCb) , the current luma coding block size `nCbS`, the luma location (xPb, yPb) , the luma prediction block width `nPbW`, the luma prediction block height `nPbH`, the luma location $(xNbY, yNbY)$ set equal to $(xNbB_k, yNbB_k)$, and the partition index `partIdx` as inputs, and the output is assigned to the prediction 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 `LongTermRefPic( currPic, currPb, refIdxLX, RefPicListX )` is equal to `LongTermRefPic( currPic, currPb, RefIdxLX[ xNbBk ][ yNbBk ], RefPicListX )`, `availableFlagLXB` is set equal to 1 and the following assignments are made:

$$mvLXB = MvLX[xNbB_k][yNbB_k] \quad (8-162)$$

$$refIdxB = RefIdxLX[xNbB_k][yNbB_k] \quad (8-163)$$

$$refPicListB = RefPicListX \quad (8-164)$$
 - Otherwise, when `PredFlagLY[ xNbBk ][ yNbBk ]` (with $Y = !X$) is equal to 1 and `LongTermRefPic( currPic, currPb, refIdxLX, RefPicListX )` is equal to `LongTermRefPic( currPic, currPb, RefIdxLY[ xNbBk ][ yNbBk ], RefPicListY )`, `availableFlagLXB` is set equal to 1 and the following assignments are made:

$$mvLXB = MvLY[xNbB_k][yNbB_k] \quad (8-165)$$

$$refIdxB = RefIdxLY[xNbB_k][yNbB_k] \quad (8-166)$$

$$refPicListB = RefPicListY \quad (8-167)$$
 - When `availableFlagLXB` is equal to 1, `DiffPicOrderCnt( refPicListB[ refIdxB ], RefPicListX[ refIdxLX ] )` is not equal to 0, and both `refPicListB[ refIdxB ]` and `RefPicListX[ refIdxLX ]` are short-term reference pictures, `mvLXB` is derived as follows:

$$tx = (16384 + (Abs(td) >> 1)) / td \quad (8-168)$$

$$distScaleFactor = Clip3(-4096, 4095, (tb * tx + 32) >> 6) \quad (8-169)$$

$$\text{mvLXB} = \text{Clip3}(-32768, 32767, \text{Sign}(\text{distScaleFactor} * \text{mvLXB}) * ((\text{Abs}(\text{distScaleFactor} * \text{mvLXB}) + 127) >> 8)) \quad (8-170)$$

where td and tb are derived as follows:

$$\text{td} = \text{Clip3}(-128, 127, \text{DiffPicOrderCnt}(\text{currPic}, \text{refPicListB}[\text{refIdxB}])) \quad (8-171)$$

$$\text{tb} = \text{Clip3}(-128, 127, \text{DiffPicOrderCnt}(\text{currPic}, \text{RefPicListX}[\text{refIdxLX}])) \quad (8-172)$$

8.5.3.2.7 Derivation process for temporal luma motion vector prediction

Inputs to this process are:

- a luma location (xPb , yPb) specifying the top-left sample of the current luma prediction block relative to the top-left luma sample of the current picture,
- two variables nPbW and nPbH specifying the width and the height of the luma prediction block,
- a reference index refIdxLX , with X being 0 or 1.

Outputs of this process are:

- the motion vector prediction mvLXCol ,
- the availability flag $\text{availableFlagLXCol}$.

The variable currPb specifies the current luma prediction block at luma location (xPb , yPb).

The variables mvLXCol and $\text{availableFlagLXCol}$ are derived as follows:

- If $\text{slice_temporal_mvp_enabled_flag}$ is equal to 0, both components of mvLXCol are set equal to 0 and $\text{availableFlagLXCol}$ is set equal to 0.
- Otherwise, the following ordered steps apply:

1. Depending on the values of slice_type , $\text{collocated_from_l0_flag}$, and $\text{collocated_ref_idx}$, the variable colPic , specifying the collocated picture, is derived as follows:

- If slice_type is equal to B and $\text{collocated_from_l0_flag}$ is equal to 0, colPic is set equal to $\text{RefPicList1}[\text{collocated_ref_idx}]$.
- Otherwise (slice_type is equal to B and $\text{collocated_from_l0_flag}$ is equal to 1 or slice_type is equal to P), colPic is set equal to $\text{RefPicList0}[\text{collocated_ref_idx}]$.

2. The bottom right collocated motion vector is derived as follows:

$$\text{xColBr} = \text{xPb} + \text{nPbW} \quad (8-173)$$

$$\text{yColBr} = \text{yPb} + \text{nPbH} \quad (8-174)$$

- If $\text{yPb} >> \text{CtbLog2SizeY}$ is equal to $\text{yColBr} >> \text{CtbLog2SizeY}$, yColBr is less than $\text{pic_height_in_luma_samples}$, and xColBr is less than $\text{pic_width_in_luma_samples}$, the following applies:

- The variable colPb specifies the luma prediction block covering the modified location given by $((\text{xColBr} >> 4) << 4, (\text{yColBr} >> 4) << 4)$ inside the collocated picture specified by colPic .

The luma location (xColPb , yColPb) is set equal to the top-left sample of the collocated luma prediction block specified by colPb relative to the top-left luma sample of the collocated picture specified by colPic .

- The derivation process for collocated motion vectors as specified in subclause 8.5.3.2.8 is invoked with currPb , colPic , colPb , (xColPb , yColPb), and refIdxLX as inputs, and the output is assigned to mvLXCol and $\text{availableFlagLXCol}$.
- Otherwise, both components of mvLXCol are set equal to 0 and $\text{availableFlagLXCol}$ is set equal to 0.

3. When $\text{availableFlagLXCol}$ is equal to 0, the central collocated motion vector is derived as follows:

$$\text{xColCtr} = \text{xPb} + (\text{nPbW} >> 1) \quad (8-175)$$

$$\text{yColCtr} = \text{yPb} + (\text{nPbH} >> 1) \quad (8-176)$$

- The variable colPb specifies the luma prediction block covering the modified location given by $((\text{xColCtr} >> 4) << 4, (\text{yColCtr} >> 4) << 4)$ inside the colPic .

- The luma location (xColPb, yColPb) is set equal to the top-left sample of the collocated luma prediction block specified by colPb relative to the top-left luma sample of the collocated picture specified by colPic.
- The derivation process for collocated motion vectors as specified in subclause 8.5.3.2.8 is invoked with currPb, colPic, colPb, (xColPb, yColPb), and refIdxLX as inputs, and the output is assigned to mvLXCol and availableFlagLXCol.

8.5.3.2.8 Derivation process for collocated motion vectors

Inputs to this process are:

- a variable currPb specifying the current prediction block,
- a variable colPic specifying the collocated picture,
- a variable colPb specifying the collocated prediction block inside the collocated picture specified by colPic,
- a luma location (xColPb, yColPb) specifying the top-left sample of the collocated luma prediction block specified by colPb relative to the top-left luma sample of the collocated picture specified by colPic,
- a reference index refIdxLX, with X being 0 or 1.

Outputs of this process are:

- the motion vector prediction mvLXCol,
- the availability flag availableFlagLXCol.

The variable currPic specifies the current picture.

The arrays predFlagLXCol[x][y], mvLXCol[x][y], and refIdxLXCol[x][y] are set equal to the corresponding arrays of the collocated picture specified by colPic, PredFlagLX[x][y], MvLX[x][y], and RefIdxLX[x][y], respectively, with X being the value of X this process is invoked for.

The variables mvLXCol and availableFlagLXCol are derived as follows:

- If colPb is coded in an intra 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 predFlagL0Col[xColPb][yColPb] is equal to 0, mvCol, refIdxCol, and listCol are set equal to mvL1Col[xColPb][yColPb], refIdxL1Col[xColPb][yColPb], and L1, respectively.
 - Otherwise, if predFlagL0Col[xColPb][yColPb] is equal to 1 and predFlagL1Col[xColPb][yColPb] is equal to 0, mvCol, refIdxCol, and listCol are set equal to mvL0Col[xColPb][yColPb], refIdxL0Col[xColPb][yColPb], and L0, respectively.
 - Otherwise (predFlagL0Col[xColPb][yColPb] is equal to 1 and predFlagL1Col[xColPb][yColPb] is equal to 1), the following assignments are made:
 - If DiffPicOrderCnt(aPic, currPic) is less than or equal to 0 for every picture aPic in every reference picture list of the current slice, mvCol, refIdxCol, and listCol are set equal to mvLXCol[xColPb][yColPb], refIdxLXCol[xColPb][yColPb] and LX, respectively.
 - Otherwise, mvCol, refIdxCol, and listCol are set equal to mvLNCol[xColPb][yColPb], refIdxLNCol[xColPb][yColPb], and LN, respectively, with N being the value of collocated_from_l0_flag.

and mvLXCol and availableFlagLXCol are derived as follows:

- If LongTermRefPic(currPic, currPb, refIdxLX, LX) is not equal to LongTermRefPic(colPic, colPb, 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, refPicListCol[refIdxCol] is set to be the picture with reference index refIdxCol in the reference picture list listCol of the slice containing prediction block currPb in the picture colPic, and the following applies:

$$\text{colPocDiff} = \text{DiffPicOrderCnt}(\text{colPic}, \text{refPicListCol}[\text{refIdxCol}]) \quad (8-177)$$

$$\text{currPocDiff} = \text{DiffPicOrderCnt}(\text{currPic}, \text{RefPicListX}[\text{refIdxLX}]) \quad (8-178)$$

- If $\text{RefPicListX}[\text{refIdxLX}]$ is a long-term reference picture, or colPocDiff is equal to currPocDiff , mvLXCol is derived as follows:

$$\text{mvLXCol} = \text{mvCol} \quad (8-179)$$

- Otherwise, mvLXCol is derived as a scaled version of the motion vector mvCol as follows:

$$\text{tx} = (16384 + (\text{Abs}(\text{td}) \gg 1)) / \text{td} \quad (8-180)$$

$$\text{distScaleFactor} = \text{Clip3}(-4096, 4095, (\text{tb} * \text{tx} + 32) \gg 6) \quad (8-181)$$

$$\text{mvLXCol} = \text{Clip3}(-32768, 32767, \text{Sign}(\text{distScaleFactor} * \text{mvCol}) * ((\text{Abs}(\text{distScaleFactor} * \text{mvCol}) + 127) \gg 8)) \quad (8-182)$$

where td and tb are derived as follows:

$$\text{td} = \text{Clip3}(-128, 127, \text{colPocDiff}) \quad (8-183)$$

$$\text{tb} = \text{Clip3}(-128, 127, \text{currPocDiff}) \quad (8-184)$$

8.5.3.2.9 Derivation process for chroma motion vectors

Input to this process is a luma motion vector mvLX .

Output of this process is a chroma motion vector mvCLX .

A chroma motion vector is derived from the corresponding luma motion vector.

For the derivation of the chroma motion vector mvCLX , the following applies:

$$\text{mvCLX}[0] = \text{mvLX}[0] \quad (8-185)$$

$$\text{mvCLX}[1] = \text{mvLX}[1] \quad (8-186)$$

8.5.3.3 Decoding process for inter prediction samples

8.5.3.3.1 General

Inputs to this process are:

- a luma location (x_{Cb}, y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{Bl}, y_{Bl}) specifying the top-left sample of the current luma prediction block relative to the top-left sample of the current luma coding block,
- a variable n_{CbS} specifying the size of the current luma coding block,
- two variables n_{PbW} and n_{PbH} specifying the width and the height of the luma prediction block,
- the luma motion vectors mvL0 and mvL1 ,
- the chroma motion vectors mvCL0 and mvCL1 ,
- the reference indices refIdxL0 and refIdxL1 ,
- the prediction list utilization flags, predFlagL0 , and predFlagL1 .

Outputs of this process are:

- an $(n_{CbS_L}) \times (n_{CbS_L})$ array predSamples_L of luma prediction samples, where n_{CbS_L} is derived as specified below,
- an $(n_{CbS_C}) \times (n_{CbS_C})$ array preSamples_{Cb} of chroma prediction samples for the component Cb , where n_{CbS_C} is derived as specified below,
- an $(n_{CbS_C}) \times (n_{CbS_C})$ array predSamples_{Cr} of chroma residual samples for the component Cr , where n_{CbS_C} is derived as specified below.

The variable n_{CbS_L} is set equal to n_{CbS} and the variable n_{CbS_C} is set equal to $n_{CbS} \gg 1$.

Let predSamplesL0_L and predSamplesL1_L be $(n_{PbW}) \times (n_{PbH})$ arrays of predicted luma sample values and predSampleL0_{Cb} , predSampleL1_{Cb} , predSampleL0_{Cr} , and predSampleL1_{Cr} be $(n_{PbW} / 2) \times (n_{PbH} / 2)$ arrays of predicted chroma sample values.

For X being each of 0 and 1, when predFlagLX is equal to 1, the following applies:

- The reference picture consisting of an ordered two-dimensional array refPicLX_L of luma samples and two ordered two-dimensional arrays refPicLX_{Cb} and refPicLX_{Cr} of chroma samples is derived by invoking the process specified in subclause 8.5.3.3.2 with refIdxLX as input.
- The arrays predSamplesLX_L , $\text{predSamplesLX}_{Cb}$, and $\text{predSamplesLX}_{Cr}$ are derived by invoking the fractional sample interpolation process specified in subclause 8.5.3.3.3 with the luma locations (x_{Cb}, y_{Cb}) and (x_{Bl}, y_{Bl}) , the luma prediction block width n_{PbW} , the luma prediction block height n_{PbH} , the motion vectors mvLX and mvCLX , and the reference arrays refPicLX_L , refPicLX_{Cb} , and refPicLX_{Cr} as inputs.

The array predSample_L of the prediction samples of luma component is derived by invoking the weighted sample prediction process specified in subclause 8.5.3.3.4 with the luma prediction block width n_{PbW} , the luma prediction block height n_{PbH} , and the sample arrays predSamplesL0_L and predSamplesL1_L , and the variables predFlagL0 , predFlagL1 , refIdxL0 , refIdxL1 , and cIdx equal to 0 as inputs.

The array predSample_{Cb} of the prediction samples of component Cb is derived by invoking the weighted sample prediction process specified in subclause 8.5.3.3.4 with the chroma prediction block width n_{PbW}_{Cb} set equal to $n_{PbW}/2$, the chroma prediction block height n_{PbH}_{Cb} set equal to $n_{PbH}/2$, the sample arrays $\text{predSamplesL0}_{Cb}$ and $\text{predSamplesL1}_{Cb}$, and the variables predFlagL0 , predFlagL1 , refIdxL0 , refIdxL1 , and cIdx equal to 1 as inputs.

The array predSample_{Cr} of the prediction samples of component Cr is derived by invoking the weighted sample prediction process specified in subclause 8.5.3.3.4 with the chroma prediction block width n_{PbW}_{Cr} set equal to $n_{PbW}/2$, the chroma prediction block height n_{PbH}_{Cr} set equal to $n_{PbH}/2$, the sample arrays $\text{predSamplesL0}_{Cr}$ and $\text{predSamplesL1}_{Cr}$, and the variables predFlagL0 , predFlagL1 , refIdxL0 , refIdxL1 , and cIdx equal to 2 as inputs.

8.5.3.3.2 Reference picture selection process

Input to this process is a reference index refIdxLX .

Output of this process is a reference picture consisting of a two-dimensional array of luma samples refPicLX_L and two two-dimensional arrays of chroma samples refPicLX_{Cb} and refPicLX_{Cr} .

The output reference picture $\text{RefPicListX}[\text{refIdxLX}]$ consists of a $\text{pic_width_in_luma_samples}$ by $\text{pic_height_in_luma_samples}$ array of luma samples refPicLX_L and two $\text{PicWidthInSamplesC}$ by $\text{PicHeightInSamplesC}$ arrays of chroma samples refPicLX_{Cb} and refPicLX_{Cr} .

The reference picture sample arrays refPicLX_L , refPicLX_{Cb} , and refPicLX_{Cr} correspond to decoded sample arrays S_L , S_{Cb} , and S_{Cr} derived in subclause 8.7 for a previously-decoded picture.

8.5.3.3.3 Fractional sample interpolation process

8.5.3.3.3.1 General

Inputs to this process are:

- a luma location (x_{Cb}, y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{Bl}, y_{Bl}) specifying the top-left sample of the current luma prediction block relative to the top-left sample of the current luma coding block,
- two variables n_{PbW} and n_{PbH} specifying the width and the height of the luma prediction block,
- a luma motion vector mvLX given in quarter-luma-sample units,
- a chroma motion vector mvCLX given in eighth-chroma-sample units,
- the selected reference picture sample arrays refPicLX_L , refPicLX_{Cb} , and refPicLX_{Cr} .

Outputs of this process are:

- an $(n_{PbW}) \times (n_{PbH})$ array predSampleLX_L of prediction luma sample values,
- two $(n_{PbW}/2) \times (n_{PbH}/2)$ arrays predSampleLX_{Cb} , and predSampleLX_{Cr} of prediction chroma sample values.

The location (x_{Pb}, y_{Pb}) given in full-sample units of the upper-left luma samples of the current prediction block relative to the upper-left luma sample location of the given reference sample arrays is derived as follows:

$$x_{Pb} = x_{Cb} + x_{Bl} \quad (8-187)$$

$$y_{Pb} = y_{Cb} + y_{Bl} \quad (8-188)$$

Let (x_{Int_L}, y_{Int_L}) be a luma location given in full-sample units and (x_{Frac_L}, y_{Frac_L}) be an offset given in quarter-sample units. These variables are used only inside this subclause for specifying fractional-sample locations inside the reference sample arrays $refPicLX_L$, $refPicLX_{Cb}$, and $refPicLX_{Cr}$.

For each luma sample location $(x_L = 0..nPbW - 1, y_L = 0..nPbH - 1)$ inside the prediction luma sample array $predSampleLX_L$, the corresponding prediction luma sample value $predSampleLX_L[x_L, y_L]$ is derived as follows:

- The variables x_{Int_L} , y_{Int_L} , x_{Frac_L} , and y_{Frac_L} are derived as follows:

$$x_{Int_L} = x_{Pb} + (mvLX[0] \gg 2) + x_L \quad (8-189)$$

$$y_{Int_L} = y_{Pb} + (mvLX[1] \gg 2) + y_L \quad (8-190)$$

$$x_{Frac_L} = mvLX[0] \& 3 \quad (8-191)$$

$$y_{Frac_L} = mvLX[1] \& 3 \quad (8-192)$$

- The prediction luma sample value $predSampleLX_L[x_L, y_L]$ is derived by invoking the process specified in subclause 8.5.3.3.3.2 with (x_{Int_L}, y_{Int_L}) , (x_{Frac_L}, y_{Frac_L}) , and $refPicLX_L$ as inputs.

Let (x_{Int_C}, y_{Int_C}) be a chroma location given in full-sample units and (x_{Frac_C}, y_{Frac_C}) be an offset given in one-eighth sample units. These variables are used only inside this subclause for specifying general fractional-sample locations inside the reference sample arrays $refPicLX_{Cb}$ and $refPicLX_{Cr}$.

For each chroma sample location $(x_C = 0..nPbW / 2 - 1, y_C = 0..nPbH / 2 - 1)$ inside the prediction chroma sample arrays $predSampleLX_{Cb}$ and $predSampleLX_{Cr}$, the corresponding prediction chroma sample values $predSampleLX_{Cb}[x_C, y_C]$ and $predSampleLX_{Cr}[x_C, y_C]$ are derived as follows:

- The variables x_{Int_C} , y_{Int_C} , x_{Frac_C} , and y_{Frac_C} are derived as follows:

$$x_{Int_C} = (x_{Pb} / 2) + (mvCLX[0] \gg 3) + x_C \quad (8-193)$$

$$y_{Int_C} = (y_{Pb} / 2) + (mvCLX[1] \gg 3) + y_C \quad (8-194)$$

$$x_{Frac_C} = mvLX[0] \& 7 \quad (8-195)$$

$$y_{Frac_C} = mvLX[1] \& 7 \quad (8-196)$$

- The prediction sample value $predSampleLX_{Cb}[x_C, y_C]$ is derived by invoking the process specified in subclause 8.5.3.3.3.3 with (x_{Int_C}, y_{Int_C}) , (x_{Frac_C}, y_{Frac_C}) , and $refPicLX_{Cb}$ as inputs.
- The prediction sample value $predSampleLX_{Cr}[x_C, y_C]$ is derived by invoking the process specified in subclause 8.5.3.3.3.3 with (x_{Int_C}, y_{Int_C}) , (x_{Frac_C}, y_{Frac_C}) , and $refPicLX_{Cr}$ as inputs.

8.5.3.3.3.2 Luma sample interpolation process

Inputs to this process are:

- a luma location in full-sample units (x_{Int_L}, y_{Int_L}) ,
- a luma location in fractional-sample units (x_{Frac_L}, y_{Frac_L}) ,
- the luma reference sample array $refPicLX_L$.

Output of this process is a predicted luma sample value $predSampleLX_L[x_L, y_L]$

$A_{-1,-1}$				$A_{0,-1}$	$a_{0,-1}$	$b_{0,-1}$	$c_{0,-1}$	$A_{1,-1}$				$A_{2,-1}$
$A_{-1,0}$				$A_{0,0}$	$a_{0,0}$	$b_{0,0}$	$c_{0,0}$	$A_{1,0}$				$A_{2,0}$
$d_{-1,0}$				$d_{0,0}$	$e_{0,0}$	$f_{0,0}$	$g_{0,0}$	$d_{1,0}$				$d_{2,0}$
$h_{-1,0}$				$h_{0,0}$	$i_{0,0}$	$j_{0,0}$	$k_{0,0}$	$h_{1,0}$				$h_{2,0}$
$n_{-1,0}$				$n_{0,0}$	$p_{0,0}$	$q_{0,0}$	$r_{0,0}$	$n_{1,0}$				$n_{2,0}$
$A_{-1,1}$				$A_{0,1}$	$a_{0,1}$	$b_{0,1}$	$c_{0,1}$	$A_{1,1}$				$A_{2,1}$
$A_{-1,2}$				$A_{0,2}$	$a_{0,2}$	$b_{0,2}$	$c_{0,2}$	$A_{1,2}$				$A_{2,2}$

Figure 8-4 – Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for quarter sample luma interpolation

In Figure 8-4, the positions labelled with upper-case letters $A_{i,j}$ within shaded blocks represent luma samples at full-sample locations inside the given two-dimensional array refPicLX_L of luma samples. These samples may be used for generating the predicted luma sample value $\text{predSampleLX}_L[x_L, y_L]$. The locations $(xA_{i,j}, yA_{i,j})$ for each of the corresponding luma samples $A_{i,j}$ inside the given array refPicLX_L of luma samples are derived as follows:

$$xA_{i,j} = \text{Clip3}(0, \text{pic_width_in_luma_samples} - 1, x\text{Int}_L + i) \quad (8-197)$$

$$yA_{i,j} = \text{Clip3}(0, \text{pic_height_in_luma_samples} - 1, y\text{Int}_L + j) \quad (8-198)$$

The positions labelled with lower-case letters within un-shaded blocks represent luma samples at quarter-pel sample fractional locations. The luma location offset in fractional-sample units $(x\text{Frac}_L, y\text{Frac}_L)$ specifies which of the generated luma samples at full-sample and fractional-sample locations is assigned to the predicted luma sample value $\text{predSampleLX}_L[x_L, y_L]$. This assignment is as specified in Table 8-7. The value of $\text{predSampleLX}_L[x_L, y_L]$ is the output.

The variables shift1 , shift2 , and shift3 are derived as follows:

- The variable shift1 is set equal to $\text{BitDepth}_Y - 8$, the variable shift2 is set equal to 6, and the variable shift3 is set equal to $14 - \text{BitDepth}_Y$.

Given the luma samples $A_{i,j}$ at full-sample locations $(xA_{i,j}, yA_{i,j})$, the luma samples $a_{0,0}$ to $r_{0,0}$ at fractional sample positions are derived as follows:

- The samples labelled $a_{0,0}$, $b_{0,0}$, $c_{0,0}$, $d_{0,0}$, $h_{0,0}$, and $n_{0,0}$ are derived by applying an 8-tap filter to the nearest integer position samples as follows:

$$a_{0,0} = (-A_{-3,0} + 4 * A_{-2,0} - 10 * A_{-1,0} + 58 * A_{0,0} + 17 * A_{1,0} - 5 * A_{2,0} + A_{3,0}) \gg \text{shift1} \quad (8-199)$$

$$b_{0,0} = (-A_{-3,0} + 4 * A_{-2,0} - 11 * A_{-1,0} + 40 * A_{0,0} + 40 * A_{1,0} - 11 * A_{2,0} + 4 * A_{3,0} - A_{4,0}) \gg \text{shift1} \quad (8-200)$$

$$c_{0,0} = (A_{-2,0} - 5 * A_{-1,0} + 17 * A_{0,0} + 58 * A_{1,0} - 10 * A_{2,0} + 4 * A_{3,0} - A_{4,0}) \gg \text{shift1} \quad (8-201)$$

$$d_{0,0} = (-A_{0,-3} + 4 * A_{0,-2} - 10 * A_{0,-1} + 58 * A_{0,0} + 17 * A_{0,1} - 5 * A_{0,2} + A_{0,3}) \gg \text{shift1} \quad (8-202)$$

$$h_{0,0} = (-A_{0,-3} + 4 * A_{0,-2} - 11 * A_{0,-1} + 40 * A_{0,0} + 40 * A_{0,1} - 11 * A_{0,2} + 4 * A_{0,3} - A_{0,4}) \gg \text{shift1} \quad (8-203)$$

$$n_{0,0} = (A_{0,-2} - 5 * A_{0,-1} + 17 * A_{0,0} + 58 * A_{0,1} - 10 * A_{0,2} + 4 * A_{0,3} - A_{0,4}) \gg \text{shift1} \quad (8-204)$$

- The samples labelled $e_{0,0}$, $i_{0,0}$, $p_{0,0}$, $f_{0,0}$, $j_{0,0}$, $q_{0,0}$, $g_{0,0}$, $k_{0,0}$, and $r_{0,0}$ are derived by applying an 8-tap filter to the samples $a_{0,i}$, $b_{0,i}$ and $c_{0,i}$ with $i = -3..4$ in the vertical direction as follows:

$$e_{0,0} = (-a_{0,-3} + 4 * a_{0,-2} - 10 * a_{0,-1} + 58 * a_{0,0} + 17 * a_{0,1} - 5 * a_{0,2} + a_{0,3}) \gg \text{shift2} \quad (8-205)$$

$$i_{0,0} = (-a_{0,-3} + 4 * a_{0,-2} - 11 * a_{0,-1} + 40 * a_{0,0} + 40 * a_{0,1} - 11 * a_{0,2} + 4 * a_{0,3} - a_{0,4}) \gg \text{shift2} \quad (8-206)$$

$$p_{0,0} = (a_{0,-2} - 5 * a_{0,-1} + 17 * a_{0,0} + 58 * a_{0,1} - 10 * a_{0,2} + 4 * a_{0,3} - a_{0,4}) \gg \text{shift2} \quad (8-207)$$

$$f_{0,0} = (-b_{0,-3} + 4 * b_{0,-2} - 10 * b_{0,-1} + 58 * b_{0,0} + 17 * b_{0,1} - 5 * b_{0,2} + b_{0,3}) \gg \text{shift2} \quad (8-208)$$

$$j_{0,0} = (-b_{0,-3} + 4 * b_{0,-2} - 11 * b_{0,-1} + 40 * b_{0,0} + 40 * b_{0,1} - 11 * b_{0,2} + 4 * b_{0,3} - b_{0,4}) \gg \text{shift2} \quad (8-209)$$

$$q_{0,0} = (b_{0,-2} - 5 * b_{0,-1} + 17 * b_{0,0} + 58 * b_{0,1} - 10 * b_{0,2} + 4 * b_{0,3} - b_{0,4}) \gg \text{shift2} \quad (8-210)$$

$$g_{0,0} = (-c_{0,-3} + 4 * c_{0,-2} - 10 * c_{0,-1} + 58 * c_{0,0} + 17 * c_{0,1} - 5 * c_{0,2} + c_{0,3}) \gg \text{shift2} \quad (8-211)$$

$$k_{0,0} = (-c_{0,-3} + 4 * c_{0,-2} - 11 * c_{0,-1} + 40 * c_{0,0} + 40 * c_{0,1} - 11 * c_{0,2} + 4 * c_{0,3} - c_{0,4}) \gg \text{shift2} \quad (8-212)$$

$$r_{0,0} = (c_{0,-2} - 5 * c_{0,-1} + 17 * c_{0,0} + 58 * c_{0,1} - 10 * c_{0,2} + 4 * c_{0,3} - c_{0,4}) \gg \text{shift2} \quad (8-213)$$

Table 8-7 – Assignment of the luma prediction sample $\text{predSampleLX}_L[x_L, y_L]$

xFracL	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
yFracL	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
predSampleLX_L[x_L, y_L]	A << shift3	d	h	n	a	e	i	p	b	f	j	q	c	g	k	r

8.5.3.3.3 Chroma sample interpolation process

Inputs to this process are:

- a chroma location in full-sample units (x_{IntC} , y_{IntC}),
- a chroma location in fractional-sample units (x_{FracC} , y_{FracC}),
- the chroma reference sample array refPicLXC .

Output of this process is a predicted chroma sample value $\text{predSampleLXC}[x_C, y_C]$

	ha _{0,-1}	hb _{0,-1}	hc _{0,-1}	hd _{0,-1}	he _{0,-1}	hf _{0,-1}	hg _{0,-1}	hh _{0,-1}	
ah _{-1,0}	B _{0,0}	ab _{0,0}	ac _{0,0}	ad _{0,0}	ae _{0,0}	af _{0,0}	ag _{0,0}	ah _{0,0}	B _{1,0}
bh _{-1,0}	ba _{0,0}	bb _{0,0}	bc _{0,0}	bd _{0,0}	be _{0,0}	bf _{0,0}	bg _{0,0}	bh _{0,0}	ba _{1,0}
ch _{-1,0}	ca _{0,0}	cb _{0,0}	cc _{0,0}	cd _{0,0}	ce _{0,0}	cf _{0,0}	cg _{0,0}	ch _{0,0}	ca _{1,0}
dh _{-1,0}	da _{0,0}	db _{0,0}	dc _{0,0}	dd _{0,0}	de _{0,0}	df _{0,0}	dg _{0,0}	dh _{0,0}	da _{1,0}
eh _{-1,0}	ea _{0,0}	eb _{0,0}	ec _{0,0}	ed _{0,0}	ee _{0,0}	ef _{0,0}	eg _{0,0}	eh _{0,0}	ea _{1,0}
fh _{-1,0}	fa _{0,0}	fb _{0,0}	fc _{0,0}	fd _{0,0}	fe _{0,0}	ff _{0,0}	fg _{0,0}	fh _{0,0}	fa _{1,0}
gh _{-1,0}	ga _{0,0}	gb _{0,0}	gc _{0,0}	gd _{0,0}	ge _{0,0}	gf _{0,0}	gg _{0,0}	gh _{0,0}	ga _{1,0}
hh _{-1,0}	ha _{0,0}	hb _{0,0}	hc _{0,0}	hd _{0,0}	he _{0,0}	hf _{0,0}	hg _{0,0}	hh _{0,0}	ha _{1,0}
	B _{0,1}	ab _{0,1}	ac _{0,1}	ad _{0,1}	ae _{0,1}	af _{0,1}	ag _{0,1}	ah _{0,1}	B _{1,1}

Figure 8-5 – Integer samples (shaded blocks with upper-case letters) and fractional sample positions (un-shaded blocks with lower-case letters) for eighth sample chroma interpolation

In Figure 8-5, the positions labelled with upper-case letters $B_{i,j}$ within shaded blocks represent chroma samples at full-sample locations inside the given two-dimensional array refPicLXC of chroma samples. These samples may be used for generating the predicted chroma sample value $\text{predSampleLXC}[x_C, y_C]$. The locations $(x_{B_{i,j}}, y_{B_{i,j}})$ for each of the corresponding chroma samples $B_{i,j}$ inside the given array refPicLXC of chroma samples are derived as follows:

$$xB_{i,j} = \text{Clip3}(0, (\text{pic_width_in_luma_samples} / \text{SubWidthC}) - 1, x\text{Int}_C + i) \quad (8-214)$$

$$yB_{i,j} = \text{Clip3}(0, (\text{pic_height_in_luma_samples} / \text{SubHeightC}) - 1, y\text{Int}_C + j) \quad (8-215)$$

The positions labelled with lower-case letters within un-shaded blocks represent chroma samples at eighth-pel sample fractional locations. The chroma location offset in fractional-sample units $(x\text{Frac}_C, y\text{Frac}_C)$ specifies which of the generated chroma samples at full-sample and fractional-sample locations is assigned to the predicted chroma sample value $\text{predSampleLXC}[x_C, y_C]$. This assignment is as specified in Table 8-8. The output is the value of $\text{predSampleLXC}[x_C, y_C]$.

The variables shift1 , shift2 , and shift3 are derived as follows:

- The variable shift1 is set equal to $\text{BitDepth}_C - 8$, the variable shift2 is set equal to 6, and the variable shift3 is set equal to $14 - \text{BitDepth}_C$.

Given the chroma samples $B_{i,j}$ at full-sample locations $(x_{B_{i,j}}, y_{B_{i,j}})$, the chroma samples $ab_{0,0}$ to $hh_{0,0}$ at fractional sample positions are derived as follows:

- The samples labelled $ab_{0,0}$, $ac_{0,0}$, $ad_{0,0}$, $ae_{0,0}$, $af_{0,0}$, $ag_{0,0}$, and $ah_{0,0}$ are derived by applying a 4-tap filter to the nearest integer position samples as follows:

$$ab_{0,0} = (-2 * B_{-1,0} + 58 * B_{0,0} + 10 * B_{1,0} - 2 * B_{2,0}) \gg \text{shift1} \quad (8-216)$$

$$ac_{0,0} = (-4 * B_{-1,0} + 54 * B_{0,0} + 16 * B_{1,0} - 2 * B_{2,0}) \gg \text{shift1} \quad (8-217)$$

$$ad_{0,0} = (-6 * B_{-1,0} + 46 * B_{0,0} + 28 * B_{1,0} - 4 * B_{2,0}) \gg \text{shift1} \quad (8-218)$$

$$ae_{0,0} = (-4 * B_{-1,0} + 36 * B_{0,0} + 36 * B_{1,0} - 4 * B_{2,0}) \gg \text{shift1} \quad (8-219)$$

$$af_{0,0} = (-4 * B_{-1,0} + 28 * B_{0,0} + 46 * B_{1,0} - 6 * B_{2,0}) \gg \text{shift1} \quad (8-220)$$

$$ag_{0,0} = (-2 * B_{-1,0} + 16 * B_{0,0} + 54 * B_{1,0} - 4 * B_{2,0}) \gg \text{shift1} \quad (8-221)$$

$$ah_{0,0} = (-2 * B_{-1,0} + 10 * B_{0,0} + 58 * B_{1,0} - 2 * B_{2,0}) \gg \text{shift1} \quad (8-222)$$

- The samples labelled $ba_{0,0}$, $ca_{0,0}$, $da_{0,0}$, $ea_{0,0}$, $fa_{0,0}$, $ga_{0,0}$, and $ha_{0,0}$ are derived by applying a 4-tap filter to the nearest integer position samples as follows:

$$ba_{0,0} = (-2 * B_{0,-1} + 58 * B_{0,0} + 10 * B_{0,1} - 2 * B_{0,2}) \gg \text{shift1} \quad (8-223)$$

$$ca_{0,0} = (-4 * B_{0,-1} + 54 * B_{0,0} + 16 * B_{0,1} - 2 * B_{0,2}) \gg \text{shift1} \quad (8-224)$$

$$da_{0,0} = (-6 * B_{0,-1} + 46 * B_{0,0} + 28 * B_{0,1} - 4 * B_{0,2}) \gg \text{shift1} \quad (8-225)$$

$$ea_{0,0} = (-4 * B_{0,-1} + 36 * B_{0,0} + 36 * B_{0,1} - 4 * B_{0,2}) \gg \text{shift1} \quad (8-226)$$

$$fa_{0,0} = (-4 * B_{0,-1} + 28 * B_{0,0} + 46 * B_{0,1} - 6 * B_{0,2}) \gg \text{shift1} \quad (8-227)$$

$$ga_{0,0} = (-2 * B_{0,-1} + 16 * B_{0,0} + 54 * B_{0,1} - 4 * B_{0,2}) \gg \text{shift1} \quad (8-228)$$

$$ha_{0,0} = (-2 * B_{0,-1} + 10 * B_{0,0} + 58 * B_{0,1} - 2 * B_{0,2}) \gg \text{shift1} \quad (8-229)$$

- The samples labelled $bX_{0,0}$, $cX_{0,0}$, $dX_{0,0}$, $eX_{0,0}$, $fX_{0,0}$, $gX_{0,0}$, and $hX_{0,0}$ for X being replaced by b, c, d, e, f, g, and h, respectively, are derived by applying an 4-tap filter to the intermediate values $aX_{0,i}$ with $i = -1..2$ in the vertical direction as follows:

$$bX_{0,0} = (-2 * aX_{0,-1} + 58 * aX_{0,0} + 10 * aX_{0,1} - 2 * aX_{0,2}) \gg \text{shift2} \quad (8-230)$$

$$cX_{0,0} = (-4 * aX_{0,-1} + 54 * aX_{0,0} + 16 * aX_{0,1} - 2 * aX_{0,2}) \gg \text{shift2} \quad (8-231)$$

$$dX_{0,0} = (-6 * aX_{0,-1} + 46 * aX_{0,0} + 28 * aX_{0,1} - 4 * aX_{0,2}) \gg \text{shift2} \quad (8-232)$$

$$eX_{0,0} = (-4 * aX_{0,-1} + 36 * aX_{0,0} + 36 * aX_{0,1} - 4 * aX_{0,2}) \gg \text{shift2} \quad (8-233)$$

$$fX_{0,0} = (-4 * aX_{0,-1} + 28 * aX_{0,0} + 46 * aX_{0,1} - 6 * aX_{0,2}) \gg \text{shift2} \quad (8-234)$$

$$gX_{0,0} = (-2 * aX_{0,-1} + 16 * aX_{0,0} + 54 * aX_{0,1} - 4 * aX_{0,2}) \gg \text{shift2} \quad (8-235)$$

$$hX_{0,0} = (-2 * aX_{0,-1} + 10 * aX_{0,0} + 58 * aX_{0,1} - 2 * aX_{0,2}) \gg \text{shift2} \quad (8-236)$$

Table 8-8 – Assignment of the chroma prediction sample $\text{predSampleLXC}[x_C, y_C]$ for (X, Y) being replaced by (1, b), (2, c), (3, d), (4, e), (5, f), (6, g), and (7, h), respectively

xFracC	0	0	0	0	0	0	0	0
yFracC	0	1	2	3	4	5	6	7
predSampleLXC[x_C, y_C]	B << shift3	ba	ca	da	ea	fa	ga	ha
xFracC	X	X	X	X	X	X	X	X
yFracC	0	1	2	3	4	5	6	7
predSampleLXC[x_C, y_C]	aY	bY	cY	dY	eY	fY	gY	hY

8.5.3.3.4 Weighted sample prediction process

8.5.3.3.4.1 General

Inputs to this process are:

- two variables nPbW and nPbH specifying the width and the height of the luma prediction block,
- two (nPbW)x(nPbH) arrays predSamplesL0 and predSamplesL1,
- the prediction list utilization flags, predFlagL0, and predFlagL1,
- the reference indices refIdxL0 and refIdxL1,
- a variable cIdx specifying colour component index.

Output of this process is the (nPbW)x(nPbH) array predSamples of prediction sample values.

The variable bitDepth is derived as follows:

- If $cIdx$ is equal to 0, $bitDepth$ is set equal to $BitDepth_Y$.
- Otherwise, $bitDepth$ is set equal to $BitDepth_C$.

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 $predSample$ of the prediction samples is derived by invoking the default weighted sample prediction process as specified in subclause 8.5.3.3.4.2 with the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, two $(nPbW) \times (nPbH)$ arrays $predSamplesL0$ and $predSamplesL1$, the prediction list utilization flags $predFlagL0$ and $predFlagL1$, and the bit depth $bitDepth$ as inputs.
- Otherwise ($weightedPredFlag$ is equal to 1), the array $predSample$ of the prediction samples is derived by invoking the weighted sample prediction process as specified in subclause 8.5.3.3.4.3 with the luma prediction block width $nPbW$, the luma prediction block height $nPbH$, two $(nPbW) \times (nPbH)$ 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.

8.5.3.3.4.2 Default weighted sample prediction process

Inputs to this process are:

- two variables $nPbW$ and $nPbH$ specifying the width and the height of the luma prediction block,
- two $(nPbW) \times (nPbH)$ arrays $predSamplesL0$ and $predSamplesL1$,
- the prediction list utilization flags, $predFlagL0$, and $predFlagL1$,
- a bit depth of samples, $bitDepth$.

Output of this process is the $(nPbW) \times (nPbH)$ array $predSamples$ of prediction sample values.

Variables $shift1$, $shift2$, $offset1$, and $offset2$ are derived as follows:

- The variable $shift1$ is set equal to $14 - bitDepth$ and the variable $shift2$ is set equal to $15 - bitDepth$.
- The variable $offset1$ is derived as follows:
 - If $shift1$ is greater than 0, $offset1$ is set equal to $1 \ll (shift1 - 1)$.
 - Otherwise ($shift1$ is equal to 0), $offset1$ is set equal to 0.
- The variable $offset2$ is set equal to $1 \ll (shift2 - 1)$.

Depending on the values of $predFlagL0$ and $predFlagL1$, the prediction samples $predSamples[x][y]$ with $x = 0..nPbW - 1$ and $y = 0..nPbH - 1$ are derived as follows:

- If $predFlagL0$ is equal to 1 and $predFlagL1$ is equal to 0, the prediction sample values are derived as follows:

$$predSamples[x][y] = Clip3(0, (1 \ll bitDepth) - 1, (predSamplesL0[x][y] + offset1) \gg shift1) \quad (8-237)$$
- Otherwise, if $predFlagL0$ is equal to 0 and $predFlagL1$ is equal to 1, the prediction sample values are derived as follows:

$$predSamples[x][y] = Clip3(0, (1 \ll bitDepth) - 1, (predSamplesL1[x][y] + offset1) \gg shift1) \quad (8-238)$$
- Otherwise ($predFlagL0$ is equal to 1 and $predFlagL1$ is equal to 1), the prediction sample values are derived as follows:

$$predSamples[x][y] = Clip3(0, (1 \ll bitDepth) - 1, (predSamplesL0[x][y] + predSamplesL1[x][y] + offset2) \gg shift2) \quad (8-239)$$

8.5.3.3.4.3 Explicit weighted sample prediction process

Inputs to this process are:

- two variables $nPbW$ and $nPbH$ specifying the width and the height of the luma prediction block,
- two $(nPbW) \times (nPbH)$ arrays $predSamplesL0$ and $predSamplesL1$,
- the prediction list utilization flags, $predFlagL0$, and $predFlagL1$,

- the reference indices, refIdxL0 and refIdxL1,
- a variable cIdx specifying colour component index,
- a bit depth of samples, bitDepth.

Output of this process is the (nPbW)x(nPbH) array predSamples of prediction sample values.

The variable shift1 is set equal to 14 – bitDepth.

The variables log2Wd, o0, o1, and w0, w1 are derived as follows:

- If cIdx is equal to 0 for luma samples, the following applies:

$$\text{log2Wd} = \text{luma_log2_weight_denom} + \text{shift1} \quad (8-240)$$

$$w0 = \text{LumaWeightL0}[\text{refIdxL0}] \quad (8-241)$$

$$w1 = \text{LumaWeightL1}[\text{refIdxL1}] \quad (8-242)$$

$$o0 = \text{luma_offset_l0}[\text{refIdxL0}] * (1 \ll (\text{bitDepth} - 8)) \quad (8-243)$$

$$o1 = \text{luma_offset_l1}[\text{refIdxL1}] * (1 \ll (\text{bitDepth} - 8)) \quad (8-244)$$

- Otherwise (cIdx is not equal to 0 for chroma samples), the following applies:

$$\text{log2Wd} = \text{ChromaLog2WeightDenom} + \text{shift1} \quad (8-245)$$

$$w0 = \text{ChromaWeightL0}[\text{refIdxL0}][\text{cIdx} - 1] \quad (8-246)$$

$$w1 = \text{ChromaWeightL1}[\text{refIdxL1}][\text{cIdx} - 1] \quad (8-247)$$

$$o0 = \text{ChromaOffsetL0}[\text{refIdxL0}][\text{cIdx} - 1] * (1 \ll (\text{bitDepth} - 8)) \quad (8-248)$$

$$o1 = \text{ChromaOffsetL1}[\text{refIdxL1}][\text{cIdx} - 1] * (1 \ll (\text{bitDepth} - 8)) \quad (8-249)$$

The prediction sample predSamples[x][y] with x = 0..nPbW – 1 and y = 0..nPbH – 1 are derived as follows:

- If the predFlagL0 is equal to 1 and predFlagL1 is equal to 0, the prediction sample values are derived as follows:

$$\begin{aligned} &\text{if}(\text{log2Wd} \geq 1) \\ &\quad \text{predSamples}[x][y] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \\ &\quad \quad ((\text{predSamplesL0}[x][y] * w0 + 2^{\text{log2Wd}-1}) \gg \text{log2Wd}) + o0) \\ &\text{else} \\ &\quad \text{predSamples}[x][y] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \text{predSamplesL0}[x][y] * w0 + o0) \end{aligned} \quad (8-250)$$

- Otherwise, if the predFlagL0 is equal to 0 and predFlagL1 is equal to 1, the prediction sample values are derived as follows:

$$\begin{aligned} &\text{if}(\text{log2Wd} \geq 1) \\ &\quad \text{predSamples}[x][y] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \\ &\quad \quad ((\text{predSamplesL1}[x][y] * w1 + 2^{\text{log2Wd}-1}) \gg \text{log2Wd}) + o1) \\ &\text{else} \\ &\quad \text{predSamples}[x][y] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \text{predSamplesL1}[x][y] * w1 + o1) \end{aligned} \quad (8-251)$$

- Otherwise (predFlagL0 is equal to 1 and predFlagL1 is equal to 1), the prediction sample values are derived as follows:

$$\begin{aligned} \text{predSamples}[x][y] = &\text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \\ &(\text{predSamplesL0}[x][y] * w0 + \text{predSamplesL1}[x][y] * w1 + \\ &((o0 + o1 + 1) \ll \text{log2Wd})) \gg (\text{log2Wd} + 1)) \end{aligned} \quad (8-252)$$

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

8.5.4.1 General

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,
- a variable log2CbSize specifying the size of the current luma coding block.

Outputs of this process are:

- an $(nCbS_L) \times (nCbS_L)$ array $resSamples_L$ of luma residual samples, where $nCbS_L$ is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cb}$ of chroma residual samples for the component Cb, where $nCbS_C$ is derived as specified below,
- an $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cr}$ of chroma residual samples for the component Cr, where $nCbS_C$ is derived as specified below.

The variable $nCbS_L$ is set equal to $1 \ll \log_2 CbSize$ and the variable $nCbS_C$ is set equal to $nCbS_L \gg 1$.

Let $resSamples_L$ be an $(nCbS_L) \times (nCbS_L)$ array of luma residual samples and let $resSamples_{Cb}$ and $resSamples_{Cr}$ be two $(nCbS_C) \times (nCbS_C)$ arrays of chroma residual samples.

Depending on the value of rqt_root_cbf , the following applies:

- If rqt_root_cbf is equal to 0 or $skip_flag[xCb][yCb]$ is equal to 1, all samples of the $(nCbS_L) \times (nCbS_L)$ array $resSamples_L$ and all samples of the two $(nCbS_C) \times (nCbS_C)$ arrays $resSamples_{Cb}$ and $resSamples_{Cr}$ are set equal to 0.
- Otherwise (rqt_root_cbf is equal to 1), the following ordered steps apply:
 1. The decoding process for luma residual blocks as specified in subclause 8.5.4.2 below is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$ set equal to $(0, 0)$, the variable $\log_2 TrafoSize$ set equal to $\log_2 CbSize$, the variable $trafoDepth$ set equal to 0, the variable $nCbS$ set equal to $nCbS_L$, and the $(nCbS_L) \times (nCbS_L)$ array $resSamples_L$ as inputs, and the output is a modified version of the $(nCbS_L) \times (nCbS_L)$ array $resSamples_L$.
 2. The decoding process for chroma residual blocks as specified in subclause 8.5.4.3 below is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$ set equal to $(0, 0)$, the variable $\log_2 TrafoSize$ set equal to $\log_2 CbSize$, the variable $trafoDepth$ set equal to 0, the variable $cldx$ set equal to 1, the variable $nCbS$ set equal to $nCbS_C$, and the $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cb}$ as inputs, and the output is a modified version of the $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cb}$.
 3. The decoding process for chroma residual blocks as specified in subclause 8.5.4.3 below is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$ set equal to $(0, 0)$, the variable $\log_2 TrafoSize$ set equal to $\log_2 CbSize$, the variable $trafoDepth$ set equal to 0, the variable $cldx$ set equal to 2, the variable $nCbS$ set equal to $nCbS_C$, and the $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cr}$ as inputs, and the output is a modified version of the $(nCbS_C) \times (nCbS_C)$ array $resSamples_{Cr}$.

8.5.4.2 Decoding process for luma residual blocks

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,
- a luma location $(xB0, yB0)$ specifying the top-left sample of the current luma block relative to the top-left sample of the current luma coding block,
- a variable $\log_2 TrafoSize$ specifying the size of the current luma block,
- a variable $trafoDepth$ specifying the hierarchy depth of the current luma block relative to the luma coding block,
- a variable $nCbS$ specifying the size of the current luma coding block,
- an $(nCbS) \times (nCbS)$ array $resSamples$ of luma residual samples.

Output of this process is a modified version of the $(nCbS) \times (nCbS)$ array of luma residual samples.

Depending on the value of $split_transform_flag[xCb + xB0][yCb + yB0][trafoDepth]$, the following applies:

- If $split_transform_flag[xCb + xB0][yCb + yB0][trafoDepth]$ is equal to 1, the following ordered steps apply:
 1. The variables $xB1$ and $yB1$ are derived as follows:
 - The variable $xB1$ is set equal to $xB0 + (1 \ll (\log_2 TrafoSize - 1))$.
 - The variable $yB1$ is set equal to $yB0 + (1 \ll (\log_2 TrafoSize - 1))$.
 2. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$, the variable $\log_2 TrafoSize$ set equal to $\log_2 TrafoSize - 1$, the variable $trafoDepth$ set equal to $trafoDepth + 1$, the variable $nCbS$, and the $(nCbS) \times (nCbS)$ array $resSamples$ as inputs, and the output is a modified version of the $(nCbS) \times (nCbS)$ array $resSamples$.

3. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B0}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable n_{CbS} , and the $(n_{CbS}) \times (n_{CbS})$ array resSamples as inputs, and the output is a modified version of the $(n_{CbS}) \times (n_{CbS})$ array resSamples .
 4. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B0} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable n_{CbS} , and the $(n_{CbS}) \times (n_{CbS})$ array resSamples as inputs, and the output is a modified version of the $(n_{CbS}) \times (n_{CbS})$ array resSamples .
 5. The decoding process for luma residual blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable n_{CbS} , and the $(n_{CbS}) \times (n_{CbS})$ array resSamples as inputs, and the output is a modified version of the $(n_{CbS}) \times (n_{CbS})$ array resSamples .
- Otherwise ($\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$ is equal to 0), the following ordered steps apply:
1. The variable n_{TbS} is set equal to $1 \ll \log_2\text{TrafoSize}$.
 2. The scaling and transformation process as specified in subclause 8.6.2 is invoked with the luma location ($x_{Cb} + x_{B0}$, $y_{Cb} + y_{B0}$), the variable trafoDepth , the variable $cIdx$ set equal to 0, and the transform size trafoSize set equal to n_{TbS} as inputs, and the output is an $(n_{TbS}) \times (n_{TbS})$ array transformBlock .
 3. The $(n_{CbS}) \times (n_{CbS})$ residual sample array of the current coding block resSamples is modified as follows:
$$\text{resSamples}[x_{B0} + i, y_{B0} + j] = \text{transformBlock}[i, j], \text{ with } i = 0..n_{TbS} - 1, j = 0..n_{TbS} - 1 \quad (8-253)$$

8.5.4.3 Decoding process for chroma residual blocks

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{B0} , y_{B0}) specifying the top-left luma sample of the current chroma block relative to the top-left sample of the current luma coding block,
- a variable $\log_2\text{TrafoSize}$ specifying the size of the current chroma block in luma samples,
- a variable trafoDepth specifying the hierarchy depth of the current chroma block relative to the chroma coding block,
- a variable $cIdx$ specifying the chroma component of the current block,
- a variable n_{CbS} specifying the size of the current chroma coding block,
- an $(n_{CbS}) \times (n_{CbS})$ array resSamples of chroma residual samples.

Output of this process is a modified version of the $(n_{CbS}) \times (n_{CbS})$ array of chroma residual samples.

The variable splitChromaFlag is derived as follows:

- If $\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$ is equal to 1 and $\log_2\text{TrafoSize}$ is greater than 3, splitChromaFlag is set equal to 1.
- Otherwise ($\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$ is equal to 0 or $\log_2\text{TrafoSize}$ is equal to 3), splitChromaFlag is set equal to 0.

Depending on the value of splitChromaFlag , the following applies:

- If splitChromaFlag is equal to 1, the following ordered steps apply:
 1. The variables x_{B1} and y_{B1} are derived as follows:
 - The variable x_{B1} is set equal to $x_{B0} + (1 \ll (\log_2\text{TrafoSize} - 1))$.
 - The variable y_{B1} is set equal to $y_{B0} + (1 \ll (\log_2\text{TrafoSize} - 1))$.
 2. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B0} , y_{B0}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable $cIdx$, the variable n_{CbS} , and the $(n_{CbS}) \times (n_{CbS})$ array resSamples as inputs, and the output is a modified version of the $(n_{CbS}) \times (n_{CbS})$ array resSamples .

3. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B0}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable $cIdx$, the variable $nCbS$, and the $(nCbS) \times (nCbS)$ array resSamples as inputs, and the output is a modified version of the $(nCbS) \times (nCbS)$ array resSamples .
 4. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B0} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable $cIdx$, the variable $nCbS$, and the $(nCbS) \times (nCbS)$ array resSamples as inputs, and the output is a modified version of the $(nCbS) \times (nCbS)$ array resSamples .
 5. The decoding process for residual chroma blocks as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable $cIdx$, the variable $nCbS$, and the $(nCbS) \times (nCbS)$ array resSamples as inputs, and the output is a modified version of the $(nCbS) \times (nCbS)$ array resSamples .
- Otherwise (splitChromaFlag is equal to 0), the following ordered steps apply:
1. The variable $nTbS$ is set equal to $1 \ll (\log_2\text{TrafoSize} - 1)$.
 2. The scaling and transformation process as specified in subclause 8.6.2 is invoked with the luma location ($x_{Cb} + x_{B0}$, $y_{Cb} + y_{B0}$), the variable trafoDepth , the variable $cIdx$, and the transform size trafoSize set equal to $nTbS$ as inputs, and the output is an $(nTbS) \times (nTbS)$ array transformBlock .
 3. The $(nCbS) \times (nCbS)$ residual sample array of the current coding block resSamples is modified as follows, for $i = 0..nTbS - 1$, $j = 0..nTbS - 1$:

$$\text{resSamples}[(x_{Cb} + x_{B0}) / 2 + i, (y_{Cb} + y_{B0}) / 2 + j] = \text{transformBlock}[i, j] \quad (8-254)$$

8.6 Scaling, transformation and array construction process prior to deblocking filter process

8.6.1 Derivation process for quantization parameters

Input to this process is a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture.

In this process, the variable Qp_Y , the luma quantization parameter Qp'_Y , and the chroma quantization parameters Qp'_{Cb} and Qp'_{Cr} are derived.

The luma location (x_{Qg} , y_{Qg}), 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 x_{Qg} and y_{Qg} are set equal to $x_{Cb} - (x_{Cb} \& ((1 \ll \text{Log2MinCuQpDeltaSize}) - 1))$ and $y_{Cb} - (y_{Cb} \& ((1 \ll \text{Log2MinCuQpDeltaSize}) - 1))$, respectively. The luma size of a quantization group, $\text{Log2MinCuQpDeltaSize}$, determines the luma size of the smallest area inside a coding tree block that shares the same qP_{Y_PREV} .

The predicted luma quantization parameter qP_{Y_PREV} is derived by the following ordered steps:

1. The variable qP_{Y_PREV} is derived as follows:
 - If one or more of the following conditions are true, qP_{Y_PREV} is set equal to SliceQp_Y :
 - The current quantization group is the first quantization group in a slice.
 - The current quantization group is the first quantization group in a tile.
 - The current quantization group is the first quantization group in a coding tree block row and $\text{entropy_coding_sync_enabled_flag}$ is equal to 1.
 - Otherwise, qP_{Y_PREV} is set equal to the luma quantization parameter Qp_Y of the last coding unit in the previous quantization group in decoding order.
2. The availability derivation process for a block in z-scan order as specified in subclause 6.4.1 is invoked with the location (x_{Curr} , y_{Curr}) set equal to (x_{Cb} , y_{Cb}) and the neighbouring location (x_{NbY} , y_{NbY}) set equal to ($x_{Qg} - 1$, y_{Qg}) as inputs, and the output is assigned to availableA . The variable qP_{Y_A} is derived as follows:
 - If one or more of the following conditions are true, qP_{Y_A} is set equal to qP_{Y_PREV} :
 - availableA is equal to FALSE.
 - the coding tree block address ctbAddrA of the coding tree block containing the luma coding block covering the luma location ($x_{Qg} - 1$, y_{Qg}) is not equal to CtbAddrInTs , where ctbAddrA is derived as follows:

$$\begin{aligned}
xTmp &= (xQg - 1) \gg \text{Log2MinTrafoSize} \\
yTmp &= yQg \gg \text{Log2MinTrafoSize} \\
minTbAddrA &= MinTbAddrZs[xTmp][yTmp] \\
ctbAddrA &= (minTbAddrA \gg 2) * (\text{CtbLog2SizeY} - \text{Log2MinTrafoSize})
\end{aligned} \quad (8-255)$$

- Otherwise, qP_{Y_A} is set equal to the luma quantization parameter Qp_Y of the coding unit containing the luma coding block covering $(xQg - 1, yQg)$.
- 3. The availability derivation process for a block in z-scan order as specified in subclause 6.4.1 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 qP_{Y_B} is derived as follows:
 - If one or more of the following conditions are true, qP_{Y_B} is set equal to qP_{Y_PREV} :
 - $availableB$ is equal to FALSE.
 - the coding tree block address $ctbAddrB$ of the coding tree block containing the luma coding block covering the luma location $(xQg, yQg - 1)$ is not equal to $ctbAddrInTs$, where $ctbAddrB$ is derived as follows:

$$\begin{aligned}
xTmp &= xQg \gg \text{Log2MinTrafoSize} \\
yTmp &= (yQg - 1) \gg \text{Log2MinTrafoSize} \\
minTbAddrB &= MinTbAddrZs[xTmp][yTmp] \\
ctbAddrB &= (minTbAddrB \gg 2) * (\text{CtbLog2SizeY} - \text{Log2MinTrafoSize})
\end{aligned} \quad (8-256)$$

- Otherwise, qP_{Y_B} is set equal to the luma quantization parameter Qp_Y of the coding unit containing the luma coding block covering $(xQg, yQg - 1)$.
- 4. The predicted luma quantization parameter qP_{Y_PRED} is derived as follows:

$$qP_{Y_PRED} = (qP_{Y_A} + qP_{Y_B} + 1) \gg 1 \quad (8-257)$$

The variable Qp_Y is derived as follows:

$$Qp_Y = ((qP_{Y_PRED} + CuQpDeltaVal + 52 + 2 * QpBdOffset_Y) \% (52 + QpBdOffset_Y)) - QpBdOffset_Y \quad (8-258)$$

The luma quantization parameter Qp'_Y is derived as follows:

$$Qp'_Y = Qp_Y + QpBdOffset_Y \quad (8-259)$$

The variables qP_{Cb} and qP_{Cr} are set equal to the value of Qp_C as specified in Table 8-9 based on the index qPi equal to qPi_{Cb} and qPi_{Cr} , respectively, and qPi_{Cb} and qPi_{Cr} are derived as follows:

$$qPi_{Cb} = \text{Clip3}(-QpBdOffset_C, 57, Qp_Y + pps_cb_qp_offset + slice_cb_qp_offset) \quad (8-260)$$

$$qPi_{Cr} = \text{Clip3}(-QpBdOffset_C, 57, Qp_Y + pps_cr_qp_offset + slice_cr_qp_offset) \quad (8-261)$$

The chroma quantization parameters for the Cb and Cr components, Qp'_{Cb} and Qp'_{Cr} , are derived as follows:

$$Qp'_{Cb} = qP_{Cb} + QpBdOffset_C \quad (8-262)$$

$$Qp'_{Cr} = qP_{Cr} + QpBdOffset_C \quad (8-263)$$

Table 8-9 – Specification of Qp_C as a function of qPi

qPi	< 30	30	31	32	33	34	35	36	37	38	39	40	41	42	43	> 43
Qp_C	= qPi	29	30	31	32	33	33	34	34	35	35	36	36	37	37	= $qPi - 6$

8.6.2 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 $trafoDepth$ specifying the hierarchy depth of the current block relative to the coding block,
- a variable $cIdx$ specifying the colour component of the current block,
- a variable $nTbS$ specifying the size of the current transform block.

Output of this process is the (nTbS)x(nTbS) array of residual samples r with elements $r[x][y]$.

The quantization parameter qP is derived as follows:

- If $cIdx$ is equal to 0,

$$qP = Qp'_Y \quad (8-264)$$

- Otherwise, if $cIdx$ is equal to 1,

$$qP = Qp'_{Cb} \quad (8-265)$$

- Otherwise ($cIdx$ is equal to 2),

$$qP = Qp'_{Cr} \quad (8-266)$$

The (nTbS)x(nTbS) array of residual samples r is derived as follows:

- If $cu_transquant_bypass_flag$ is equal to 1, the (nTbS)x(nTbS) array r is set equal to the (nTbS)x(nTbS) array of transform coefficients $TransCoeffLevel[xTbY][yTbY][cIdx]$.

- Otherwise, the following ordered steps apply:

1. The scaling process for transform coefficients as specified in subclause 8.6.3 is invoked with the transform block location ($xTbY, yTbY$), the size of the transform block $nTbS$, the colour component variable $cIdx$, and the quantization parameter qP as inputs, and the output is an (nTbS)x(nTbS) array of scaled transform coefficients d .

2. The (nTbS)x(nTbS) array of residual samples r is derived as follows:

- If $transform_skip_flag[xTbY][yTbY][cIdx]$ is equal to 1, the residual sample array values $r[x][y]$ with $x = 0..nTbS - 1, y = 0..nTbS - 1$ are derived as follows:

$$r[x][y] = (d[x][y] \ll 7) \quad (8-267)$$

- Otherwise ($transform_skip_flag[xTbY][yTbY][cIdx]$ is equal to 0), the transformation process for scaled transform coefficients as specified in subclause 8.6.4 is invoked with the transform block location ($xTbY, yTbY$), the size of the transform block $nTbS$, the colour component variable $cIdx$, and the (nTbS)x(nTbS) array of scaled transform coefficients d as inputs, and the output is an (nTbS)x(nTbS) array of residual samples r .

3. The variable $bdShift$ is derived as follows:

$$bdShift = (cIdx == 0) ? 20 - BitDepth_Y : 20 - BitDepth_C \quad (8-268)$$

4. The residual sample values $r[x][y]$ with $x = 0..nTbS - 1, y = 0..nTbS - 1$ are modified as follows:

$$r[x][y] = (r[x][y] + (1 \ll (bdShift - 1))) \gg bdShift \quad (8-269)$$

8.6.3 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 $nTbS$ specifying the size of the current transform block,
- a variable $cIdx$ specifying the colour component of the current block,
- a variable qP specifying the quantization parameter.

Output of this process is the (nTbS)x(nTbS) array d of scaled transform coefficients with elements $d[x][y]$.

The variable $bdShift$ is derived as follows:

- If $cIdx$ is equal to 0,

$$bdShift = BitDepth_Y + \log_2(nTbS) - 5 \quad (8-270)$$

- Otherwise,

$$bdShift = BitDepth_C + \log_2(nTbS) - 5 \quad (8-271)$$

The list $levelScale[k]$ is specified as $levelScale[k] = \{ 40, 45, 51, 57, 64, 72 \}$ with $k = 0..5$.

For the derivation of the scaled transform coefficients $d[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$, the following applies:

- The scaling factor $m[x][y]$ is derived as follows:

- If `scaling_list_enabled_flag` is equal to 0,

$$m[x][y] = 16 \quad (8-272)$$

- Otherwise (`scaling_list_enabled_flag` is equal to 1),

$$m[x][y] = \text{ScalingFactor}[\text{sizeId}][\text{matrixId}][x][y] \quad (8-273)$$

Where `sizeId` is specified in Table 7-3 for the size of the quantization matrix equal to $(nTbS) \times (nTbS)$ and `matrixId` is specified in Table 7-4 for `sizeId`, `CuPredMode[xTbY][yTbY]`, and `cIdx`, respectively.

- The scaled transform coefficient $d[x][y]$ is derived as follows:

$$d[x][y] = \text{Clip3}(-32768, 32767, ((\text{TransCoeffLevel}[xTbY][yTbY][cIdx][x][y] * m[x][y] * \text{levelScale}[qP\%6] << (qP / 6)) + (1 << (bdShift - 1))) >> bdShift) \quad (8-274)$$

8.6.4 Transformation process for scaled transform coefficients

8.6.4.1 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 `nTbS` specifying the size of the current transform block,
- a variable `cIdx` specifying the colour component of the current block,
- an $(nTbS) \times (nTbS)$ array d of scaled transform coefficients with elements $d[x][y]$.

Output of this process is the $(nTbS) \times (nTbS)$ array r of residual samples with elements $r[x][y]$.

Depending on the values of `CuPredMode[xTbY][yTbY]`, `nTbS`, and `cIdx`, the variable `trType` is derived as follows:

- If `CuPredMode[xTbY][yTbY]` is equal to `MODE_INTRA`, `nTbS` is equal to 4, and `cIdx` is equal to 0, `trType` is set equal to 1.
- Otherwise, `trType` is set equal to 0.

The $(nTbS) \times (nTbS)$ array r of residual samples is derived as follows:

1. Each (vertical) column of scaled transform coefficients $d[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$ is transformed to $e[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$ by invoking the one-dimensional transformation process as specified in subclause 8.6.4.2 for each column $x = 0..nTbS - 1$ with the size of the transform block `nTbS`, the list $d[x][y]$ with $y = 0..nTbS - 1$, and the transform type variable `trType` as inputs, and the output is the list $e[x][y]$ with $y = 0..nTbS - 1$.

2. The intermediate sample values $g[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$ are derived as follows:

$$g[x][y] = \text{Clip3}(-32768, 32767, (e[x][y] + 64) >> 7) \quad (8-275)$$

3. Each (horizontal) row of the resulting array $g[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$ is transformed to $r[x][y]$ with $x = 0..nTbS - 1$, $y = 0..nTbS - 1$ by invoking the one-dimensional transformation process as specified in subclause 8.6.4.2 for each row $y = 0..nTbS - 1$ with the size of the transform block `nTbS`, the list $g[x][y]$ with $x = 0..nTbS - 1$, and the transform type variable `trType` as inputs, and the output is the list $r[x][y]$ with $x = 0..nTbS - 1$.

8.6.4.2 Transformation process

Inputs to this process are:

- a variable `nTbS` specifying the sample size of scaled transform coefficients,
- a list of scaled transform coefficients x with elements $x[j]$, with $j = 0..nTbS - 1$.
- a transform type variable `trType`

Output of this process is the list of transformed samples y with elements $y[i]$, with $i = 0..nTbS - 1$.

Depending on the value of trType, the following applies:

- If trType is equal to 1, the following transform matrix multiplication applies:

$$y[i] = \sum_{j=0}^{nTbS-1} \text{transMatrix}[i][j] * x[j] \text{ with } i = 0..nTbS - 1 \quad (8-276)$$

where the transform coefficient array transMatrix is specified as follows:

$$\text{transMatrix} = \begin{Bmatrix} \{29 & 55 & 74 & 84\} \\ \{74 & 74 & 0 & -74\} \\ \{84 & -29 & -74 & 55\} \\ \{55 & -84 & 74 & -29\} \end{Bmatrix} \quad (8-277)$$

- Otherwise (trType is equal to 0), the following transform matrix multiplication applies:

$$y[i] = \sum_{j=0}^{nTbS-1} \text{transMatrix}[i][j * 2^{5-\text{Log}_2(nTbS)}] * x[j] \text{ with } i = 0..nTbS - 1, \quad (8-278)$$

where the transform coefficient array transMatrix is specified as follows:

$$\text{transMatrix}[m][n] = \text{transMatrixCol0to15}[m][n] \text{ with } m = 0..15, n = 0..31 \quad (8-279)$$

$$\text{transMatrixCol0to15} = \begin{Bmatrix} \{64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64\} \\ \{90 & 90 & 88 & 85 & 82 & 78 & 73 & 67 & 61 & 54 & 46 & 38 & 31 & 22 & 13 & 4\} \\ \{90 & 87 & 80 & 70 & 57 & 43 & 25 & 9 & -9 & -25 & -43 & -57 & -70 & -80 & -87 & -90\} \\ \{90 & 82 & 67 & 46 & 22 & -4 & -31 & -54 & -73 & -85 & -90 & -88 & -78 & -61 & -38 & -13\} \\ \{89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89\} \\ \{88 & 67 & 31 & -13 & -54 & -82 & -90 & -78 & -46 & -4 & 38 & 73 & 90 & 85 & 61 & 22\} \\ \{87 & 57 & 9 & -43 & -80 & -90 & -70 & -25 & 25 & 70 & 90 & 80 & 43 & -9 & -57 & -87\} \\ \{85 & 46 & -13 & -67 & -90 & -73 & -22 & 38 & 82 & 88 & 54 & -4 & -61 & -90 & -78 & -31\} \\ \{83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & 36 & 36 & 83\} \\ \{82 & 22 & -54 & -90 & -61 & 13 & 78 & 85 & 31 & -46 & -90 & -67 & 4 & 73 & 88 & 38\} \\ \{80 & 9 & -70 & -87 & -25 & 57 & 90 & 43 & -43 & -90 & -57 & 25 & 87 & 70 & -9 & -80\} \\ \{78 & -4 & -82 & -73 & 13 & 85 & 67 & -22 & 88 & -61 & 31 & 90 & 54 & -38 & -90 & -46\} \\ \{75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & 75 & 18 & 89 & 50 & -50 & -89 & -18 & 75\} \\ \{73 & -31 & -90 & -22 & 78 & 67 & -38 & -90 & -13 & 82 & 61 & -46 & -88 & -4 & 85 & 54\} \\ \{70 & -43 & -87 & 9 & 90 & 25 & -80 & -57 & 57 & 80 & -25 & -90 & -9 & 87 & 43 & -70\} \\ \{67 & -54 & -78 & 38 & 85 & -22 & -90 & 4 & 90 & 13 & -88 & -31 & 82 & 46 & -73 & -61\} \\ \{64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64\} \\ \{61 & -73 & -46 & 82 & 31 & -88 & -13 & 90 & -4 & -90 & 22 & 85 & -38 & -78 & 54 & 67\} \\ \{57 & -80 & -25 & 90 & -9 & -87 & 43 & 70 & -70 & -43 & 87 & 9 & -90 & 25 & 80 & -57\} \\ \{54 & -85 & -4 & 88 & -46 & -61 & 82 & 13 & -90 & 38 & 67 & -78 & -22 & 90 & -31 & -73\} \\ \{50 & -89 & 18 & 75 & 75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50\} \\ \{46 & -90 & 38 & 54 & -90 & 31 & 61 & -88 & 22 & 67 & -85 & 13 & 73 & -82 & 4 & 78\} \\ \{43 & -90 & 57 & 25 & -87 & 70 & 9 & -80 & 80 & -9 & -70 & 87 & -25 & -57 & 90 & -43\} \\ \{38 & -88 & 73 & -4 & -67 & 90 & -46 & -31 & 85 & -78 & 13 & 61 & -90 & 54 & 22 & -82\} \\ \{36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36\} \\ \{31 & -78 & 90 & -61 & 4 & 54 & -88 & 82 & -38 & -22 & 73 & -90 & 67 & -13 & -46 & 85\} \\ \{25 & -70 & 90 & -80 & 43 & 9 & -57 & 87 & -87 & 57 & -9 & -43 & 80 & -90 & 70 & -25\} \\ \{22 & -61 & 85 & -90 & 73 & -38 & -4 & 46 & -78 & 90 & -82 & 54 & -13 & -31 & 67 & -88\} \\ \{18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18\} \\ \{13 & -38 & 61 & -78 & 88 & -90 & 85 & -73 & 54 & -31 & 4 & 22 & -46 & 67 & -82 & 90\} \\ \{9 & -25 & 43 & -57 & 70 & -80 & 87 & -90 & 90 & -87 & 80 & -70 & 57 & -43 & 25 & -9\} \\ \{4 & -13 & 22 & -31 & 38 & -46 & 54 & -61 & 67 & -73 & 78 & -82 & 85 & -88 & 90 & -90\} \end{Bmatrix}, \quad (8-280)$$

transMatrix[m][n] = transMatrixCol16to31[m - 16][n] with m = 16..31, n = 0..31, (8-281)

transMatrixCol16to31 = (8-282)

```
{
{ 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 }
{ -4 -13 -22 -31 -38 -46 -54 -61 -67 -73 -78 -82 -85 -88 -90 -90 }
{ -90 -87 -80 -70 -57 -43 -25 -9 9 25 43 57 70 80 87 90 }
{ 13 38 61 78 88 90 85 73 54 31 4 -22 -46 -67 -82 -90 }
{ 89 75 50 18 -18 -50 -75 -89 -89 -75 -50 -18 18 50 75 89 }
{ -22 -61 -85 -90 -73 -38 4 46 78 90 82 54 13 -31 -67 -88 }
{ -87 -57 -9 43 80 90 70 25 -25 -70 -90 -80 -43 9 57 87 }
{ 31 78 90 61 4 -54 -88 -82 -38 22 73 90 67 13 -46 -85 }
{ 83 36 -36 -83 -83 -36 36 83 83 36 -36 -83 -83 -36 36 83 }
{ -38 -88 -73 -4 67 90 46 -31 -85 -78 -13 61 90 54 -22 -82 }
{ -80 -9 70 87 25 -57 -90 -43 43 90 57 -25 -87 -70 9 80 }
{ 46 90 38 -54 -90 -31 61 88 22 -67 -85 -13 73 82 4 -78 }
{ 75 -18 -89 -50 50 89 18 -75 -75 18 89 50 -50 -89 -18 75 }
{ -54 -85 4 88 46 -61 -82 13 90 38 -67 -78 22 90 31 -73 }
{ -70 43 87 -9 -90 -25 80 57 -57 -80 25 90 9 -87 -43 70 }
{ 61 73 -46 -82 31 88 -13 -90 -4 90 22 -85 -38 78 54 -67 }
{ 64 -64 -64 64 64 -64 -64 64 64 -64 64 64 -64 64 64 }
{ -67 -54 78 38 -85 -22 90 4 -90 13 88 -31 -82 46 73 -61 }
{ -57 80 25 -90 9 87 -43 -70 70 43 -87 -9 90 -25 -80 57 }
{ 73 31 -90 22 78 -67 -38 90 -13 -82 61 46 -88 4 85 -54 }
{ 50 -89 18 75 -75 -18 89 -50 -50 89 -18 -75 75 18 -89 50 }
{ -78 -4 82 -73 -13 85 -67 -22 88 -61 -31 90 -54 -38 90 -46 }
{ -43 90 -57 -25 87 -70 -9 80 -80 9 70 -87 25 57 -90 43 }
{ 82 -22 -54 90 -61 -13 78 -85 31 46 -90 67 4 -73 88 -38 }
{ 36 -83 83 -36 -36 83 -83 36 36 -83 83 -36 -36 83 -83 36 }
{ -85 46 13 -67 90 -73 22 38 -82 88 -54 -4 61 -90 78 -31 }
{ -25 70 -90 80 -43 -9 57 -87 87 -57 9 43 -80 90 -70 25 }
{ 88 -67 31 13 -54 82 -90 78 -46 4 38 -73 90 -85 61 -22 }
{ 18 -50 75 -89 89 -75 50 -18 -18 50 -75 89 -89 75 -50 18 }
{ -90 82 -67 46 -22 -4 31 -54 73 -85 90 -88 78 -61 38 -13 }
{ -9 25 -43 57 -70 80 -87 90 -90 87 -80 70 -57 43 -25 9 }
{ 90 -90 88 -85 82 -78 73 -67 61 -54 46 -38 31 -22 13 -4 }
}
```

8.6.5 Picture construction process prior to in-loop filter process

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,
- a variable nCurrS specifying the size of the current block,
- a variable cIdx specifying the colour component of the current block,
- an (nCurrS)x(nCurrS) array predSamples specifying the predicted samples of the current block,
- an (nCurrS)x(nCurrS) array resSamples specifying the residual samples of the current block.

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 S_L and the function clipCidx1 corresponds to Clip1_Y.
- Otherwise, if cIdx is equal to 1, recSamples corresponds to the reconstructed chroma sample array S_{Cb} and the function clipCidx1 corresponds to Clip1_C.
- Otherwise (cIdx is equal to 2), recSamples corresponds to the reconstructed chroma sample array S_{Cr} and the function clipCidx1 corresponds to Clip1_C.

The (nCurrS)x(nCurrS) block of the reconstructed sample array recSamples at location (xCurr, yCurr) is derived as follows:

$$\text{recSamples}[\text{xCurr} + i][\text{yCurr} + j] = \text{clipCidx1}(\text{predSamples}[i][j] + \text{resSamples}[i][j]) \quad (8-283)$$

with $i = 0..n\text{CurrS} - 1, j = 0..n\text{CurrS} - 1$

8.7 In-loop filter process

8.7.1 General

The two in-loop filters, namely deblocking filter and sample adaptive offset filter, are applied as specified by the following ordered steps:

1. For the deblocking filter, the following applies:
 - The deblocking filter process as specified in subclause 8.7.2 is invoked with the reconstructed picture sample arrays S_L , S_{Cb} , and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L , S'_{Cb} , and S'_{Cr} after deblocking as outputs.
 - The arrays S'_L , S'_{Cb} , S'_{Cr} are assigned to the arrays S_L , S_{Cb} , S_{Cr} (which represent the decoded picture), respectively.
2. When `sample_adaptive_offset_enabled_flag` is equal to 1, the following applies:
 - The sample adaptive offset process as specified in subclause 8.7.3 is invoked with the reconstructed picture sample arrays S_L , S_{Cb} , and S_{Cr} as inputs, and the modified reconstructed picture sample arrays S'_L , S'_{Cb} , and S'_{Cr} after sample adaptive offset as outputs.
 - The arrays S'_L , S'_{Cb} , S'_{Cr} are assigned to the arrays S_L , S_{Cb} , S_{Cr} (which represent the decoded picture), respectively.

8.7.2 Deblocking filter process

8.7.2.1 General

Inputs to this process are the reconstructed picture sample arrays prior to deblocking recPicture_L , recPicture_{Cb} , and recPicture_{Cr} .

Outputs of this process are the modified reconstructed picture sample arrays after deblocking recPicture_L , recPicture_{Cb} , and recPicture_{Cr} .

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 coding tree blocks of each coding tree unit are processed separately on a coding unit basis. The vertical edges of the coding blocks in a coding unit are filtered starting with the edge on the left-hand side of the coding blocks proceeding through the edges towards the right-hand side of the coding blocks in their geometrical order. The horizontal edges of the coding blocks in a coding unit are filtered starting with the edge on the top of the coding blocks proceeding through the edges towards the bottom of the coding blocks in their geometrical order.

NOTE – Although the filtering process is specified on a picture basis in this specification, the filtering process can be implemented on a coding unit basis with an equivalent result, 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 prediction block edges and transform block edges of a picture, except the edges that are at the boundary of the picture, for which the deblocking filter process is disabled by `slice_deblocking_filter_disabled_flag`, that coincide with tile boundaries when `loop_filter_across_tiles_enabled_flag` is equal to 0, or that coincide with upper or left slice boundaries of slices with `slice_loop_filter_across_slices_enabled_flag` equal to 0. For the transform units and prediction units with luma block edges less than 8 samples in either vertical or horizontal direction, only the edges lying on the 8x8 sample grid are filtered.

The edge type, vertical or horizontal, is represented by the variable `edgeType` as specified in Table 8-10.

Table 8-10 – 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, for each coding unit with luma coding block size $\log_2\text{CbSize}$ and location of top-left sample of the luma coding block (x_{Cb} , y_{Cb}), the vertical edges are filtered by the following ordered steps:

1. The luma coding block size n_{CbS} is set equal to $1 \ll \log_2\text{CbSize}$.
2. The variable `filterLeftCbEdgeFlag` is derived as follows:
 - If one or more of the following conditions are true, `filterLeftCbEdgeFlag` is set equal to 0:
 - The left boundary of the current luma coding block is the left boundary of the picture.
 - The left boundary of the current luma coding block is the left boundary of the tile and `loop_filter_across_tiles_enabled_flag` is equal to 0.

- The left boundary of the current luma coding block is the left boundary of the slice and `slice_loop_filter_across_slices_enabled_flag` is equal to 0.
 - Otherwise, `filterLeftCbEdgeFlag` is set equal to 1.
3. All elements of the two-dimensional $(nCbS) \times (nCbS)$ array `verEdgeFlags` are initialized to be equal to zero.
 4. The derivation process of transform block boundary specified in subclause 8.7.2.2 is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$ set equal to $(0, 0)$, the transform block size `log2TrafoSize` set equal to `log2CbSize`, the variable `trafoDepth` set equal to 0, the variable `filterLeftCbEdgeFlag`, the array `verEdgeFlags`, and the variable `edgeType` set equal to `EDGE_VER` as inputs, and the modified array `verEdgeFlags` as output.
 5. The derivation process of prediction block boundary specified in subclause 8.7.2.3 is invoked with the luma coding block size `log2CbSize`, the prediction partition mode `PartMode`, the array `verEdgeFlags`, and the variable `edgeType` set equal to `EDGE_VER` as inputs, and the modified array `verEdgeFlags` as output.
 6. The derivation process of the boundary filtering strength specified in subclause 8.7.2.4 is invoked with the reconstructed luma picture sample array prior to deblocking `recPictureL`, the luma location (xCb, yCb) , the luma coding block size `log2CbSize`, the variable `edgeType` set equal to `EDGE_VER`, and the array `verEdgeFlags` as inputs, and an $(nCbS) \times (nCbS)$ array `verBs` as output.
 7. The vertical edge filtering process for a coding unit as specified in subclause 8.7.2.5.1 is invoked with the reconstructed picture sample arrays prior to deblocking `recPictureL`, `recPictureCb`, and `recPictureCr`, the luma location (xCb, yCb) , the luma coding block size `log2CbSize`, and the array `verBs` as inputs, and the modified reconstructed picture sample arrays `recPictureL`, `recPictureCb`, and `recPictureCr` as outputs.

When `slice_deblocking_filter_disabled_flag` of the current slice is equal to 0, for each coding unit with luma coding block size `log2CbSize` and location of top-left sample of the luma coding block (xCb, yCb) , the horizontal edges are filtered by the following ordered steps:

1. The luma coding block size `nCbS` is set equal to $1 \ll \log2CbSize$.
2. The variable `filterTopCbEdgeFlag` is derived as follows:
 - If one or more of the following conditions are true, the variable `filterTopCbEdgeFlag` 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 luma coding block is the top boundary of the tile and `loop_filter_across_tiles_enabled_flag` is equal to 0.
 - The top boundary of the current luma coding block is the top boundary of the slice and `slice_loop_filter_across_slices_enabled_flag` is equal to 0.
 - Otherwise, the variable `filterTopCbEdgeFlag` is set equal to 1.
3. All elements of the two-dimensional $(nCbS) \times (nCbS)$ array `horEdgeFlags` are initialized to zero.
4. The derivation process of transform block boundary specified in subclause 8.7.2.2 is invoked with the luma location (xCb, yCb) , the luma location $(xB0, yB0)$ set equal to $(0, 0)$, the transform block size `log2TrafoSize` set equal to `log2CbSize`, the variable `trafoDepth` set equal to 0, the variable `filterTopCbEdgeFlag`, the array `horEdgeFlags`, and the variable `edgeType` set equal to `EDGE_HOR` as inputs, and the modified array `horEdgeFlags` as output.
5. The derivation process of prediction block boundary specified in subclause 8.7.2.3 is invoked with the luma coding block size `log2CbSize`, the prediction partition mode `PartMode`, the array `horEdgeFlags`, and the variable `edgeType` set equal to `EDGE_HOR` as inputs, and the modified array `horEdgeFlags` as output.
6. The derivation process of the boundary filtering strength specified in subclause 8.7.2.4 is invoked with the reconstructed luma picture sample array prior to deblocking `recPictureL`, the luma location (xCb, yCb) , the luma coding block size `log2CbSize`, the variable `edgeType` set equal to `EDGE_HOR`, and the array `horEdgeFlags` as inputs, and an $(nCbS) \times (nCbS)$ array `horBs` as output.
7. The horizontal edge filtering process for a coding unit as specified in subclause 8.7.2.5.2 is invoked with the modified reconstructed picture sample arrays `recPictureL`, `recPictureCb`, and `recPictureCr`, the luma location (xCb, yCb) , the luma coding block size `log2CbSize` and the array `horBs` as inputs, and the modified reconstructed picture sample arrays `recPictureL`, `recPictureCb`, and `recPictureCr` as outputs.

8.7.2.2 Derivation process of transform block boundary

Inputs to this process are:

- a luma location (x_{Cb} , y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a luma location (x_{B0} , y_{B0}) specifying the top-left sample of the current luma block relative to the top-left sample of the current luma coding block,
- a variable $\log_2\text{TrafoSize}$ specifying the size of the current block,
- a variable trafoDepth ,
- a variable filterEdgeFlag ,
- a two-dimensional $(n_{CbS}) \times (n_{CbS})$ array edgeFlags ,
- a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional $(n_{CbS}) \times (n_{CbS})$ array edgeFlags .

Depending on the value of $\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$, the following applies:

- If $\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$ is equal to 1, the following ordered steps apply:
 1. The variables x_{B1} and y_{B1} are derived as follows:
 - The variable x_{B1} is set equal to $x_{B0} + (1 \ll (\log_2\text{TrafoSize} - 1))$.
 - The variable y_{B1} is set equal to $y_{B0} + (1 \ll (\log_2\text{TrafoSize} - 1))$.
 2. The derivation process of transform block boundary as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B0} , y_{B0}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable filterEdgeFlag , the array edgeFlags , and the variable edgeType as inputs, and the output is the modified version of array edgeFlags .
 3. The derivation process of transform block boundary as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B0}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable filterEdgeFlag , the array edgeFlags , and the variable edgeType as inputs, and the output is the modified version of array edgeFlags .
 4. The derivation process of transform block boundary as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B0} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable filterEdgeFlag , the array edgeFlags , and the variable edgeType as inputs, and the output is the modified version of array edgeFlags .
 5. The derivation process of transform block boundary as specified in this subclause is invoked with the luma location (x_{Cb} , y_{Cb}), the luma location (x_{B1} , y_{B1}), the variable $\log_2\text{TrafoSize}$ set equal to $\log_2\text{TrafoSize} - 1$, the variable trafoDepth set equal to $\text{trafoDepth} + 1$, the variable filterEdgeFlag , the array edgeFlags , and the variable edgeType as inputs, and the output is the modified version of array edgeFlags .
- Otherwise ($\text{split_transform_flag}[x_{Cb} + x_{B0}][y_{Cb} + y_{B0}][\text{trafoDepth}]$ is equal to 0), the following applies:
 - If edgeType is equal to EDGE_VER, the value of $\text{edgeFlags}[x_{B0}][y_{B0} + k]$ for $k = 0..(1 \ll \log_2\text{TrafoSize}) - 1$ is derived as follows:
 - If x_{B0} is equal to 0, $\text{edgeFlags}[x_{B0}][y_{B0} + k]$ is set equal to filterEdgeFlag .
 - Otherwise, $\text{edgeFlags}[x_{B0}][y_{B0} + k]$ is set equal to 1.
 - Otherwise (edgeType is equal to EDGE_HOR), the value of $\text{edgeFlags}[x_{B0} + k][y_{B0}]$ for $k = 0..(1 \ll \log_2\text{TrafoSize}) - 1$ is derived as follows:
 - If y_{B0} is equal to 0, $\text{edgeFlags}[x_{B0} + k][y_{B0}]$ is set equal to filterEdgeFlag .
 - Otherwise, $\text{edgeFlags}[x_{B0} + k][y_{B0}]$ is set equal to 1.

8.7.2.3 Derivation process of prediction block boundary

Inputs to this process are:

- a variable $\log_2\text{CbSize}$ specifying the luma coding block size,
- a prediction partition mode PartMode ,
- a two-dimensional $(n_{CbS}) \times (n_{CbS})$ array edgeFlags ,
- a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered.

Output of this process is the modified two-dimensional (nCbs)x(nCbs) array edgeFlags.

Depending on the values of edgeType and PartMode, the following applies for $k = 0..(1 \ll \log_2 \text{CbSize}) - 1$:

- If edgeType is equal to EDGE_VER, the following applies:
 - When PartMode is equal to PART_Nx2N or PART_NxN, edgeFlags[$1 \ll (\log_2 \text{CbSize} - 1) \ll k$] is set equal to 1.
 - When PartMode is equal to PART_nLx2N, edgeFlags[$1 \ll (\log_2 \text{CbSize} - 2) \ll k$] is set equal to 1.
 - When PartMode is equal to PART_nRx2N, edgeFlags[$3 * (1 \ll (\log_2 \text{CbSize} - 2)) \ll k$] is set equal to 1.
- Otherwise (edgeType is equal to EDGE_HOR), the following applies:
 - When PartMode is equal to PART_2NxN or PART_NxN, edgeFlags[$k \ll (1 \ll (\log_2 \text{CbSize} - 1))$] is set equal to 1.
 - When PartMode is equal to PART_2NxN, edgeFlags[$k \ll (1 \ll (\log_2 \text{CbSize} - 2))$] is set equal to 1.
 - When PartMode is equal to PART_2NxN, edgeFlags[$k \ll (3 * (1 \ll (\log_2 \text{CbSize} - 2)))$] is set equal to 1.

8.7.2.4 Derivation process of boundary filtering strength

Inputs to this process are:

- a luma picture sample array recPicture_L,
- 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 log2CbSize specifying the size of the current luma coding block,
- a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,
- a two-dimensional (nCbs)x(nCbs) array edgeFlags.

Output of this process is a two-dimensional (nCbs)x(nCbs) 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 is set equal to $(i \ll 3)$, yDj is set equal to $(j \ll 2)$, xN is set equal to $(1 \ll (\log_2 \text{CbSize} - 3)) - 1$, and yN is set equal to $(1 \ll (\log_2 \text{CbSize} - 2)) - 1$.
- Otherwise (edgeType is equal to EDGE_HOR), xDi is set equal to $(i \ll 2)$, yDj is set equal to $(j \ll 3)$, xN is set equal to $(1 \ll (\log_2 \text{CbSize} - 2)) - 1$, and yN is set equal to $(1 \ll (\log_2 \text{CbSize} - 3)) - 1$.

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 (edgeFlags[xDi][yDj] is equal to 1), 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_L[xCb + xDi - 1][yCb + yDj] and q₀ is set equal to recPicture_L[xCb + xDi][yCb + yDj].
 - Otherwise (edgeType is equal to EDGE_HOR), p₀ is set equal to recPicture_L[xCb + xDi][yCb + yDj - 1] and q₀ is set equal to recPicture_L[xCb + xDi][yCb + yDj].
 - The variable bS[xDi][yDj] is derived as follows:
 - If the sample p₀ or q₀ is in the luma coding block of a coding unit coded with intra prediction mode, bS[xDi][yDj] is set equal to 2.
 - Otherwise, if the block edge is also a transform block edge and the sample p₀ or q₀ is in a luma transform block which contains one or more non-zero transform coefficient levels, bS[xDi][yDj] is set equal to 1.
 - Otherwise, if one or more of the following conditions are true, bS[xDi][yDj] is set equal to 1:
 - For the prediction of the luma prediction block containing the sample p₀ different reference pictures or a different number of motion vectors are used than for the prediction of the luma prediction block containing the sample q₀.

NOTE 1 – The determination of whether the reference pictures used for the two luma prediction blocks are the same or different is based only on which pictures are referenced, without regard to whether a prediction is formed using an index into reference picture list 0 or an index into reference picture list 1, and also without regard to whether the index position within a reference picture list is different.

NOTE 2 – The number of motion vectors that are used for the prediction of a luma prediction block with top-left luma sample covering (xPb, yPb) , is equal to $PredFlagL0[xPb][yPb] + PredFlagL1[xPb][yPb]$.

- One motion vector is used to predict the luma prediction block containing the sample p_0 and one motion vector is used to predict the luma prediction block containing the sample q_0 , 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 luma prediction block containing the sample p_0 , two motion vectors for the same two reference pictures are used to predict the luma prediction block containing the sample q_0 , and the absolute difference between the horizontal or vertical component of the two motion vectors used in the prediction of the two luma prediction blocks for the same reference picture is greater than or equal to 4 in units of quarter luma samples.
- Two motion vectors for the same reference picture are used to predict the luma prediction block containing the sample p_0 , two motion vectors for the same reference picture are used to predict the luma prediction block containing the sample q_0 , 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 luma prediction blocks 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 luma prediction blocks 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 luma prediction block containing the sample p_0 and the list 1 motion vector used in the prediction of the luma prediction block containing the sample q_0 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 luma prediction block containing the sample p_0 and list 0 motion vector used in the prediction of the luma prediction block containing the sample q_0 is greater than or equal to 4 in units of quarter luma samples.
- Otherwise, the variable $bS[xD_i][yD_j]$ is set equal to 0.

8.7.2.5 Edge filtering process

8.7.2.5.1 Vertical edge filtering process

Inputs to this process are:

- the picture sample arrays $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$,
- 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 $\log2CbSize$ specifying the size of the current luma coding block,
- an array bS specifying the boundary filtering strength.

Outputs of this process are the modified picture sample arrays $recPicture_L$, $recPicture_{Cb}$, and $recPicture_{Cr}$.

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to $1 \ll (\log2CbSize - 3)$.
2. For xD_k equal to $k \ll 3$ with $k = 0..nD - 1$ and yD_m equal to $m \ll 2$ with $m = 0..nD * 2 - 1$, the following applies:
 - When $bS[xD_k][yD_m]$ is greater than 0, the following ordered steps apply:

- a. The decision process for luma block edges as specified in subclause 8.7.2.5.3 is invoked with the luma picture sample array recPicture_L , the location of the luma coding block (x_{Cb}, y_{Cb}) , the luma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_VER , and the boundary filtering strength $\text{bS}[x_{D_k}][y_{D_m}]$ as inputs, and the decisions dE , dEp , and dEq , and the variables β and t_c as outputs.
- b. The filtering process for luma block edges as specified in subclause 8.7.2.5.4 is invoked with the luma picture sample array recPicture_L , the location of the luma coding block (x_{Cb}, y_{Cb}) , the luma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_VER , the decisions dE , dEp , and dEq , and the variables β and t_c as inputs, and the modified luma picture sample array recPicture_L as output.

The filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable nD is set equal to $1 \ll (\log_2 \text{CbSize} - 3)$.
2. For x_{D_k} equal to $k \ll 2$ with $k = 0..nD - 1$ and y_{D_m} equal to $m \ll 2$ with $m = 0..nD - 1$, the following applies:
 - When $\text{bS}[x_{D_k} * 2][y_{D_m} * 2]$ is equal to 2 and $((x_{Cb} / 2 + x_{D_k}) \gg 3) \ll 3$ is equal to $x_{Cb} / 2 + x_{D_k}$, the following ordered steps apply:
 - a. The filtering process for chroma block edges as specified in subclause 8.7.2.5.5 is invoked with the chroma picture sample array recPicture_{Cb} , the location of the chroma coding block $(x_{Cb} / 2, y_{Cb} / 2)$, the chroma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_VER , and a variable cQpPicOffset set equal to pps_cb_qp_offset as inputs, and the modified chroma picture sample array recPicture_{Cb} as output.
 - b. The filtering process for chroma block edges as specified in subclause 8.7.2.5.5 is invoked with the chroma picture sample array recPicture_{Cr} , the location of the chroma coding block $(x_{Cb} / 2, y_{Cb} / 2)$, the chroma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_VER , and a variable cQpPicOffset set equal to pps_cr_qp_offset as inputs, and the modified chroma picture sample array recPicture_{Cr} as output.

8.7.2.5.2 Horizontal edge filtering process

Inputs to this process are:

- the picture sample arrays recPicture_L , recPicture_{Cb} , and recPicture_{Cr} ,
- a luma location (x_{Cb}, y_{Cb}) specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture,
- a variable $\log_2 \text{CbSize}$ specifying the size of the current luma coding block,
- an array bS specifying the boundary filtering strength.

Outputs of this process are the modified picture sample arrays recPicture_L , recPicture_{Cb} , and recPicture_{Cr} .

The filtering process for edges in the luma coding block of the current coding unit consists of the following ordered steps:

1. The variable nD is set equal to $1 \ll (\log_2 \text{CbSize} - 3)$.
2. For y_{D_m} equal to $m \ll 3$ with $m = 0..nD - 1$, and x_{D_k} equal to $k \ll 2$ with $k = 0..nD * 2 - 1$, the following applies:
 - When $\text{bS}[x_{D_k}][y_{D_m}]$ is greater than 0, the following ordered steps apply:
 - a. The decision process for luma block edges as specified in subclause 8.7.2.5.3 is invoked with the luma picture sample array recPicture_L , the location of the luma coding block (x_{Cb}, y_{Cb}) , the luma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_HOR , and the boundary filtering strength $\text{bS}[x_{D_k}][y_{D_m}]$ as inputs, and the decisions dE , dEp , and dEq , and the variables β and t_c as outputs.
 - b. The filtering process for luma block edges as specified in subclause 8.7.2.5.4 is invoked with the luma picture sample array recPicture_L , the location of the luma coding block (x_{Cb}, y_{Cb}) , the luma location of the block (x_{D_k}, y_{D_m}) , a variable edgeType set equal to EDGE_HOR , the decisions dEp , dEp , and dEq , and the variables β and t_c as inputs, and the modified luma picture sample array recPicture_L as output.

The filtering process for edges in the chroma coding blocks of current coding unit consists of the following ordered steps:

1. The variable nD is set equal to $1 \ll (\log_2 CbSize - 3)$.
2. For yD_m equal to $m \ll 2$ with $m = 0..nD - 1$ and xD_k equal to $k \ll 2$ with $k = 0..nD - 1$, the following applies:
 - When $bS[xD_k * 2][yD_m * 2]$ is equal to 2 and $((yCb / 2 + yD_m) \gg 3) \ll 3$ is equal to $yCb / 2 + yD_m$, the following ordered steps apply:
 - a. The filtering process for chroma block edges as specified in subclause 8.7.2.5.5 is invoked with the chroma picture sample array $recPicture_{Cb}$, the location of the chroma coding block $(xCb / 2, yCb / 2)$, the chroma location of the block (xD_k, yD_m) , a variable $edgeType$ set equal to $EDGE_HOR$, and a variable $cQpPicOffset$ set equal to $pps_cb_qp_offset$ as inputs, and the modified chroma picture sample array $recPicture_{Cb}$ as output.
 - b. The filtering process for chroma block edges as specified in subclause 8.7.2.5.5 is invoked with the chroma picture sample array $recPicture_{Cr}$, the location of the chroma coding block $(xCb / 2, yCb / 2)$, the chroma location of the block (xD_k, yD_m) , a variable $edgeType$ set equal to $EDGE_HOR$, and a variable $cQpPicOffset$ set equal to $pps_cr_qp_offset$ as inputs, and the modified chroma picture sample array $recPicture_{Cr}$ as output.

8.7.2.5.3 Decision process for luma block edges

Inputs to this process are:

- a luma picture sample array $recPicture_L$,
- 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 luma location (xBl, yBl) specifying the top-left sample of the current luma block relative to the top-left sample of the current luma coding block,
- a variable $edgeType$ specifying whether a vertical ($EDGE_VER$) or a horizontal ($EDGE_HOR$) edge is filtered,
- a variable bS specifying the boundary filtering strength.

Outputs of this process are:

- the variables dE , dEp , and dEq containing decisions,
- the variables β and t_C .

If $edgeType$ is equal to $EDGE_VER$, the sample values $p_{i,k}$ and $q_{i,k}$ with $i = 0..3$ and $k = 0$ and 3 are derived as follows:

$$q_{i,k} = recPicture_L[xCb + xBl + i][yCb + yBl + k] \quad (8-284)$$

$$p_{i,k} = recPicture_L[xCb + xBl - i - 1][yCb + yBl + k] \quad (8-285)$$

Otherwise ($edgeType$ is equal to $EDGE_HOR$), the sample values $p_{i,k}$ and $q_{i,k}$ with $i = 0..3$ and $k = 0$ and 3 are derived as follows:

$$q_{i,k} = recPicture_L[xCb + xBl + k][yCb + yBl + i] \quad (8-286)$$

$$p_{i,k} = recPicture_L[xCb + xBl + k][yCb + yBl - i - 1] \quad (8-287)$$

The variables Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

A variable qP_L is derived as follows:

$$qP_L = ((Qp_Q + Qp_P + 1) \gg 1) \quad (8-288)$$

The value of the variable β' is determined as specified in Table 8-11 based on the luma quantization parameter Q derived as follows:

$$Q = Clip3(0, 51, qP_L + (slice_beta_offset_div2 \ll 1)) \quad (8-289)$$

where $slice_beta_offset_div2$ is the value of the syntax element $slice_beta_offset_div2$ for the slice that contains sample $q_{0,0}$.

The variable β is derived as follows:

$$\beta = \beta' * (1 \ll (\text{BitDepth}_Y - 8)) \quad (8-290)$$

The value of the variable t_c' is determined as specified in Table 8-11 based on the luma quantization parameter Q derived as follows:

$$Q = \text{Clip3}(0, 53, qP_L + 2 * (bS - 1) + (\text{slice_tc_offset_div2} \ll 1)) \quad (8-291)$$

where $\text{slice_tc_offset_div2}$ is the value of the syntax element $\text{slice_tc_offset_div2}$ for the slice that contains sample $q_{0,0}$.

The variable t_c is derived as follows:

$$t_c = t_c' * (1 \ll (\text{BitDepth}_Y - 8)) \quad (8-292)$$

Depending on the value of edgeType , the following applies:

– If edgeType is equal to EDGE_VER , the following ordered steps apply:

1. The variables dpq0 , dpq3 , dp , dq , and d are derived as follows:

$$\text{dp0} = \text{Abs}(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8-293)$$

$$\text{dp3} = \text{Abs}(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8-294)$$

$$\text{dq0} = \text{Abs}(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8-295)$$

$$\text{dq3} = \text{Abs}(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8-296)$$

$$\text{dpq0} = \text{dp0} + \text{dq0} \quad (8-297)$$

$$\text{dpq3} = \text{dp3} + \text{dq3} \quad (8-298)$$

$$\text{dp} = \text{dp0} + \text{dp3} \quad (8-299)$$

$$\text{dq} = \text{dq0} + \text{dq3} \quad (8-300)$$

$$\text{d} = \text{dpq0} + \text{dpq3} \quad (8-301)$$

2. The variables dE , dEp , and dEq are set equal to 0.

3. When d is less than β , the following ordered steps apply:

a. The variable dpq is set equal to $2 * \text{dpq0}$.

b. For the sample location $(x_{Cb} + x_{Bl}, y_{Cb} + y_{Bl})$, the decision process for a luma sample as specified in subclause 8.7.2.5.6 is invoked with sample values $p_{i,0}$, $q_{i,0}$ with $i = 0..3$, the variables dpq , β , and t_c as inputs, and the output is assigned to the decision dSam0 .

c. The variable dpq is set equal to $2 * \text{dpq3}$.

d. For the sample location $(x_{Cb} + x_{Bl}, y_{Cb} + y_{Bl} + 3)$, the decision process for a luma sample as specified in subclause 8.7.2.5.6 is invoked with sample values $p_{i,3}$, $q_{i,3}$ with $i = 0..3$, the variables dpq , β , and t_c as inputs, and the output is assigned to the decision dSam3 .

e. The variable dE is set equal to 1.

f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.

g. When dp is less than $(\beta + (\beta \gg 1)) \gg 3$, the variable dEp is set equal to 1.

h. When dq is less than $(\beta + (\beta \gg 1)) \gg 3$, the variable dEq is set equal to 1.

– Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The variables dpq0 , dpq3 , dp , dq , and d are derived as follows:

$$\text{dp0} = \text{Abs}(p_{2,0} - 2 * p_{1,0} + p_{0,0}) \quad (8-302)$$

$$\text{dp3} = \text{Abs}(p_{2,3} - 2 * p_{1,3} + p_{0,3}) \quad (8-303)$$

$$\text{dq0} = \text{Abs}(q_{2,0} - 2 * q_{1,0} + q_{0,0}) \quad (8-304)$$

$$\text{dq3} = \text{Abs}(q_{2,3} - 2 * q_{1,3} + q_{0,3}) \quad (8-305)$$

$$\text{dpq0} = \text{dp0} + \text{dq0} \quad (8-306)$$

$$\text{dpq3} = \text{dp3} + \text{dq3} \quad (8-307)$$

$$dp = dp0 + dp3 \quad (8-308)$$

$$dq = dq0 + dq3 \quad (8-309)$$

$$d = dpq0 + dpq3 \quad (8-310)$$

2. The variables dE, dEp, and dEq are set equal to 0.
3. When d is less than β , the following ordered steps apply:
 - a. The variable dpq is set equal to $2 * dpq0$.
 - b. For the sample location (xCb + xBl, yCb + yBl), the decision process for a luma sample as specified in subclause 8.7.2.5.6 is invoked with sample values $p_{0,0}$, $p_{3,0}$, $q_{0,0}$, and $q_{3,0}$, the variables dpq, β , and t_c as inputs, and the output is assigned to the decision dSam0.
 - c. The variable dpq is set equal to $2 * dpq3$.
 - d. For the sample location (xCb + xBl + 3, yCb + yBl), the decision process for a luma sample as specified in subclause 8.7.2.5.6 is invoked with sample values $p_{0,3}$, $p_{3,3}$, $q_{0,3}$, and $q_{3,3}$, the variables dpq, β , and t_c as inputs, and the output is assigned to the decision dSam3.
 - e. The variable dE is set equal to 1.
 - f. When dSam0 is equal to 1 and dSam3 is equal to 1, the variable dE is set equal to 2.
 - g. When dp is less than $(\beta + (\beta \gg 1)) \gg 3$, the variable dEp is set equal to 1.
 - h. When dq is less than $(\beta + (\beta \gg 1)) \gg 3$, the variable dEq is set equal to 1.

Table 8-11 – Derivation of threshold variables β' and t_c' from input Q

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
β'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	7	8
t_c'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Q	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
β'	9	10	11	12	13	14	15	16	17	18	20	22	24	26	28	30	32	34	36
t_c'	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4
Q	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53			
β'	38	40	42	44	46	48	50	52	54	56	58	60	62	64	-	-			
t_c'	5	5	6	6	7	8	9	10	11	13	14	16	18	20	22	24			

8.7.2.5.4 Filtering process for luma block edges

Inputs to this process are:

- a luma picture sample array recPicture_L,
- 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 luma location (xBl, yBl) specifying the top-left sample of the current luma block relative to the top-left sample of the current luma coding block,
- a variable edgeType specifying whether a vertical (EDGE_VER) or a horizontal (EDGE_HOR) edge is filtered,
- the variables dE, dEp, and dEq containing decisions,
- the variables β and t_c .

Output of this process is the modified luma picture sample array recPicture_L.

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 $p_{i,k}$ and $q_{i,k}$ with $i = 0..3$ and $k = 0..3$ are derived as follows:

$$q_{i,k} = \text{recPicture}_L[\text{xCb} + \text{xBl} + i][\text{yCb} + \text{yBl} + k] \quad (8-311)$$

$$p_{i,k} = \text{recPicture}_L[\text{xCb} + \text{xBl} - i - 1][\text{yCb} + \text{yBl} + k] \quad (8-312)$$

2. When dE is not equal to 0, for each sample location $(\text{xCb} + \text{xBl}, \text{yCb} + \text{yBl} + k)$, $k = 0..3$, the following ordered steps apply:

- a. The filtering process for a luma sample as specified in subclause 8.7.2.5.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with $i = 0..3$, the locations (xP_i, yP_i) set equal to $(\text{xCb} + \text{xBl} - i - 1, \text{yCb} + \text{yBl} + k)$ and (xQ_i, yQ_i) set equal to $(\text{xCb} + \text{xBl} + i, \text{yCb} + \text{yBl} + k)$ with $i = 0..2$, the decision dE , the variables dEp and dEq , and the variable t_c as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary, and the filtered sample values p_i' and q_j' as outputs.

- b. When nDp is greater than 0, the filtered sample values p_i' with $i = 0..nDp - 1$ replace the corresponding samples inside the sample array recPicture_L as follows:

$$\text{recPicture}_L[\text{xCb} + \text{xBl} - i - 1][\text{yCb} + \text{yBl} + k] = p_i' \quad (8-313)$$

- c. When nDq is greater than 0, the filtered sample values q_j' with $j = 0..nDq - 1$ replace the corresponding samples inside the sample array recPicture_L as follows:

$$\text{recPicture}_L[\text{xCb} + \text{xBl} + j][\text{yCb} + \text{yBl} + k] = q_j' \quad (8-314)$$

- Otherwise (edgeType is equal to EDGE_HOR), the following ordered steps apply:

1. The sample values $p_{i,k}$ and $q_{i,k}$ with $i = 0..3$ and $k = 0..3$ are derived as follows:

$$q_{i,k} = \text{recPicture}_L[\text{xCb} + \text{xBl} + k][\text{yCb} + \text{yBl} + i] \quad (8-315)$$

$$p_{i,k} = \text{recPicture}_L[\text{xCb} + \text{xBl} + k][\text{yCb} + \text{yBl} - i - 1] \quad (8-316)$$

2. When dE is not equal to 0, for each sample location $(\text{xCb} + \text{xBl} + k, \text{yCb} + \text{yBl})$, $k = 0..3$, the following ordered steps apply:

- a. The filtering process for a luma sample as specified in subclause 8.7.2.5.7 is invoked with the sample values $p_{i,k}$, $q_{i,k}$ with $i = 0..3$, the locations (xP_i, yP_i) set equal to $(\text{xCb} + \text{xBl} + k, \text{yCb} + \text{yBl} - i - 1)$ and (xQ_i, yQ_i) set equal to $(\text{xCb} + \text{xBl} + k, \text{yCb} + \text{yBl} + i)$ with $i = 0..2$, the decision dE , the variables dEp and dEq , and the variable t_c as inputs, and the number of filtered samples nDp and nDq from each side of the block boundary and the filtered sample values p_i' and q_j' as outputs.

- b. When nDp is greater than 0, the filtered sample values p_i' with $i = 0..nDp - 1$ replace the corresponding samples inside the sample array recPicture_L as follows:

$$\text{recPicture}_L[\text{xCb} + \text{xBl} + k][\text{yCb} + \text{yBl} - i - 1] = p_i' \quad (8-317)$$

- c. When nDq is greater than 0, the filtered sample values q_j' with $j = 0..nDq - 1$ replace the corresponding samples inside the sample array recPicture_L as follows:

$$\text{recPicture}_L[\text{xCb} + \text{xBl} + k][\text{yCb} + \text{yBl} + j] = q_j' \quad (8-318)$$

8.7.2.5.5 Filtering process for chroma block edges

Inputs to this process are:

- a chroma picture sample array s' ,
- 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.

Output of this process is the modified chroma picture sample array s' .

If edgeType is equal to EDGE_VER , the values p_i and q_i with $i = 0..1$ and $k = 0..3$ are derived as follows:

$$q_{i,k} = s'[\text{xCb} + \text{xBl} + i][\text{yCb} + \text{yBl} + k] \quad (8-319)$$

$$p_{i,k} = s'[\text{xCb} + \text{xBl} - i - 1][\text{yCb} + \text{yBl} + k] \quad (8-320)$$

Otherwise (edgeType is equal to EDGE_HOR), the sample values p_i and q_i with $i = 0..1$ and $k = 0..3$ are derived as follows:

$$q_{i,k} = s'[xCb + xBl + k][yCb + yBl + i] \quad (8-321)$$

$$p_{i,k} = s'[xCb + xBl + k][yCb + yBl - i - 1] \quad (8-322)$$

The variables Qp_Q and Qp_P are set equal to the Qp_Y values of the coding units which include the coding blocks containing the sample $q_{0,0}$ and $p_{0,0}$, respectively.

The variable Qp_C is determined as specified in Table 8-9 based on the index qPi derived as follows:

$$qPi = ((Qp_Q + Qp_P + 1) \gg 1) + cQpPicOffset \quad (8-323)$$

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 t_C' is determined as specified in Table 8-11 based on the chroma quantization parameter Q derived as follows:

$$Q = \text{Clip3}(0, 53, Qp_C + 2 + (\text{slice_tc_offset_div2} \ll 1)) \quad (8-324)$$

where $\text{slice_tc_offset_div2}$ is the value of the syntax element $\text{slice_tc_offset_div2}$ for the slice that contains sample $q_{0,0}$.

The variable t_C is derived as follows:

$$t_C = t_C' * (1 \ll (\text{BitDepth}_C - 8)) \quad (8-325)$$

Depending on the value of edgeType, the following applies:

- If edgeType is equal to EDGE_VER, for each sample location $(xCb + xBl, yCb + yBl + k)$, $k = 0..3$, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in subclause 8.7.2.5.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with $i = 0..1$, the locations $(xCb + xBl - 1, yCb + yBl + k)$ and $(xCb + xBl, yCb + yBl + k)$, and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s' as follows:

$$s'[xCb + xBl][yCb + yBl + k] = q_0' \quad (8-326)$$

$$s'[xCb + xBl - 1][yCb + yBl + k] = p_0' \quad (8-327)$$

- Otherwise (edgeType is equal to EDGE_HOR), for each sample location $(xCb + xBl + k, yCb + yBl)$, $k = 0..3$, the following ordered steps apply:

1. The filtering process for a chroma sample as specified in subclause 8.7.2.5.8 is invoked with the sample values $p_{i,k}$, $q_{i,k}$, with $i = 0..1$, the locations $(xCb + xBl + k, yCb + yBl - 1)$ and $(xCb + xBl + k, yCb + yBl)$, and the variable t_C as inputs, and the filtered sample values p_0' and q_0' as outputs.
2. The filtered sample values p_0' and q_0' replace the corresponding samples inside the sample array s' as follows:

$$s'[xCb + xBl + k][yCb + yBl] = q_0' \quad (8-328)$$

$$s'[xCb + xBl + k][yCb + yBl - 1] = p_0' \quad (8-329)$$

8.7.2.5.6 Decision process for a luma sample

Inputs to this process are:

- the sample values p_0 , p_3 , q_0 , and q_3 ,
- the variables dpq , β , and t_C .

Output of this process is the variable $dSam$ containing a decision.

The variable $dSam$ is specified as follows:

- If dpq is less than $(\beta \gg 2)$, $\text{Abs}(p_3 - p_0) + \text{Abs}(q_0 - q_3)$ is less than $(\beta \gg 3)$, and $\text{Abs}(p_0 - q_0)$ is less than $(5 * t_C + 1) \gg 1$, $dSam$ is set equal to 1.
- Otherwise, $dSam$ is set equal to 0.

8.7.2.5.7 Filtering process for a luma sample

Inputs to this process are:

- the luma sample values p_i and q_i with $i = 0..3$,
- the luma locations of p_i and q_i , (xP_i, yP_i) and (xQ_i, yQ_i) with $i = 0..2$,
- a variable dE ,
- the variables dEp and dEq containing decisions to filter samples $p1$ and $q1$ respectively,
- a variable t_C .

Outputs of this process are:

- the number of filtered samples nDp and nDq ,
- the filtered sample values p'_i and q'_j 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:

$$p'_0 = \text{Clip3}(p_0 - 2 * t_C, p_0 + 2 * t_C, (p_2 + 2 * p_1 + 2 * p_0 + 2 * q_0 + q_1 + 4) \gg 3) \quad (8-330)$$

$$p'_1 = \text{Clip3}(p_1 - 2 * t_C, p_1 + 2 * t_C, (p_2 + p_1 + p_0 + q_0 + 2) \gg 2) \quad (8-331)$$

$$p'_2 = \text{Clip3}(p_2 - 2 * t_C, p_2 + 2 * t_C, (2 * p_3 + 3 * p_2 + p_1 + p_0 + q_0 + 4) \gg 3) \quad (8-332)$$

$$q'_0 = \text{Clip3}(q_0 - 2 * t_C, q_0 + 2 * t_C, (p_1 + 2 * p_0 + 2 * q_0 + 2 * q_1 + q_2 + 4) \gg 3) \quad (8-333)$$

$$q'_1 = \text{Clip3}(q_1 - 2 * t_C, q_1 + 2 * t_C, (p_0 + q_0 + q_1 + q_2 + 2) \gg 2) \quad (8-334)$$

$$q'_2 = \text{Clip3}(q_2 - 2 * t_C, q_2 + 2 * t_C, (p_0 + q_0 + q_1 + 3 * q_2 + 2 * q_3 + 4) \gg 3) \quad (8-335)$$

- Otherwise, nDp and nDq are set both equal to 0, and the following weak filtering applies:

- The following applies:

$$\Delta = (9 * (q_0 - p_0) - 3 * (q_1 - p_1) + 8) \gg 4 \quad (8-336)$$

- When $\text{Abs}(\Delta)$ is less than $t_C * 10$, the following ordered steps apply:

- The filtered sample values p'_0 and q'_0 are specified as follows:

$$\Delta = \text{Clip3}(-t_C, t_C, \Delta) \quad (8-337)$$

$$p'_0 = \text{Clip1}_Y(p_0 + \Delta) \quad (8-338)$$

$$q'_0 = \text{Clip1}_Y(q_0 - \Delta) \quad (8-339)$$

- When dEp is equal to 1, the filtered sample value p'_1 is specified as follows:

$$\Delta p = \text{Clip3}(-(t_C \gg 1), t_C \gg 1, (((p_2 + p_0 + 1) \gg 1) - p_1 + \Delta) \gg 1) \quad (8-340)$$

$$p'_1 = \text{Clip1}_Y(p_1 + \Delta p) \quad (8-341)$$

- When dEq is equal to 1, the filtered sample value q'_1 is specified as follows:

$$\Delta q = \text{Clip3}(-(t_C \gg 1), t_C \gg 1, (((q_2 + q_0 + 1) \gg 1) - q_1 - \Delta) \gg 1) \quad (8-342)$$

$$q'_1 = \text{Clip1}_Y(q_1 + \Delta q) \quad (8-343)$$

- 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 p_0 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 q_0 is equal to 1.

8.7.2.5.8 Filtering process for a chroma sample

Inputs to this process are:

- the chroma sample values p_i and q_i with $i = 0..1$,
- the chroma locations of p_0 and q_0 , (xP_0, yP_0) and (xQ_0, yQ_0) ,
- a variable t_C .

Outputs of this process are the filtered sample values p_0' and q_0' .

The filtered sample values p_0' and q_0' are derived as follows:

$$\Delta = \text{Clip3}(-t_C, t_C, (((q_0 - p_0) \ll 2) + p_1 - q_1 + 4) \gg 3)) \quad (8-344)$$

$$p_0' = \text{Clip1}_C(p_0 + \Delta) \quad (8-345)$$

$$q_0' = \text{Clip1}_C(q_0 - \Delta) \quad (8-346)$$

When one or more of the following conditions are true, the filtered sample value, p_0' is substituted by the corresponding input sample value p_0 :

- `pcm_loop_filter_disabled_flag` is equal to 1 and `pcm_flag[2 * xP0][2 * yP0]` is equal to 1.
- `cu_transquant_bypass_flag` of the coding unit that includes the coding block containing the sample p_0 is equal to 1.

When one or more of the following conditions are true, the filtered sample value, q_0' is substituted by the corresponding input sample value q_0 :

- `pcm_loop_filter_disabled_flag` is equal to 1 and `pcm_flag[2 * xQ0][2 * yQ0]` is equal to 1.
- `cu_transquant_bypass_flag` of the coding unit that includes the coding block containing the sample q_0 is equal to 1.

8.7.3 Sample adaptive offset process

8.7.3.1 General

Inputs to this process are the reconstructed picture sample arrays prior to sample adaptive offset `recPictureL`, `recPictureCb`, and `recPictureCr`.

Outputs of this process are the modified reconstructed picture sample arrays after sample adaptive offset `saoPictureL`, `saoPictureCb`, and `saoPictureCr`.

This process is performed on a coding tree block basis after the completion of the deblocking filter process for the decoded picture.

The sample values in the modified reconstructed picture sample arrays `saoPictureL`, `saoPictureCb`, and `saoPictureCr` are initially set equal to the sample values in the reconstructed picture sample arrays `recPictureL`, `recPictureCb`, and `recPictureCr`.

For every coding tree unit with coding tree block 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 coding tree block modification process as specified in subclause 8.7.3.2 is invoked with `recPicture` set equal to `recPictureL`, `cIdx` set equal to 0, (rx, ry) , and `nCtbS` set equal to `CtbSizeY` as inputs, and the modified luma picture sample array `saoPictureL` as output.
- When `slice_sao_chroma_flag` of the current slice is equal to 1, the coding tree block modification process as specified in subclause 8.7.3.2 is invoked with `recPicture` set equal to `recPictureCb`, `cIdx` set equal to 1, (rx, ry) , and `nCtbS` set equal to $(1 \ll (CtbLog2SizeY - 1))$ as inputs, and the modified chroma picture sample array `saoPictureCb` as output.
- When `slice_sao_chroma_flag` of the current slice is equal to 1, the coding tree block modification process as specified in subclause 8.7.3.2 is invoked with `recPicture` set equal to `recPictureCr`, `cIdx` set equal to 2, (rx, ry) , and `nCtbS` set equal to $(1 \ll (CtbLog2SizeY - 1))$ as inputs, and the modified chroma picture sample array `saoPictureCr` as output.

8.7.3.2 Coding tree block 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 coding tree block location,
- the coding tree block size `nCtbS`.

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 `BitDepthV`.
- Otherwise, `bitDepth` is set equal to `BitDepthC`.

The location (`xCtb`, `yCtb`), specifying the top-left sample of the current coding tree block for the colour component `cIdx` relative to the top-left sample of the current picture component `cIdx`, is derived as follows:

$$(xCtb, yCtb) = (rx * nCtbS, ry * nCtbS) \quad (8-347)$$

The sample locations inside the current coding tree block are derived as follows:

$$(xS_i, yS_j) = (xCtb + i, yCtb + j) \quad (8-348)$$

$$(xY_i, yY_j) = (cIdx == 0) ? (xS_i, yS_j) : (xS_i << 1, yS_j << 1) \quad (8-349)$$

For all sample locations (`xSi`, `ySj`) and (`xYi`, `yYj`) with $i = 0..nCtbS - 1$ and $j = 0..nCtbS - 1$, 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.
- 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 8-12 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:

$$(xS_{ik'}, yS_{jk'}) = (xS_i + hPos[k], yS_j + vPos[k]) \quad (8-350)$$

$$(xY_{ik'}, yY_{jk'}) = (cIdx == 0) ? (xS_{ik'}, yS_{jk'}) : (xS_{ik'} << 1, yS_{jk'} << 1) \quad (8-351)$$
 - 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.
 - The sample at location (`xSik'`, `ySjk'`) belongs to a different slice and one of the following two conditions is true:
 - `MinTbAddrZs[ xYik' >> Log2MinTrafoSize ][ yYjk' >> Log2MinTrafoSize ]` is less than `MinTbAddrZs[ xYi >> Log2MinTrafoSize ][ yYj >> Log2MinTrafoSize ]` and `slice_loop_filter_across_slices_enabled_flag` in the slice which the sample `recPicture[ xSi ][ ySj ]` belongs to is equal to 0.
 - `MinTbAddrZs[ xYi >> Log2MinTrafoSize ][ yYj >> Log2MinTrafoSize ]` is less than `MinTbAddrZs[ xYik' >> Log2MinTrafoSize ][ yYjk' >> Log2MinTrafoSize ]` and `slice_loop_filter_across_slices_enabled_flag` in the slice which the sample `recPicture[ xSik' ][ ySjk' ]` belongs to is equal to 0.

- loop_filter_across_tiles_enabled_flag is equal to 0 and the sample at location ($x_{S_{ik}'}$, $y_{S_{jk}'}$) belongs to a different tile.
- Otherwise, edgeIdx is derived as follows:

- The following applies:

$$\text{edgeIdx} = 2 + \text{Sign}(\text{recPicture}[x_{S_i}][y_{S_j}] - \text{recPicture}[x_{S_i} + \text{hPos}[0]][y_{S_j} + \text{vPos}[0]]) + \text{Sign}(\text{recPicture}[x_{S_i}][y_{S_j}] - \text{recPicture}[x_{S_i} + \text{hPos}[1]][y_{S_j} + \text{vPos}[1]]) \quad (8-352)$$

- When edgeIdx is equal to 0, 1, or 2, edgeIdx is modified as follows:

$$\text{edgeIdx} = (\text{edgeIdx} == 2) ? 0 : (\text{edgeIdx} + 1) \quad (8-353)$$

3. The modified picture sample array $\text{saoPicture}[x_{S_i}][y_{S_j}]$ is derived as follows:

$$\text{saoPicture}[x_{S_i}][y_{S_j}] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \text{recPicture}[x_{S_i}][y_{S_j}] + \text{SaoOffsetVal}[\text{cIdx}][\text{rx}][\text{ry}][\text{edgeIdx}]) \quad (8-354)$$

- Otherwise ($\text{SaoTypeIdx}[\text{cIdx}][\text{rx}][\text{ry}]$ is equal to 1), the following ordered steps apply:

1. The variable bandShift is set equal to $\text{bitDepth} - 5$.
2. The variable saoLeftClass is set equal to $\text{sao_band_position}[\text{cIdx}][\text{rx}][\text{ry}]$.
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:

$$\text{for}(k = 0; k < 4; k++) \quad \text{bandTable}[(k + \text{saoLeftClass}) \& 31] = k + 1 \quad (8-355)$$

4. The variable bandIdx is set equal to $\text{bandTable}[\text{recPicture}[x_{S_i}][y_{S_j}] \gg \text{bandShift}]$.

5. The modified picture sample array $\text{saoPicture}[x_{S_i}][y_{S_j}]$ is derived as follows:

$$\text{saoPicture}[x_{S_i}][y_{S_j}] = \text{Clip3}(0, (1 \ll \text{bitDepth}) - 1, \text{recPicture}[x_{S_i}][y_{S_j}] + \text{SaoOffsetVal}[\text{cIdx}][\text{rx}][\text{ry}][\text{bandIdx}]) \quad (8-356)$$

Table 8-12 – Specification of hPos and vPos according to the sample adaptive offset class

$\text{SaoEoClass}[\text{cIdx}][\text{rx}][\text{ry}]$	0	1	2	3
$\text{hPos}[0]$	–1	0	–1	1
$\text{hPos}[1]$	1	0	1	–1
$\text{vPos}[0]$	0	–1	–1	–1
$\text{vPos}[1]$	0	1	1	1

9 Parsing process

9.1 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 in subclause 7.3 is equal to $\text{ue}(v)$, $\text{se}(v)$ (see subclause 9.2), or $\text{ae}(v)$ (see subclause 9.3).

9.2 Parsing process for 0-th order Exp-Golomb codes

9.2.1 General

This process is invoked when the descriptor of a syntax element in the syntax tables in subclause 7.3 is equal to $\text{ue}(v)$ or $\text{se}(v)$.

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. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

```

leadingZeroBits = -1
for( b = 0; !b; leadingZeroBits++ )
    b = read_bits( 1 )

```

(9-1)

The variable `codeNum` is then assigned as follows:

$$\text{codeNum} = 2^{\text{leadingZeroBits}} - 1 + \text{read_bits}(\text{leadingZeroBits}) \quad (9-2)$$

where the value returned from `read_bits(leadingZeroBits)` is interpreted as a binary representation of an unsigned integer with most significant bit written first.

Table 9-1 illustrates the structure of the Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed 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 x_i in Table 9-1, with i in the range of 0 to `leadingZeroBits` - 1, inclusive. Each x_i 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 <code>codeNum</code>
1	0
0 1 x_0	1..2
0 0 1 $x_1 x_0$	3..6
0 0 0 1 $x_2 x_1 x_0$	7..14
0 0 0 0 1 $x_3 x_2 x_1 x_0$	15..30
0 0 0 0 0 1 $x_4 x_3 x_2 x_1 x_0$	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	<code>codeNum</code>
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 subclause 9.2.2 with $codeNum$ as input.

9.2.2 Mapping process for signed Exp-Golomb codes

Input to this process is $codeNum$ as specified in subclause 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	$(-1)^{k+1} \lceil k/2 \rceil$

9.3 CABAC parsing process for slice segment data

9.3.1 General

This process is invoked when parsing syntax elements with descriptor $ae(v)$ in subclauses 7.3.8.1 through 7.3.8.11.

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 of the CABAC parsing process as specified in subclause 9.3.2 is invoked when starting the parsing of one or more of the following:

- the slice segment data syntax specified in subclause 7.3.8.1
- the coding tree unit syntax specified in subclause 7.3.8.2 and the coding tree unit is the first coding tree unit in a tile
- the coding tree unit syntax specified in subclause 7.3.8.2, $entropy_coding_sync_enabled_flag$ is equal to 1, and the associated luma coding tree block is the first luma coding tree block in a coding tree unit row

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as specified in subclause 9.3.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in subclause 9.3.4.

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 9.3.2.5.

The storage process for context variables is applied as follows:

- When ending the parsing of the coding tree unit syntax in subclause 7.3.8.2, `entropy_coding_sync_enabled_flag` is equal to 1 and `CtbAddrInRs % PicWidthInCtbsY` is equal to 1, the storage process for context variables as specified in subclause 9.3.2.3 is invoked with `TableStateIdxWpp` and `TableMpsValWpp` as outputs.
- When ending the parsing of the general slice segment data syntax in subclause 7.3.8.1, `dependent_slice_segments_enabled_flag` is equal to 1 and `end_of_slice_segment_flag` is equal to 1, the storage process for context variables as specified in subclause 9.3.2.3 is invoked with `TableStateIdxDs` and `TableMpsValDs` as outputs.

The whole CABAC parsing process for a syntax element `synEl` is illustrated in Figure 9-1.

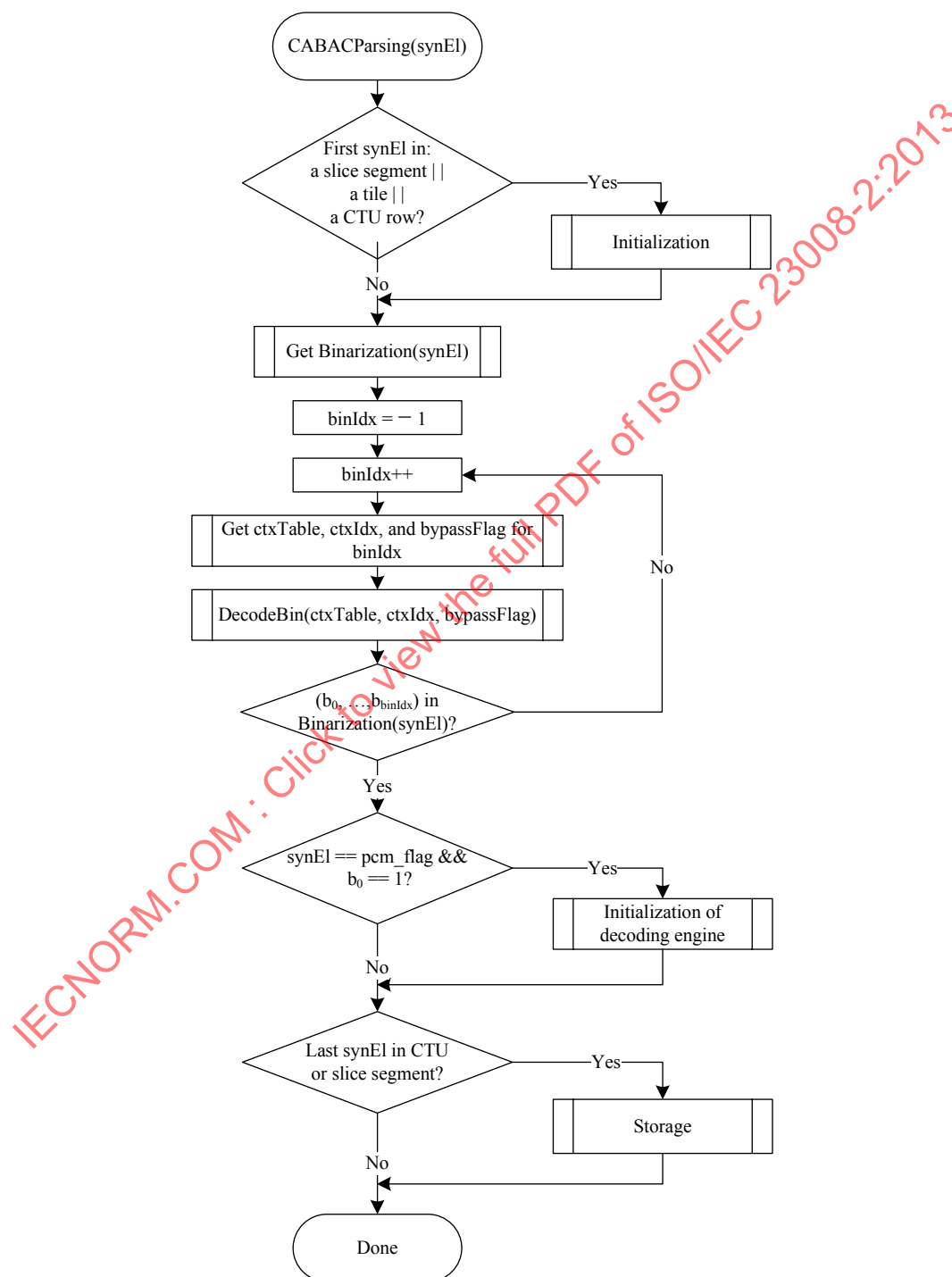


Figure 9-1 – Illustration of CABAC parsing process for a syntax element `synEl` (informative)

9.3.2 Initialization process

9.3.2.1 General

Outputs of this process are initialized CABAC internal variables.

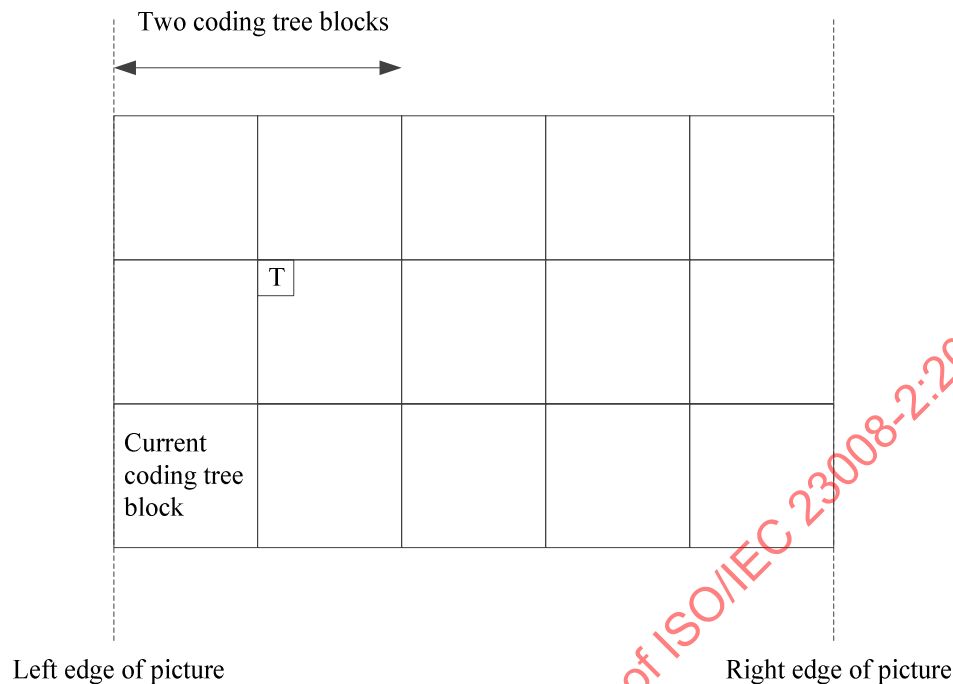


Figure 9-2 – Spatial neighbour T that is used to invoke the coding tree block availability derivation process relative to the current coding tree block (informative)

The context variables of the arithmetic decoding engine are initialized as follows:

- If the coding tree unit is the first coding tree unit in a tile, the initialization process for context variables is invoked as specified in subclause 9.3.2.2.
- Otherwise, if `entropy_coding_sync_enabled_flag` is equal to 1 and `CtbAddrInRs % PicWidthInCtbsY` is equal to 0, the following applies:
 - The location $(xNbT, yNbT)$ of the top-left luma sample of the spatial neighbouring block T (Figure 9-2) is derived using the location $(x0, y0)$ of the top-left luma sample of the current coding tree block as follows:

$$(xNbT, yNbT) = (x0 + CtbSizeY, y0 - CtbSizeY) \quad (9-3)$$
 - The availability derivation process for a block in z-scan order as specified in subclause 6.4.1 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 subclause 9.3.2.4 is invoked with `TableStateIdxWpp` and `TableMpsValWpp` as inputs.
 - Otherwise, the initialization process for context variables is invoked as specified in subclause 9.3.2.2.
- Otherwise, if `CtbAddrInRs` is equal to `slice_segment_address` and `dependent_slice_segment_flag` is equal to 1, the synchronization process for context variables as specified in subclause 9.3.2.4 is invoked with `TableStateIdxDs` and `TableMpsValDs` as inputs.
- Otherwise, the initialization process for context variables is invoked as specified in subclause 9.3.2.2.

The initialization process for the arithmetic decoding engine is invoked as specified in subclause 9.3.2.5.

The whole initialization process for a syntax element `synEl` is illustrated in the flowchart of Figure 9-3.

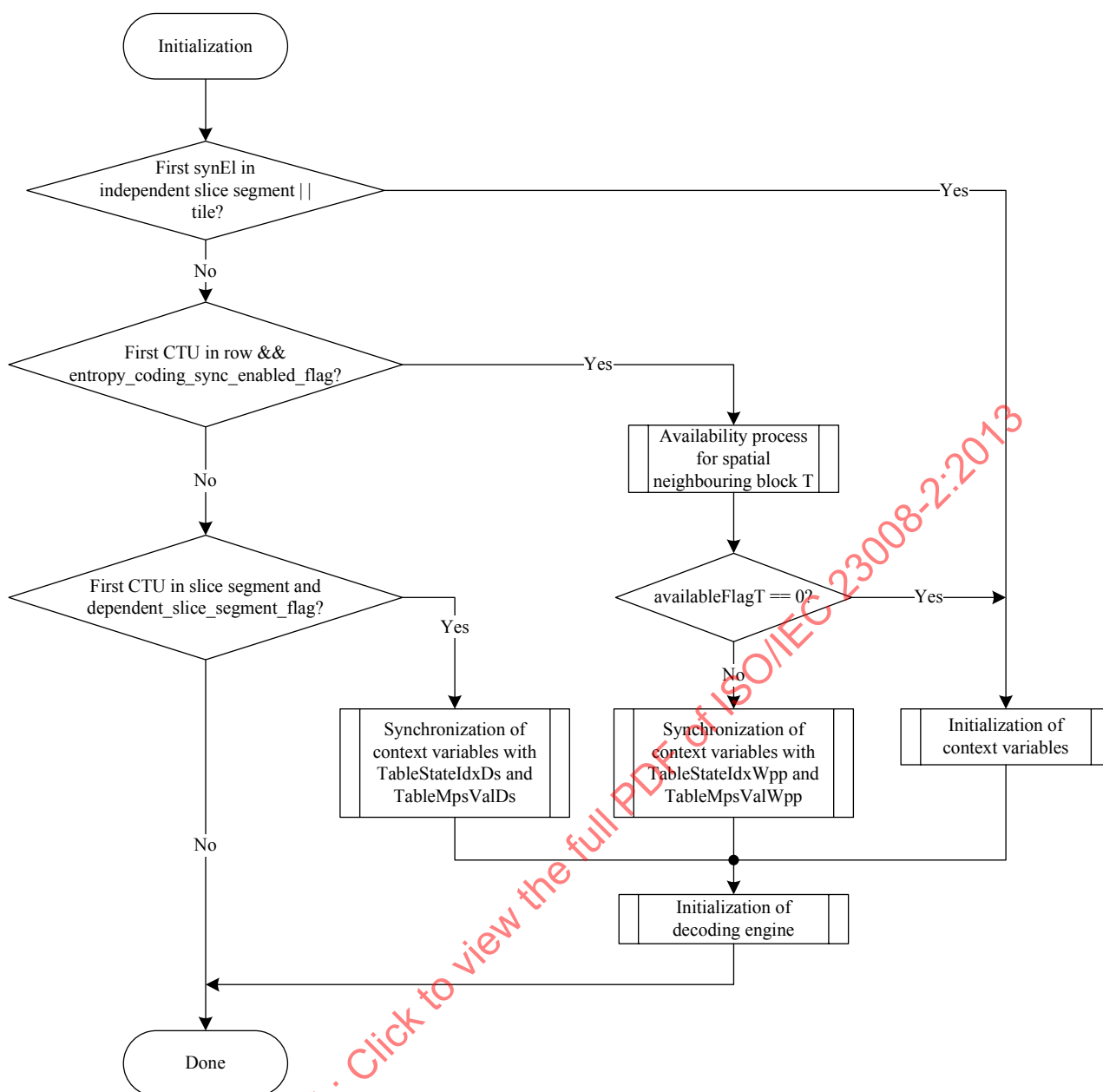


Figure 9-3 – Illustration of CABAC initialization process (informative)

9.3.2.2 Initialization process for context variables

Outputs of this process are the initialized CABAC context variables indexed by ctxTable and ctxIdx.

Table 9-5 to Table 9-31 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 7.3.8.1 through 7.3.8.11, except end_of_slice_segment_flag, end_of_sub_stream_one_bit, and pcm_flag.

For each context variable, the two variables pStateIdx and valMps are initialized.

NOTE 1 – The variable pStateIdx corresponds to a probability state index and the variable valMps corresponds to the value of the most probable symbol as further described in subclause 9.3.4.3.

From the 8 bit table entry initValue, the two 4 bit variables slopeIdx and offsetIdx are derived as follows:

$$\begin{aligned} \text{slopeIdx} &= \text{initValue} \gg 4 \\ \text{offsetIdx} &= \text{initValue} \& 15 \end{aligned} \quad (9-4)$$

The variables m and n, used in the initialization of context variables, are derived from slopeIdx and offsetIdx as follows:

$$\begin{aligned} m &= \text{slopeIdx} * 5 - 45 \\ n &= (\text{offsetIdx} \ll 3) - 16 \end{aligned} \quad (9-5)$$

The two values assigned to `pStateIdx` and `valMps` for the initialization are derived from `SliceQpY`, which is derived in Equation 7-40. Given the variables `m` and `n`, the initialization is specified as follows:

$$\begin{aligned} \text{preCtxState} &= \text{Clip3}(1, 126, ((m * \text{Clip3}(0, 51, \text{SliceQp}_Y)) \gg 4) + n) \\ \text{valMps} &= (\text{preCtxState} \leq 63) ? 0 : 1 \\ \text{pStateIdx} &= \text{valMps} ? (\text{preCtxState} - 64) : (63 - \text{preCtxState}) \end{aligned} \quad (9-6)$$

In Table 9-4, the `ctxIdx` 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 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:

$$\begin{aligned} &\text{if}(\text{slice_type} == I) \\ &\quad \text{initType} = 0 \\ &\text{else if}(\text{slice_type} == P) \\ &\quad \text{initType} = \text{cabac_init_flag} ? 2 : 1 \\ &\text{else} \\ &\quad \text{initType} = \text{cabac_init_flag} ? 1 : 2 \end{aligned} \quad (9-7)$$

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

Table 9-4 – Association of ctxIdx and syntax elements for each initializationType in the initialization process

Syntax structure	Syntax element	ctxTable	initType		
			0	1	2
sao()	sao_merge_left_flag sao_merge_up_flag	Table 9-5	0	1	2
	sao_type_idx_luma sao_type_idx_chroma	Table 9-6	0	1	2
coding_quadtree()	split_cu_flag[][]	Table 9-7	0..2	3..5	6..8
coding_unit()	cu_transquant_bypass_flag	Table 9-8	0	1	2
	cu_skip_flag	Table 9-9		0..2	3..5
	pred_mode_flag	Table 9-10		0	1
	part_mode	Table 9-11	0	1..4	5..8
	prev_intra_luma_pred_flag[][]	Table 9-12	0	1	2
	intra_chroma_pred_mode[][]	Table 9-13	0	1	2
	rqt_root_cbf	Table 9-14		0	1
prediction_unit()	merge_flag[][]	Table 9-15		0	1
	merge_idx[][]	Table 9-16		0	1
	inter_pred_idc[][]	Table 9-17		0..4	5..9
	ref_idx_l0[][], ref_idx_l1[][]	Table 9-18		0..1	2..3
	mvp_l0_flag[][], mvp_l1_flag[][]	Table 9-19		0	1
transform_tree()	split_transform_flag[][][]	Table 9-20	0..2	3..5	6..8
	cbf_luma[][][]	Table 9-21	0..1	2..3	4..5
	cbf_cb[][][], cbf_cr[][][]	Table 9-22	0..3	4..7	8..11
mvd_coding()	abs_mvd_greater0_flag[]	Table 9-23		0	2
	abs_mvd_greater1_flag[]	Table 9-23		1	3
transform_unit()	cu_qp_delta_abs	Table 9-24	0..1	2..3	4..5
residual_coding()	transform_skip_flag[][][0]	Table 9-25	0	1	2
	transform_skip_flag[][][1] transform_skip_flag[][][2]	Table 9-25	3	4	5
	last_sig_coeff_x_prefix	Table 9-26	0..17	18..35	36..53
	last_sig_coeff_y_prefix	Table 9-27	0..17	18..35	36..53
	coded_sub_block_flag[][]	Table 9-28	0..3	4..7	8..11
	sig_coeff_flag[][]	Table 9-29	0..41	42..83	84..125
	coeff_abs_level_greater1_flag[]	Table 9-30	0..23	24..47	48..71
	coeff_abs_level_greater2_flag[]	Table 9-31	0..5	6..11	12..17

NOTE 2 – ctxTable equal to 0 and ctxIdx equal to 0 are associated with end_of_slice_segment_flag, end_of_sub_stream_one_bit, and pcm_flag. The decoding process specified in subclause 9.3.4.3.5 applies to ctxTable equal to 0 and ctxIdx equal to 0. This decoding process, however, may also be implemented by using the decoding process specified in subclause 9.3.4.3.2. In this case, the initial values associated with ctxTable equal to 0 and ctxIdx equal to 0 are specified to be pStateIdx = 63 and valMps = 0, where pStateIdx = 63 represents a non-adapting probability state.

Table 9-5 – Values of initValue for ctxIdx of sao_merge_left_flag and sao_merge_up_flag

Initialization variable	ctxIdx of sao_merge_left_flag and sao_merge_up_flag		
	0	1	2
initValue	153	153	153

Table 9-6 – Values of initValue for ctxIdx of sao_type_idx_luma and sao_type_idx_chroma

Initialization variable	ctxIdx of sao_type_idx_luma and sao_type_idx_chroma		
	0	1	2
initValue	200	185	160

Table 9-7 – Values of initValue for ctxIdx of split_cu_flag

Initialization variable	ctxIdx of split_cu_flag								
	0	1	2	3	4	5	6	7	8
initValue	139	141	157	107	139	126	107	139	126

Table 9-8 – Values of initValue for ctxIdx of cu_transquant_bypass_flag

Initialization variable	ctxIdx of cu_transquant_bypass_flag		
	0	1	2
initValue	154	154	154

Table 9-9 – Values of initValue for ctxIdx of cu_skip_flag

Initialization variable	ctxIdx of cu_skip_flag					
	0	1	2	3	4	5
initValue	197	185	201	197	185	201

Table 9-10 – Values of initValue for ctxIdx of pred_mode_flag

Initialization variable	ctxIdx of pred_mode_flag	
	0	1
initValue	149	134

Table 9-11 – Values of initValue for ctxIdx of part_mode

Initialization variable	ctxIdx of part_mode								
	0	1	2	3	4	5	6	7	8
initValue	184	154	139	154	154	154	139	154	154

Table 9-12 – Values of initValue for ctxIdx of prev_intra_luma_pred_flag

Initialization variable	ctxIdx of prev_intra_luma_pred_flag		
	0	1	2
initValue	184	154	183

Table 9-13 – Values of initValue for ctxIdx of intra_chroma_pred_mode

Initialization variable	ctxIdx of intra_chroma_pred_mode		
	0	1	2
initValue	63	152	152

Table 9-14 – Values of initValue for ctxIdx of rqt_root_cbf

Initialization variable	ctxIdx of rqt_root_cbf	
	0	1
initValue	79	79

Table 9-15 – Value of initValue for ctxIdx of merge_flag

Initialization variable	ctxIdx of merge_flag	
	0	1
initValue	110	154

Table 9-16 – Values of initValue for ctxIdx of merge_idx

Initialization variable	ctxIdx of merge_idx	
	0	1
initValue	122	137

Table 9-17 – Values of initValue for ctxIdx of inter_pred_idc

Initialization variable	ctxIdx of inter_pred_idc									
	0	1	2	3	4	5	6	7	8	9
initValue	95	79	63	31	31	95	79	63	31	31

Table 9-18 – Values of initValue for ctxIdx of ref_idx_l0 and ref_idx_l1

Initialization variable	ctxIdx of ref_idx_l0 and ref_idx_l1			
	0	1	2	3
initValue	153	153	153	153

Table 9-19 – Values of initValue for ctxIdx of mvp_l0_flag and mvp_l1_flag

Initialization variable	ctxIdx of mvp_l0_flag and mvp_l1_flag	
	0	1
initValue	168	168

Table 9-20 – Values of initValue for ctxIdx of split_transform_flag

Initialization variable	ctxIdx of split_transform_flag								
	0	1	2	3	4	5	6	7	8
initValue	153	138	138	124	138	94	224	167	122

Table 9-21 – Values of initValue for ctxIdx of cbf_luma

Initialization variable	ctxIdx of cbf_luma					
	0	1	2	3	4	5
initValue	111	141	153	111	153	111

Table 9-22 – Values of initValue for ctxIdx of cbf_cb and cbf_cr

Initialization variable	ctxIdx of cbf_cb and cbf_cr											
	0	1	2	3	4	5	6	7	8	9	10	11
initValue	94	138	182	154	149	107	167	154	149	92	167	154

Table 9-23 – Values of initValue for ctxIdx of abs_mvd_greater0_flag and abs_mvd_greater1_flag

Initialization variable	ctxIdx of abs_mvd_greater0_flag and abs_mvd_greater1_flag			
	0	1	2	3
initValue	140	198	169	198

Table 9-24 – Values of initValue for ctxIdx of cu_qp_delta_abs

Initialization variable	ctxIdx of cu_qp_delta_abs					
	0	1	2	3	4	5
initValue	154	154	154	154	154	154

Table 9-25 – Values of initValue for ctxIdx of transform_skip_flag

Initialization variable	ctxIdx of transform_skip_flag					
	0	1	2	3	4	5
initValue	139	139	139	139	139	139

Table 9-26 – Values of initValue for ctxIdx of last_sig_coeff_x_prefix

Initialization variable	ctxIdx of last_sig_coeff_x_prefix																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
initValue	110	110	124	125	140	153	125	127	140	109	111	143	127	111	79	108	123	63
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
initValue	125	110	94	110	95	79	125	111	110	78	110	111	111	95	94	108	123	108
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
initValue	125	110	124	110	95	94	125	111	111	79	125	126	111	111	79	108	123	93

Table 9-27 – Values of initValue for ctxIdx of last_sig_coeff_y_prefix

Initialization variable	ctxIdx of last_sig_coeff_y_prefix																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
initValue	110	110	124	125	140	153	125	127	140	109	111	143	127	111	79	108	123	63
	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
initValue	125	110	94	110	95	79	125	111	110	78	110	111	111	95	94	108	123	108
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
initValue	125	110	124	110	95	94	125	111	111	79	125	126	111	111	79	108	123	93

Table 9-28 – Values of initValue for ctxIdx of coded_sub_block_flag

Initialization variable	ctxIdx of coded_sub_block_flag											
	0	1	2	3	4	5	6	7	8	9	10	11
initValue	91	171	134	141	121	140	61	154	121	140	61	154

Table 9-29 – Values of initValue for ctxIdx of sig_coeff_flag

Initialization variable	ctxIdx of sig_coeff_flag															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	111	111	125	110	110	94	124	108	124	107	125	141	179	153	125	107
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	125	141	179	153	125	107	125	141	179	153	125	140	139	182	182	152
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	136	152	136	153	136	139	111	136	139	111	155	154	139	153	139	123
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
initValue	123	63	153	166	183	140	136	153	154	166	183	140	136	153	154	166
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
initValue	183	140	136	153	154	170	153	123	123	107	121	107	121	167	151	183
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
initValue	140	151	183	140	170	154	139	153	139	123	123	63	124	166	183	140
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
initValue	136	153	154	166	183	140	136	153	154	166	183	140	136	153	154	170
	112	113	114	115	116	117	118	119	120	121	122	123	124	125		
initValue	153	138	138	122	121	122	121	167	151	183	140	151	183	140		

Table 9-30 – Values of initValue for ctxIdx of coeff_abs_level_greater1_flag

Initialization variable	ctxIdx of coeff_abs_level_greater1_flag															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initValue	140	92	137	138	140	152	138	139	153	74	149	92	139	107	122	152
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
initValue	140	179	166	182	140	227	122	197	154	196	196	167	154	152	167	182
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
initValue	182	134	149	136	153	121	136	137	169	194	166	167	154	167	137	182
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
initValue	154	196	167	167	154	152	167	182	182	134	149	136	153	121	136	122
	64	65	66	67	68	69	70	71								
initValue	169	208	166	167	154	152	167	182								

Table 9-31 – Values of initValue for ctxIdx of coeff_abs_level_greater2_flag

Initialization variable	ctxIdx of coeff_abs_level_greater2_flag								
	0	1	2	3	4	5	6	7	8
initValue	138	153	136	167	152	152	107	167	91
	9	10	11	12	13	14	15	16	17
initValue	122	107	167	107	167	91	107	107	167

9.3.2.3 Storage process for context variables

Inputs to this process are the CABAC context variables indexed by `ctxTable` and `ctxIdx`.

Outputs of this process are variables `tableStateSync` and `tableMPSSync` containing the values of the variables `pStateIdx` and `valMps` used in the initialization process of context variables that are assigned to all syntax elements in subclauses 7.3.8.1 through 7.3.8.11, except `end_of_slice_segment_flag`, `end_of_sub_stream_one_bit`, and `pcm_flag`.

For each context variable, the corresponding entries `pStateIdx` and `valMps` of tables `tableStateSync` and `tableMPSSync` are initialized to the corresponding `pStateIdx` and `valMps`.

The storage process for context variables is illustrated in the flowchart of Figure 9-4.

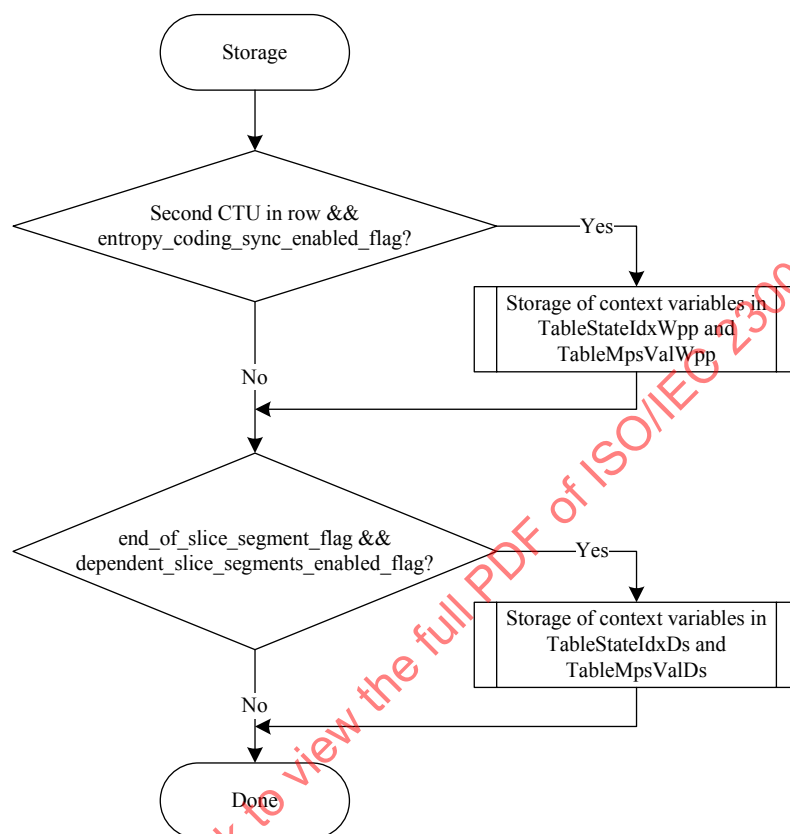


Figure 9-4 – Illustration of CABAC storage process (informative)

9.3.2.4 Synchronization process for context variables

Inputs to this process are variables `tableStateSync` and `tableMPSSync` containing the values of the variables `pStateIdx` and `valMps` used in the storage process of context variables that are assigned to all syntax elements in subclauses 7.3.8.1 through 7.3.8.11, except `end_of_slice_segment_flag`, `end_of_sub_stream_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 `pStateIdx` and `valMps` are initialized to the corresponding entries `pStateIdx` and `valMps` of tables `tableStateSync` and `tableMPSSync`.

9.3.2.5 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 9.3.4.3. The arithmetic decoding process for a binary decision (`DecodeDecision`) as specified in subclause 9.3.4.3.2 and the decoding process for a binary decision before termination (`DecodeTerminate`) as specified in subclause 9.3.4.3.5 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 9.3.4.3.4 requires a minimum register precision of 10 bits for the variable `ivlOffset` and a minimum register precision of 9 bits for the variable `ivlCurrRange`.

9.3.3 Binarization process

9.3.3.1 General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 9-32 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the k-th order Exp-Golomb (EGk) binarization process, and the fixed-length (FL) binarization process are given in subclauses 9.3.3.2 through 9.3.3.4, respectively. Other binarizations are specified in subclauses 9.3.3.5 through 9.3.3.9.

Table 9-32 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
<code>slice_segment_data()</code>	<code>end_of_slice_segment_flag</code>	FL	<code>cMax = 1</code>
	<code>end_of_sub_stream_one_bit</code>	FL	<code>cMax = 1</code>
<code>sao()</code>	<code>sao_merge_left_flag</code>	FL	<code>cMax = 1</code>
	<code>sao_merge_up_flag</code>	FL	<code>cMax = 1</code>
	<code>sao_type_idx_luma</code>	TR	<code>cMax = 2, cRiceParam = 0</code>
	<code>sao_type_idx_chroma</code>	TR	<code>cMax = 2, cRiceParam = 0</code>
	<code>sao_offset_abs[][][]</code>	TR	<code>cMax = (1 << (Min(bitDepth, 10) - 5)) - 1, cRiceParam = 0</code>
	<code>sao_offset_sign[][][]</code>	FL	<code>cMax = 1</code>
	<code>sao_band_position[][][]</code>	FL	<code>cMax = 31</code>
	<code>sao_eo_class_luma</code>	FL	<code>cMax = 3</code>
	<code>sao_eo_class_chroma</code>	FL	<code>cMax = 3</code>
<code>coding_quadtree()</code>	<code>split_cu_flag[][]</code>	FL	<code>cMax = 1</code>
<code>coding_unit()</code>	<code>cu_transquant_bypass_flag</code>	FL	<code>cMax = 1</code>
	<code>cu_skip_flag</code>	FL	<code>cMax = 1</code>
	<code>pred_mode_flag</code>	FL	<code>cMax = 1</code>
	<code>part_mode</code>	9.3.3.5	<code>(xCb, yCb) = (x0, y0), log2CbSize</code>
	<code>pcm_flag[][]</code>	FL	<code>cMax = 1</code>
	<code>prev_intra_luma_pred_flag[][]</code>	FL	<code>cMax = 1</code>
	<code>mpm_idx[][]</code>	TR	<code>cMax = 2, cRiceParam = 0</code>
	<code>rem_intra_luma_pred_mode[][]</code>	FL	<code>cMax = 31</code>
	<code>intra_chroma_pred_mode[][]</code>	9.3.3.6	-
	<code>rqt_root_cbf</code>	FL	<code>cMax = 1</code>

Table 9-32 – Syntax elements and associated binarizations

Syntax structure	Syntax element	Binarization	
		Process	Input parameters
prediction_unit()	merge_flag[][]	FL	cMax = 1
	merge_idx[][]	TR	cMax = MaxNumMergeCand – 1, cRiceParam = 0
	inter_pred_idc[x0][y0]	9.3.3.7	nPbW, nPbH
	ref_idx_l0[][]	TR	cMax = num_ref_idx_l0_active_minus1, cRiceParam = 0
	mvp_l0_flag[][]	FL	cMax = 1
	ref_idx_l1[][]	TR	cMax = num_ref_idx_l1_active_minus1, cRiceParam = 0
	mvp_l1_flag[][]	FL	cMax = 1
transform_tree()	split_transform_flag[][][]	FL	cMax = 1
	cbf_luma[][][]	FL	cMax = 1
	cbf_cb[][][]	FL	cMax = 1
	cbf_cr[][][]	FL	cMax = 1
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()	cu_qp_delta_abs	9.3.3.8	-
	cu_qp_delta_sign_flag	FL	cMax = 1
residual_coding()	transform_skip_flag[][][]	FL	cMax = 1
	last_sig_coeff_x_prefix	TR	cMax = (log2TrafoSize << 1) – 1, cRiceParam = 0
	last_sig_coeff_y_prefix	TR	cMax = (log2TrafoSize << 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
	coeff_abs_level_greater1_flag[]	FL	cMax = 1
	coeff_abs_level_greater2_flag[]	FL	cMax = 1
	coeff_abs_level_remaining[]	9.3.3.9	current sub-block scan index i, baseLevel
	coeff_sign_flag[]	FL	cMax = 1

9.3.3.2 Truncated Rice (TR) binarization process

Input to this process is a request for a TR binarization for a syntax element with value synVal, cMax, and cRiceParam.

Output of this process is the TR binarization of the syntax element.

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 synVal, prefixVal, is derived as follows:

$$\text{prefixVal} = \text{synVal} \gg \text{cRiceParam} \quad (9-8)$$

- 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 9-33 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 9-33 – 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 synVal, the suffix of the TR bin string is present and it is derived as follows:

- The suffix value of synVal, suffixVal, is derived as follows:

$$\text{suffixVal} = \text{synVal} - ((\text{prefixVal}) \ll \text{cRiceParam}) \quad (9-9)$$

- The suffix of the TR bin string is specified by the binary representation of suffixVal.

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.

9.3.3.3 k-th order Exp-Golomb (EGk) binarization process

Inputs to this process is a request for an EGk binarization for a syntax element.

Output of this process is the EGk binarization of the syntax element.

The bin string of the EGk binarization process of a syntax element synVal is specified 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( synVal )
stopLoop = 0
do {
    if( absV >= ( 1 << k ) ) {
        put( 1 )
        absV = absV - ( 1 << k )
        k++
    } else {
        put( 0 )
        while( k-- )
            put( ( absV >> k ) & 1 )
        stopLoop = 1
    }
} while( !stopLoop )

```

(9-10)

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 0-th order as specified in subclause 9.2.

9.3.3.4 Fixed-length (FL) binarization process

Inputs to this process are a request for a FL binarization for a syntax element and cMax.

Output of this process is the FL binarization of the syntax element.

FL binarization is constructed by using a fixedLength-bit unsigned integer bin string of the syntax element value, where $\text{fixedLength} = \text{Ceil}(\text{Log}_2(\text{cMax} + 1))$. The indexing of bins for the FL binarization is such that the $\text{binIdx} = 0$ relates to the most significant bit with increasing values of binIdx towards the least significant bit.

9.3.3.5 Binarization process for part_mode

Inputs to this process are a request for a binarization for the syntax element `part_mode` a luma location (x_{Cb} , y_{Cb}), specifying the top-left sample of the current luma coding block relative to the top-left luma sample of the current picture, and a variable $\log_2\text{CbSize}$ specifying the current luma coding block size.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element `part_mode` is specified in Table 9-34 depending on the values of $\text{CuPredMode}[x_{Cb}][y_{Cb}]$ and $\log_2\text{CbSize}$.

Table 9-34 – Binarization for part_mode

$\text{CuPredMode}[x_{Cb}][y_{Cb}]$	<code>part_mode</code>	<code>PartMode</code>	Bin string			
			$\log_2\text{CbSize} > \text{MinCbLog}_2\text{SizeY}$		$\log_2\text{CbSize} == \text{MinCbLog}_2\text{SizeY}$	
			<code>!amp_enabled_flag</code>	<code>amp_enabled_flag</code>	$\log_2\text{CbSize} == 3$	$\log_2\text{CbSize} > 3$
MODE_INTRA	0	PART_2Nx2N	-	-	1	1
	1	PART_NxN	-	-	0	0
MODE_INTER	0	PART_2Nx2N	1	1	1	1
	1	PART_2NxN	01	011	01	01
	2	PART_Nx2N	00	001	00	001
	3	PART_NxN	-	-	-	000
	4	PART_2NxN _U	-	0100	-	-
	5	PART_2NxN _D	-	0101	-	-
	6	PART_nLx2N	-	0000	-	-
	7	PART_nRx2N	-	0001	-	-

9.3.3.6 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 9-35.

Table 9-35 – Binarization for intra_chroma_pred_mode

Value of intra_chroma_pred_mode	Bin string
4	0
0	100
1	101
2	110
3	111

9.3.3.7 Binarization process for inter_pred_idc

Inputs to this process are a request for a binarization for the syntax element `inter_pred_idc`, the current luma prediction block width `nPbW`, and the current luma prediction block height `nPbH`.

Output of this process is the binarization of the syntax element.

The binarization for the syntax element `inter_pred_idc` is specified in Table 9-36.

Table 9-36 – Binarization for inter_pred_idc

Value of inter_pred_idc	Name of inter_pred_idc	Bin string	
		(nPbW + nPbH) != 12	(nPbW + nPbH) == 12
0	PRED_L0	00	0
1	PRED_L1	01	1
2	PRED_BI	1	-

9.3.3.8 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:

$$\text{prefixVal} = \text{Min}(\text{cu_qp_delta_abs}, 5) \quad (9-11)$$

- The prefix bin string is specified by invoking the TR binarization process as specified in subclause 9.3.3.2 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:

$$\text{suffixVal} = \text{cu_qp_delta_abs} - 5 \quad (9-12)$$

- The suffix bin string is specified by invoking the EGk binarization process as specified in subclause 9.3.3.3 for `suffixVal` with the Exp-Golomb order `k` set equal to 0.

9.3.3.9 Binarization process for coeff_abs_level_remaining

Input to this process is a request for a binarization for the syntax element `coeff_abs_level_remaining[ n ]`, the current sub-block scan index `i`, and `baseLevel`.

Output of this process is the binarization of the syntax element.

The variables `cLastAbsLevel` and `cLastRiceParam` are derived as follows:

- If this process is invoked for the first time for the current sub-block scan index i , $cLastAbsLevel$ and $cLastRiceParam$ are set equal to 0.
- Otherwise (this process is not invoked for the first time for the current sub-block scan index i), $cLastAbsLevel$ and $cLastRiceParam$ are set equal to the values of $cAbsLevel$ and $cRiceParam$, respectively, that have been derived during the last invocation of the binarization process for the syntax element $coeff_abs_level_remaining$ as specified in this subclause.

The variable $cAbsLevel$ is set equal to $baseLevel + coeff_abs_level_remaining[n]$.

The variable $cRiceParam$ is derived from $cLastAbsLevel$ and $cLastRiceParam$ as:

$$cRiceParam = \text{Min}(cLastRiceParam + (cLastAbsLevel > (3 * (1 << cLastRiceParam))) ? 1 : 0, 4) \quad (9-13)$$

The variable $cMax$ is derived from $cRiceParam$ as:

$$cMax = 4 << cRiceParam \quad (9-14)$$

The binarization of the syntax element $coeff_abs_level_remaining[n]$ 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 $cu_qp_delta_abs$, $prefixVal$, is derived as follows:

$$prefixVal = \text{Min}(cMax, coeff_abs_level_remaining[n]) \quad (9-15)$$

- The prefix bin string is specified by invoking the TR binarization process as specified in subclause 9.3.3.2 for $prefixVal$ with the variables $cMax$ and $cRiceParam$ as inputs.

When the prefix bin string is equal to the bit string of length 4 with all bits equal to 1, 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:

$$suffixVal = coeff_abs_level_remaining[n] - cMax \quad (9-16)$$

- The suffix bin string is specified by invoking the EGK binarization process as specified in subclause 9.3.3.3 for $suffixVal$ with the Exp-Golomb order k set equal to $cRiceParam + 1$.

9.3.4 Decoding process flow

9.3.4.1 General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in subclause 9.3.3.

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 subclause 9.3.4.2 is invoked with $binIdx$ as input and $ctxTable$, $ctxIdx$, and $bypassFlag$ as outputs.
2. The arithmetic decoding process as specified in subclause 9.3.4.3 is invoked with $ctxTable$, $ctxIdx$, and $bypassFlag$ as inputs and the value of the bin as output.

9.3.4.2 Derivation process for $ctxTable$, $ctxIdx$ and $bypassFlag$

9.3.4.2.1 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 9-37:

- If the entry in Table 9-37 is not equal to "bypass", "terminate", and "na", the values of `binIdx` are decoded by invoking the `DecodeDecision` process as specified in subclause 9.3.4.3.2 and the following applies:
 - `ctxTable` is specified in Table 9-4.
 - The variable `ctxInc` is specified by the corresponding entry in Table 9-37 and when more than one value is listed in Table 9-37 for a `binIdx`, the assignment process for `ctxInc` for that `binIdx` is further specified in the subclauses given in parenthesis.
 - The variable `ctxIdxOffset` is specified by the lowest value of `ctxIdx` in Table 9-4 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 9-37 is equal to "bypass", the values of `binIdx` are decoded by invoking the `DecodeBypass` process as specified in subclause 9.3.4.3.4 and the following applies:
 - `ctxTable` is set equal to 0.
 - `ctxIdx` is set equal to 0.
 - `bypassFlag` is set equal to 1.
- Otherwise, if the entry in Table 9-37 is equal to "terminate", the values of `binIdx` are decoded by invoking the `DecodeTerminate` process as specified in subclause 9.3.4.3.5 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 9-37 is equal to "na"), the values of `binIdx` do not occur for the corresponding syntax element.

Table 9-37 – Assignment of `ctxInc` to syntax elements with context coded bins

Syntax element	binIdx					
	0	1	2	3	4	≥ 5
end of slice segment flag	terminate	na	na	na	na	na
end of sub stream one bit	terminate	na	na	na	na	na
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	bypass
sao_offset_sign[][][]	bypass	na	na	na	na	na
sao_band_position[][][]	bypass	bypass	bypass	bypass	bypass	bypass
sao_eo_class_luma	bypass	bypass	bypass	na	na	na
sao_eo_class_chroma	bypass	bypass	bypass	na	na	na
split_cu_flag[][]	0,1,2 (subclause 9.3.4.2.2)	na	na	na	na	na
cu_transquant_bypass_flag	0	na	na	na	na	na
cu_skip_flag	0,1,2 (subclause 9.3.4.2.2)	na	na	na	na	na
pred_mode_flag	0	na	na	na	na	na
part_mode log2CbSize == MinCbLog2SizeY	0	1	2	bypass	na	na

Table 9-37 – Assignment of ctxInc to syntax elements with context coded bins

Syntax element	binIdx					
	0	1	2	3	4	>= 5
part_mode log2CbSize > MinCbLog2SizeY	0	1	3	bypass	na	na
pcm_flag[][]	terminate	na	na	na	na	na
prev_intra_luma_pred_flag[][]	0	na	na	na	na	na
mpm_idx[][]	bypass	bypass	na	na	na	na
rem_intra_luma_pred_mode[][]	bypass	bypass	bypass	bypass	bypass	bypass
intra_chroma_pred_mode[][]	0	bypass	bypass	na	na	na
rqt_root_cbf	0	na	na	na	na	na
merge_flag[][]	0	na	na	na	na	na
merge_idx[][]	0	bypass	bypass	bypass	na	na
inter_pred_idc[x0][y0]	(nPbW + nPbH) != 12 ? CtDepth[x0][y0] : 4	4	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
split_transform_flag[][][]	5 – log2TrafoSize	na	na	na	na	na
cbf_cb[][][]	trafoDepth	na	na	na	na	na
cbf_cr[][][]	trafoDepth	na	na	na	na	na
cbf_luma[][][]	trafoDepth == 0 ? 1 : 0	na	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
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
last_sig_coeff_x_prefix	0..17 (subclause 9.3.4.2.3)					
last_sig_coeff_y_prefix	0..17 (subclause 9.3.4.2.3)					
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[][]	0..3 (subclause 9.3.4.2.4)	na	na	na	na	na
sig_coeff_flag[][]	0..41 (subclause 9.3.4.2.5)	na	na	na	na	na
coeff_abs_level_greater1_flag[]	0..23 (subclause 9.3.4.2.6)	na	na	na	na	na
coeff_abs_level_greater2_flag[]	0..5 (subclause 9.3.4.2.7)	na	na	na	na	na
coeff_abs_level_remaining[]	bypass	bypass	bypass	bypass	bypass	bypass
coeff_sign_flag[]	bypass	na	na	na	na	na

9.3.4.2.2 Derivation process of ctxInc using left and above syntax elements

Input to this process is the luma location (x_0, y_0) specifying the top-left luma sample of the current luma block relative to the top-left sample of the current picture.

Output of this process is ctxInc.

The location (x_{NbL}, y_{NbL}) is set equal to ($x_0 - 1, y_0$) 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 6.4.1 with the location (x_{Curr}, y_{Curr}) set equal to (x_0, y_0) and the neighbouring location (x_{NbY}, y_{NbY}) set equal to (x_{NbL}, y_{NbL}) as inputs, and the output is assigned to availableL.

The location (x_{NbA}, y_{NbA}) is set equal to ($x_0, y_0 - 1$) 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 6.4.1 with the location (x_{Curr}, y_{Curr}) set equal to (x_0, y_0) and the neighbouring location (x_{NbY}, y_{NbY}) set equal to (x_{NbA}, y_{NbA}) as inputs, and the output is assigned to availableA.

The assignment of ctxInc for the syntax elements split_cu_flag[x_0][y_0] and cu_skip_flag[x_0][y_0] is specified in Table 9-38.

Table 9-38 – Specification of ctxInc using left and above syntax elements

Syntax element	condL	condA	ctxInc
split_cu_flag[x_0][y_0]	CtDepth[x_{NbL}][y_{NbL}] > cqtDepth	CtDepth[x_{NbA}][y_{NbA}] > cqtDepth	(condL && availableL) + (condA && availableA)
cu_skip_flag[x_0][y_0]	cu_skip_flag[x_{NbL}][y_{NbL}]	cu_skip_flag[x_{NbA}][y_{NbA}]	(condL && availableL) + (condA && availableA)

9.3.4.2.3 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, and the transform block size log2TrafoSize.

Output of this process is the variable ctxInc.

The variables ctxOffset and ctxShift are derived as follows:

- If cIdx is equal to 0, ctxOffset is set equal to $3 * (\log_2 \text{TrafoSize} - 2) + ((\log_2 \text{TrafoSize} - 1) \gg 2)$ and ctxShift is set equal to $(\log_2 \text{TrafoSize} + 1) \gg 2$.
- Otherwise (cIdx is greater than 0), ctxOffset is set equal to 15 and ctxShift is set equal to $\log_2 \text{TrafoSize} - 2$.

The variable ctxInc is derived as follows:

$$\text{ctxInc} = (\text{binIdx} \gg \text{ctxShift}) + \text{ctxOffset} \quad (9-17)$$

9.3.4.2.4 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 (x_S, y_S), the previously decoded bins of the syntax element coded_sub_block_flag, and the transform block size log2TrafoSize.

Output of this process is the variable ctxInc.

The variable csbfCtx is derived using the current location (x_S, y_S), two previously decoded bins of the syntax element coded_sub_block_flag in scan order, and the transform block size log2TrafoSize, as follows:

- csbfCtx is initialized with 0 as follows:

$$\text{csbfCtx} = 0 \quad (9-18)$$

- When x_S is less than $(1 \ll (\log_2 \text{TrafoSize} - 2)) - 1$, csbfCtx is modified as follows:

$$\text{csbfCtx} += \text{coded_sub_block_flag}[x_S + 1][y_S] \quad (9-19)$$

- When y_S is less than $(1 \ll (\log_2 \text{TrafoSize} - 2)) - 1$, csbfCtx is modified as follows:

$$\text{csbfCtx} += \text{coded_sub_block_flag}[x_S][y_S + 1] \quad (9-20)$$

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:

$$ctxInc = \text{Min}(csbfCtx, 1) \quad (9-21)$$

- Otherwise ($cIdx$ is greater than 0), $ctxInc$ is derived as follows:

$$ctxInc = 2 + \text{Min}(csbfCtx, 1) \quad (9-22)$$

9.3.4.2.5 Derivation process of $ctxInc$ for the syntax element sig_coeff_flag

Inputs to this process are the colour component index $cIdx$, the current coefficient scan location (xC , yC), the scan order index $scanIdx$, and the transform block size $\log2TrafoSize$.

Output of this process is the variable $ctxInc$.

The variable $sigCtx$ depends on the current location (xC , yC), the colour component index $cIdx$, the transform block size, and previously decoded bins of the syntax element $coded_sub_block_flag$. For the derivation of $sigCtx$, the following applies:

- If $\log2TrafoSize$ is equal to 2, $sigCtx$ is derived using $ctxIdxMap[]$ specified in Table 9-39 as follows:

$$sigCtx = ctxIdxMap[(yC \ll 2) + xC] \quad (9-23)$$

- Otherwise, if $xC + yC$ is equal to 0, $sigCtx$ is derived as follows:

$$sigCtx = 0 \quad (9-24)$$

- Otherwise, $sigCtx$ is derived using previous values of $coded_sub_block_flag$ as follows:

- The sub-block location (xS , yS) is set equal to ($xC \gg 2$, $yC \gg 2$).

- The variable $prevCsbF$ is set equal to 0.

- When xS is less than ($1 \ll (\log2TrafoSize - 2) - 1$), the following applies:

$$prevCsbF += coded_sub_block_flag[xS + 1][yS] \quad (9-25)$$

- When yS is less than ($1 \ll (\log2TrafoSize - 2) - 1$), the following applies:

$$prevCsbF += (coded_sub_block_flag[xS][yS + 1] \ll 1) \quad (9-26)$$

- The inner sub-block location (xP , yP) is set equal to ($xC \& 3$, $yC \& 3$).

- The variable $sigCtx$ is derived as follows:

- If $prevCsbF$ is equal to 0, the following applies:

$$sigCtx = (xP + yP == 0) ? 2 : (xP + yP < 3) ? 1 : 0 \quad (9-27)$$

- Otherwise, if $prevCsbF$ is equal to 1, the following applies:

$$sigCtx = (yP == 0) ? 2 : (yP == 1) ? 1 : 0 \quad (9-28)$$

- Otherwise, if $prevCsbF$ is equal to 2, the following applies:

$$sigCtx = (xP == 0) ? 2 : (xP == 1) ? 1 : 0 \quad (9-29)$$

- Otherwise ($prevCsbF$ is equal to 3), the following applies:

$$sigCtx = 2 \quad (9-30)$$

- The variable $sigCtx$ is modified as follows:

- If $cIdx$ is equal to 0, the following applies:

- When ($xS + yS$) is greater than 0, the following applies:

$$sigCtx += 3 \quad (9-31)$$

- The variable $sigCtx$ is modified as follows:

- If $\log2TrafoSize$ is equal to 3, the following applies:

$$sigCtx += (scanIdx == 0) ? 9 : 15 \quad (9-32)$$

- Otherwise, the following applies:

$$\text{sigCtx} += 21 \quad (9-33)$$

- Otherwise (cIdx is greater than 0), the following applies:

- If $\log_2\text{TrafoSize}$ is equal to 3, the following applies:

$$\text{sigCtx} += 9 \quad (9-34)$$

- Otherwise, the following applies:

$$\text{sigCtx} += 12 \quad (9-35)$$

The context index increment ctxInc is derived using the colour component index cIdx and sigCtx as follows:

- If cIdx is equal to 0, ctxInc is derived as follows:

$$\text{ctxInc} = \text{sigCtx} \quad (9-36)$$

- Otherwise (cIdx is greater than 0), ctxInc is derived as follows:

$$\text{ctxInc} = 27 + \text{sigCtx} \quad (9-37)$$

Table 9-39 – Specification of $\text{ctxIdxMap}[i]$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\text{ctxIdxMap}[i]$	0	1	4	5	2	3	4	5	6	6	8	8	7	7	8

9.3.4.2.6 Derivation process of ctxInc for the syntax element `coeff_abs_level_greater1_flag`

Inputs to this process are the colour component index cIdx , the current sub-block scan index i , and the current coefficient scan index n within the current sub-block.

Output of this process is the variable ctxInc .

The variable ctxSet specifies the current context set and for its derivation the following applies:

- If this process is invoked for the first time for the current sub-block scan index i , the following applies:

- The variable ctxSet is initialized as follows:

- If the current sub-block scan index i is equal to 0 or cIdx is greater than 0, the following applies:

$$\text{ctxSet} = 0 \quad (9-38)$$

- Otherwise (i is greater than 0 and cIdx is equal to 0), the following applies:

$$\text{ctxSet} = 2 \quad (9-39)$$

- The variable lastGreater1Ctx is derived as follows:

- If the current sub-block with scan index i is the first one to be processed in this subclause for the current transform block, the variable lastGreater1Ctx is set equal to 1.

- Otherwise, the following applies:

- The variable lastGreater1Ctx is set equal to the value of greater1Ctx that has been derived during the last invocation of the process specified in this subclause for a previous sub-block.

- When lastGreater1Ctx is greater than 0, the variable lastGreater1Flag is set equal to the value of the syntax element `coeff_abs_level_greater1_flag` that has been used during the last invocation of the process specified in this subclause for a previous sub-block and lastGreater1Ctx is modified as follows:

- If lastGreater1Flag is equal to 1, lastGreater1Ctx is set equal to 0.

- Otherwise (lastGreater1Flag is equal to 0), lastGreater1Ctx is incremented by 1.

- When lastGreater1Ctx is equal to 0, ctxSet is incremented by one as follows:

$$\text{ctxSet} = \text{ctxSet} + 1 \quad (9-40)$$

- The variable greater1Ctx is set equal to 1.

- Otherwise (this process is not invoked for the first time for the current sub-block scan index i), the following applies:
 - The variable $ctxSet$ is set equal to the variable $ctxSet$ that has been derived during the last invocation of the process specified in this subclause.
 - The variable $greater1Ctx$ is set equal to the variable $greater1Ctx$ that has been derived during the last invocation of the process specified in this subclause.
 - When $greater1Ctx$ is greater than 0, the variable $lastGreater1Flag$ is set equal to the syntax element $coeff_abs_level_greater1_flag$ that has been used during the last invocation of the process specified in this subclause and $greater1Ctx$ is modified as follows:
 - If $lastGreater1Flag$ is equal to 1, $greater1Ctx$ is set equal to 0.
 - Otherwise ($lastGreater1Flag$ is equal to 0), $greater1Ctx$ is incremented by 1.

The context index increment $ctxInc$ is derived using the current context set $ctxSet$ and the current context $greater1Ctx$ as follows:

$$ctxInc = (ctxSet * 4) + \text{Min}(3, greater1Ctx) \quad (9-41)$$

When $cIdx$ is greater than 0, $ctxInc$ is modified as follows:

$$ctxInc = ctxInc + 16 \quad (9-42)$$

9.3.4.2.7 Derivation process of $ctxInc$ for the syntax element $coeff_abs_level_greater2_flag$

Inputs to this process are the colour component index $cIdx$, the current sub-block scan index i , and the current coefficient scan index n within the current sub-block.

Output of this process is the variable $ctxInc$.

The variable $ctxSet$ specifies the current context set and is set equal to the value of the variable $ctxSet$ that has been derived in subclause 9.3.4.2.6 for the same subset i .

The context index increment $ctxInc$ is set equal to the variable $ctxSet$ as follows:

$$ctxInc = ctxSet \quad (9-43)$$

When $cIdx$ is greater than 0, $ctxInc$ is modified as follows:

$$ctxInc = ctxInc + 4 \quad (9-44)$$

9.3.4.3 Arithmetic decoding process

9.3.4.3.1 General

Inputs to this process are $ctxTable$, $ctxIdx$, and $bypassFlag$, as derived in subclause 9.3.4.2, and the state variables $ivlCurrRange$ and $ivlOffset$ of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 9-5 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table $ctxTable$ and the $ctxIdx$ are passed to the arithmetic decoding process $\text{DecodeBin}(ctxTable, ctxIdx)$, which is specified as follows:

- If $bypassFlag$ is equal to 1, $\text{DecodeBypass}()$ as specified in subclause 9.3.4.3.4 is invoked.
- Otherwise, if $bypassFlag$ is equal to 0, $ctxTable$ is equal to 0, and $ctxIdx$ is equal to 0, $\text{DecodeTerminate}()$ as specified in subclause 9.3.4.3.5 is invoked.
- Otherwise ($bypassFlag$ is equal to 0 and $ctxTable$ is not equal to 0), $\text{DecodeDecision}()$ as specified in subclause 9.3.4.3.2 is invoked.

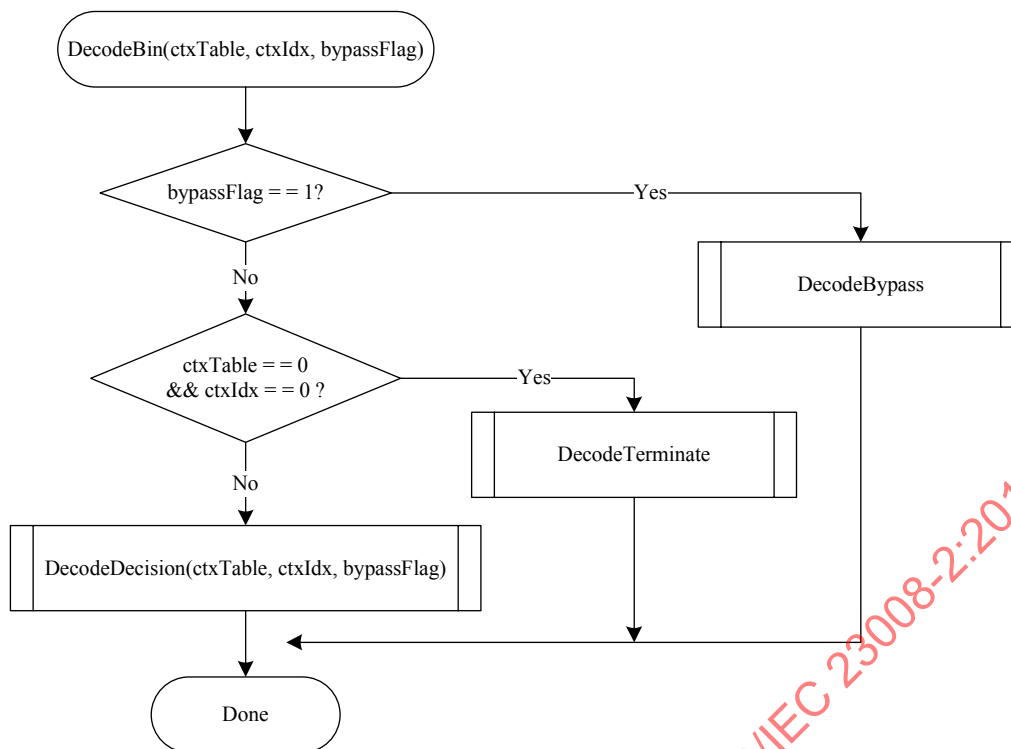


Figure 9-5 – Overview of the arithmetic decoding process for a single bin (informative)

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 p_{LPS} 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 $\{p_{LPS}(pStateIdx) \mid 0 \leq pStateIdx < 64\}$ for the LPS probability p_{LPS} . The numbering of the states is arranged in such a way that the probability state with index $pStateIdx = 0$ 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 $\{Q_1, \dots, Q_4\}$ of pre-set quantization values prior to the calculation of the new interval range. Storing a table containing all 64×4 pre-computed product values of $Q_i * p_{LPS}(pStateIdx)$ allows a multiplication-free approximation of the product $ivlCurrRange * p_{LPS}(pStateIdx)$.
- 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.

9.3.4.3.2 Arithmetic decoding process for a binary decision

9.3.4.3.2.1 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 9-6 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:

$$qRangeIdx = (ivlCurrRange \gg 6) \& 3 \quad (9-45)$$

- Given $qRangeIdx$ and $pStateIdx$ associated with $ctxTable$ and $ctxIdx$, the value of the variable $rangeTabLps$ as specified in Table 9-40 is assigned to $ivlLpsRange$:

$$ivlLpsRange = rangeTabLps[pStateIdx][qRangeIdx] \quad (9-46)$$

2. 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 $1 - \text{valMps}$, `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 is performed as specified in subclause 9.3.4.3.2.2. Depending on the current value of `ivlCurrRange`, renormalization is performed as specified in subclause 9.3.4.3.3.

9.3.4.3.2.2 State transition process

Inputs to this process are the current `pStateIdx`, the decoded value `binVal` and `valMps` values of the context variable associated with `ctxTable` and `ctxIdx`.

Outputs of this process are the updated `pStateIdx` and `valMps` of the context variable associated with `ctxIdx`.

Depending on the decoded value `binVal`, the update of the two variables `pStateIdx` and `valMps` associated with `ctxIdx` is derived as follows:

```

if( binVal == valMps )
    pStateIdx = transIdxMps( pStateIdx )
else {
    if( pStateIdx == 0 )
        valMps = 1 – valMps
    pStateIdx = transIdxLps( pStateIdx )
}

```

(9-47)

Table 9-41 specifies the transition rules `transIdxMps()` and `transIdxLps()` after decoding the value of `valMps` and $1 - \text{valMps}$, respectively.

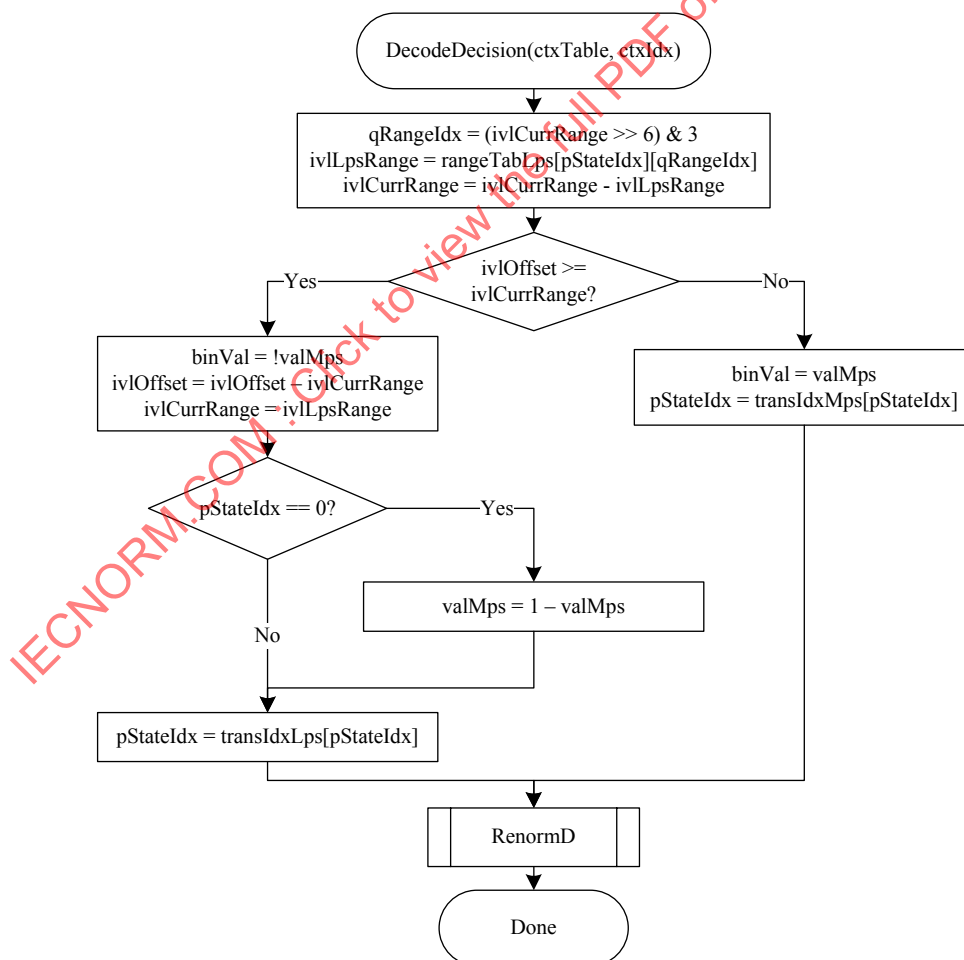


Figure 9-6 – Flowchart for decoding a decision

Table 9-40 – Specification of rangeTabLps depending on the values of pStateIdx and qRangeIdx

pStateIdx	qRangeIdx				pStateIdx	qRangeIdx			
	0	1	2	3		0	1	2	3
0	128	176	208	240	32	27	33	39	45
1	128	167	197	227	33	26	31	37	43
2	128	158	187	216	34	24	30	35	41
3	123	150	178	205	35	23	28	33	39
4	116	142	169	195	36	22	27	32	37
5	111	135	160	185	37	21	26	30	35
6	105	128	152	175	38	20	24	29	33
7	100	122	144	166	39	19	23	27	31
8	95	116	137	158	40	18	22	26	30
9	90	110	130	150	41	17	21	25	28
10	85	104	123	142	42	16	20	23	27
11	81	99	117	135	43	15	19	22	25
12	77	94	111	128	44	14	18	21	24
13	73	89	105	122	45	14	17	20	23
14	69	85	100	116	46	13	16	19	22
15	66	80	95	110	47	12	15	18	21
16	62	76	90	104	48	12	14	17	20
17	59	72	86	99	49	11	14	16	19
18	56	69	81	94	50	11	13	15	18
19	53	65	77	89	51	10	12	15	17
20	51	62	73	85	52	10	12	14	16
21	48	59	69	80	53	9	11	13	15
22	46	56	66	76	54	9	11	12	14
23	43	53	63	72	55	8	10	12	14
24	41	50	59	69	56	8	9	11	13
25	39	48	56	65	57	7	9	11	12
26	37	45	54	62	58	7	9	10	12
27	35	43	51	59	59	7	8	10	11
28	33	41	48	56	60	6	8	9	11
29	32	39	46	53	61	6	7	9	10
30	30	37	43	50	62	6	7	8	9
31	29	35	41	48	63	2	2	2	2

Table 9-41 – State transition table

pStateIdx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
transIdxLps	0	0	1	2	2	4	4	5	6	7	8	9	9	11	11	12
transIdxMps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pStateIdx	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
transIdxLps	13	13	15	15	16	16	18	18	19	19	21	21	22	22	23	24
transIdxMps	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
pStateIdx	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
transIdxLps	24	25	26	26	27	27	28	29	29	30	30	30	31	32	32	33
transIdxMps	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
pStateIdx	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
transIdxLps	33	33	34	34	35	35	35	36	36	36	37	37	37	38	38	63
transIdxMps	49	50	51	52	53	54	55	56	57	58	59	60	61	62	62	63

9.3.4.3.3 Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice segment 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 9-7. 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.

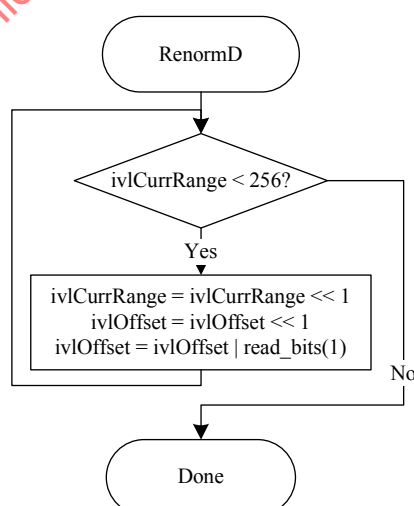


Figure 9-7 – Flowchart of renormalization

9.3.4.3.4 Bypass decoding process for binary decisions

Inputs to this process are bits from slice segment 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 9-8 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.

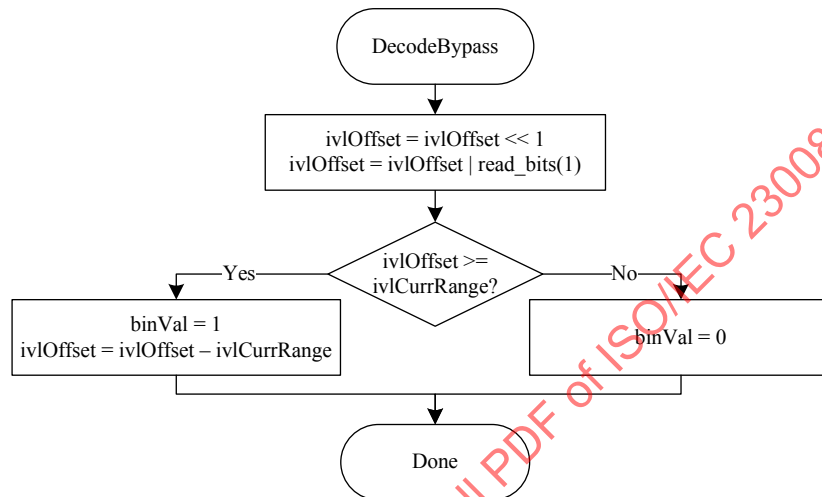


Figure 9-8 – Flowchart of bypass decoding process

9.3.4.3.5 Decoding process for binary decisions before termination

Inputs to this process are bits from slice segment 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_slice_segment_flag`, `end_of_sub_stream_one_bit` and `pcm_flag` corresponding to `ctxTable` equal to 0 and `ctxIdx` equal to 0. Figure 9-9 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_slice_segment_flag`, this last bit inserted in register `ivlOffset` is interpreted as `rbps_stop_one_bit`. When decoding `end_of_sub_stream_one_bit`, this last bit inserted in register `ivlOffset` is interpreted as `alignment_bit_equal_to_one`.
- Otherwise (`ivlOffset` is less than `ivlCurrRange`), the variable `binVal` is set equal to 0 and renormalization is performed as specified in subclause 9.3.4.3.3.

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.

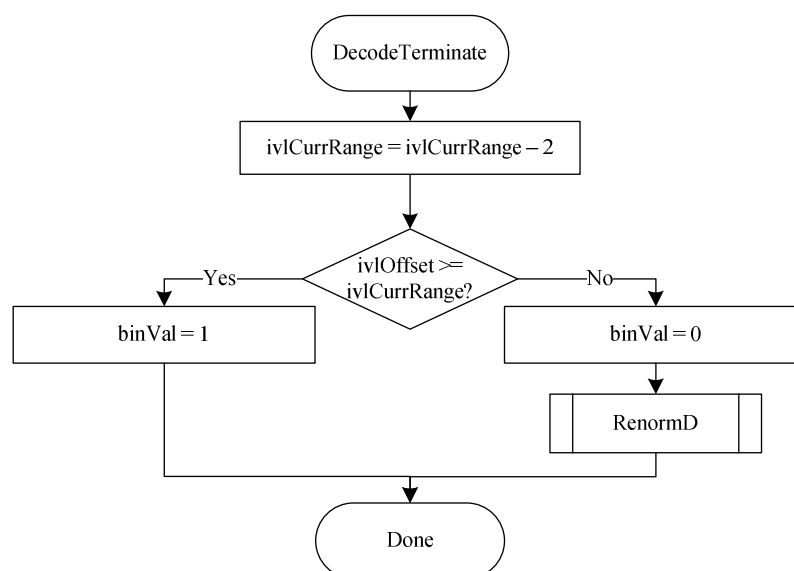


Figure 9-9 – Flowchart of decoding a decision before termination

9.3.5 Arithmetic encoding process (informative)

9.3.5.1 General

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

Inputs to this process are decisions that are to be encoded and written.

Outputs of this process are bits that are written to the RBSP.

This informative subclause describes an arithmetic encoding engine that matches the arithmetic decoding engine described in subclause 9.3.4.3. The encoding engine is essentially symmetric with the decoding engine, i.e. procedures are called in the same order. The following procedures are described in this subclause: InitEncoder, EncodeDecision, EncodeBypass, EncodeTerminate, which correspond to InitDecoder, DecodeDecision, DecodeBypass, and DecodeTerminate, respectively. The state of the arithmetic encoding engine is represented by a value of the variable *ivlLow* pointing to the lower end of a sub-interval and a value of the variable *ivlCurrRange* specifying the corresponding range of that sub-interval.

9.3.5.2 Initialization process for the arithmetic encoding engine (informative)

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

This process is invoked before encoding the first coding block of a slice segment, and after encoding any *pcm_alignment_zero_bit* and all *pcm_sample_luma* and *pcm_sample_chroma* data for a coding unit with *pcm_flag* equal to 1.

Outputs of this process are the values *ivlLow*, *ivlCurrRange*, *firstBitFlag*, *bitsOutstanding*, and *BinCountsInNalUnits* of the arithmetic encoding engine.

In the initialization procedure of the encoder, *ivlLow* is set equal to 0, and *ivlCurrRange* is set equal to 510. Furthermore, *firstBitFlag* is set equal to 1 and the counter *bitsOutstanding* is set equal to 0.

Depending on whether the current slice segment is the first slice segment of a coded picture, the following applies:

- If the current slice segment is the first slice segment of a coded picture, the counter *BinCountsInNalUnits* is set equal to 0.
- Otherwise (the current slice segment is not the first slice segment of a coded picture), the counter *BinCountsInNalUnits* is not modified. The value of *BinCountsInNalUnits* is the result of encoding all the slice segments of a coded picture that precede the current slice segment in decoding order. After initializing for the first slice segment of a coded picture as specified in this subclause, *BinCountsInNalUnits* is incremented as specified in subclauses 9.3.5.3, 9.3.5.5, and 9.3.5.6.

NOTE – The minimum register precision required for storing the values of the variables *ivlLow* and *ivlCurrRange* after invocation of any of the arithmetic encoding processes specified in subclauses 9.3.5.3, 9.3.5.5, and 9.3.5.6 is 10 bits and 9 bits, respectively. The encoding process for a binary decision (EncodeDecision) as specified in subclause 9.3.5.3 and the encoding process for a

binary decision before termination (EncodeTerminate) as specified in subclause 9.3.5.6 require a minimum register precision of 10 bits for the variable `ivlLow` and a minimum register precision of 9 bits for the variable `ivlCurrRange`. The bypass encoding process for binary decisions (EncodeBypass) as specified in subclause 9.3.5.5 requires a minimum register precision of 11 bits for the variable `ivlLow` and a minimum register precision of 9 bits for the variable `ivlCurrRange`. The precision required for the counters `bitsOutstanding` and `BinCountsInNalUnits` should be sufficiently large to prevent overflow of the related registers. When `maxBinCountInSlice` denotes the maximum total number of binary decisions to encode in one slice segment and `maxBinCountInPic` denotes the maximum total number of binary decisions to encode a picture, the minimum register precision required for the variables `bitsOutstanding` and `BinCountsInNalUnits` is given by $\text{Ceil}(\text{Log2}(\text{maxBinCountInSlice} + 1))$ and $\text{Ceil}(\text{Log2}(\text{maxBinCountInPic} + 1))$, respectively.

9.3.5.3 Encoding process for a binary decision (informative)

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

Inputs to this process are the context index `ctxIdx`, the value of `binVal` to be encoded, and the variables `ivlCurrRange`, `ivlLow` and `BinCountsInNalUnits`.

Outputs of this process are the variables `ivlCurrRange`, `ivlLow`, and `BinCountsInNalUnits`.

Figure 9-10 shows the flowchart for encoding a single decision. In a first step, the variable `ivlLpsRange` is derived as follows:

Given the current value of `ivlCurrRange`, `ivlCurrRange` is mapped to the index `qRangeIdx` of a quantized value of `ivlCurrRange` by using Equation 9-45. The value of `qRangeIdx` and the value of `pStateIdx` associated with `ctxIdx` are used to determine the value of the variable `rangeTabLps` as specified in Table 9-40, which is assigned to `ivlLpsRange`. The value of `ivlCurrRange - ivlLpsRange` is assigned to `ivlCurrRange`.

In a second step, the value of `binVal` is compared to `valMps` associated with `ctxIdx`. When `binVal` is different from `valMps`, `ivlCurrRange` is added to `ivlLow` and `ivlCurrRange` is set equal to the value `ivlLpsRange`. Given the encoded decision, the state transition is performed as specified in subclause 9.3.4.3.2.2. Depending on the current value of `ivlCurrRange`, renormalization is performed as specified in subclause 9.3.5.4. Finally, the variable `BinCountsInNalUnits` is incremented by 1.

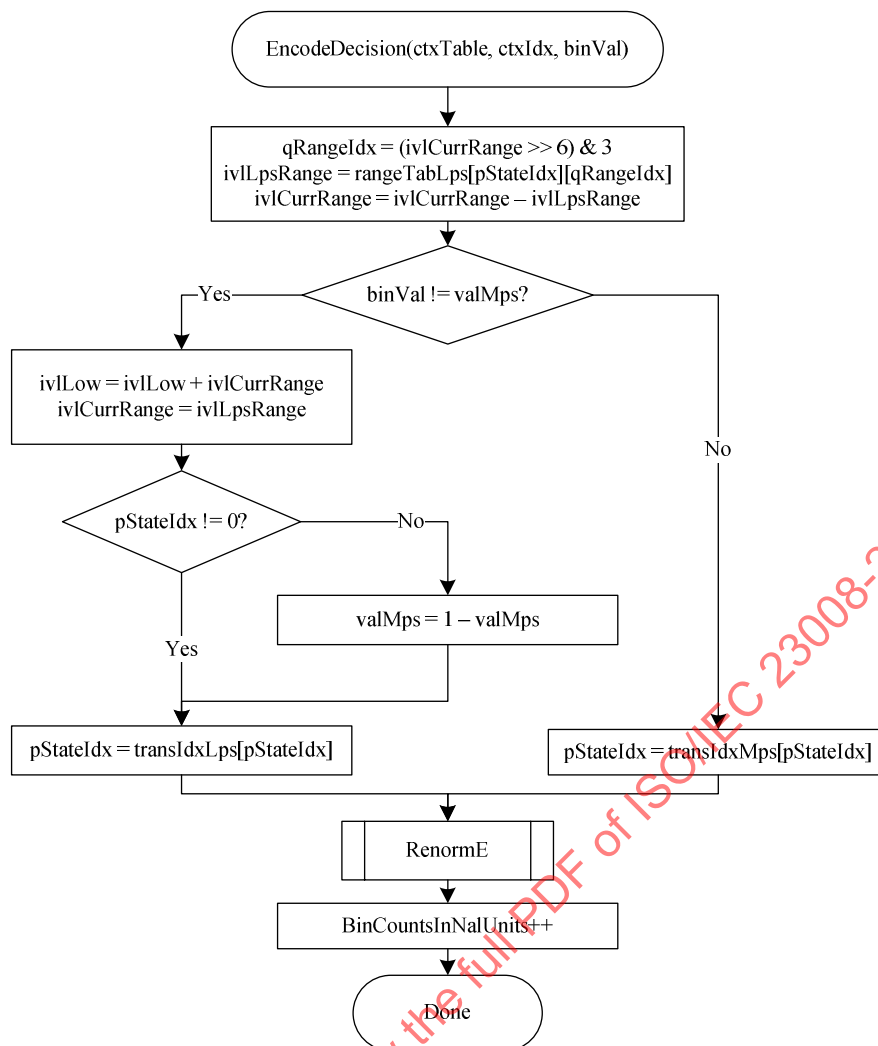


Figure 9-10 – Flowchart for encoding a decision

9.3.5.4 Renormalization process in the arithmetic encoding engine (informative)

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

Inputs to this process are the variables ivlCurrRange, ivlLow, firstBitFlag, and bitsOutstanding.

Outputs of this process are zero or more bits written to the RBSP and the updated variables ivlCurrRange, ivlLow, firstBitFlag, and bitsOutstanding.

Renormalization is illustrated in Figure 9-11.

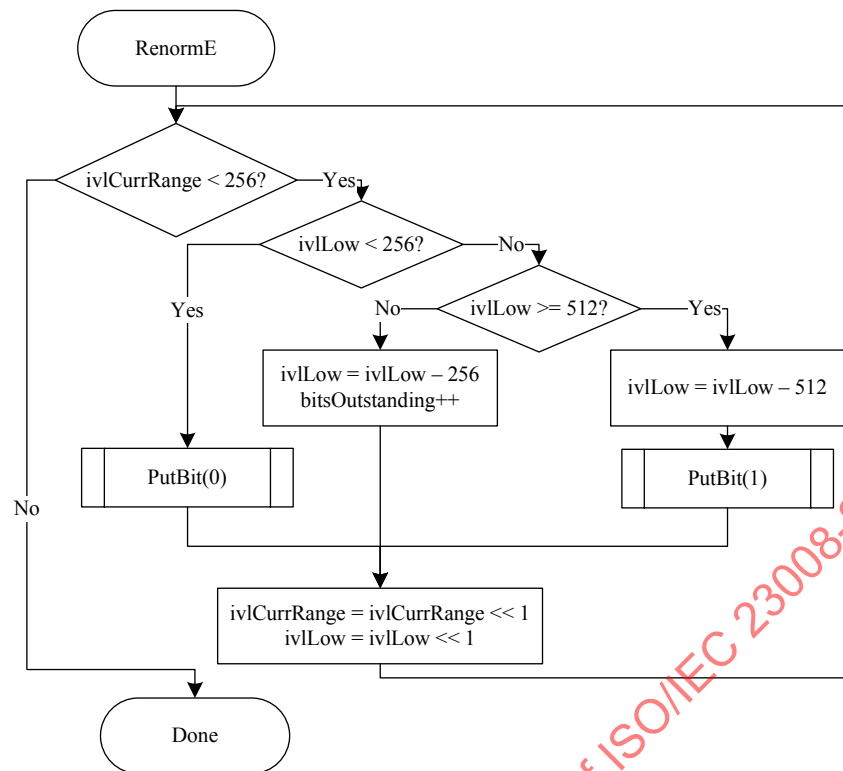


Figure 9-11 – Flowchart of renormalization in the encoder

The PutBit() procedure described in Figure 9-12 provides carry over control. It uses the function WriteBits(B, N) that writes N bits with value B to the bitstream and advances the bitstream pointer by N bit positions. This function assumes the existence of a bitstream pointer with an indication of the position of the next bit to be written to the bitstream by the encoding process.

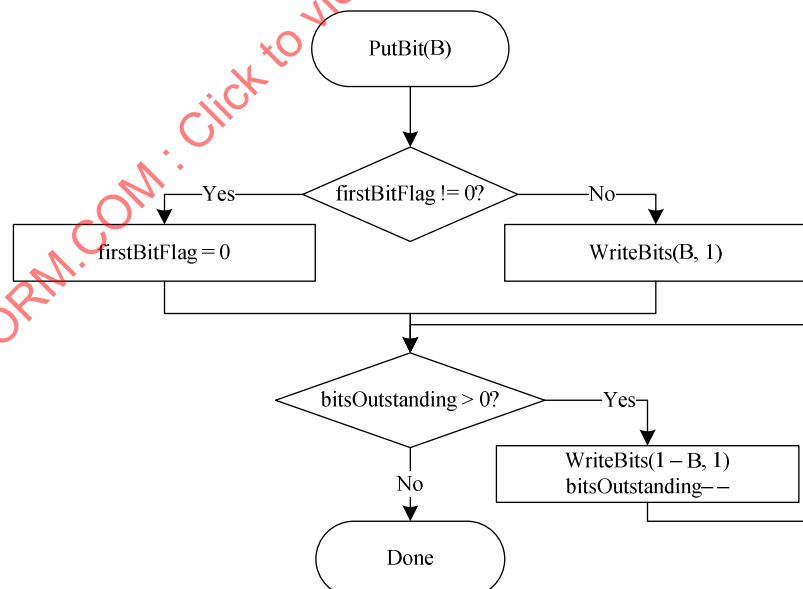


Figure 9-12 – Flowchart of PutBit(B)

9.3.5.5 Bypass encoding process for binary decisions (informative)

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

Inputs to this process are the variables binVal, ivlLow, ivlCurrRange, bitsOutstanding, and BinCountsInNalUnits.

Output of this process is a bit written to the RBSP and the updated variables `ivlLow`, `bitsOutstanding`, and `BinCountsInNalUnits`.

This encoding process applies to all binary decisions with `bypassFlag` equal to 1. Renormalization is included in the specification of this process as given in Figure 9-13.

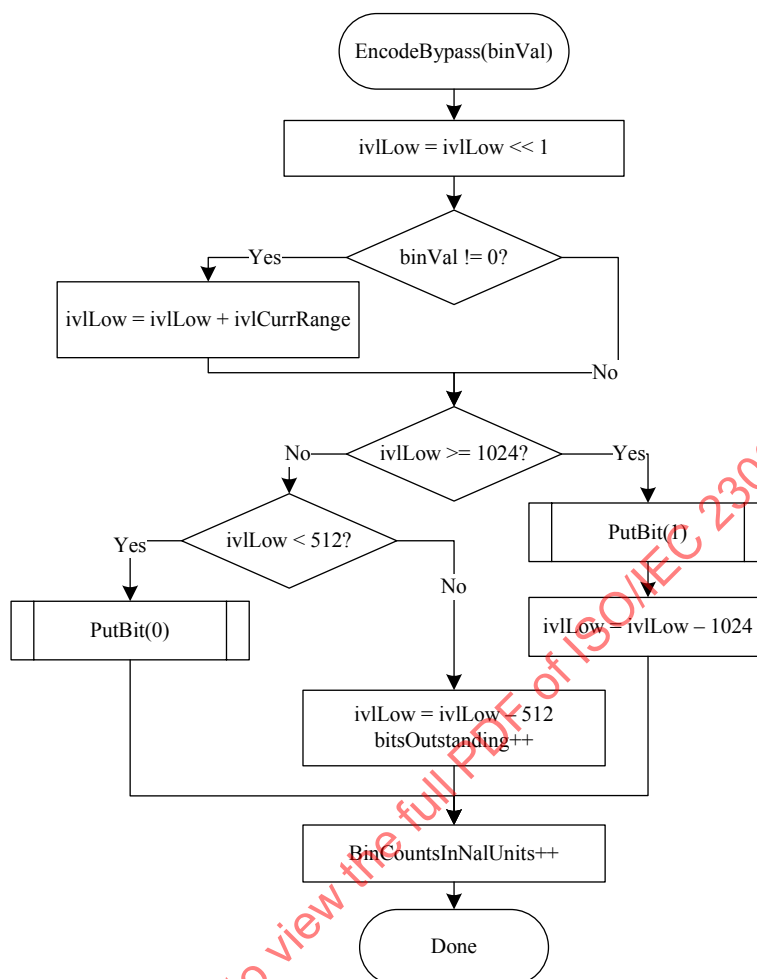


Figure 9-13 – Flowchart of encoding bypass

9.3.5.6 Encoding process for a binary decision before termination (informative)

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

Inputs to this process are the variables `binVal`, `ivlCurrRange`, `ivlLow`, `bitsOutstanding`, and `BinCountsInNalUnits`.

Outputs of this process are zero or more bits written to the RBSP and the updated variables `ivlLow`, `ivlCurrRange`, `bitsOutstanding`, and `BinCountsInNalUnits`.

This encoding routine shown in Figure 9-14 applies to encoding of `end_of_slice_segment_flag`, `end_of_sub_stream_one_bit`, and `pcm_flag`, all associated with `ctxIdx` equal to 0.

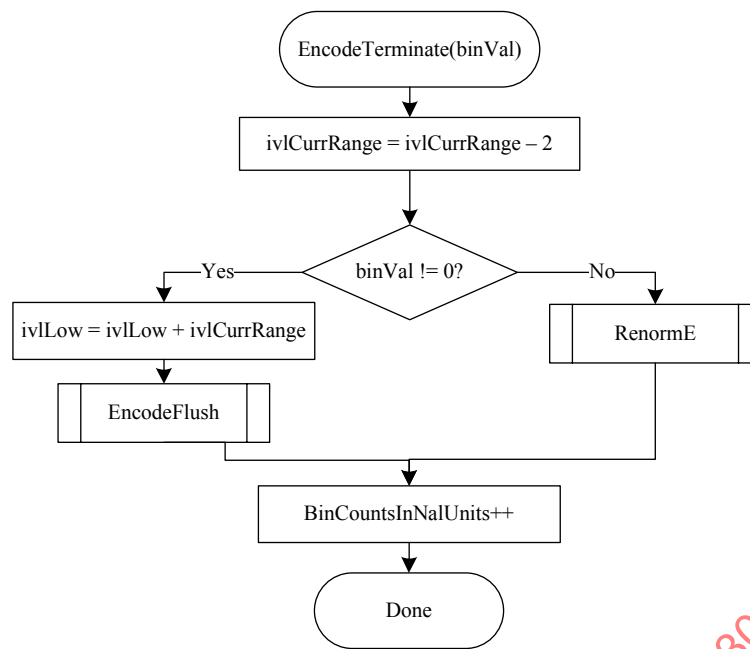


Figure 9-14 – Flowchart of encoding a decision before termination

When the value of binVal to encode is equal to 1, CABAC encoding is terminated and the flushing procedure shown in Figure 9-15 is applied. In this flushing procedure, the last bit written by WriteBits(B, N) is equal to 1. When encoding end_of_slice_segment_flag, this last bit is interpreted as rbsp_stop_one_bit. When encoding end_of_sub_stream_one_bit, this last bit is interpreted as alignment_bit equal_to_one.

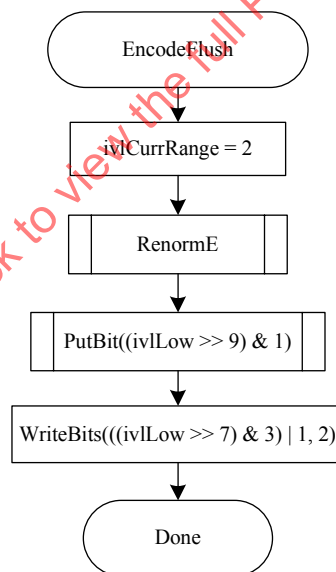


Figure 9-15 – Flowchart of flushing at termination

9.3.5.7 Byte stuffing process (informative)

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

This process is invoked after encoding the last coding block of the last slice segment of a picture and after encapsulation.

Inputs to this process are the number of bytes NumBytesInVclNalUnits of all VCL NAL units of a picture, the number of minimum CUs PicSizeInMinCbsY in the picture, and the number of binary symbols BinCountsInNalUnits resulting from encoding the contents of all VCL NAL units of the picture.

NOTE – The value of BinCountsInNalUnits is the result of encoding all slice segments of a coded picture. After initializing for the first slice segment of a coded picture as specified in subclause 9.3.5.2, BinCountsInNalUnits is incremented as specified in subclauses 9.3.5.3, 9.3.5.5, and 9.3.5.6.

Outputs of this process are zero or more bytes appended to the NAL unit.

Let the variable k be set equal to $\text{Ceil}((\text{Ceil}(3 * (\text{BinCountsInNalUnits} - \text{RawMinCuBits} * \text{PicSizeInMinCbsY}) \div 1024) - \text{NumBytesInVclNalUnits}) \div 3)$. Depending on the value of k the following applies:

- If k is less than or equal to 0, no `cabac_zero_word` is appended to the NAL unit.
- Otherwise (k is greater than 0), the 3-byte sequence 0x000003 is appended k times to the NAL unit after encapsulation, where the first two bytes 0x0000 represent a `cabac_zero_word` and the third byte 0x03 represents an `emulation_prevention_three_byte`.

10 Sub-bitstream extraction process

Inputs to this process are a bitstream, a target highest TemporalId value `tIdTarget`, and a target layer identifier list `layerIdListTarget`.

Output of this process is a sub-bitstream.

It is a requirement of bitstream conformance for the input bitstream that any output sub-bitstream of the process specified in this subclause with `tIdTarget` equal to any value in the range of 0 to 6, inclusive, and `layerIdListTarget` equal to the layer identifier list associated with a layer set specified in the active video parameter set shall be a conforming bitstream.

NOTE 1 – A conforming bitstream contains one or more coded slice segment NAL units with `nuh_layer_id` equal to 0 and TemporalId equal to 0.

The output sub-bitstream is derived as follows:

- When one or more of the following two conditions are true, remove all SEI NAL units that have `nuh_layer_id` equal to 0 and that contain a non-nested buffering period SEI message, a non-nested picture timing SEI message, or a non-nested decoding unit information SEI message:
 - `layerIdListTarget` does not include all the values of `nuh_layer_id` in all NAL units in the bitstream.
 - `tIdTarget` is less than the greatest TemporalId in all NAL units in the bitstream.

NOTE 2 – A "smart" bitstream extractor may include appropriate non-nested buffering picture SEI messages, non-nested picture timing SEI messages, and non-nested decoding unit information SEI messages in the extracted sub-bitstream, provided that the SEI messages applicable to the sub-bitstream were present as nested SEI messages in the original bitstream.
- Remove all NAL units with TemporalId greater than `tIdTarget` or `nuh_layer_id` not among the values included in `layerIdListTarget`.

Annex A

Profiles, tiers and levels

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

A.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.

NOTE 1 – This Specification does not include individually selectable "options" at the decoder, as this would increase interoperability difficulties.

Each profile specifies a subset of algorithmic features and limits that shall be supported by all decoders conforming to that profile.

NOTE 2 – Encoders are not required to make use of any particular subset of features supported in a profile.

Each level of a tier specifies a set of limits on the values that may be taken by the syntax elements of this Specification. The same set of tier and level definitions is used with all profiles, but individual implementations may support a different tier and within a tier a different level for each supported profile. For any given profile, a level of a tier generally corresponds to a particular decoder processing load and memory capability.

The profiles that are specified in subclause A.3 are also referred to as the profiles specified in Annex A.

A.2 Requirements on video decoder capability

Capabilities of video decoders conforming to this Specification are specified in terms of the ability to decode video streams conforming to the constraints of profiles, tiers and levels specified in this annex. When expressing the capabilities of a decoder for a specified profile, the tier and level supported for that profile should also be expressed.

Specific values are specified in this annex for the syntax elements `general_profile_idc`, `general_tier_flag`, and `general_level_idc`. All other values of `general_profile_idc`, `general_tier_flag`, and `general_level_idc` are reserved for future use by ITU-T | ISO/IEC.

NOTE – Decoders should not infer that a reserved value of `general_profile_idc` between the values specified in this Specification indicates intermediate capabilities between the specified profiles, as there are no restrictions on the method to be chosen by ITU-T | ISO/IEC for the use of such future reserved values. However, decoders should infer that a reserved value of `general_level_idc` associated with a particular value of `general_tier_flag` between the values specified in this Specification indicates intermediate capabilities between the specified levels of the tier.

A.3 Profiles

A.3.1 General

All constraints for PPSs that are specified are constraints for PPSs that are activated when the bitstream is decoded. All constraints for SPSS that are specified are constraints for SPSSs that are activated when the bitstream is decoded.

The variable `RawCtuBits` is derived as follows:

$$\text{RawCtuBits} = \text{CtbSizeY} * \text{CtbSizeY} * \text{BitDepth}_Y + 2 * (\text{CtbWidthC} * \text{CtbHeightC}) * \text{BitDepth}_C \quad (\text{A-1})$$

A.3.2 Main profile

Bitstreams conforming to the Main profile shall obey the following constraints:

- SPSSs shall have `chroma_format_idc` equal to 1 only.
- SPSSs shall have `bit_depth_luma_minus8` equal to 0 only.
- SPSSs shall have `bit_depth_chroma_minus8` equal to 0 only.
- `CtbLog2SizeY` shall be in the range of 4 to 6, inclusive.
- When a PPS has `tiles_enabled_flag` is equal to 1, it shall have `entropy_coding_sync_enabled_flag` equal to 0.

- When a PPS has `tiles_enabled_flag` is equal to 1, `ColumnWidthInLumaSamples[ i ]` shall be greater than or equal to 256 for all values of `i` in the range of 0 to `num_tile_columns_minus1`, inclusive, and `RowHeightInLumaSamples[ j ]` shall be greater than or equal to 64 for all values of `j` in the range of 0 to `num_tile_rows_minus1`, inclusive.
- The number of times `read_bits( 1 )` is called in subclauses 9.3.4.3.3 and 9.3.4.3.4 when parsing `coding_tree_unit( )` data for any coding tree unit shall be less than or equal to $5 * \text{RawCtuBits} / 3$.
- The level constraints specified for the Main profile in subclause A.4 shall be fulfilled.

Conformance of a bitstream to the Main profile is indicated by `general_profile_idc` being equal to 1 or `general_profile_compatibility_flag[ 1 ]` being equal to 1.

NOTE – When `general_profile_compatibility_flag[ 1 ]` is equal to 1, `general_profile_compatibility_flag[ 2 ]` should also be equal to 1.

Decoders conforming to the Main profile at a specific level (identified by a specific value of `general_level_idc`) of a specific tier (identified by a specific value of `general_tier_flag`) shall be capable of decoding all bitstreams for which all of the following conditions apply:

- `general_profile_compatibility_flag[ 1 ]` is equal to 1.
- `general_level_idc` represents a level lower than or equal to the specified level.
- `general_tier_flag` represents a tier lower than or equal to the specified tier.

A.3.3 Main 10 profile

Bitstreams conforming to the Main 10 profile shall obey the following constraints:

- SPSs shall have `chroma_format_idc` equal to 1 only.
- SPSs shall have `bit_depth_luma_minus8` in the range of 0 to 2, inclusive.
- SPSs shall have `bit_depth_chroma_minus8` in the range of 0 to 2, inclusive.
- `CtbLog2SizeY` shall be in the range of 4 to 6, inclusive.
- When a PPS has `tiles_enabled_flag` is equal to 1, it shall have `entropy_coding_sync_enabled_flag` equal to 0.
- When a PPS has `tiles_enabled_flag` is equal to 1, `ColumnWidthInLumaSamples[ i ]` shall be greater than or equal to 256 for all values of `i` in the range of 0 to `num_tile_columns_minus1`, inclusive, and `RowHeightInLumaSamples[ j ]` shall be greater than or equal to 64 for all values of `j` in the range of 0 to `num_tile_rows_minus1`, inclusive.
- The number of times `read_bits( 1 )` is called in subclauses 9.3.4.3.3 and 9.3.4.3.4 when parsing `coding_tree_unit( )` data for any coding tree unit shall be less than or equal to $5 * \text{RawCtuBits} / 3$.
- The level constraints specified for the Main 10 profile in subclause A.4 shall be fulfilled.

Conformance of a bitstream to the Main 10 profile is indicated by `general_profile_idc` being equal to 2 or `general_profile_compatibility_flag[ 2 ]` being equal to 1.

Decoders conforming to the Main 10 profile at a specific level (identified by a specific value of `general_level_idc`) shall be capable of decoding all bitstreams for which all of the following conditions apply:

- `general_profile_compatibility_flag[ 1 ]` is equal to 1 or `general_profile_compatibility_flag[ 2 ]` is equal to 1.
- `general_level_idc` represents a level lower than or equal to the specified level.
- `general_tier_flag` represents a tier lower than or equal to the specified tier.

A.3.4 Main Still Picture profile

Bitstreams conforming to the Main Still Picture profile shall obey the following constraints:

- The bitstream shall contain only one picture.
- SPSs shall have `chroma_format_idc` equal to 1 only.
- SPSs shall have `bit_depth_luma_minus8` equal to 0 only.
- SPSs shall have `bit_depth_chroma_minus8` equal to 0 only.
- SPSs shall have `sps_max_dec_pic_buffering_minus1[ sps_max_sub_layers_minus1 ]` equal to 0 only.
- `CtbLog2SizeY` shall be in the range of 4 to 6, inclusive.

- When a PPS has `tiles_enabled_flag` is equal to 1, it shall have `entropy_coding_sync_enabled_flag` equal to 0.
- When a PPS has `tiles_enabled_flag` is equal to 1, `ColumnWidthInLumaSamples[ i ]` shall be greater than or equal to 256 for all values of `i` in the range of 0 to `num_tile_columns_minus1`, inclusive, and `RowHeightInLumaSamples[ j ]` shall be greater than or equal to 64 for all values of `j` in the range of 0 to `num_tile_rows_minus1`, inclusive.
- The number of times `read_bits( 1 )` is called in subclauses 9.3.4.3.3 and 9.3.4.3.4 when parsing `coding_tree_unit()` data for any coding tree unit shall be less than or equal to $5 * \text{RawCtuBits} / 3$.
- The level constraints specified for the Main Still Picture profile in subclause A.4 shall be fulfilled.

Conformance of a bitstream to the Main Still Picture profile is indicated by `general_profile_idc` being equal to 3 or `general_profile_compatibility_flag[ 3 ]` being equal to 1.

Decoders conforming to the Main Still Picture profile at a specific level (identified by a specific value of `general_level_idc`) shall be capable of decoding all bitstreams for which all of the following conditions apply:

- `general_profile_compatibility_flag[ 3 ]` is equal to 1.
- `general_level_idc` represents a level lower than or equal to the specified level.
- `general_tier_flag` represents a tier lower than or equal to the specified tier.

A.4 Tiers and levels

A.4.1 General tier and level limits

For purposes of comparison of tier capabilities, the tier with `general_tier_flag` equal to 0 is considered to be a lower tier than the tier with `general_tier_flag` equal to 1.

For purposes of comparison of level capabilities, a particular level of a specific tier is considered to be a lower level than some other level of the same tier when the value of the `general_level_idc` of the particular level is less than that of the other level.

The following is specified for expressing the constraints in this annex:

- Let access unit `n` be the `n`-th access unit in decoding order, with the first access unit being access unit 0 (i.e. the 0-th access unit).
- Let picture `n` be the coded picture or the corresponding decoded picture of access unit `n`.
- Let the variable `CpbBrVclFactor` be equal to 1000.
- Let the variable `CpbBrNalFactor` be equal to 1100.

Bitstreams conforming to a profile at a specified tier and level shall obey the following constraints for each bitstream conformance test as specified in Annex C:

- a) `PicSizeInSamplesY` shall be less than or equal to `MaxLumaPs`, where `MaxLumaPs` is specified in Table A-1.
- b) The value of `pic_width_in_luma_samples` shall be less than or equal to $\text{Sqrt}(\text{MaxLumaPs} * 8)$.
- c) The value of `pic_height_in_luma_samples` shall be less than or equal to $\text{Sqrt}(\text{MaxLumaPs} * 8)$.
- d) The value of `sps_max_dec_pic_buffering_minus1[ HighestTid ] + 1` shall be less than or equal to `MaxDpbSize`, which is derived as follows:

```

if( PicSizeInSamplesY <= ( MaxLumaPs >> 2 ) )
    MaxDpbSize = Min( 4 * maxDpbPicBuf, 16 )
else if( PicSizeInSamplesY <= ( MaxLumaPs >> 1 ) )
    MaxDpbSize = Min( 2 * maxDpbPicBuf, 16 )
else if( PicSizeInSamplesY <= ( ( 3 * MaxLumaPs ) >> 2 ) )
    MaxDpbSize = Min( ( 4 * maxDpbPicBuf ) / 3, 16 )
else
    MaxDpbSize = maxDpbPicBuf

```

(A-2)

where `MaxLumaPs` is specified in Table A-1 and `maxDpbPicBuf` is equal to 6.

- e) For level 5 and higher levels, the value of `CtbSizeY` shall be equal to 32 or 64.
- f) The value of `NumPocTotalCurr` shall be less than or equal to 8.

- g) The value of num_tile_columns_minus1 shall be less than MaxTileCols and num_tile_rows_minus1 shall be less than MaxTileRows, where MaxTileCols and MaxTileRows are specified in Table A-1.
- h) For the VCL HRD parameters, CpbSize[i] shall be less than or equal to CpbBrVclFactor * MaxCPB for at least one value of i in the range of 0 to cpb_cnt_minus1[HighestTid], inclusive, where CpbSize[i] is specified in subclause E.2.3 based on parameters selected as specified in subclause C.1 and MaxCPB is specified in Table A-1 in units of CpbBrVclFactor bits.
- i) For the NAL HRD parameters, CpbSize[i] shall be less than or equal to CpbBrNalFactor * MaxCPB for at least one value of i in the range of 0 to cpb_cnt_minus1[HighestTid], inclusive, where CpbSize[i] is specified in subclause E.2.3 based on parameters selected as specified in subclause C.1 and MaxCPB is specified in Table A-1 in units of CpbBrNalFactor bits.

Table A-1 specifies the limits for each level of each tier.

A tier and level to which the bitstream conforms are indicated by the syntax elements general_tier_flag and general_level_idc as follows:

- general_tier_flag equal to 0 indicates conformance to the Main tier, and general_tier_flag equal to 1 indicates conformance to the High tier, according to the tier constraints specified in Table A-1. general_tier_flag shall be equal to 0 for levels below level 4 (corresponding to the entries in Table A-1 marked with "-").
- general_level_idc shall be set equal to a value of 30 times the level number specified in Table A-1.

Table A-1 – General tier and level limits

Level	Max luma picture size MaxLumaPs (samples)	Max CPB size MaxCPB (1000 bits)		Max slice segments per picture MaxSliceSegmentsPerPicture	Max # of tile rows MaxTileRows	Max # of tile columns MaxTileCols
		Main tier	High tier			
1	36 864	350	-	16	1	1
2	122 880	1 500	-	16	1	1
2.1	245 760	3 000	-	20	1	1
3	552 960	6 000	-	30	2	2
3.1	983 040	10 000	-	40	3	3
4	2 228 224	12 000	30 000	75	5	5
4.1	2 228 224	20 000	50 000	75	5	5
5	8 912 896	25 000	100 000	200	11	10
5.1	8 912 896	40 000	160 000	200	11	10
5.2	8 912 896	60 000	240 000	200	11	10
6	35 651 584	60 000	240 000	600	22	20
6.1	35 651 584	120 000	480 000	600	22	20
6.2	35 651 584	240 000	800 000	600	22	20

Informative subclause A.4.3 shows the effect of these limits on picture rates for several example picture formats.

A.4.2 Profile-specific level limits for the Main and Main 10 profiles

The following is specified for expressing the constraints in this annex:

- Let the variable fR be set equal to $1 \div 300$.

Bitstreams conforming to the Main or Main 10 profile at a specified tier and level shall obey the following constraints for each bitstream conformance test as specified in Annex C:

- a) The nominal removal time of access unit n (with n greater than 0) from the CPB, as specified in subclause C.2.3, shall satisfy the constraint that $AuNominalRemovalTime[n] - AuCpbRemovalTime[n - 1]$ is greater than or equal to $Max(PicSizeInSamplesY \div MaxLumaSr, fR)$ for the value of $PicSizeInSamplesY$ of picture $n - 1$, where $MaxLumaSr$ is the value specified in Table A-2 that applies to picture $n - 1$.
- b) The difference between consecutive output times of pictures from the DPB, as specified in subclause C.3.3, shall satisfy the constraint that $DpbOutputInterval[n]$ is greater than or equal to $Max(PicSizeInSamplesY \div MaxLumaSr, fR)$ for the value of $PicSizeInSamplesY$ of picture n , where $MaxLumaSr$ is the value specified in Table A-2 for picture n , provided that picture n is a picture that is output and is not the last picture of the bitstream that is output.
- c) The removal time of access unit 0 shall satisfy the constraint that the number of slice segments in picture 0 is less than or equal to $Min(MaxSliceSegmentsPerPicture * MaxLumaSr / MaxLumaPs * (AuCpbRemovalTime[0] - AuNominalRemovalTime[0]) + MaxSliceSegmentsPerPicture * PicSizeInSamplesY / MaxLumaPs, MaxSliceSegmentsPerPicture)$, for the value of $PicSizeInSamplesY$ of picture 0, where $MaxSliceSegmentsPerPicture$, $MaxLumaPs$ and $MaxLumaSr$ are the values specified in Table A-1 and Table A-2, respectively, that apply to picture 0.
- d) The difference between consecutive CPB removal times of access units n and $n - 1$ (with n greater than 0) shall satisfy the constraint that the number of slice segments in picture n is less than or equal to $Min(MaxSliceSegmentsPerPicture * MaxLumaSr / MaxLumaPs * (AuCpbRemovalTime[n] - AuCpbRemovalTime[n - 1]), MaxSliceSegmentsPerPicture)$, where $MaxSliceSegmentsPerPicture$, $MaxLumaPs$ and $MaxLumaSr$ are the values specified in Table A-1 and Table A-2, respectively, that apply to picture n .
- e) For the VCL HRD parameters, $BitRate[i]$ shall be less than or equal to $CpbBrVclFactor * MaxBR$ for at least one value of i in the range of 0 to $cpb_cnt_minus1[HighestTid]$, inclusive, where $BitRate[i]$ is specified in subclause E.2.3 based on parameters selected as specified in subclause C.1 and $MaxBR$ is specified in Table A-2 in units of $CpbBrVclFactor$ bits/s.
- f) For the NAL HRD parameters, $BitRate[i]$ shall be less than or equal to $CpbBrNalFactor * MaxBR$ for at least one value of i in the range of 0 to $cpb_cnt_minus1[HighestTid]$, inclusive, where $BitRate[i]$ is specified in subclause E.2.3 based on parameters selected as specified in subclause C.1 and $MaxBR$ is specified in Table A-2 in units of $CpbBrNalFactor$ bits/s.
- g) The sum of the $NumBytesInNalUnit$ variables for access unit 0 shall be less than or equal to $1.5 * (Max(PicSizeInSamplesY, fR * MaxLumaSr) + MaxLumaSr * (AuCpbRemovalTime[0] - AuNominalRemovalTime[0])) \div MinCr$ for the value of $PicSizeInSamplesY$ of picture 0, where $MaxLumaSr$ and $MinCr$ are the values specified in Table A-2 that apply to picture 0.
- h) The sum of the $NumBytesInNalUnit$ variables for access unit n (with n greater than 0) shall be less than or equal to $1.5 * MaxLumaSr * (AuCpbRemovalTime[n] - AuCpbRemovalTime[n - 1]) \div MinCr$, where $MaxLumaSr$ and $MinCr$ are the values specified in Table A-2 that apply to picture n .
- i) The removal time of access unit 0 shall satisfy the constraint that the number of tiles in picture 0 is less than or equal to $Min(MaxTileCols * MaxTileRows * 120 * (AuCpbRemovalTime[0] - AuNominalRemovalTime[0]) + MaxTileCols * MaxTileRows * PicSizeInSamplesY / MaxLumaPs, MaxTileCols * MaxTileRows)$, for the value of $PicSizeInSamplesY$ of picture 0, where $MaxTileCols$ and $MaxTileRows$ are the values specified in Table A-1 that apply to picture 0.
- j) The difference between consecutive CPB removal times of access units n and $n - 1$ (with n greater than 0) shall satisfy the constraint that the number of tiles in picture n is less than or equal to $Min(MaxTileCols * MaxTileRows * 120 * (AuCpbRemovalTime[n] - AuCpbRemovalTime[n - 1]), MaxTileCols * MaxTileRows)$, where $MaxTileCols$ and $MaxTileRows$ are the values specified in Table A-1 that apply to picture n .

Table A-2 – Tier and level limits for the Main and Main 10 profiles

Level	Max luma sample rate MaxLumaSr (samples/sec)	Max bit rate MaxBR (1000 bits/s)		Min Compression Ratio MinCr
		Main tier	High tier	
1	552 960	128	-	2
2	3 686 400	1 500	-	2
2.1	7 372 800	3 000	-	2
3	16 588 800	6 000	-	2
3.1	33 177 600	10 000	-	2
4	66 846 720	12 000	30 000	4
4.1	133 693 440	20 000	50 000	4
5	267 386 880	25 000	100 000	6
5.1	534 773 760	40 000	160 000	8
5.2	1 069 547 520	60 000	240 000	8
6	1 069 547 520	60 000	240 000	8
6.1	2 139 095 040	120 000	480 000	8
6.2	4 278 190 080	240 000	800 000	6

A.4.3 Effect of level limits on picture rate for the Main and Main 10 profiles (informative)

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

Informative Tables A-3 and A-4 provide examples of maximum picture rates for the Main and Main 10 profiles for various picture formats when MinCbSizeY is equal to 64.

Table A-3 – Maximum picture rates (pictures per second) at level 1 to 4.3 for some example picture sizes when MinCbSizeY is equal to 64

Level:				1	2	2.1	3	3.1	4	4.1
Max luma picture size (samples):				36 864	122 880	245 760	552 960	983 040	2 228 224	2 228 224
Max luma sample rate (samples/sec)				552 960	3 686 400	7 372 800	16 588 800	33 177 600	66 846 720	133 693 440
Format nickname	Luma width	Luma height	Luma picture size							
SQCIF	128	96	16 384	33.7	225.0	300.0	300.0	300.0	300.0	300.0
QCIF	176	144	36 864	15.0	100.0	200.0	300.0	300.0	300.0	300.0
QVGA	320	240	81 920	-	45.0	90.0	202.5	300.0	300.0	300.0
525 SIF	352	240	98 304	-	37.5	75.0	168.7	300.0	300.0	300.0
CIF	352	288	122 880	-	30.0	60.0	135.0	270.0	300.0	300.0
525 HHR	352	480	196 608	-	-	37.5	84.3	168.7	300.0	300.0
625 HHR	352	576	221 184	-	-	33.3	75.0	150.0	300.0	300.0
Q720p	640	360	245 760	-	-	30.0	67.5	135.0	272.0	300.0
VGA	640	480	327 680	-	-	-	50.6	101.2	204.0	300.0
525 4SIF	704	480	360 448	-	-	-	46.0	92.0	185.4	300.0
525 SD	720	480	393 216	-	-	-	42.1	84.3	170.0	300.0
4CIF	704	576	405 504	-	-	-	40.9	81.8	164.8	300.0
625 SD	720	576	442 368	-	-	-	37.5	75.0	151.1	300.0
480p (16:9)	864	480	458 752	-	-	-	36.1	72.3	145.7	291.4
SVGA	800	600	532 480	-	-	-	31.1	62.3	125.5	251.0
QHD	960	540	552 960	-	-	-	30.0	60.0	120.8	241.7
XGA	1024	768	786 432	-	-	-	-	42.1	85.0	170.0
720p HD	1280	720	983 040	-	-	-	-	33.7	68.0	136.0
4VGA	1280	960	1 228 800	-	-	-	-	-	54.4	108.8
SXGA	1280	1024	1 310 720	-	-	-	-	-	51.0	102.0
525 16SIF	1408	960	1 351 680	-	-	-	-	-	49.4	98.9
16CIF	1408	1152	1 622 016	-	-	-	-	-	41.2	82.4
4SVGA	1600	1200	1 945 600	-	-	-	-	-	34.3	68.7
1080 HD	1920	1080	2 088 960	-	-	-	-	-	32.0	64.0
2Kx1K	2048	1024	2 097 152	-	-	-	-	-	31.8	63.7
2Kx1080	2048	1080	2 228 224	-	-	-	-	-	30.0	60.0
4XGA	2048	1536	3 145 728	-	-	-	-	-	-	-
16VGA	2560	1920	4 915 200	-	-	-	-	-	-	-
3616x1536 (2.35:1)	3616	1536	5 603 328	-	-	-	-	-	-	-
3672x1536 (2.39:1)	3680	1536	5 701 632	-	-	-	-	-	-	-
3840x2160 (4*HD)	3840	2160	8 355 840	-	-	-	-	-	-	-
4Kx2K	4096	2048	8 388 608	-	-	-	-	-	-	-
4096x2160	4096	2160	8 912 896	-	-	-	-	-	-	-
4096x2304 (16:9)	4096	2304	9 437 184	-	-	-	-	-	-	-
7680x4320	7680	4320	33 423 360	-	-	-	-	-	-	-
8192x4096	8192	4096	33 554 432	-	-	-	-	-	-	-
8192x4320	8192	4320	35 651 584	-	-	-	-	-	-	-

Table A-4 – Maximum picture rates (pictures per second) at level 5 to 6.2 for some example picture sizes when MinCbSizeY is equal to 64

Level:				5	5.1	5.2	6	6.1	6.2
Max luma picture size (samples):				8 912 896	8 912 896	8 912 896	35 651 584	35 651 584	35 651 584
Max luma sample rate (samples/sec)				267 386 880	534 773 760	1 069 547 520	1 069 547 520	2 139 095 040	4 278 190 080
Format nickname	Luma width	Luma height	Luma picture size						
SQCIF	128	96	16 384	300.0	300.0	300.0	300.0	300.0	300.0
QCIF	176	144	36 864	300.0	300.0	300.0	300.0	300.0	300.0
QVGA	320	240	81 920	300.0	300.0	300.0	300.0	300.0	300.0
525 SIF	352	240	98 304	300.0	300.0	300.0	300.0	300.0	300.0
CIF	352	288	122 880	300.0	300.0	300.0	300.0	300.0	300.0
525 HHR	352	480	196 608	300.0	300.0	300.0	300.0	300.0	300.0
625 HHR	352	576	221 184	300.0	300.0	300.0	300.0	300.0	300.0
Q720p	640	360	245 760	300.0	300.0	300.0	300.0	300.0	300.0
VGA	640	480	327 680	300.0	300.0	300.0	300.0	300.0	300.0
525 4SIF	704	480	360 448	300.0	300.0	300.0	300.0	300.0	300.0
525 SD	720	480	393 216	300.0	300.0	300.0	300.0	300.0	300.0
4CIF	704	576	405 504	300.0	300.0	300.0	300.0	300.0	300.0
625 SD	720	576	442 368	300.0	300.0	300.0	300.0	300.0	300.0
480p (16:9)	864	480	458 752	300.0	300.0	300.0	300.0	300.0	300.0
SVGA	800	600	532 480	300.0	300.0	300.0	300.0	300.0	300.0
QHD	960	540	552 960	300.0	300.0	300.0	300.0	300.0	300.0
XGA	1024	768	786 432	300.0	300.0	300.0	300.0	300.0	300.0
720p HD	1280	720	983 040	272.0	300.0	300.0	300.0	300.0	300.0
4VGA	1280	960	1 228 800	217.6	300.0	300.0	300.0	300.0	300.0
SXGA	1280	1024	1 310 720	204.0	300.0	300.0	300.0	300.0	300.0
525 16SIF	1408	960	1 351 680	197.8	300.0	300.0	300.0	300.0	300.0
16CIF	1408	1152	1 622 016	164.8	300.0	300.0	300.0	300.0	300.0
4SVGA	1600	1200	1 945 600	137.4	274.8	300.0	300.0	300.0	300.0
1080 HD	1920	1080	2 088 960	128.0	256.0	300.0	300.0	300.0	300.0
2Kx1K	2048	1024	2 097 152	127.5	255.0	300.0	300.0	300.0	300.0
2Kx1080	2048	1080	2 228 224	120.0	240.0	300.0	300.0	300.0	300.0
4XGA	2048	1536	3 145 728	85.0	170.0	300.0	300.0	300.0	300.0
16VGA	2560	1920	4 915 200	54.4	108.8	217.6	217.6	300.0	300.0
3616x1536 (2.35:1)	3616	1536	5 603 328	47.7	95.4	190.8	190.8	300.0	300.0
3672x1536 (2.39:1)	3680	1536	5 701 632	46.8	93.7	187.5	187.5	300.0	300.0
3840x2160 (4*HD)	3840	2160	8 355 840	32.0	64.0	128.0	256.0	300.0	300.0
4Kx2K	4096	2048	8 388 608	31.8	63.7	127.5	127.5	255.0	300.0
4096x2160	4096	2160	8 912 896	30.0	60.0	120.0	120.0	240.0	300.0
4096x2304 (16:9)	4096	2304	9 437 184	-	-	-	113.3	226.6	300.0
4096x3072	4096	3072	12 582 912	-	-	-	85.0	170.0	300.0
7680x4320	7680	4320	33 423 360	-	-	-	32.0	64.0	128.0
8192x4096	8192	4096	33 554 432	-	-	-	31.8	63.7	127.5
8192x4320	8192	4320	35 651 584	-	-	-	30.0	60.0	120.0

The following should be noted in regard to the examples shown in Tables A-3 and A-4:

- This Specification is a variable-picture-size specification. The specific listed picture sizes are illustrative examples only.
- The example luma picture sizes were computed by rounding up the luma width and luma height to multiples of 64 before computing the product of these quantities, to reflect the potential use of MinCbSizeY equal to 64 for these picture sizes, as `pic_width_in_luma_samples` and `pic_height_in_luma_samples` are each required to be a multiple of MinCbSizeY. For some illustrated values of luma width and luma height, a somewhat higher number of pictures per second can be supported when MinCbSizeY is less than 64.

- As used in the examples, "525" refers to typical use for environments using 525 analogue scan lines (of which approximately 480 lines contain the visible picture region), and "625" refers to environments using 625 analogue scan lines (of which approximately 576 lines contain the visible picture region).
- XGA is also known as (aka) XVGA, 4SVGA aka UXGA, 16XGA aka 4Kx3K, CIF aka 625 SIF, 625 HHR aka 2CIF aka half 625 D-1, aka half 625 ITU-R BT.601, 525 SD aka 525 D-1 aka 525 ITU-R BT.601, 625 SD aka 625 D-1 aka 625 ITU-R BT.601.

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

Annex B

Byte stream format

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

B.1 General

This annex specifies syntax and semantics of a byte stream format specified for use by applications that deliver some or all of the NAL unit stream as an ordered stream of bytes or bits within which the locations of NAL unit boundaries need to be identifiable from patterns in the data, such as Rec. ITU-T H.222.0 | ISO/IEC 13818-1 systems or Rec. ITU-T H.320 systems. For bit-oriented delivery, the bit order for the byte stream format is specified to start with the MSB of the first byte, proceed to the LSB of the first byte, followed by the MSB of the second byte, etc.

The byte stream format consists of a sequence of byte stream NAL unit syntax structures. Each byte stream NAL unit syntax structure contains one start code prefix followed by one `nal_unit( NumBytesInNalUnit )` syntax structure. It may (and under some circumstances, it shall) also contain an additional `zero_byte` syntax element. It may also contain one or more additional `trailing_zero_8bits` syntax elements. When it is the first byte stream NAL unit in the bitstream, it may also contain one or more additional `leading_zero_8bits` syntax elements.

B.2 Byte stream NAL unit syntax and semantics

B.2.1 Byte stream NAL unit syntax

byte_stream_nal_unit(NumBytesInNalUnit) {	Descriptor
while(next_bits(24) != 0x000001 && next_bits(32) != 0x00000001)	
leading_zero_8bits /* equal to 0x00 */	f(8)
if(next_bits(24) != 0x000001)	
zero_byte /* equal to 0x00 */	f(8)
start_code_prefix_one_3bytes /* equal to 0x000001 */	f(24)
nal_unit(NumBytesInNalUnit)	
while(more_data_in_byte_stream() && next_bits(24) != 0x000001 && next_bits(32) != 0x00000001)	
trailing_zero_8bits /* equal to 0x00 */	f(8)
}	

B.2.2 Byte stream NAL unit semantics

The order of byte stream NAL units in the byte stream shall follow the decoding order of the NAL units contained in the byte stream NAL units (see subclause 7.4.2.4). The content of each byte stream NAL unit is associated with the same access unit as the NAL unit contained in the byte stream NAL unit (see subclause 7.4.2.4.4).

leading_zero_8bits is a byte equal to 0x00.

NOTE – The `leading_zero_8bits` syntax element can only be present in the first byte stream NAL unit of the bitstream, because (as shown in the syntax diagram of subclause B.2.1) any bytes equal to 0x00 that follow a NAL unit syntax structure and precede the four-byte sequence 0x00000001 (which is to be interpreted as a `zero_byte` followed by a `start_code_prefix_one_3bytes`) will be considered to be `trailing_zero_8bits` syntax elements that are part of the preceding byte stream NAL unit.

zero_byte is a single byte equal to 0x00.

When one or more of the following conditions are true, the `zero_byte` syntax element shall be present:

- The `nal_unit_type` within the `nal_unit( )` syntax structure is equal to `VPS_NUT`, `SPS_NUT` or `PPS_NUT`.
- The byte stream NAL unit syntax structure contains the first NAL unit of an access unit in decoding order, as specified in subclause 7.4.2.4.4.

start_code_prefix_one_3bytes is a fixed-value sequence of 3 bytes equal to 0x000001. This syntax element is called a start code prefix.

trailing_zero_8bits is a byte equal to 0x00.

B.3 Byte stream NAL unit decoding process

Input to this process consists of an ordered stream of bytes consisting of a sequence of byte stream NAL unit syntax structures.

Output of this process consists of a sequence of NAL unit syntax structures.

At the beginning of the decoding process, the decoder initializes its current position in the byte stream to the beginning of the byte stream. It then extracts and discards each `leading_zero_8bits` syntax element (when present), moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next four bytes in the bitstream form the four-byte sequence `0x00000001`.

The decoder then performs the following step-wise process repeatedly to extract and decode each NAL unit syntax structure in the byte stream until the end of the byte stream has been encountered (as determined by unspecified means) and the last NAL unit in the byte stream has been decoded:

1. When the next four bytes in the bitstream form the four-byte sequence `0x00000001`, the next byte in the byte stream (which is a `zero_byte` syntax element) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this discarded byte.
2. The next three-byte sequence in the byte stream (which is a `start_code_prefix_one_3bytes`) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this three-byte sequence.
3. `NumBytesInNalUnit` is set equal to the number of bytes starting with the byte at the current position in the byte stream up to and including the last byte that precedes the location of one or more of the following conditions:
 - A subsequent byte-aligned three-byte sequence equal to `0x000000`,
 - A subsequent byte-aligned three-byte sequence equal to `0x000001`,
 - The end of the byte stream, as determined by unspecified means.
4. `NumBytesInNalUnit` bytes are removed from the bitstream and the current position in the byte stream is advanced by `NumBytesInNalUnit` bytes. This sequence of bytes is `nal_unit(NumBytesInNalUnit)` and is decoded using the NAL unit decoding process.
5. When the current position in the byte stream is not at the end of the byte stream (as determined by unspecified means) and the next bytes in the byte stream do not start with a three-byte sequence equal to `0x000001` and the next bytes in the byte stream do not start with a four byte sequence equal to `0x00000001`, the decoder extracts and discards each `trailing_zero_8bits` syntax element, moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next bytes in the byte stream form the four-byte sequence `0x00000001` or the end of the byte stream has been encountered (as determined by unspecified means).

B.4 Decoder byte-alignment recovery (informative)

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

Many applications provide data to a decoder in a manner that is inherently byte aligned, and thus have no need for the bit-oriented byte alignment detection procedure described in this subclause.

A decoder is said to have byte alignment with a bitstream when the decoder has determined whether or not the positions of data in the bitstream are byte-aligned. When a decoder does not have byte alignment with the bitstream, the decoder may examine the incoming bitstream for the binary pattern '00000000 00000000 00000000 00000001' (31 consecutive bits equal to 0 followed by a bit equal to 1). The bit immediately following this pattern is the first bit of an aligned byte following a start code prefix. Upon detecting this pattern, the decoder will be byte-aligned with the bitstream and positioned at the start of a NAL unit in the bitstream.

Once byte aligned with the bitstream, the decoder can examine the incoming bitstream data for subsequent three-byte sequences `0x000001` and `0x000003`.

When the three-byte sequence `0x000001` is detected, this is a start code prefix.

When the three-byte sequence `0x000003` is detected, the third byte (`0x03`) is an `emulation_prevention_three_byte` to be discarded as specified in subclause 7.4.2.

When an error in the bitstream syntax is detected (e.g. a non-zero value of the `forbidden_zero_bit` or one of the three-byte or four-byte sequences that are prohibited in subclause 7.4.2), the decoder may consider the detected condition as an indication that byte alignment may have been lost and may discard all bitstream data until the detection of byte alignment at a later position in the bitstream as described above in this subclause.

Annex C

Hypothetical reference decoder

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

C.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 `nal_unit_type` 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).

Figure C-1 shows the types of bitstream conformance points checked by the HRD.

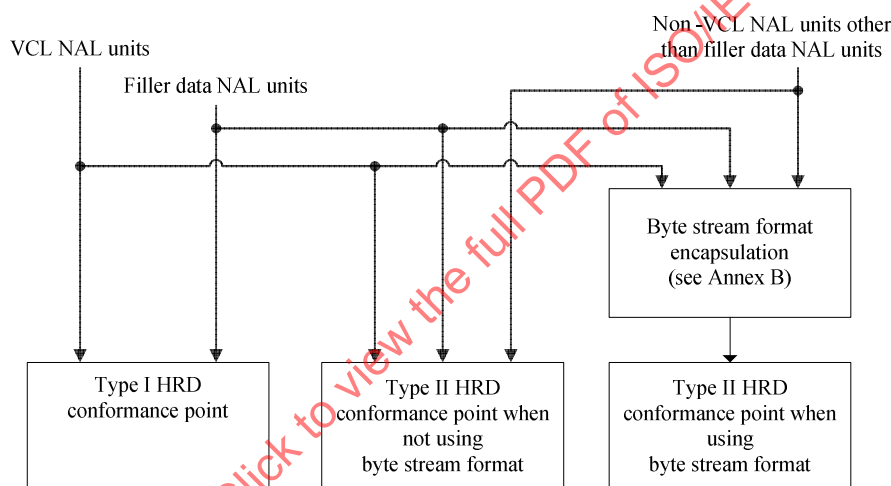


Figure C-1 – Structure of byte streams and NAL unit streams for HRD conformance checks

The syntax elements of non-VCL NAL units (or their default values for some of the syntax elements), required for the HRD, are specified in the semantic subclauses of clause 7, Annexes D and E.

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 may be part of the SPS syntax structure or the VPS syntax structure.

Multiple tests may be needed for checking the conformance of a bitstream, which is referred to as the bitstream under test. For each test, the following steps apply in the order listed:

1. An operation point under test, denoted as `TargetOp`, is selected. The layer identifier list `OpLayerIdList` of `TargetOp` consists of the list of `nuh_layer_id` values, in increasing order of `nuh_layer_id` values, present in the bitstream subset associated with `TargetOp`, which is a subset of the `nuh_layer_id` values present in the bitstream under test. The `OpTid` of `TargetOp` is equal to the highest `TemporalId` present in the bitstream subset associated with `TargetOp`.
2. `TargetDecLayerIdList` is set equal to `OpLayerIdList` of `TargetOp`, `HighestTid` is set equal to `OpTid` of `TargetOp`, and the sub-bitstream extraction process as specified in clause 10 is invoked with the bitstream under test, `HighestTid`, and `TargetDecLayerIdList` as inputs, and the output is assigned to `BitstreamToDecode`.
3. The `hrd_parameters()` syntax structure and the `sub_layer_hrd_parameters()` syntax structure applicable to `TargetOp` are selected. If `TargetDecLayerIdList` contains all `nuh_layer_id` values present in the bitstream under

test, the `hrd_parameters()` syntax structure in the active SPS (or provided through an external means not specified in this Specification) is selected. Otherwise, the `hrd_parameters()` syntax structure in the active VPS (or provided through some external means not specified in this Specification) that applies to `TargetOp` 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`.

4. 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.
5. 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, 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, and when `SubPicHrdFlag` is equal to 1 and `sub_pic_cpb_params_in_pic_timing_sei_flag` is equal to 0, the decoding unit information SEI messages (present in `BitstreamToDecode` or available through external means not specified in this Specification) that are associated with decoding units in the access unit and apply to `TargetOp` are selected.
6. A value of `SchedSelIdx` is selected. The selected `SchedSelIdx` shall be in the range of 0 to `cpb_cnt_minus1[ HighestTid ]`, inclusive, where `cpb_cnt_minus1[ HighestTid ]` is found in the `sub_layer_hrd_parameters(HighestTid)` syntax structure as selected above.
7. When the coded picture in access unit 0 has `nal_unit_type` equal to `CRA_NUT` or `BLA_W_LP`, and `irap_cpb_params_present_flag` in the selected buffering period SEI message is equal to 1, either of the following applies for selection of the initial CPB removal delay and delay offset:
 - If `NalHrdModeFlag` is equal to 1, the default initial CPB removal delay and delay offset represented by `nal_initial_cpb_removal_delay[ SchedSelIdx ]` and `nal_initial_cpb_removal_offset[ SchedSelIdx ]`, respectively, in the selected buffering period SEI message are selected. Otherwise, the default initial CPB removal delay and delay offset represented by `vcl_initial_cpb_removal_delay[ SchedSelIdx ]` and `vcl_initial_cpb_removal_offset[ SchedSelIdx ]`, respectively, in the selected buffering period SEI message are selected. The variable `DefaultInitCpbParamsFlag` is set equal to 1.
 - If `NalHrdModeFlag` is equal to 0, the alternative initial CPB removal delay and delay offset represented by `nal_initial_alt_cpb_removal_delay[ SchedSelIdx ]` and `nal_initial_alt_cpb_removal_offset[ SchedSelIdx ]`, respectively, in the selected buffering period SEI message are selected. Otherwise, the alternative initial CPB removal delay and delay offset represented by `vcl_initial_alt_cpb_removal_delay[ SchedSelIdx ]` and `vcl_initial_alt_cpb_removal_offset[ SchedSelIdx ]`, respectively, in the selected buffering period SEI message are selected. The variable `DefaultInitCpbParamsFlag` is set equal to 0, and the RASL access units associated with access unit 0 are discarded from `BitstreamToDecode` and the remaining bitstream is assigned to `BitstreamToDecode`.
8. When `sub_pic_hrd_params_present_flag` in the selected `hrd_parameters()` syntax structure is equal to 1, the CPB is scheduled to operate either at the access unit level (in which case the variable `SubPicHrdFlag` is set equal to 0) or at the sub-picture level (in which case the variable `SubPicHrdFlag` is set equal to 1).

For each operation point under test, the number of bitstream conformance tests to be performed is equal to $n0 * n1 * (n2 * 2 + n3) * n4$, where the values of $n0$, $n1$, $n2$, $n3$, and $n4$ 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 `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` and for each of which both of the following conditions are true:
 - `nal_unit_type` is equal to `CRA_NUT` or `BLA_W_LP` for the VCL NAL units;

- The associated buffering period SEI message applicable to TargetOp has `irap_cpb_params_present_flag` equal to 1.
- `n3` is the number of access units in BitstreamToDecode that each is associated with a buffering period SEI message applicable to TargetOp and for each of which one or both of the following conditions are true:
 - `nal_unit_type` is equal to neither `CRA_NUT` nor `BLA_W_LP` for the VCL NAL units;
 - The associated buffering period SEI message applicable to TargetOp has `irap_cpb_params_present_flag` equal to 0.
- `n4` is derived as follows:
 - If `sub_pic_hrd_params_present_flag` in the selected `hrd_parameters()` syntax structure is equal to 0, `n4` is equal to 1;
 - Otherwise, `n4` is equal to 2.

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 1 – NAL HRD parameters established by a value of `SchedSelIdx` for the Type II conformance point shown in Figure C-1 are sufficient to also establish VCL HRD conformance for the Type I conformance point shown in Figure C-1 for the same values of `InitCpbRemovalDelay[ SchedSelIdx ]`, `BitRate[ SchedSelIdx ]`, and `CpbSize[ SchedSelIdx ]` for the VBR case (`cbr_flag[ SchedSelIdx ]` equal to 0). This is because the data flow into the Type I conformance point is a subset of the data flow into the Type II conformance point and because, for the VBR case, the CPB is allowed to become empty and stay empty until the time a next picture is scheduled to begin to arrive. For example, when decoding a CVS conforming to one or more of the profiles specified in Annex A using the decoding process specified in clauses 2 through 10, when NAL HRD parameters are provided for the Type II conformance point that not only fall within the bounds set for NAL HRD parameters for profile conformance in item f) of subclause A.4.2 but also fall within the bounds set for VCL HRD parameters for profile conformance in item e) of subclause A.4.2, conformance of the VCL HRD for the Type I conformance point is also assured to fall within the bounds of item e) of subclause A.4.2.

All VPSs, SPSs and PPSs referred to in the VCL NAL units, and the corresponding buffering period, picture timing and decoding unit information 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, buffering period SEI messages, picture timing SEI messages, or decoding unit information 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 2 – 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 3 – When HRD information is contained within the bitstream, it is possible to verify the conformance of a bitstream to the requirements of this subclause based solely on information contained in the bitstream. When the HRD information is not present in the bitstream, as is the case for all "stand-alone" Type I bitstreams, conformance can only be verified when the HRD data 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.

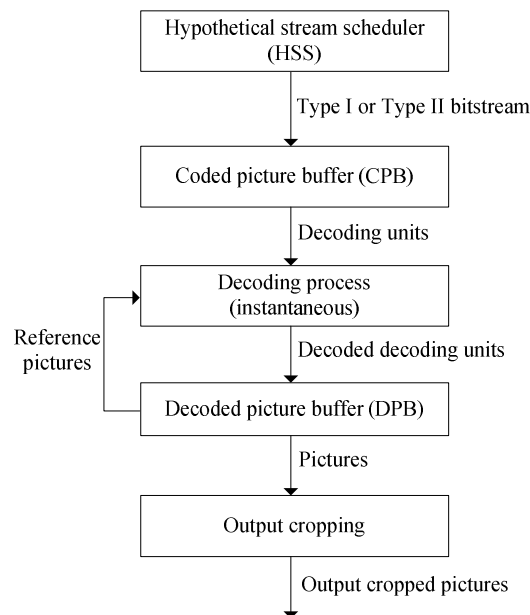


Figure C-2 – HRD buffer model

For each bitstream conformance test, the CPB size (number of bits) is $CpbSize[SchedSelIdx]$ as specified in subclause E.2.3, where $SchedSelIdx$ and the HRD parameters are specified above in this subclause. The DPB size (number of picture storage buffers) is $sps_max_dec_pic_buffering_minus1[HighestTid] + 1$.

The variable $SubPicHrdPreferredFlag$ is either specified by external means, or when not specified by external means, set equal to 0.

When the value of the variable $SubPicHrdFlag$ has not been set by step 8 above in this subclause, it is derived as follows:

$$SubPicHrdFlag = SubPicHrdPreferredFlag \ \&\& \ sub_pic_hrd_params_present_flag \quad (C-1)$$

If $SubPicHrdFlag$ is equal to 0, the HRD operates at access unit level and each decoding unit is an access unit. Otherwise the HRD operates at sub-picture level and each decoding unit is a subset of an access unit.

NOTE 4 – If the HRD operates at access unit level, each time a decoding unit that is an entire access unit is removed from the CPB. Otherwise (the HRD operates at sub-picture level), each time a decoding unit that is a subset of an access unit is removed from the CPB. In both cases, each time an entire decoded picture is output from the DPB, though the picture output time is derived based on the differently derived CPB removal times and the differently signalled DPB output delays.

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.
- Each decoding unit is referred to as decoding unit m , where the number m identifies the particular decoding unit. The first decoding unit in decoding order in access unit 0 is referred to as decoding unit 0. The value of m is incremented by 1 for each subsequent decoding unit in decoding order.

NOTE 5 – The numbering of decoding units is relative to the first decoding unit in access unit 0.

- 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 decoding unit 0, with the both the CPB and the DPB being set to be empty (the DPB fullness is set equal to 0).

NOTE 6 – 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 HSS.
- The data associated with each decoding unit are removed and decoded instantaneously by the instantaneous decoding process at the CPB removal time of the decoding 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 subclause C.2, the instantaneous decoder operation is specified in clauses 2 through 10, the operation of the DPB is specified in subclause C.3, and the output cropping is specified in subclause C.3.3 and subclause 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 subclauses E.1.2 and E.2.2. The HRD is initialized as specified by the buffering period SEI message specified in subclauses D.2.2 and D.3.2. The removal timing of decoding units from the CPB and output timing of decoded pictures from the DPB is specified using information in picture timing SEI messages (specified in subclauses D.2.3 and D.3.3) or in decoding unit information SEI messages (specified in subclauses D.2.21 and D.3.21). All timing information relating to a specific decoding unit shall arrive prior to the CPB removal time of the decoding unit.

The requirements for bitstream conformance are specified in subclause C.4, and the HRD is used to check conformance of bitstreams as specified above in this subclause and to check conformance of decoders as specified in subclause C.5.

NOTE 7 – 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 decoding unit is not necessarily an integer.

The variable ClockTick is derived as follows and is called a clock tick:

$$\text{ClockTick} = \text{vui_num_units_in_tick} \div \text{vui_time_scale} \quad (\text{C-2})$$

The variable ClockSubTick is derived as follows and is called a clock sub-tick:

$$\text{ClockSubTick} = \text{ClockTick} \div (\text{tick_divisor_minus2} + 2) \quad (\text{C-3})$$

C.2 Operation of coded picture buffer (CPB)

C.2.1 General

The specifications in this subclause apply independently to each set of CPB parameters that is present and to both the Type I and Type II conformance points shown in Figure C-1, and the set of CPB parameters is selected as specified in subclause C.1.

C.2.2 Timing of decoding unit arrival

If SubPicHrdFlag is equal to 0, the variable subPicParamsFlag is set equal to 0, and the process in specified in the remainder of this subclause is invoked with a decoding unit being considered as an access unit, for derivation of the initial and final CPB arrival times for access unit n.

Otherwise (SubPicHrdFlag is equal to 1), the process in specified in the remainder of this subclause is first invoked with the variable subPicParamsFlag set equal to 0 and a decoding unit being considered as an access unit, for derivation of the initial and final CPB arrival times for access unit n, and then invoked with subPicParamsFlag set equal to 1 and a decoding unit being considered as a subset of an access unit, for derivation of the initial and final CPB arrival times for the decoding units in access unit n.

The variables InitCpbRemovalDelay[SchedSelIdx] and InitCpbRemovalDelayOffset[SchedSelIdx] are derived as follows:

- If one or more of the following conditions are true, InitCpbRemovalDelay[SchedSelIdx] and InitCpbRemovalDelayOffset[SchedSelIdx] are set equal to the values of the buffering period SEI message syntax elements nal_initial_alt_cpb_removal_delay[SchedSelIdx] and nal_initial_alt_cpb_removal_offset[SchedSelIdx], respectively, when NalHrdModeFlag is equal to 1, or vcl_initial_alt_cpb_removal_delay[SchedSelIdx] and vcl_initial_alt_cpb_removal_offset[SchedSelIdx], respectively, when NalHrdModeFlag is equal to 0, where the buffering period SEI message syntax elements are selected as specified in subclause C.1:
 - Access unit 0 is a BLA access unit for which the coded picture has nal_unit_type equal to BLA_W_RADL or BLA_N_LP, and the value of irap_cpb_params_present_flag of the buffering period SEI message is equal to 1.
 - Access unit 0 is a BLA access unit for which the coded picture has nal_unit_type equal to BLA_W_LP or is a CRA access unit, and the value of irap_cpb_params_present_flag of the buffering period SEI message is equal to 1, and one or more of the following conditions are true:
 - UseAltCpbParamsFlag for access unit 0 is equal to 1.

- DefaultInitCpbParamsFlag is equal to 0.
- The value of subPicParamsFlag is equal to 1.
- Otherwise, 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 subclause C.1.

The time at which the first bit of decoding unit m begins to enter the CPB is referred to as the initial arrival time $\text{initArrivalTime}[m]$.

The initial arrival time of decoding unit m is derived as follows:

- If the decoding unit is decoding unit 0 (i.e. $m = 0$), $\text{initArrivalTime}[0] = 0$,
- Otherwise (the decoding unit is decoding unit m with $m > 0$), the following applies:
 - If $\text{cbr_flag}[\text{SchedSelIdx}]$ is equal to 1, the initial arrival time for decoding unit m is equal to the final arrival time (which is derived below) of decoding unit $m - 1$, i.e.

$$\begin{aligned} &\text{if(!subPicParamsFlag)} \\ &\quad \text{initArrivalTime}[m] = \text{AuFinalArrivalTime}[m - 1] \\ &\text{else} \\ &\quad \text{initArrivalTime}[m] = \text{DuFinalArrivalTime}[m - 1] \end{aligned} \quad (\text{C-4})$$

- Otherwise ($\text{cbr_flag}[\text{SchedSelIdx}]$ is equal to 0), the initial arrival time for decoding unit m is derived as follows:

$$\begin{aligned} &\text{if(!subPicParamsFlag)} \\ &\quad \text{initArrivalTime}[m] = \text{Max}(\text{AuFinalArrivalTime}[m - 1], \text{initArrivalEarliestTime}[m]) \\ &\text{else} \\ &\quad \text{initArrivalTime}[m] = \text{Max}(\text{DuFinalArrivalTime}[m - 1], \text{initArrivalEarliestTime}[m]) \end{aligned} \quad (\text{C-5})$$

where $\text{initArrivalEarliestTime}[m]$ is derived as follows:

- The variable $\text{tmpNominalRemovalTime}$ is derived as follows:

$$\begin{aligned} &\text{if(!subPicParamsFlag)} \\ &\quad \text{tmpNominalRemovalTime} = \text{AuNominalRemovalTime}[m] \\ &\text{else} \\ &\quad \text{tmpNominalRemovalTime} = \text{DuNominalRemovalTime}[m] \end{aligned} \quad (\text{C-6})$$

where $\text{AuNominalRemovalTime}[m]$ and $\text{DuNominalRemovalTime}[m]$ are the nominal CPB removal time of access unit m and decoding unit m , respectively, as specified in subclause C.2.3.

- If decoding unit m is not the first decoding unit of a subsequent buffering period, $\text{initArrivalEarliestTime}[m]$ is derived as follows:

$$\begin{aligned} \text{initArrivalEarliestTime}[m] = &\text{tmpNominalRemovalTime} - (\text{InitCpbRemovalDelay}[\text{SchedSelIdx}] \\ &+ \text{InitCpbRemovalDelayOffset}[\text{SchedSelIdx}]) \div 90000 \end{aligned} \quad (\text{C-7})$$

- Otherwise (decoding unit m is the first decoding unit of a subsequent buffering period), $\text{initArrivalEarliestTime}[m]$ is derived as follows:

$$\begin{aligned} \text{initArrivalEarliestTime}[m] = &\text{tmpNominalRemovalTime} - \\ &(\text{InitCpbRemovalDelay}[\text{SchedSelIdx}] \div 90000) \end{aligned} \quad (\text{C-8})$$

The final arrival time for decoding unit m is derived as follows:

$$\begin{aligned} &\text{if(!subPicParamsFlag)} \\ &\quad \text{AuFinalArrivalTime}[m] = \text{initArrivalTime}[m] + \text{sizeInbits}[m] \div \text{BitRate}[\text{SchedSelIdx}] \\ &\text{else} \\ &\quad \text{DuFinalArrivalTime}[m] = \text{initArrivalTime}[m] + \text{sizeInbits}[m] \div \text{BitRate}[\text{SchedSelIdx}] \end{aligned} \quad (\text{C-9})$$

where $\text{sizeInbits}[m]$ is the size in bits of decoding unit m , counting the bits of the VCL NAL units and the filler data NAL units for the Type I conformance point or all bits of the Type II bitstream for the Type II conformance point, where the Type I and Type II conformance points are as shown in Figure C-1.

The values of SchedSelIdx , $\text{BitRate}[\text{SchedSelIdx}]$, and $\text{CpbSize}[\text{SchedSelIdx}]$ are constrained as follows:

- If the content of the selected `hrd_parameters()` syntax structures for the access unit containing decoding unit `m` and the previous access unit differ, the HSS selects a value `SchedSelIdx1` of `SchedSelIdx` from among the values of `SchedSelIdx` provided in the selected `hrd_parameters()` syntax structures for the access unit containing decoding unit `m` that results in a `BitRate[ SchedSelIdx1 ]` or `CpbSize[ SchedSelIdx1 ]` for the access unit containing decoding unit `m`. The value of `BitRate[ SchedSelIdx1 ]` or `CpbSize[ SchedSelIdx1 ]` may differ from the value of `BitRate[ SchedSelIdx0 ]` or `CpbSize[ SchedSelIdx0 ]` for the value `SchedSelIdx0` of `SchedSelIdx` that was in use for the previous access unit.
- Otherwise, the HSS continues to operate with the previous values of `SchedSelIdx`, `BitRate[ SchedSelIdx ]` and `CpbSize[ SchedSelIdx ]`.

When the HSS selects values of `BitRate[ SchedSelIdx ]` or `CpbSize[ SchedSelIdx ]` that differ from those of the previous access unit, the following applies:

- The variable `BitRate[ SchedSelIdx ]` comes into effect at 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.

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

The variables `InitCpbRemovalDelay[ SchedSelIdx ]`, `InitCpbRemovalDelayOffset[ SchedSelIdx ]`, `CpbDelayOffset`, and `DpbDelayOffset` are derived as follows:

- If one or more of the following conditions are true, `CpbDelayOffset` is set equal to the value of the buffering period SEI message syntax element `cpb_delay_offset`, `DpbDelayOffset` is set equal to the value of the buffering period SEI message syntax element `dpb_delay_offset`, and `InitCpbRemovalDelay[ SchedSelIdx ]` and `InitCpbRemovalDelayOffset[ SchedSelIdx ]` are set equal to the values of the buffering period SEI message syntax elements `nal_initial_alt_cpb_removal_delay[ SchedSelIdx ]` and `nal_initial_alt_cpb_removal_offset[ SchedSelIdx ]`, respectively, when `NalHrdModeFlag` is equal to 1, or `vcl_initial_alt_cpb_removal_delay[ SchedSelIdx ]` and `vcl_initial_alt_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 subclause C.1:
 - Access unit 0 is a BLA access unit for which the coded picture has `nal_unit_type` equal to `BLA_W_RADL` or `BLA_N_LP`, and the value of `irap_cpb_params_present_flag` of the buffering period SEI message is equal to 1.
 - Access unit 0 is a BLA access unit for which the coded picture has `nal_unit_type` equal to `BLA_W_LP` or is a CRA access unit, and the value of `irap_cpb_params_present_flag` of the buffering period SEI message is equal to 1, and one or more of the following conditions are true:
 - `UseAltCpbParamsFlag` for access unit 0 is equal to 1.
 - `DefaultInitCpbParamsFlag` is equal to 0.
- Otherwise, `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 subclause C.1, `CpbDelayOffset` and `DpbDelayOffset` are both set equal to 0.

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:

$$AuNominalRemovalTime[0] = InitCpbRemovalDelay[SchedSelIdx] \div 90000 \quad (C-10)$$

- 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 = AuNominalRemovalTime[ firstPicInPrevBuffPeriod ]
```

```

    tmpCpbRemovalDelay = AuCpbRemovalDelayVal
  } else {
    baseTime = AuNominalRemovalTime[ prevNonDiscardablePic ]
    tmpCpbRemovalDelay =
      Max( ( auCpbRemovalDelayDeltaMinus1 + 1 ),
        Ceil( ( InitCpbRemovalDelay[ SchedSelIdx ] ÷ 90000 +
          AuFinalArrivalTime[ n - 1 ] - AuNominalRemovalTime[ n - 1 ] ) ÷ ClockTick )
      )
  }
  AuNominalRemovalTime( n ) = baseTime + ClockTick * ( tmpCpbRemovalDelay - CpbDelayOffset )

```

(C-11)

where $AuNominalRemovalTime[firstPicInPrevBuffPeriod]$ is the nominal removal time of the first access unit of the previous buffering period, $AuNominalRemovalTime[prevNonDiscardablePic]$ is the nominal removal time of the preceding picture in decoding order with TemporalId equal to 0 that is not a RASL, RADL or sub-layer non-reference picture, $AuCpbRemovalDelayVal$ is the value of $AuCpbRemovalDelayVal$ derived according to $au_cpb_removal_delay_minus1$ in the picture timing SEI message, selected as specified in subclause C.1, associated with access unit n , and concatenationFlag and $auCpbRemovalDelayDeltaMinus1$ are the values of the syntax elements concatenation_flag and $au_cpb_removal_delay_delta_minus1$, respectively, in the buffering period SEI message, selected as specified in subclause C.1, associated with access unit n .

After the derivation of the nominal CPB removal time and before the derivation of the DPB output time of access unit n , the values of $CpbDelayOffset$ and $DpbDelayOffset$ are updated as follows:

- If one or more of the following conditions are true, $CpbDelayOffset$ is set equal to the value of the buffering period SEI message syntax element cpb_delay_offset , and $DpbDelayOffset$ is set equal to the value of the buffering period SEI message syntax element dpb_delay_offset , where the buffering period SEI message containing the syntax elements is selected as specified in subclause C.1:
 - Access unit n is a BLA access unit for which the coded picture has nal_unit_type equal to BLA_W_RADL or BLA_N_LP , and the value of $irap_cpb_params_present_flag$ of the buffering period SEI message is equal to 1.
 - Access unit n is a BLA access unit for which the coded picture has nal_unit_type equal to BLA_W_LP or is a CRA access unit, and the value of $irap_cpb_params_present_flag$ of the buffering period SEI message is equal to 1, and $UseAltCpbParamsFlag$ for access unit n is equal to 1.
- Otherwise, $CpbDelayOffset$ and $DpbDelayOffset$ are both set equal to 0.
- 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:

$$AuNominalRemovalTime[n] = AuNominalRemovalTime[firstPicInCurrBuffPeriod] + ClockTick * (AuCpbRemovalDelayVal - CpbDelayOffset) \quad (C-12)$$

where $AuNominalRemovalTime[firstPicInCurrBuffPeriod]$ is the nominal removal time of the first access unit of the current buffering period, and $AuCpbRemovalDelayVal$ is the value of $AuCpbRemovalDelayVal$ derived according to $au_cpb_removal_delay_minus1$ in the picture timing SEI message, selected as specified in subclause C.1, associated with access unit n .

When $SubPicHrdFlag$ is equal to 1, the following applies:

- The variable $duCpbRemovalDelayInc$ is derived as follows:
 - If $sub_pic_cpb_params_in_pic_timing_sei_flag$ is equal to 0, $duCpbRemovalDelayInc$ is set equal to the value of $du_spt_cpb_removal_delay_increment$ in the decoding unit information SEI message, selected as specified in subclause C.1, associated with decoding unit m .
 - Otherwise, if $du_common_cpb_removal_delay_flag$ is equal to 0, $duCpbRemovalDelayInc$ is set equal to the value of $du_cpb_removal_delay_increment_minus1[i] + 1$ for decoding unit m in the picture timing SEI message, selected as specified in subclause C.1, associated with access unit n , where the value of i is 0 for the first $num_nalus_in_du_minus1[0] + 1$ consecutive NAL units in the access unit that contains decoding unit m , 1 for the subsequent $num_nalus_in_du_minus1[1] + 1$ NAL units in the same access unit, 2 for the subsequent $num_nalus_in_du_minus1[2] + 1$ NAL units in the same access unit, etc.
 - Otherwise, $duCpbRemovalDelayInc$ is set equal to the value of $du_common_cpb_removal_delay_increment_minus1 + 1$ in the picture timing SEI message, selected as specified in subclause C.1, associated with access unit n .
- The nominal removal time of decoding unit m from the CPB is specified as follows, where $AuNominalRemovalTime[n]$ is the nominal removal time of access unit n :

- If decoding unit m is the last decoding unit in access unit n , the nominal removal time of decoding unit m $DuNominalRemovalTime[m]$ is set equal to $AuNominalRemovalTime[n]$.
- Otherwise (decoding unit m is not the last decoding unit in access unit n), the nominal removal time of decoding unit m $DuNominalRemovalTime[m]$ is derived as follows:

$$\begin{aligned}
 & \text{if(sub_pic_cpb_params_in_pic_timing_sei_flag)} \\
 & \quad DuNominalRemovalTime[m] = DuNominalRemovalTime[m + 1] - \\
 & \quad \quad ClockSubTick * duCpbRemovalDelayInc \\
 & \text{else} \\
 & \quad DuNominalRemovalTime[m] = AuNominalRemovalTime(n) - \\
 & \quad \quad ClockSubTick * duCpbRemovalDelayInc
 \end{aligned} \tag{C-13}$$

If $SubPicHrdFlag$ is equal to 0, the removal time of access unit n from the CPB is specified as follows, where $AuFinalArrivalTime[n]$ and $AuNominalRemovalTime[n]$ are the final CPB arrival time and nominal CPB removal time, respectively, of access unit n :

$$\begin{aligned}
 & \text{if(!low_delay_hrd_flag[HighestTid] || } AuNominalRemovalTime[n] \geq AuFinalArrivalTime[n] \text{)} \\
 & \quad AuCpbRemovalTime[n] = AuNominalRemovalTime[n] \\
 & \text{else} \\
 & \quad AuCpbRemovalTime[n] = AuNominalRemovalTime[n] + ClockTick * \\
 & \quad \quad Ceil((AuFinalArrivalTime[n] - AuNominalRemovalTime[n]) \div ClockTick)
 \end{aligned} \tag{C-14}$$

NOTE 1 – When $low_delay_hrd_flag[HighestTid]$ is equal to 1 and $AuNominalRemovalTime[n]$ is less than $AuFinalArrivalTime[n]$, the size of access unit n is so large that it prevents removal at the nominal removal time.

Otherwise ($SubPicHrdFlag$ is equal to 1), the removal time of decoding unit m from the CPB is specified as follows:

$$\begin{aligned}
 & \text{if(!low_delay_hrd_flag[HighestTid] || } DuNominalRemovalTime[m] \geq DuFinalArrivalTime[m] \text{)} \\
 & \quad DuCpbRemovalTime[m] = DuNominalRemovalTime[m] \\
 & \text{else} \\
 & \quad DuCpbRemovalTime[m] = DuFinalArrivalTime[m]
 \end{aligned} \tag{C-15}$$

NOTE 2 – When $low_delay_hrd_flag[HighestTid]$ is equal to 1 and $DuNominalRemovalTime[m]$ is less than $DuFinalArrivalTime[m]$, the size of decoding unit m is so large that it prevents removal at the nominal removal time.

If $SubPicHrdFlag$ is equal to 0, at the CPB removal time of access unit n , the access unit is instantaneously decoded.

Otherwise ($SubPicHrdFlag$ is equal to 1), at the CPB removal time of decoding unit m , the decoding unit is instantaneously decoded, and when decoding unit m is the last decoding unit of access unit n , the following applies:

- Picture n is considered as decoded.
- The final CPB arrival time of access unit n , i.e. $AuFinalArrivalTime[n]$, is set equal to the final CPB arrival time of the last decoding unit in access unit n , i.e. $DuFinalArrivalTime[m]$.
- The nominal CPB removal time of access unit n , i.e. $AuNominalRemovalTime[n]$, is set equal to the nominal CPB removal time of the last decoding unit in access unit n , i.e. $DuNominalRemovalTime[m]$.
- The CPB removal time of access unit n , i.e. $AuCpbRemovalTime[n]$, is set equal to the CPB removal time of the last decoding unit in access unit n , i.e. $DuCpbRemovalTime[m]$.

C.3 Operation of the decoded picture buffer (DPB)

C.3.1 General

The specifications in this subclause apply independently to each set of DPB parameters selected as specified in subclause 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 subclauses C.3.2, C.3.3 and C.3.4 are sequentially applied as specified below.

C.3.2 Removal of pictures from the DPB

The removal of pictures from the DPB before decoding of the current picture (but after parsing the slice header of the first slice of the current picture) happens instantaneously at the CPB removal time of the first decoding unit of access unit n (containing the current picture) and proceeds as follows:

- The decoding process for RPS as specified in subclause 8.3.2 is invoked.

- When the current picture is an IRAP picture with NoRaslOutputFlag 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, 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, 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 PicOutputFlag equal to 0 or its DPB output time is less than or equal to the CPB removal time of the first decoding unit (denoted as decoding unit m) of the current picture n; i.e. DpbOutputTime[k] is less than or equal to CpbRemovalTime(m)
- For each picture that is removed from the DPB, the DPB fullness is decremented by one.

C.3.3 Picture output

The processes specified in this subclause happen instantaneously at the CPB removal time of access unit n, AuCpbRemovalTime[n].

When picture n has PicOutputFlag equal to 1, its DPB output time DpbOutputTime[n] is derived as follows, where the variable firstPicInBufferingPeriodFlag is equal to 1 if access unit n is the first access unit of a buffering period and 0 otherwise:

```

if( !SubPicHrdFlag ) {
    DpbOutputTime[ n ] = AuCpbRemovalTime[ n ] + ClockTick * picDpbOutputDelay
    if( firstPicInBufferingPeriodFlag )
        DpbOutputTime[ n ] -= ClockTick * DpbDelayOffset
} else
    DpbOutputTime[ n ] = AuCpbRemovalTime[ n ] + ClockSubTick * picSptDpbOutputDuDelay
  
```

(C-16)

where picDpbOutputDelay is the value of pic_dpb_output_delay in the picture timing SEI message associated with access unit n, and picSptDpbOutputDuDelay is the value of pic_spt_dpb_output_du_delay, when present, in the decoding unit information SEI messages associated with access unit n, or the value of pic_dpb_output_du_delay in the picture timing SEI message associated with access unit n when there is no decoding unit information SEI message associated with access unit n or no decoding unit information SEI message associated with access unit n has pic_spt_dpb_output_du_delay present.

NOTE – When the syntax element pic_spt_dpb_output_du_delay is not present in any decoding unit information SEI message associated with access unit n, the value is inferred to be equal to pic_dpb_output_du_delay in the picture timing SEI message associated with access unit n.

The output of the current picture is specified as follows:

- If PicOutputFlag is equal to 1 and DpbOutputTime[n] is equal to AuCpbRemovalTime[n], the current picture is output.
- Otherwise, if PicOutputFlag is equal to 0, the current picture is not output, but will be stored in the DPB as specified in subclause C.3.4.
- Otherwise (PicOutputFlag is equal to 1 and DpbOutputTime[n] is greater than AuCpbRemovalTime[n]), the current picture is output later and will be stored in the DPB (as specified in subclause 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:

$$\text{DpbOutputInterval}[n] = \text{DpbOutputTime}[\text{nextPicInOutputOrder}] - \text{DpbOutputTime}[n] \quad (\text{C-17})$$

where nextPicInOutputOrder is the picture that follows picture n in output order and has PicOutputFlag equal to 1.

C.3.4 Current decoded picture marking and storage

The process specified in this subclause happens instantaneously at the CPB removal time of access unit n, CpbRemovalTime[n].

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".

C.4 Bitstream conformance

A bitstream of coded data conforming to this Specification shall fulfil all requirements specified in this subclause.

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, a CRA picture or a BLA picture.

The bitstream is tested by the HRD for conformance as specified in subclause 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 TemporalId equal to 0 and that is not a RASL picture, a RADL picture, or a sub-layer non-reference picture.
- The short-term reference pictures in the RPS of the current picture.
- All pictures n that have PicOutputFlag equal to 1, AuCpbRemovalTime[n] less than AuCpbRemovalTime[currPic], and DpbOutputTime[n] greater than or equal to AuCpbRemovalTime[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:

$$\text{deltaTime90k}[n] = 90000 * (\text{AuNominalRemovalTime}[n] - \text{AuFinalArrivalTime}[n-1]) \quad (\text{C-18})$$

The value of InitCpbRemovalDelay[SchedSelIdx] is constrained as follows:

- If cbr_flag[SchedSelIdx] is equal to 0, the following condition shall be true:

$$\text{InitCpbRemovalDelay}[\text{SchedSelIdx}] \leq \text{Ceil}(\text{deltaTime90k}[n]) \quad (\text{C-19})$$

- Otherwise (cbr_flag[SchedSelIdx] is equal to 1), the following condition shall be true:

$$\text{Floor}(\text{deltaTime90k}[n]) \leq \text{InitCpbRemovalDelay}[\text{SchedSelIdx}] \leq \text{Ceil}(\text{deltaTime90k}[n]) \quad (\text{C-20})$$

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.

2. 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.
3. A CPB underflow is specified as the condition in which the nominal CPB removal time of decoding unit m DuNominalRemovalTime(m) is less than the final CPB arrival time of decoding unit m DuFinalArrivalTime(m) for at least one value of m. When low_delay_hrd_flag[HighestTid] is equal to 0, the CPB shall never underflow.

4. When `SubPicHrdFlag` is equal to 1, `low_delay_hrd_flag[ HighestTid ]` is equal to 1, and the nominal removal time of a decoding unit `m` of access unit `n` is less than the final CPB arrival time of decoding unit `m` (i.e. `DuNominalRemovalTime[ m ] < DuFinalArrivalTime[ m ]`), the nominal removal time of access unit `n` shall be less than the final CPB arrival time of access unit `n` (i.e. `AuNominalRemovalTime[ n ] < AuFinalArrivalTime[ n ]`).
5. The nominal removal times of pictures from the CPB (starting from the second picture in decoding order) shall satisfy the constraints on `AuNominalRemovalTime[ n ]` and `AuCpbRemovalTime[ n ]` expressed in subclauses A.4.1 through A.4.2.
6. For each current picture, after invocation of the process for removal of pictures from the DPB as specified in subclause 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 `PicOutputFlag` equal to 1 and `AuCpbRemovalTime[ n ]` less than `AuCpbRemovalTime[ currPic ]`, where `currPic` is the current picture, shall be less than or equal to `sps_max_dec_pic_buffering_minus1[ HighestTid ]`.
7. All reference pictures shall be present in the DPB when needed for prediction. Each picture that has `PicOutputFlag` 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 subclause C.3.
8. For each current picture, the value of `maxPicOrderCnt - minPicOrderCnt` shall be less than `MaxPicOrderCntLsb / 2`.
9. The value of `DpbOutputInterval[ n ]` as given by Equation C-17, which is the difference between the output time of a picture and that of the first picture following it in output order and having `PicOutputFlag` equal to 1, shall satisfy the constraint expressed in subclause A.4.1 for the profile, tier and level specified in the bitstream using the decoding process specified in clauses 2 through 10.
10. For each current picture, when `sub_pic_cpb_params_in_pic_timing_sei_flag` is equal to 1, let `tmpCpbRemovalDelaySum` be derived as follows:

$$\begin{aligned}
 &\text{tmpCpbRemovalDelaySum} = 0 \\
 &\text{for } (i = 0; i < \text{num_decoding_units_minus1}; i++) \\
 &\quad \text{tmpCpbRemovalDelaySum} += \text{du_cpb_removal_delay_increment_minus1}[i] + 1
 \end{aligned}
 \tag{C-21}$$

The value of `ClockSubTick * tmpCpbRemovalDelaySum` shall be equal to the difference between the nominal CPB removal time of the current access unit and the nominal CPB removal time of the first decoding unit in the current access unit in decoding order.

C.5 Decoder conformance

C.5.1 General

A decoder conforming to this Specification shall fulfil all requirements specified in this subclause.

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 subclause C.4, in the manner specified in Annex A, provided that all VPSSs, SPSSs and PPSSs 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 subclause 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 `PicOutputFlag` 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 `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 $(f(r) \div r)$ as follows:

$$\alpha = (r - \text{BitRate}[\text{SchedSelIdx} - 1]) \div (\text{BitRate}[\text{SchedSelIdx}] - \text{BitRate}[\text{SchedSelIdx} - 1]), \quad (\text{C-22})$$

$$c(r) = \alpha * \text{CpbSize}[\text{SchedSelIdx}] + (1 - \alpha) * \text{CpbSize}[\text{SchedSelIdx} - 1], \quad (\text{C-23})$$

$$f(r) = \alpha * \text{InitCpbRemovalDelay}[\text{SchedSelIdx}] * \text{BitRate}[\text{SchedSelIdx}] + (1 - \alpha) * \text{InitCpbRemovalDelay}[\text{SchedSelIdx} - 1] * \text{BitRate}[\text{SchedSelIdx} - 1] \quad (\text{C-24})$$

for any $\text{SchedSelIdx} > 0$ and r such that $\text{BitRate}[\text{SchedSelIdx} - 1] \leq r \leq \text{BitRate}[\text{SchedSelIdx}]$ 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 – $\text{InitCpbRemovalDelay}[\text{SchedSelIdx}]$ can be different from one buffering period to another and have to be recalculated.

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 $\text{CpbSize}[\text{SchedSelIdx}]$ as specified in subclause E.2.3, where SchedSelIdx and the HRD parameters are selected as specified in subclause C.1. The DPB size is given by $\text{sps_max_dec_pic_buffering_minus1}[\text{HighestTid}] + 1$. 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 subclauses C.5.2 through C.5.2.3.

C.5.2 Operation of the output order DPB

C.5.2.1 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 as specified in subclause C.5.2.2 is invoked, followed by the invocation of the process for picture decoding, marking, additional bumping, and storage as specified in subclause C.5.2.3. The "bumping" process is specified in subclause C.5.2.4 and is invoked as specified in subclauses C.5.2.2 and C.5.2.3.

C.5.2.2 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 RPS as specified in subclause 8.3.2 is invoked.
- If the current picture is an IRAP picture with `NoRaslOutputFlag` 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`, 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`, 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 subclause C.5.2.4, and the DPB fullness is set equal to 0.
- Otherwise (the current picture is not an IRAP picture with `NoRaslOutputFlag` 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 subclause 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`.

C.5.2.3 Picture decoding, marking, additional bumping, and storage

The processes specified in this subclause happen instantaneously when the last decoding unit of access unit *n* containing the current picture is removed from the CPB.

For each picture in the DPB that is marked as "needed for output", the associated variable `PicLatencyCount` is set equal to `PicLatencyCount + 1`.

The current picture is considered as decoded after the last decoding unit of the picture is decoded. The current decoded picture is stored in an empty picture storage buffer in the DPB, and the following applies:

- If the current decoded picture has `PicOutputFlag` equal to 1, it is marked as "needed for output" and its associated variable `PicLatencyCount` is set equal to 0.
- Otherwise (the current decoded picture has `PicOutputFlag` equal to 0), it is marked as "not needed for output".

The current decoded picture is marked as "used for short-term reference".

When one or more of the following conditions are true, the "bumping" process specified in subclause 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 ]`.

C.5.2.4 "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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

Annex D

Supplemental enhancement information

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

D.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). Some SEI message information is required to check bitstream conformance and for output timing decoder conformance.

In subclause 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 subclause 7.3.5 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 23008-2:2013

D.2 SEI payload syntax**D.2.1 General SEI message syntax**

sei_payload(payloadType, payloadSize) {	Descriptor
if(nal_unit_type == PREFIX_SEI_NUT)	
if(payloadType == 0)	
buffering_period(payloadSize)	
else if(payloadType == 1)	
pic_timing(payloadSize)	
else if(payloadType == 2)	
pan_scan_rect(payloadSize)	
else if(payloadType == 3)	
filler_payload(payloadSize)	
else if(payloadType == 4)	
user_data_registered_itu_t_t35(payloadSize)	
else if(payloadType == 5)	
user_data_unregistered(payloadSize)	
else if(payloadType == 6)	
recovery_point(payloadSize)	
else if(payloadType == 9)	
scene_info(payloadSize)	
else if(payloadType == 15)	
picture_snapshot(payloadSize)	
else if(payloadType == 16)	
progressive_refinement_segment_start(payloadSize)	
else if(payloadType == 17)	
progressive_refinement_segment_end(payloadSize)	
else if(payloadType == 19)	
film_grain_characteristics(payloadSize)	
else if(payloadType == 22)	
post_filter_hint(payloadSize)	
else if(payloadType == 23)	
tone_mapping_info(payloadSize)	
else if(payloadType == 45)	
frame_packing_arrangement(payloadSize)	
else if(payloadType == 47)	
display_orientation(payloadSize)	
else if(payloadType == 128)	
structure_of_pictures_info(payloadSize)	
else if(payloadType == 129)	
active_parameter_sets(payloadSize)	
else if(payloadType == 130)	
decoding_unit_info(payloadSize)	
else if(payloadType == 131)	
temporal_sub_layer_zero_index(payloadSize)	
else if(payloadType == 133)	
scalable_nesting(payloadSize)	
else if(payloadType == 134)	

region_refresh_info(payloadSize)	
else	
reserved_sei_message(payloadSize)	
else /* nal_unit_type == SUFFIX_SEI_NUT */	
if(payloadType == 3)	
filler_payload(payloadSize)	
else if(payloadType == 4)	
user_data_registered_itu_t_t35(payloadSize)	
else if(payloadType == 5)	
user_data_unregistered(payloadSize)	
else if(payloadType == 17)	
progressive_refinement_segment_end(payloadSize)	
else if(payloadType == 22)	
post_filter_hint(payloadSize)	
else 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)
}	
}	

D.2.2 Buffering period SEI message syntax

buffering_period(payloadSize) {	Descriptor
bp_seq_parameter_set_id	ue(v)
if(!sub_pic_hrd_params_present_flag)	
irap_cpb_params_present_flag	u(1)
if(irap_cpb_params_present_flag) {	
cpb_delay_offset	u(v)
dpb_delay_offset	u(v)
}	
concatenation_flag	u(1)
au_cpb_removal_delay_delta_minus1	u(v)
if(NalHrdBpPresentFlag) {	
for(i = 0; i <= CpbCnt; i++) {	
nal_initial_cpb_removal_delay[i]	u(v)
nal_initial_cpb_removal_offset[i]	u(v)
if(sub_pic_hrd_params_present_flag irap_cpb_params_present_flag) {	
nal_initial_alt_cpb_removal_delay[i]	u(v)
nal_initial_alt_cpb_removal_offset[i]	u(v)
}	
}	
}	
if(VclHrdBpPresentFlag) {	
for(i = 0; i <= CpbCnt; i++) {	
vcl_initial_cpb_removal_delay[i]	u(v)
vcl_initial_cpb_removal_offset[i]	u(v)
if(sub_pic_hrd_params_present_flag irap_cpb_params_present_flag) {	
vcl_initial_alt_cpb_removal_delay[i]	u(v)
vcl_initial_alt_cpb_removal_offset[i]	u(v)
}	
}	
}	
}	

D.2.3 Picture timing SEI message syntax

pic_timing(payloadSize) {	Descriptor
if(frame_field_info_present_flag) {	
pic_struct	u(4)
source_scan_type	u(2)
duplicate_flag	u(1)
}	
if(CpbDpbDelaysPresentFlag) {	
au_cpb_removal_delay_minus1	u(v)
pic_dpb_output_delay	u(v)
if(sub_pic_hrd_params_present_flag)	
pic_dpb_output_du_delay	u(v)
if(sub_pic_hrd_params_present_flag && sub_pic_cpb_params_in_pic_timing_sei_flag) {	
num_decoding_units_minus1	ue(v)
du_common_cpb_removal_delay_flag	u(1)
if(du_common_cpb_removal_delay_flag)	
du_common_cpb_removal_delay_increment_minus1	u(v)
for(i = 0; i <= num_decoding_units_minus1; i++) {	
num_nalus_in_du_minus1[i]	ue(v)
if(!du_common_cpb_removal_delay_flag && i < num_decoding_units_minus1)	
du_cpb_removal_delay_increment_minus1[i]	u(v)
}	
}	
}	
}	

D.2.4 Pan-scan rectangle SEI message syntax

pan_scan_rect(payloadSize) {	Descriptor
pan_scan_rect_id	ue(v)
pan_scan_rect_cancel_flag	u(1)
if(!pan_scan_rect_cancel_flag) {	
pan_scan_cnt_minus1	ue(v)
for(i = 0; i <= pan_scan_cnt_minus1; i++) {	
pan_scan_rect_left_offset[i]	se(v)
pan_scan_rect_right_offset[i]	se(v)
pan_scan_rect_top_offset[i]	se(v)
pan_scan_rect_bottom_offset[i]	se(v)
}	
pan_scan_rect_persistence_flag	u(1)
}	
}	

D.2.5 Filler payload SEI message syntax

filler_payload(payloadSize) {	Descriptor
for(k = 0; k < payloadSize; k++)	
ff_byte /* equal to 0xFF */	f(8)
}	

D.2.6 User data registered by Rec. ITU-T T.35 SEI message syntax

user_data_registered_itu_t_t35(payloadSize) {	Descriptor
itu_t_t35_country_code	b(8)
if(itu_t_t35_country_code != 0xFF)	
i = 1	
else {	
itu_t_t35_country_code_extension_byte	b(8)
i = 2	
}	
do {	
itu_t_t35_payload_byte	b(8)
i++	
} while(i < payloadSize)	
}	

D.2.7 User data unregistered SEI message syntax

user_data_unregistered(payloadSize) {	Descriptor
uuid_iso_iec_11578	u(128)
for(i = 16; i < payloadSize; i++)	
user_data_payload_byte	b(8)
}	

D.2.8 Recovery point SEI message syntax

recovery_point(payloadSize) {	Descriptor
recovery_poc_cnt	se(v)
exact_match_flag	u(1)
broken_link_flag	u(1)
}	

D.2.9 Scene information SEI message syntax

scene_info(payloadSize) {	Descriptor
scene_info_present_flag	u(1)
if(scene_info_present_flag) {	
prev_scene_id_valid_flag	u(1)
scene_id	ue(v)
scene_transition_type	ue(v)
if(scene_transition_type > 3)	
second_scene_id	ue(v)
}	
}	

D.2.10 Picture snapshot SEI message syntax

picture_snapshot(payloadSize) {	Descriptor
snapshot_id	ue(v)
}	

D.2.11 Progressive refinement segment start SEI message syntax

progressive_refinement_segment_start(payloadSize) {	Descriptor
progressive_refinement_id	ue(v)
pic_order_cnt_delta	ue(v)
}	

D.2.12 Progressive refinement segment end SEI message syntax

progressive_refinement_segment_end(payloadSize) {	Descriptor
progressive_refinement_id	ue(v)
}	

D.2.13 Film grain characteristics SEI message syntax

film_grain_characteristics(payloadSize) {	Descriptor
film_grain_characteristics_cancel_flag	u(1)
if(!film_grain_characteristics_cancel_flag) {	
film_grain_model_id	u(2)
separate_colour_description_present_flag	u(1)
if(separate_colour_description_present_flag) {	
film_grain_bit_depth_luma_minus8	u(3)
film_grain_bit_depth_chroma_minus8	u(3)
film_grain_full_range_flag	u(1)
film_grain_colour_primaries	u(8)
film_grain_transfer_characteristics	u(8)
film_grain_matrix_coeffs	u(8)
}	
blending_mode_id	u(2)
log2_scale_factor	u(4)
for(c = 0; c < 3; c++)	
comp_model_present_flag[c]	u(1)
for(c = 0; c < 3; c++)	
if(comp_model_present_flag[c]) {	
num_intensity_intervals_minus1[c]	u(8)
num_model_values_minus1[c]	u(3)
for(i = 0; i <= num_intensity_intervals_minus1[c]; i++) {	
intensity_interval_lower_bound[c][i]	u(8)
intensity_interval_upper_bound[c][i]	u(8)
for(j = 0; j <= num_model_values_minus1[c]; j++)	
comp_model_value[c][i][j]	se(v)
}	
}	
}	
film_grain_characteristics_persistence_flag	u(1)
}	
}	

D.2.14 Post-filter hint SEI message syntax

post_filter_hint(payloadSize) {	Descriptor
filter_hint_size_y	ue(v)
filter_hint_size_x	ue(v)
filter_hint_type	u(2)
for(cIdx = 0; cIdx < (chroma_format_idc == 0 ? 1 : 3); cIdx++)	
for(cy = 0; cy < filter_hint_size_y; cy ++)	
for(cx = 0; cx < filter_hint_size_x; cx ++)	
filter_hint_value[cIdx][cy][cx]	se(v)
}	

D.2.15 Tone mapping information SEI message syntax

tone_mapping_info(payloadSize) {	Descriptor
tone_map_id	ue(v)
tone_map_cancel_flag	u(1)
if(!tone_map_cancel_flag) {	
tone_map_persistence_flag	u(1)
coded_data_bit_depth	u(8)
target_bit_depth	u(8)
tone_map_model_id	ue(v)
if(tone_map_model_id == 0) {	
min_value	u(32)
max_value	u(32)
} else if(tone_map_model_id == 1) {	
sigmoid_midpoint	u(32)
sigmoid_width	u(32)
} else if(tone_map_model_id == 2)	
for(i = 0; i < (1 << target_bit_depth); i++)	
start_of_coded_interval[i]	u(v)
else if(tone_map_model_id == 3) {	
num_pivots	u(16)
for(i = 0; i < num_pivots; i++) {	
coded_pivot_value[i]	u(v)
target_pivot_value[i]	u(v)
}	
} else if(tone_map_model_id == 4) {	
camera_iso_speed_idc	u(8)
if(camera_iso_speed_idc == EXTENDED_ISO)	
camera_iso_speed_value	u(32)
exposure_index_idc	u(8)
if(exposure_index_idc == EXTENDED_ISO)	
exposure_index_value	u(32)
exposure_compensation_value_sign_flag	u(1)
exposure_compensation_value_numerator	u(16)
exposure_compensation_value_denom_idc	u(16)
ref_screen_luminance_white	u(32)
extended_range_white_level	u(32)
nominal_black_level_code_value	u(16)
nominal_white_level_code_value	u(16)
extended_white_level_code_value	u(16)
}	
}	
}	

D.2.16 Frame packing arrangement SEI message syntax

frame_packing_arrangement(payloadSize) {	Descriptor
frame_packing_arrangement_id	ue(v)
frame_packing_arrangement_cancel_flag	u(1)
if(!frame_packing_arrangement_cancel_flag) {	
frame_packing_arrangement_type	u(7)
quincunx_sampling_flag	u(1)
content_interpretation_type	u(6)
spatial_flipping_flag	u(1)
frame0_flipped_flag	u(1)
field_views_flag	u(1)
current_frame_is_frame0_flag	u(1)
frame0_self_contained_flag	u(1)
frame1_self_contained_flag	u(1)
if(!quincunx_sampling_flag && frame_packing_arrangement_type != 5) {	
frame0_grid_position_x	u(4)
frame0_grid_position_y	u(4)
frame1_grid_position_x	u(4)
frame1_grid_position_y	u(4)
}	
frame_packing_arrangement_reserved_byte	u(8)
frame_packing_arrangement_persistence_flag	u(1)
}	
upsampled_aspect_ratio_flag	u(1)
}	

D.2.17 Display orientation SEI message syntax

display_orientation(payloadSize) {	Descriptor
display_orientation_cancel_flag	u(1)
if(!display_orientation_cancel_flag) {	
hor_flip	u(1)
ver_flip	u(1)
anticlockwise_rotation	u(16)
display_orientation_persistence_flag	u(1)
}	
}	

D.2.18 Structure of pictures information SEI message syntax

structure_of_pictures_info(payloadSize) {	Descriptor
sop_seq_parameter_set_id	ue(v)
num_entries_in_sop_minus1	ue(v)
for(i = 0; i <= num_entries_in_sop_minus1; i++) {	
sop_vcl_nut[i]	u(6)
sop_temporal_id[i]	u(3)
if(sop_vcl_nut[i] != IDR_W_RADL && sop_vcl_nut[i] != IDR_N_LP)	
sop_short_term_rps_idx[i]	ue(v)
if(i > 0)	
sop_poc_delta[i]	se(v)
}	
}	

D.2.19 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)
}	

D.2.20 Active parameter sets SEI message syntax

active_parameter_sets(payloadSize) {	Descriptor
active_video_parameter_set_id	u(4)
self_contained_cvs_flag	u(1)
no_parameter_set_update_flag	u(1)
num_sps_ids_minus1	ue(v)
for(i = 0; i <= num_sps_ids_minus1; i++)	
active_seq_parameter_set_id[i]	ue(v)
}	

D.2.21 Decoding unit information SEI message syntax

decoding_unit_info(payloadSize) {	Descriptor
decoding_unit_idx	ue(v)
if(!sub_pic_cpb_params_in_pic_timing_sei_flag)	
du_spt_cpb_removal_delay_increment	u(v)
dpb_output_du_delay_present_flag	u(1)
if(dpb_output_du_delay_present_flag)	
pic_spt_dpb_output_du_delay	u(v)
}	

D.2.22 Temporal sub-layer zero index SEI message syntax

temporal_sub_layer_zero_index(payloadSize) {	Descriptor
temporal_sub_layer_zero_idx	u(8)
irap_pic_id	u(8)
}	

D.2.23 Scalable nesting SEI message syntax

scalable_nesting(payloadSize) {	Descriptor
bitstream_subset_flag	u(1)
nesting_op_flag	u(1)
if(nesting_op_flag) {	
default_op_flag	u(1)
nesting_num_ops_minus1	ue(v)
for(i = default_op_flag; i <= nesting_num_ops_minus1; i++) {	
nesting_max_temporal_id_plus1[i]	u(3)
nesting_op_idx[i]	ue(v)
}	
} else {	
all_layers_flag	u(1)
if(!all_layers_flag) {	
nesting_no_op_max_temporal_id_plus1	u(3)
nesting_num_layers_minus1	ue(v)
for(i = 0; i <= nesting_num_layers_minus1; i++)	
nesting_layer_id[i]	u(6)
}	
}	
while(!byte_aligned())	
nesting_zero_bit /* equal to 0 */	u(1)
do	
sei_message()	
while(more_rbsp_data())	
}	

D.2.24 Region refresh information SEI message syntax

region_refresh_info(payloadSize) {	Descriptor
refreshed_region_flag	u(1)
}	

D.2.25 Reserved SEI message syntax

reserved_sei_message(payloadSize) {	Descriptor
for(i = 0; i < payloadSize; i++)	
reserved_sei_message_payload_byte	b(8)
}	

D.3 SEI payload semantics**D.3.1 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 $8 * \text{payloadSize} - \text{nEarlierBits} - \text{nPayloadZeroBits} - 1$, 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.