
**Information technology – General
video coding —**

**Part 2:
Low complexity enhancement video
coding**

Technologies de l'information – Codage vidéo général —

Partie 2: Codage vidéo d'amélioration de faible complexité



Reference number
ISO/IEC 23094-2:2021(E)

Documents
Document Sharing Hub
© ISO/IEC 2021



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2021

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword	v
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Abbreviated terms	6
5 Conventions	7
5.1 General	7
5.2 Arithmetic operators	7
5.3 Logical operators	8
5.4 Relational operators	8
5.5 Bit-wise operators	8
5.6 Assignment operators	9
5.7 Range notation	9
5.8 Mathematical functions	9
5.9 Order of operation precedence	10
5.10 Variables, syntax elements and tables	11
5.11 Text description of logical operations	12
5.12 Processes	13
6 Bitstream and picture formats, partitioning, scanning processes and neighbouring relationships	13
6.1 Bitstream formats	13
6.2 Source, decoded and output picture formats	14
6.3 Partitioning of pictures	17
6.3.1 Organization of the hierarchical structure	17
6.3.2 Partitioning of residuals plane	17
7 Syntax and semantics	18
7.1 Method of specifying syntax in tabular form	18
7.2 Specification of syntax functions and descriptors	19
7.3 Syntax in tabular form	20
7.3.1 Syntax order	20
7.3.2 NAL unit and NAL unit header syntax	20
7.3.3 Process block syntax	21
7.3.4 Process payload – sequence configuration	22
7.3.5 Process payload – global configuration	22
7.3.6 Process payload – picture configuration	24
7.3.7 Process payload – encoded data	25
7.3.8 Process payload – encoded tiled data	26
7.3.9 Process payload – surface	28
7.3.10 Process payload – additional info	28
7.3.11 Process payload – filler	29
7.3.12 Byte alignment syntax	29
7.4 Semantics	29
7.4.1 General	29
7.4.2 NAL unit semantics	30
7.4.3 Data block unit configuration semantics	32
8 Decoding process	41
8.1 General decoding process	41
8.2 Payload data block unit process	41
8.3 Picture enhancement decoding process	42
8.3.1 General enhancement decoding process	42
8.3.2 Decoding process for picture enhancement encoded data (payload_ encoded_data)	42

8.3.3	Decoding process for picture enhancement encoded tiled data (payload_	45
	encoded_tiled_data)	
8.3.4	Decoding process for enhancement sub-layer 1 (L-1) encoded data	50
8.3.5	Decoding process for enhancement sub-layer 2 (L-2) encoded data	52
8.4	Decoding process for the temporal prediction	53
8.4.1	General decoding process for temporal prediction	53
8.4.2	Tiled temporal refresh	54
8.5	Decoding process for the dequantization	54
8.5.1	Decoding process for the dequantization overview	54
8.5.2	Scaling process for transform coefficients	54
8.5.3	Derivation of dequantization offset and stepwidth modifier	55
8.5.4	Derivation of quantization matrix	56
8.6	Decoding process for the transform	59
8.6.1	General upscaling process description	59
8.6.2	Transform inputs and outputs, transform types, and residual samples	
	derivation	63
8.6.3	2x2 directional decomposition transform	64
8.6.4	4x4 directional decomposition transform	65
8.7	Decoding process for the upscaling	66
8.7.1	Nearest sample upsampler kernel description	66
8.7.2	Bilinear upsampler kernel description	68
8.7.3	Cubic upsampler kernel description	71
8.7.4	Modified Cubic upsampler kernel description	73
8.7.5	Predicted residual process description	74
8.7.6	Adaptive Cubic upsampler kernel description	75
8.8	Decoding process for the residual reconstruction	75
8.8.1	Reconstructed residual of each block derivation	75
8.8.2	Residual reconstruction for L-1 block	76
8.8.3	Residual reconstruction for L-2 block	76
8.9	Decoding process for the L-1 filter	77
8.9.1	L-1 residual filter overview	77
8.9.2	Decoding process for filtering L-1 block	77
8.10	Decoding process for base decoder data extraction	79
8.11	Decoding process for dither filter	79
9	Parsing process	79
9.1	Parsing process inputs and outputs, process overview	79
9.1.1	Parsing process for entropy encoded transform coefficients	79
9.1.2	Parsing process for entropy encoded temporal signal coefficient group	82
9.2	Prefix Coding decoder	83
9.2.1	Prefix Coding decoder description	83
9.2.2	Prefix Coding decoder table generation	84
9.2.3	Prefix Coding decoder for tile data sizes	88
9.3	RLE decoder	89
9.3.1	RLE process inputs and outputs	89
9.3.2	RLE decoder for coefficient groups	89
9.3.3	RLE decoder description	90
9.3.4	RLE decoder for temporal signal coefficient group	91
9.3.5	RLE decoder for tile entropy_enabled_flag fields	93
9.4	Parsing process for 0-th order Exp-Golomb codes	94
	Annex A (normative) Profiles and levels	96
	Annex B (normative) Byte stream format	98
	Annex C (normative) Hypothetical reference decoder	101
	Annex D (normative) Supplemental enhancement information	108
	Annex E (normative) Video usability information	115
	Bibliography	119

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

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. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO/IEC 23094 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Information technology – General video coding —

Part 2:

Low complexity enhancement video coding

1 Scope

This document specifies low complexity enhancement video coding.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 11578:1996, *Information technology — Open systems interconnection — Remote procedure call (RPC)*

ITU-T H.273 | ISO/IEC 23091-2:2019, *Information technology — Coding-independent code points — Part 2: Video*

ITU-T Recommendation T.35:2000, *Procedure for the allocation of ITU-T defined codes for non-standard facilities*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1

access unit

AU

set of *NAL units* (3.35) that are associated with a particular output time, are consecutive in *decoding order* (3.20), and contain exactly one *coded picture* (3.9)

3.2

bitstream

sequence of bits, in the form of a *NAL unit stream* (3.36) or a *byte stream* (3.6), that forms the representation of *coded pictures* (3.9), and associated data forming one or more coded video sequences (CVSSs)

3.3

block

MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients* (3.57)

3.4

byte

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

3.5
byte-aligned

position in a *bitstream* (3.2) in which the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*

Note 1 to entry: A bit, *byte* (3.4) or *syntax element* (3.53) is said to be *byte-aligned* when the position at which it appears in a *bitstream* (3.2) is *byte-aligned*.

3.6
byte stream

encapsulation of a *NAL unit stream* (3.36) containing *start code prefixes* (3.51) and *NAL units* (3.35)

3.7
chroma

sample array or single sample is representing one of the two colour difference signals related to the primary colours, represented by the symbols Cb and Cr

Note 1 to entry: 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.

3.8
chunk

entropy coded portion of data containing the quantized *transform coefficient* (3.57) belonging to a coefficient group

3.9
coded picture

coded representation (3.10) of a *picture* (3.40) containing all *TUs* (3.58) of the *picture*

3.10
coded representation

data element as represented in its coded form

3.11
coded video sequence

CVS
coded sequence of *access units* (3.1)

3.12
coding block

MxN *block* (3.3) of samples for some values of M and N

3.13
coding unit

CU
32 x 32 *block* (3.3) of samples resulting from the parsing of the entropy encoded *transform coefficients* (3.57) in the *decoding process* (3.22)

3.14
coefficient group

CG
syntactical structure containing coded data related to a specific set of *transform coefficients* (3.57)

3.15
component

array or single sample from one of the three arrays (*luma* (3.34) and two *chroma* (3.7)) that compose a *picture* (3.40) 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

3.16**data block**

syntax structure (3.54) containing *bytes* (3.4) corresponding to a type of data

3.17**decoded base picture**

decoded picture (3.18) derived by decoding a *coded picture* (3.9) with a *base decoder* (3.20)

3.18**decoded picture**

picture (3.40) derived by decoding a *coded picture* (3.9), and which is either a *decoded frame* (3.29) or a *decoded field* (3.28)

3.19**decoded picture buffer****DPB**

buffer holding *decoded pictures* (3.18) for reference or output reordering

3.20**decoder**

embodiment of a *decoding process* (3.22)

3.21**decoding order**

order in which *syntax elements* (3.53) are processed by the *decoding process* (3.22)

3.22**decoding process**

process specified that reads a *bitstream* (3.2) and derives *decoded pictures* (3.18) from it

3.23**emulation prevention byte**

byte (3.4) equal to 0x03 that may be present within a *NAL unit* (3.35), the presence of which ensures that no sequence of consecutive *byte-aligned* (3.5) *bytes* in the *NAL unit* contains a *start code prefix* (3.51)

3.24**encoder**

embodiment of an *encoding process* (3.25)

3.25**encoding process**

process that produces a *bitstream* (3.2) conforming to this document

3.26**enhancement layer**

layer (3.32) within the *bitstream* (3.2) pertaining to the *residual planes* (3.47)

3.27**enhancement sub-layer**

layer (3.32) of the *enhancement layer* (3.26)

3.28**field**

assembly of alternate rows of a *frame* (3.29)

3.29**frame**

array of *luma* (3.34) samples in monochrome format or array of *luma* samples and two corresponding arrays of *chroma* (3.7) samples in 4:2:0, 4:2:2, and 4:4:4 colour format, and which consists of two *fields* (3.28): a top field and a bottom field

3.30

instantaneous decoding refresh picture

IDR picture

picture (3.40) for which a *NAL unit* (3.35) contains a global configuration data block and does not refer to any other *picture* for operation of the *decoding process* (3.22) of this *picture* and for which no subsequent *pictures* in *decoding order* (3.21) refer to any *picture* that precedes it in *decoding order*

Note 1 to entry: An *IDR picture* shall occur at least when an *IDR picture* for the base *decoder* (3.20) occurs. The *IDR picture* for a base *decoder* is not specified in this document.

3.31

inverse transform

part of the *decoding process* (3.22) by which a set of *transform coefficients* (3.57) is converted into *residuals* (3.46)

3.32

layer

one of a set of syntactical structures in a non-branching hierarchical relationship

3.33

level

defined set of constraints on the values that may be taken by the *syntax elements* (3.53) and variables of this document

Note 1 to entry: The same set of *levels* is defined for all *profiles* (3.41), with most aspects of the definition of each *level* being in common across different *profiles*. Individual implementations may, within specified constraints, support a different *level* for each supported *profile*.

3.34

luma

sample array or single sample representing the monochrome signal related to the primary colours, represented by the symbol or subscript Y or L

Note 1 to entry: 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.

3.35

network abstraction layer unit

NAL unit

syntax structure (3.54) containing an indication of the type of data to follow and *bytes* (3.4) containing that data in the form of an *RBSP* (3.42) interspersed as necessary with *emulation prevention bytes* (3.23)

3.36

network abstraction layer unit stream

NAL unit stream

sequence of *NAL units* (3.35)

3.37

output order

order in which the *decoded pictures* (3.18) are output from the *decoded picture buffer* (3.19) (for the *decoded pictures* that are to be output from the *decoded picture buffer*)

3.38

partitioning

division of a set into subsets such that each element of the set is in exactly one of the subsets

3.39

plane

collection of data related to plane Y (*luma* (3.34)) or C (*chroma* (3.7))

3.40**picture**

field (3.28) or *frame* (3.29)

3.41**profile**

specified subset of the syntax of this document

3.42**raw byte sequence payload****RBSP**

syntax structure (3.54) containing an integer number of *bytes* (3.4) that is encapsulated in a *NAL unit* (3.35) and which is either empty or has the form of a *string of data bits* (3.52) containing *syntax elements* (3.53) followed by an *RBSP stop bit* (3.43) and followed by zero or more subsequent bits equal to 0

3.43**raw byte sequence payload stop bit****RBSP stop bit**

bit equal to 1 present within an *RBSP* (3.42) after a *string of data bits* (3.52)

Note 1 to entry: The location of the end of the *string of data bits* within an *RBSP* can be identified by searching from the end of the *RBSP* for the *RBSP stop bit*, which is the last non-zero bit in the *RBSP*.

3.44**reserved**

value of a particular *syntax element* (3.53) for future use by ISO/IEC and not to be used in *bitstreams* (3.2) conforming to this document, but could be used in *bitstreams* conforming to future revised editions of this document

3.45**reserved_zeros**

value of a particular *syntax element* (3.53) for future use by ISO/IEC and not to be used in *bitstreams* (3.2) conforming to this document, but could be used in *bitstreams* conforming to future revised editions of this document

Note 1 to entry: In this document, the value of any *reserved_zeros* bit is zero.

3.46**residual**

difference between a prediction of a sample or data element and a reference of that same sample or data element

3.47**residual plane**

collection of *residuals* (3.46)

3.48**run length encoding****RLE**

method for encoding a sequence of values in which consecutive occurrences of the same value are represented as a single value together with its number of occurrences

3.49**sample aspect ratio**

the ratio between the intended horizontal distance between the columns and the intended vertical distance between the rows of the *luma* (3.34) sample array in a *picture* (3.40), 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

3.50

source

video material or some of its attributes before encoding

3.51

start code prefix

unique sequence of three *bytes* (3.4) equal to 0 x 000001 embedded in the *byte stream* (3.6) as a prefix to each *NAL unit* (3.35)

Note 1 to entry: The location of a *start code prefix* can be used by a *decoder* (3.20) 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* (3.23).

3.52

string of data bits

SODB

sequence of some number of bits representing *syntax elements* (3.53) present within a *raw byte sequence payload* (3.42) prior to the *raw byte sequence payload stop bit* (3.43), and within which 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

3.53

syntax element

element of data represented in the *bitstream* (3.2)

3.54

syntax structure

zero or more *syntax elements* (3.53) present together in the *bitstream* (3.2) in a specified order

3.55

tile

rectangular region of *TUs* (3.58) within a particular *picture* (3.40)

3.56

transform

part of the *decoding process* (3.22) by which a *block* (3.3) of *transform coefficients* (3.57) is converted to a *block* of spatial-domain values

3.57

transform coefficient

scalar quantity, considered to be in a transformed domain, that is associated with a particular index in an *inverse transform* (3.31) part of the *decoding process* (3.22)

3.58

transform unit

TU

$M \times N$ *block* (3.3) of samples resulting from a *transform* (3.56) in the *decoding process* (3.22)

3.59

unspecified

value of a particular *syntax element* (3.53) with no specified meaning in this document and that will not have a specified meaning in any future revised editions of this document

3.60

video coding layer NAL unit

VCL NAL unit

NAL units (3.35) that have reserved values of *NalUnitType* that are classified as VCL NAL units in this document

4 Abbreviated terms

CLVS	coded layer-wise video sequence
CPB	coded picture buffer
CPBB	coded picture buffer of the base
CPBL	coded picture buffer LCEVC
CVS	coded video sequence
DPBB	decoded picture buffer of the base
DUT	decoder under test
HBD	hypothetical base decoder
HDM	hypothetical demuxer
HRD	hypothetical reference decoder
HSS	hypothetical stream scheduler
I	intra
LCEVC	low complexity enhancement video coding
LSB	least significant bit
MSB	most significant bit
SEI	supplemental enhancement information
VUI	video usability information

5 Conventions

5.1 General

NOTE The mathematical operators used in this document are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are defined more precisely, and additional operations are defined, such as exponentiation and real-valued division. Numbering and counting conventions generally begin from 0, e.g., “the first” is equivalent to the 0-th, “the second” is equivalent to the 1-th, etc.

5.2 Arithmetic operators

+	addition
–	subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	multiplication, including matrix multiplication
	exponentiation; specifies x to the power of y
x^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.
÷	division in mathematical equations where no truncation or rounding is intended division in mathematical equations where no truncation or rounding is intended
$\sum_{i=x}^y f(i)$	summation of f(i) with i taking all integer values from x up to and including y
x % y	Modulus Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0.

5.3 Logical operators

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

>	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

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

	bit-wise “or”
	When operating on integer arguments, operates on a two’s complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.
^	bit-wise “exclusive or”
	When operating on integer arguments, operates on a two’s complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.
x >> y	arithmetic right shift of a two’s complement integer representation of x by y binary digits
	This function is defined only for non-negative integer values of y. Bits shifted into the most significant bits (MSBs) as a result of the right shift have a value equal to the MSB of x prior to the shift operation.
x << y	arithmetic left shift of a two’s complement integer representation of x by y binary digits
	This function is defined only for non-negative integer values of y. Bits shifted into the least significant bits (LSBs) as a result of the left shift have a value equal to 0.

5.6 Assignment operators

=	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

$x = y...z$	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
$x = y \text{ to } z$	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

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

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

$$\text{Clip3}(x, y, z) = \begin{cases} x & ; \quad z < x \\ y & ; \quad z > y \\ z & ; \quad \text{otherwise} \end{cases} \quad (3)$$

$$\text{Floor}(x) \text{ largest integer less than or equal to } x \quad (4)$$

$$\text{Ln}(x) \text{ natural logarithm of } x \text{ (the base-}e\text{ logarithm, where } e \text{ is the natural logarithm base constant } 2.718\,281\,828\dots) \quad (5)$$

$$\text{Log10}(x) \text{ base-10 logarithm of } x \quad (6)$$

$$\text{Min}(x, y) = \begin{cases} x & ; \quad x \leq y \\ y & ; \quad x > y \end{cases} \quad (7)$$

$$\text{Max}(x, y) = \begin{cases} x & ; \quad x \geq y \\ y & ; \quad x < y \end{cases} \quad (8)$$

$$\text{Round}(x) = \text{Sign}(x) * \text{Floor}(\text{Abs}(x) + 0.5) \quad (9)$$

$$\text{Sign}(x) = \begin{cases} 1 & ; \quad x > 0 \\ 0 & ; \quad x = 0 \\ -1 & ; \quad x < 0 \end{cases} \quad (10)$$

$$\text{Sqrt}(x) = \sqrt{x} \quad (11)$$

5.9 Order of operation precedence

When the 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 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 document is the same as used in the C programming language.

Table 1 — Operation precedence from highest (top of the table) to lowest (bottom of the 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", "", "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"

Table 1 (continued)

operations (with operands x, y, and z)
"x == y", "x != y"
"x & y"
"x y"
"x && y"
"x y"
"x ? y : z"
"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 clause in which they are derived.

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in 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 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*_{*y**x*}. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix *s* at horizontal position *x* may be referred to as the list *s*[*x*].

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

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

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

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

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

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 dependent 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, then 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, partitioning, scanning processes and neighbouring relationships

6.1 Bitstream formats

This clause specifies the relationship between the network abstraction layer (NAL) unit stream and byte stream, either of which are referred to as the bitstream.

The bitstream can be in one of two formats: the NAL unit stream format or the byte stream format. The NAL unit stream format is conceptually the more “basic” type. It consists of a sequence of syntax structures called NAL units. This sequence is ordered in decoding order. There are constraints imposed on the decoding order (and contents) of the NAL units in the NAL unit stream. The byte stream format can be constructed from the NAL unit stream format by ordering the NAL units in decoding order and prefixing each NAL unit with a start code prefix and zero or more zero-valued bytes to form a stream of bytes. The NAL unit stream format can be extracted from the byte stream format by searching for the location of the unique start code prefix pattern within this stream of bytes. Methods of framing the NAL units in a manner other than use of the byte stream format are outside the scope of this document. The byte stream format is specified in [Annex B](#).

6.2 Source, decoded and output picture formats

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

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

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

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

For convenience of notation and terminology in this document, 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 B](#).

The variables PictureWidth and PictureHeight are defined as the width and height of the luma array and are derived from the value of the variable resolution_type ([7.4.3.3](#)). TileWidth and TileHeight are defined as the width and height of the luma array for a tile and are derived from the value of the variable tile_dimensions_type ([7.4.3.3](#)).

The variables ShiftWidthC and ShiftHeightC are specified in [Table 2](#), 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, ShiftWidthC and ShiftHeightC may be specified in the future by ISO/IEC.

Table 2 — ShiftWidthC and ShiftHeightC values derived from chroma_sampling_type ([7.3.4](#))

chroma_sampling_type	Chroma format	ShiftWidthC	ShiftHeightC
0	Monochrome	0	0
1	4:2:0	1	1
2	4:2:2	1	0
3	4:4:4	0	0

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, each of the two chroma arrays has the same height and width as the luma array.

The variables `PictureWidthC` and `PictureHeightC` are defined as the width and height of the chroma array and are derived as $(\text{PictureWidth} \gg \text{ShiftWidthC})$ and $(\text{PictureHeight} \gg \text{ShiftHeightC})$. `TileWidthC` and `TileHeightC` are defined as the width and height of the chroma array for a tile and are derived as $(\text{TileWidth} \gg \text{ShiftWidthC})$ and $(\text{TileHeight} \gg \text{ShiftHeightC})$.

The number of bits necessary for the representation of each of the samples in the luma and chroma arrays in a video sequence is in the range of 8 to 16, inclusive, and the number of bits used in the luma array may differ from the number of bits used in the chroma arrays.

When the value of `chroma_sampling_type` is equal to 1, the nominal vertical and horizontal relative locations of luma and chroma samples in pictures are as shown in [Figure 1](#).

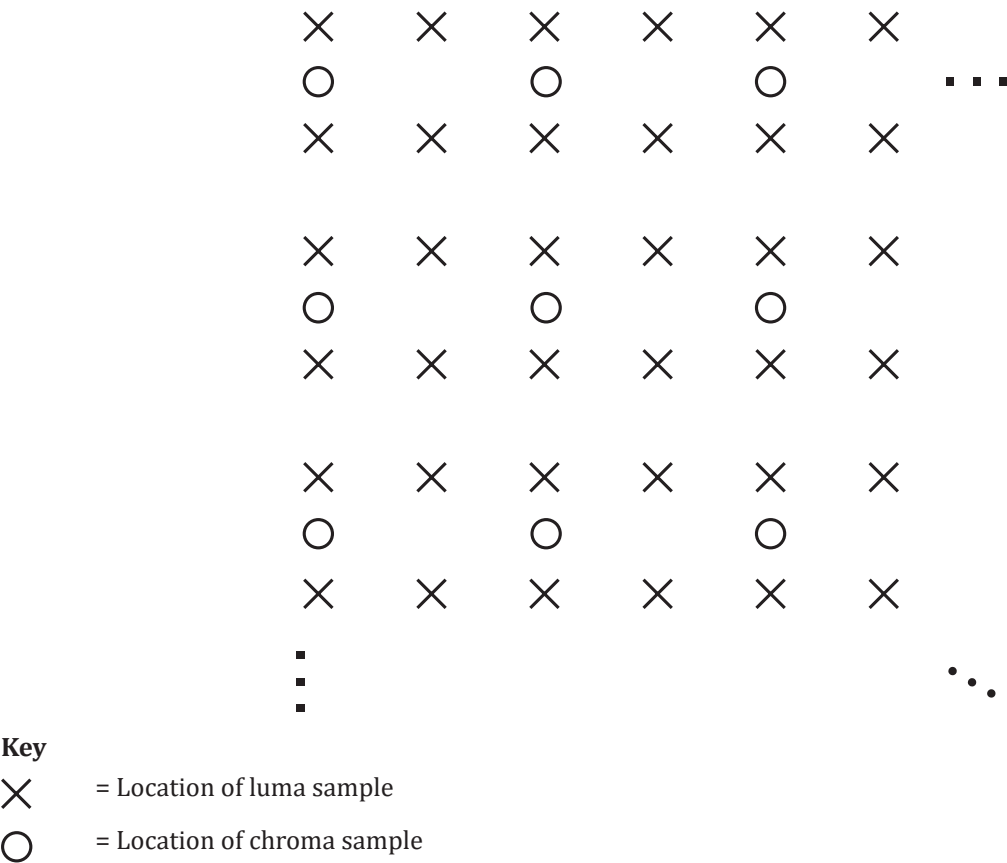


Figure 1 — Nominal vertical and horizontal locations of 4:2:0 luma and chroma samples in a picture

When the value of `chroma_sampling_type` 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 2](#).

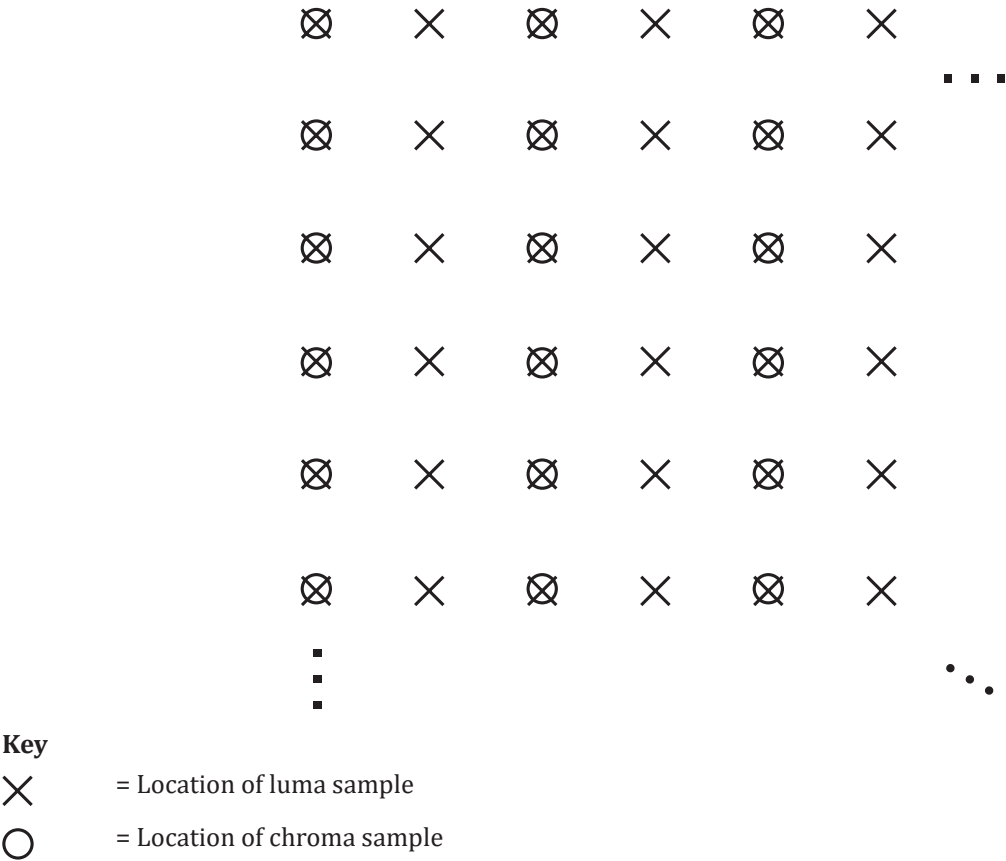


Figure 2 — Nominal vertical and horizontal locations of 4:2:2 luma and chroma samples in a picture

When the value of chroma_sampling_type 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 3](#).



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

6.3 Partitioning of pictures

6.3.1 Organization of the hierarchical structure

Each picture is organized in a hierarchical structure. The hierarchal structure is organized across three hierarchical levels: hierarchical level 0, comprising the decoded base picture; hierarchical level 1, comprising the preliminary intermediate picture, residual sub-layer 1 and the combined intermediate picture; and hierarchical level 2, comprising the preliminary output picture, residual sub-layer 2 and the combined output picture. Each picture in a hierarchical level is composed of three planes. A residual sub-layer is composed of three residual planes. The following sections specify how the different planes are organized. The decoded base picture corresponds to the decoded output of a base decoder.

NOTE The bitstream syntax and decoding process for the base decoder are not part of this document.

Residuals planes are partitioned as described below.

6.3.2 Partitioning of residuals plane

A residuals plane is divided into TUs whose size depends on the size of the transform used. The TUs have either dimension 2x2 if a 2x2 directional decomposition transform is used or dimension 4x4 if a 4x4 directional decomposition transform is used.

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 document.

[Table 3](#) 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.

Table 3 — Syntax specification format examples

Syntax specification	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	u(n)
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 are omitted if no alternative statement evaluation is needed */	
if (condition)	
primary statement	

Table 3 (continued)

Syntax specification	Descriptor
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 in [Table 4](#) 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.

Table 4 — Syntax functions and descriptors

Syntax function	Use
byte_stream_has_data()	If the byte-stream has more data, then returns TRUE; otherwise returns FALSE.
byte_aligned()	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.
bytestream_current(bitstream)	Returns the current bitstream pointer.
bytestream_seek(bitstream, n)	Returns the current bitstream pointer at the position in the bitstream corresponding to n bytes.
more_data_in_byte_stream()	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()	If byte_aligned() is equal to TRUE and the current position in the sei_payload() syntax structure is 8 * payloadSize bits from the beginning of the sei_payload() syntax structure, the return value of more_data_in_payload() is equal to FALSE. Otherwise, the return value of more_data_in_payload() is equal to TRUE.
next_bits(n)	It provides the next bits in the bitstream for comparison purposes, without advancing the bitstream pointer. It provides a look at the next n bits in the bitstream with n being its argument. When used within the byte stream format as specified in Annex B and fewer than n bits remain within the byte stream, next_bits(n) returns a value of 0.
payload_extension_present()	If the current position in the sei_payload() syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 * payloadSize bits from the beginning of the syntax structure (i.e., the position of the payload_bit_equal_to_one syntax element), the return value of payload_extension_present() is equal to TRUE. Otherwise, the return value of payload_extension_present() is equal to FALSE.

Table 4 (continued)

Syntax function	Use
process_payload_function(payload_type, payload_byte_size)	Behaves like a function lookup table, by selecting and invoking the process payload function relating to the payload_type as outlined in 7.3.4 .
read_bits(n)	Reads the next n bits from the bitstream. Following the read operation, the bitstream pointer is advanced by n bit positions. When n is equal to 0, read_bits(n) returns a value equal to 0 and the bitstream pointer is not advanced.
read_byte(bitstream)	Reads a byte in the bitstream returning its value. Following the return of the value, the bitstream pointer is advanced by a byte.
read_multibyte(bitstream)	Executes a read_byte(bitstream) until the MSB of the read byte is equal to zero. The return value for each read byte is the value represented by the bits other than the MSB.

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

- 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).
- 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 [9.4](#).
- mb: read multiple bytes. The parsing process for this descriptor is specified by the return value of the function read_multibyte(bitstream) interpreted as a binary representation of multiple unsigned char with most significant bit written first, and most significant byte of the sequence of unsigned char written first.
- rle(v): variable length code from run length encoding. The parsing process for such descriptor is specified in specific clauses referenced in [7.3](#).

7.3 Syntax in tabular form

7.3.1 Syntax order

The order in which the syntax is presented is from MSB to LSB.

7.3.2 NAL unit and NAL unit header syntax

NAL unit and NAL unit header syntax is specified in [Table 5](#) and [Table 6](#), respectively.

Table 5 — NAL unit

Syntax	Descriptor
nal_unit(NumBytesInNalUnit) {	
nal_unit_header()	

Table 5 (continued)

Syntax	Descriptor
NumBytesInRBSP = 0	
for (i = 2; i < NumBytesInNALUnit; i++) {	
if (i + 2 < NumBytesInNALUnit && next_bits(24) == 0x000003) {	
rbsp_byte [NumBytesInRBSP++]	u(8)
rbsp_byte [NumBytesInRBSP++]	u(8)
i += 2	
emulation_prevention_three_byte /* equal to 0x03 */	u(8)
} else	
rbsp_byte [NumBytesInRBSP++]	u(8)
}	
}	

Table 6 — NAL unit header

Syntax	Descriptor
nal_unit_header() {	
forbidden_zero_bit	u(1)
forbidden_one_bit	u(1)
nal_unit_type	u(5)
reserved_flag	u(9)
}	

7.3.3 Process block syntax

Process block syntax is specified in [Table 7](#).

Table 7 — Process block syntax

Syntax	Descriptor
process_block() {	
payload_size_type	u(3)
payload_type	u(5)
payload_size = 0	
if (payload_size_type == 7) {	
custom_byte_size	mb
payload_size = custom_byte_size	
} else {	
if (payload_size_type == 0) payload_size = 0	
if (payload_size_type == 1) payload_size = 1	
if (payload_size_type == 2) payload_size = 2	
if (payload_size_type == 3) payload_size = 3	
if (payload_size_type == 4) payload_size = 4	
if (payload_size_type == 5) payload_size = 5	
}	
if (payload_type == 0)	
process_payload_sequence_config(payload_size)	

Table 7 (continued)

Syntax	Descriptor
else if (payload_type == 1)	
process_payload_global_config(payload_size)	
else if (payload_type == 2)	
process_payload_picture_config(payload_size)	
else if (payload_type == 3)	
process_payload_encoded_data(payload_size)	
else if (payload_type == 4)	
process_payload_encoded_data_tiled(payload_size)	
else if (payload_type == 5)	
process_payload_additional_info(payload_size)	
else if (payload_type == 6)	
process_payload_filler(payload_size)	
}	

7.3.4 Process payload – sequence configuration

Process payload – sequence configuration syntax is specified in [Table 8](#).

Table 8 — Process payload – sequence configuration

Syntax	Descriptor
process_payload_sequence_config(payload_size) {	
profile_idc	u(4)
level_idc	u(4)
sublevel_idc	u(2)
conformance_window_flag	u(1)
reserved_zeros_5bit	u(5)
if (profile_idc == 15 level_idc == 15) {	
extended_profile_idc	u(3)
extended_level_idc	u(4)
reserved_zeros_1bit	u(1)
}	
if (conformance_window_flag == 1) {	
conf_win_left_offset	mb
conf_win_right_offset	mb
conf_win_top_offset	mb
conf_win_bottom_offset	mb
}	
}	

7.3.5 Process payload – global configuration

Process payload – global configuration syntax is specified in [Table 9](#).

Table 9 — Process payload – global configuration

Syntax	Descriptor
process_payload_global_config(payload_size) {	
processed_planes_type_flag	u(1)
resolution_type	u(6)
transform_type	u(1)
chroma_sampling_type	u(2)
base_depth_type	u(2)
enhancement_depth_type	u(2)
temporal_step_width_modifier_signalled_flag	u(1)
predicted_residual_mode_flag	u(1)
temporal_tile_intra_signalling_enabled_flag	u(1)
temporal_enabled_flag	u(1)
upsample_type	u(3)
level1_filtering_signalled_flag	u(1)
scaling_mode_level1	u(2)
scaling_mode_level2	u(2)
tile_dimensions_type	u(2)
user_data_enabled	u(2)
level1_depth_flag	u(1)
chroma_step_width_flag	u(1)
if (processed_planes_type_flag == 1) {	
planes_type	u(4)
reserved_zeros_4bit	u(4)
}	
if (temporal_step_width_modifier_signalled_flag == 1) {	
temporal_step_width_modifier	u(8)
} else {	
temporal_step_width_modifier = 48	
}	
if (upsample_type == 4) {	
upsampler_coeff1	u(16)
upsampler_coeff2	u(16)
upsampler_coeff3	u(16)
upsampler_coeff4	u(16)
}	
if (level1_filtering_signalled_flag) {	
level1_filtering_first_coefficient	u(4)
level1_filtering_second_coefficient	u(4)
}	
if (tile_dimensions_type > 0) {	
if (tile_dimensions_type == 3) {	
custom_tile_width	u(16)
custom_tile_height	u(16)
}	
reserved_zeros_5bit	u(5)

Table 9 (continued)

Syntax	Descriptor
compression_type_entropy_enabled_per_tile_flag	u(1)
compression_type_size_per_tile	u(2)
}	
if (resolution_type == 63) {	
custom_resolution_width	u(16)
custom_resolution_height	u(16)
}	
if (chroma_stepwidth_flag) {	
chroma_step_width_multiplier	u(8)
} else {	
chroma_step_width_multiplier = 64	
}	
}	

7.3.6 Process payload – picture configuration

Process payload – picture configuration syntax is specified in [Table 10](#).

Table 10 — Process payload – picture configuration

Syntax	Descriptor
process_payload_picture_config(payload_size) {	
no_enhancement_bit_flag	u(1)
if (no_enhancement_bit_flag == 0) {	
quant_matrix_mode	u(3)
dequant_offset_signalled_flag	u(1)
picture_type_bit_flag	u(1)
temporal_refresh_bit_flag	u(1)
step_width_sublayer1_enabled_flag	u(1)
step_width_sublayer2	u(15)
dithering_control_flag	u(1)
} else {	
reserved_zeros_4bit	u(4)
picture_type_bit_flag	u(1)
temporal_refresh_bit_flag	u(1)
temporal_signalling_present_flag	u(1)
}	
if (picture_type_bit_flag == 1) {	
field_type_bit_flag	u(1)
reserved_zeros_7bit	u(7)
}	
if (step_width_sublayer1_enabled_flag == 1) {	
step_width_sublayer1	u(15)
level1_filtering_enabled_flag	u(1)
}	

Table 10 (continued)

Syntax	Descriptor
if (quant_matrix_mode == 2 quant_matrix_mode == 3 quant_matrix_mode == 5) {	
for(layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
qm_coefficient_0[layerIdx]	u(8)
}	
}	
if (quant_matrix_mode == 4 quant_matrix_mode == 5) {	
for(layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
qm_coefficient_1[layerIdx]	u(8)
}	
}	
if (dequant_offset_signalled_flag) {	
dequant_offset_mode_flag	u(1)
dequant_offset	u(7)
}	
if (dithering_control_flag == 1) {	
dithering_type	u(2)
reserverd_zero	u(1)
if (dithering_type != 0) {	
dithering_strength	u(5)
} else {	
reserved_zeros_5bit	u(5)
}	
}	
}	

7.3.7 Process payload – encoded data

Process payload – encoded data syntax is specified in [Table 11](#).

Table 11 — Process payload – encoded data

Syntax	Descriptor
process_payload_encoded_data(payload_size) {	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
if (no_enhancement_bit_flag == 0) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
surfaces[planeIdx][levelIdx][layerIdx].entropy_enabled_flag	u(1)
surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag	u(1)
}	
}	
}	
if (temporal_signalling_present_flag == 1){	

Table 11 (continued)

Syntax	Descriptor
temporal_surfaces[planeIdx].entropy_enabled_flag	u(1)
temporal_surfaces[planeIdx].rle_only_flag	u(1)
}	
}	
byte_alignment()	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++)	
process_surface(surfaces[planeIdx][levelIdx][layerIdx])	
}	
if (temporal_signalling_present_flag == 1)	
process_surface(temporal_surfaces[planeIdx])	
}	
}	

7.3.8 Process payload – encoded tiled data

Process payload – encoded tiled data syntax is specified in [Table 12](#).

Table 12 — Process payload – encoded tiled data

Syntax	Descriptor
process_payload_encoded_data_tiled(payload_size) {	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
if (no_enhancement_bit_flag == 0) {	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++)	
surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag	u(1)
}	
}	
if (temporal_signalling_present_flag == 1)	
temporal_surfaces[planeIdx].rle_only_flag	u(1)
}	
byte_alignment()	
if (compression_type_entropy_enabled_per_tile_flag == 0) {	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
if (no_enhancement_bit_flag == 0) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
if (levelIdx == 1)	
nTiles = nTilesL1	
else	
nTiles = nTilesL2	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
for (tileIdx = 0; tileIdx < nTiles; tileIdx++)	

Table 12 (continued)

Syntax	Descriptor
surfaces [planeIdx][levelIdx][layerIdx]. tiles [tileIdx]. entropy_enabled_flag	u(1)
}	
}	
}	
if (temporal_signalling_present_flag == 1) {	
for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++)	
temporal_surfaces [planeIdx]. tiles [tileIdx]. entropy_enabled_flag	u(1)
}	
}	
} else {	
entropy_enabled_per_tile_compressed_data_rle	rle(v) 9.3.5
}	
byte_alignment()	
if (compression_type_size_per_tile == 0) {	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
if (levelIdx == 1)	
nTiles = nTilesL1	
else	
nTiles = nTilesL2	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
for (tileIdx = 0; tileIdx < nTiles; tileIdx++)	
process_surface(surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx])	
}	
}	
if (temporal_signalling_present_flag == 1) {	
for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++)	
process_surface(temporal_surfaces[planeIdx].tiles[tileIdx])	
}	
}	
} else {	
for (planeIdx = 0; planeIdx < nPlanes; planeIdx++) {	
for (levelIdx = 1; levelIdx <= 2; levelIdx++) {	
if (levelIdx == 1)	
nTiles = nTilesL1	
else	
nTiles = nTilesL2	
for (layerIdx = 0; layerIdx < nLayers; layerIdx++) {	
if(surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag) {	
compressed_size_per_tile_prefix	rle(v) 9.2.3
} else {	
compressed_size_per_tile_prefix	rle(v) 9.2.3
}	

Table 12 (continued)

Syntax	Descriptor
for (tileIdx=0; tileIdx < nTiles; tileIdx++)	
process_surface(surfaces[planeIdx][levelIdx][layerIdx].	
tiles[tileIdx])	
}	
}	
if (temporal_signalling_present_flag == 1) {	
if(temporal_surfaces[planeIdx].rle_only_flag) {	
compressed_size_per_tile_prefix	rle(v) 9.2.3
} else {	
compressed_size_per_tile_prefix	rle(v) 9.2.3
}	
for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++)	
process_surface(temporal_surfaces[planeIdx].tiles[tileIdx])	
}	
}	
}	

7.3.9 Process payload – surface

Process payload – surface syntax is specified in [Table 13](#).

Table 13 — Process payload – surface

Syntax	Descriptor
process_surface(surface) {	
if (compression_type_size_per_tile == 0) {	
if (surface.entropy_enabled_flag) {	
surface.size	mb
if (surface.rle_only_flag) {	
surface.data_rle	surface.size
} else {	
surface.data_prefix_coding	surface.size
}	
}	
} else {	
if (surface.entropy_enabled_flag) {	
surface.size	mb
surface.data_prefix_coding	surface.size
}	
}	
}	

7.3.10 Process payload – additional info

Process payload – additional info syntax is specified in [Table 14](#).

Table 14 — Process payload – additional info

Syntax	Descriptor
process_payload_additional_info(payload_size) {	
additional_info_type	u(8)
if (additional_info_type == 0) {	
payload_type	u(8)
sei_payload(payload_type, payload_size - 2)	
} else if (additional_info_type == 1)	
vui_parameters (payload_size - 1)	
else // (additional_info_type >= 2)	
// reserved for future use	
}	

7.3.11 Process payload – filler

Process payload – filler syntax is specified in [Table 15](#).

Table 15 — Process payload – filler

Syntax	Descriptor
process_payload_filler(payload_size) {	
for (x = 0; x < payload_size; x++) {	
filler_byte // equal to 0xAA	u(8)
}	
}	

7.3.12 Byte alignment syntax

Byte alignment syntax is specified in [Table 16](#).

Table 16 — Byte alignment syntax

Syntax	Descriptor
byte_alignment() {	
alignment_bit_equal_to_one /* equal to 1 */	f(1)
while (!byte_aligned())	
alignment_bit_equal_to_zero /* equal to 0 */	f(1)
}	

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 clause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

The following bitstream conformance constraints apply only to coded pictures present in the bitstream and do not apply to decoded base pictures that are provided by external means.

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 document.

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), then 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 contain 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.

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 17](#).

NOTE 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.

forbidden_one_bit shall be equal to 1.

nal_unit_type specifies the type of RBSP data structure contained in the NAL unit as specified in [Table 17](#).

NAL units that have **nal_unit_type** in the range of UNSPEC0...UNSPEC27, inclusive, and UNSPEC31 for which semantics are not specified, shall not affect the decoding process specified in this document.

reserved_flag shall be equal to the bit sequence 11111111.

Table 17 — 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...27	UNSPEC0...UNSPEC27	Unspecified	Non-VCL
28	LCEVC_NON_IDR	Non-IDR segment	VCL/Non-VCL
29	LCEVC_IDR	IDR segment	VCL/Non-VCL
30	LCEVC_RSV	reserved	VCL/Non-VCL
31	UNSPEC31	Unspecified	Non-VCL

NOTE 1 NAL unit types in the range of UNSPEC0...UNSPEC27 and UNSPEC31 can be used as determined by the application. No decoding process for these values of **nal_unit_type** is specified in this document. Since different applications might use these NAL unit types for different purposes, particular care is expected to 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 document.

7.4.2.3 Data block unit general semantics

payload_size_type specifies the size of the payload, and it shall take a value between 0 and 7, as specified by [Table 18](#).

Table 18 — Payload sizes

payload_size_type	Size (bytes)
0	0
1	1
2	2
3	3
4	4
5	5
6	reserved
7	Custom

payload_type specifies the type of the payload used, and it should take a value between 0 and 31, as specified by [Table 19](#).

Table 19 — Content of payload and minimum frequency of appearance of such content within a bitstream

payload_type	Content of payload	Minimum frequency
0	process_payload_sequence_config()	at least with first IDR in sequence
1	process_payload_global_config()	per IDR picture
2	process_payload_picture_config()	per picture
3	process_payload_encoded_data()	per picture (if no_enhancement_bit_flag == 0)
4	process_payload_encoded_data_tiled()	not specified (optional)
5	process_payload_additional_info()	not specified (optional)
6	process_payload_filler()	not specified (optional)
7-30	Reserved	
31	Unspecified	

custom_byte_size specifies a custom size of a payload not included in [Table 18](#).

7.4.3 Data block unit configuration semantics

7.4.3.1 Data block semantics

The following subclauses describe the semantics for each of the data block units.

7.4.3.2 Data block unit sequence configuration semantics

profile_idc indicates a profile as specified in [Annex A](#). Bitstreams shall not contain values of profile_idc other than those specified in [Annex A](#). Other values of profile_idc are reserved for future use by ISO/IEC.

level_idc indicates a level as specified in [Annex A](#). Bitstreams shall not contain values of level_idc other than those specified in [Annex A](#). Other values of level_idc are reserved for future use by ISO/IEC.

sublevel_idc indicates a sublevel as specified in [Annex A](#).

conformance_window_flag equal to 1 indicates that the conformance cropping window offset parameters are present in the sequence configuration data block. conformance_window_flag equal to 0 indicates that the conformance cropping window offset parameters are not present.

extended_profile_idc indicates an extended profile is being invoked and will be specified in [Annex A](#). Bitstreams shall not contain values of extended_profile_idc other than those specified in [Annex A](#). Other values of extended_profile_idc are reserved for future use by ISO/IEC.

extended_level_idc indicates an extended level is being invoked and will be specified in [Annex A](#). Bitstreams shall not contain values of extended_level_idc other than those specified in [Annex A](#). Other values of extended_level_idc are reserved for future use by ISO/IEC.

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 (conf_win_left_offset << ShiftWidthC) to (width - ((conf_win_right_offset << ShiftWidthC) + 1)) and vertical picture coordinates from (conf_win_top_offset << ShiftHeightC) to (height - ((conf_win_bottom_offset << ShiftHeightC) + 1)), inclusive.