INTERNATIONAL STANDARD

ISO/IEC 23094-2

First edition
2021-11

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)

Contents

Page

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 (CVSs)

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
MxN *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
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.
÷	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 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

$$\mathrm{Abs}(x) = \begin{cases} x & ; \quad x >= 0 \\ -x & ; \quad x < 0 \end{cases} \tag{1}$$

Ceil(x) smallest integer greater than or equal to x (2)

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

Floor(x) largest integer less than or equal to x (4)

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

Log10(x) base-10 logarithm of x (6)

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

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

Round(x) = Sign(x) * Floor(Abs(x) + 0.5) (9)

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

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

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_{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 { {...} {...} }, 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.

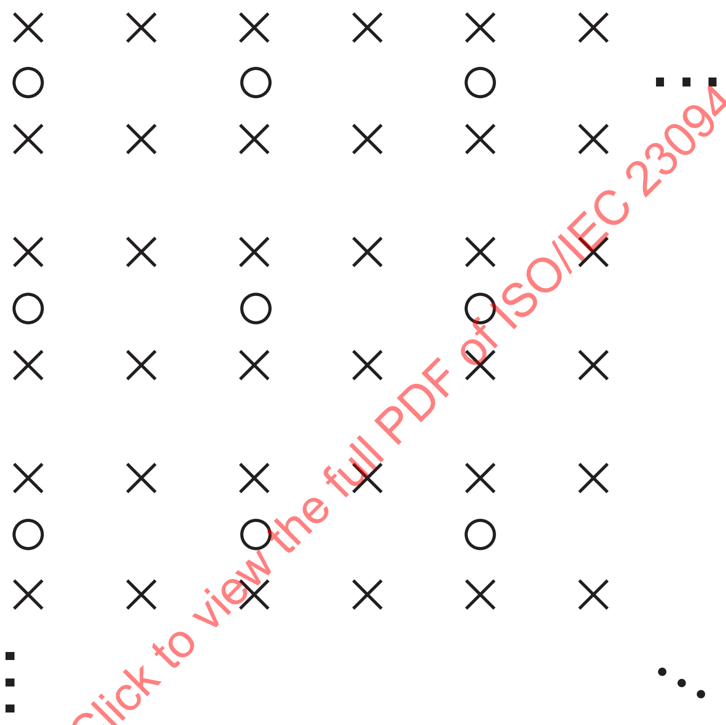
In 4:2:2 sampling, each of the two chroma arrays has the same height and half the width of the luma array.

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 (PictureWidth >> ShiftWidthC) and (PictureHeight >> ShiftHeightC). TileWidthC and TileHeightC are defined as the width and height of the chroma array for a tile and are derived as (TileWidth >> ShiftWidthC) and (TileHeight >> 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.



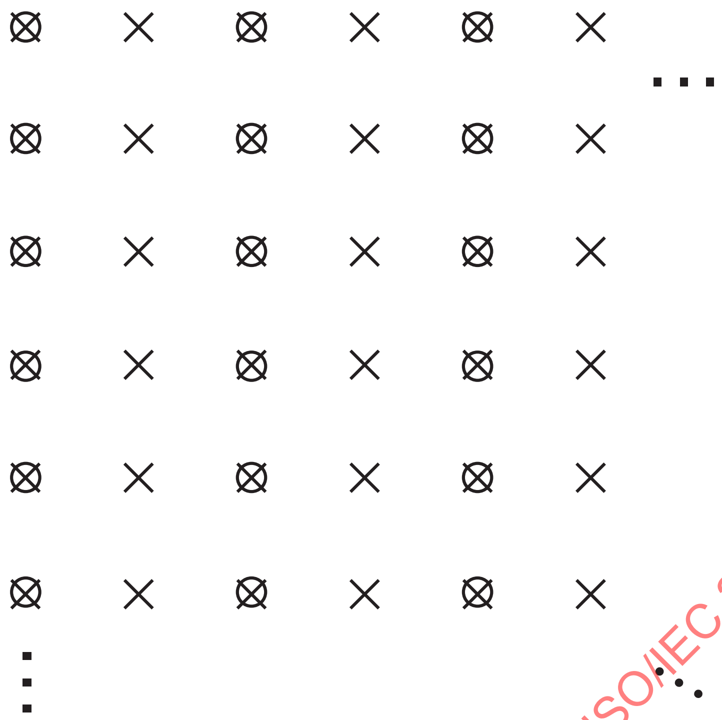
Key

× = Location of luma sample

○ = Location of chroma sample

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.

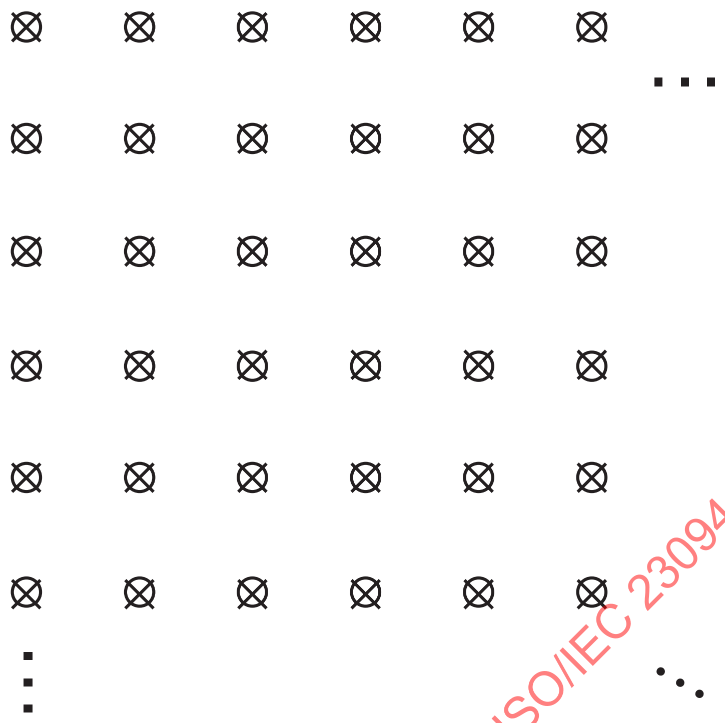


Key

×	= Location of luma sample
○	= Location of chroma sample

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.



Key

×	= Location of luma sample
○	= Location of chroma sample

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 111111111.

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.

The value of ((conf_win_left_offset + conf_win_right_offset) << ShiftWidthC) shall be less than width, and the value of ((conf_win_top_offset + conf_win_bottom_offset) << ShiftHeightC) shall be less than height.

If chroma_sampling_type is not equal to 0, then the corresponding specified samples of the two chroma arrays are the samples having picture coordinates (x >> ShiftWidthC, y >> ShiftHeightC), where (x, y) are the picture coordinates of the specified luma samples.

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

7.4.3.3 Data block unit global configuration semantics

processed_planes_type_flag specifies the planes to be processed by the decoder. It should be equal to 0 or 1. If equal to 0, only the Luma (Y) plane should be processed. If equal to 1, the value of planes_type is specified. If processed_planes_type_flag is equal to 0, nPlanes should be equal to 1.

resolution_type specifies the resolution of the Luma (Y) plane of the enhanced decoded picture, and it should take a value between 0 and 63, as specified by Table 20. The value of the type is expressed as NxM, where N is the width of the Luma (Y) plane of the enhanced decoded picture and M is height of the Luma (Y) plane of the enhanced decoded picture. In order to prevent incomplete TUs, as defined in 6.3.2, N and M shall be an integer multiple of the TU size (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1) for each sub-layer and for each plane within a sub-layer. When N is not a multiple of the TU size, N is inferred to be ceiling(N/nTbS) * nTbS. When M is not a multiple of the TU size, M is inferred to be ceiling(M/nTbS) * nTbS. When N or M are not a multiple of the TU size, the mechanism of conformance cropping window specified by conformance_window_flag shall be used as specified in 7.4.3.2

Table 20 — Resolution of the decoded picture

resolution_type	Value of type
0	unused /* Escape code prevention */
1	360x200
2	400x240
3	480x320
4	640x360
5	640x480
6	768x480
7	800x600
8	852x480
9	854x480
10	856x480
11	960x540
12	960x640
13	1024x576
14	1024x600
15	1024x768
16	1152x864
17	1280x720
18	1280x800
19	1280x1024
20	1360x768
21	1366x768

Table 20 *(continued)*

resolution_type	Value of type
22	1400x1050
23	1440x900
24	1600x1200
25	1680x1050
26	1920x1080
27	1920x1200
28	2048x1080
29	2048x1152
30	2048x1536
31	2160x1440
32	2560x1440
33	2560x1600
34	2560x2048
35	3200x1800
36	3200x2048
37	3200x2400
38	3440x1440
39	3840x1600
40	3840x2160
41	3840x2400
42	4096x2160
43	4096x3072
44	5120x2880
45	5120x3200
46	5120x4096
47	6400x4096
48	6400x4800
49	7680x4320
50	7680x4800
51-62	Reserved
63	Custom

chroma_sampling_type defines the colour format for the enhanced decoded picture (see 6.2) in accordance with Table 21.

Table 21 — Colour format for the decoded picture

chroma_sampling_type	Value of type
0	Monochrome
1	4:2:0
2	4:2:2
3	4:4:4

transform_type defines the type of transform to be used in accordance with Table 22. If transform_type is equal to 0, nLayers should be equal to 4 and if transform_type is equal to 1, nLayers should be equal to 16.

Table 22 — Transform used for decoding

transform_type	Value of type
0	2x2 directional decomposition transform
1	4x4 directional decomposition transform

base_depth_type defines the bit depth of the decoded base picture in accordance with Table 23. The value of type shall be the same as the value of the bit depth used for the decoded base picture.

Table 23 — Bit depth of the decoded base picture

base_depth_type	Value of type
0	8
1	10
2	12
3	14

enhancement_depth_type defines the bit depth of the enhanced decoded picture in accordance with Table 24.

Table 24 — Bit depth of the decoded picture

enhancement_depth_type	Value of type
0	8
1	10
2	12
3	14

planes_type specifies the planes to be processed by the decoder according to Table 25.

Table 25 — Planes to be processed in the decoded picture

planes_type	Planes
0	Luma (Y) plane only
1	Luma (Y) and Chroma (U and V) planes
2-15	Reserved

If planes_type is equal to 0, nPlanes should be equal to 1. If planes_type is equal to 1, nPlanes should be equal to 3.

temporal_step_width_modifier_signalled_flag specifies if the value of the temporal_step_width_modifier parameter is signalled. It should be equal to 0 or 1. If equal to 0, the temporal_step_width_modifier parameter is not signalled.

upsampler_coeff1 ... upsampler_coeff4 specify the coefficient values for Adaptive Cubic upsampler.

predicted_residual_mode_flag specifies whether the decoder should activate the predicted residual process during the decoding process. If the value is 0, the predicted residual process should be disabled.

temporal_tile_intra_signalling_enabled_flag specifies whether temporal tile prediction should be used when decoding a 32x32 tile. If the value is 1, the temporal tile prediction process should be enabled.

temporal_enabled_flag specifies whether temporal prediction is enabled or not.

upsample_type specifies the type of upsampler to be used in the decoding process in accordance with Table 26.

Table 26 — Upsampler type

upsample_type	Value of type
0	Nearest
1	Linear
2	Cubic
3	Modified cubic
4	Adaptive cubic
5-6	Reserved
7	Unspecified

level1_filtering_signalled_flag specifies whether deblocking filter should use the signalled parameters instead of default parameters. If equal to 1, the values of the deblocking coefficients are signalled.

temporal_step_width_modifier specifies the value used to calculate a variable step width modifier for transforms that use temporal prediction. If temporal_step_width_modifier_signalled_flag is equal to 0, temporal_step_width_modifier is set to 48.

level1_filtering_first_coefficient specifies the value of the first coefficient in the deblocking mask namely 4x4 block corner residual weight. The value of the first coefficient should be between 0 and 15.

level1_filtering_second_coefficient specifies the value of the second coefficient in the deblocking mask namely 4x4 block side residual weight. The value of the second coefficient should be between 0 and 15.

scaling_mode_level1 specifies whether and how the upsampling process should be performed between decoded base picture and preliminary intermediate picture in accordance with Table 27. The preliminary intermediate picture corresponds to the output of process of 8.6.1.2.

Table 27 — Scaling mode level1 values

scaling_mode_level1	Value of type
0	no scaling
1	one-dimensional 2:1 scaling only across the horizontal dimension
2	two-dimensional 2:1 scaling across both dimensions
3	Reserved

scaling_mode_level2 specifies whether and how the upsampling process should be performed between combined intermediate picture and preliminary output picture in accordance with Table 28. The combined intermediate picture corresponds to the output of process of 8.8.2. The preliminary output picture corresponds to the output of process of 8.6.1.3. Scaling mode level2values are specified in Table 28.

Table 28 — Scaling mode level2values

scaling_mode_level2	Value of type
0	no scaling
1	one-dimensional 2:1 scaling only across the horizontal dimension
2	two-dimensional 2:1 scaling across both dimensions
3	Reserved

NOTE 1 In case scaling_mode_level1 or scaling_mode_level2 equal to 1, for one-dimensional 2:1 scaling only across the horizontal dimension, in order to maintain the source display aspect ratio, the sample aspect ratio of the decoded base picture is expected to be doubled, for each scaling_model_level equal to 1, in the horizontal value with respect to the source sample aspect ratio.

user_data_enabled specifies whether user data are included in the bitstream and the size of the user data, as specified in Table 29.

Table 29 — User data

user_data_enabled	Value of type
0	Disabled
1	enabled 2-bits
2	enabled 6-bits
3	Reserved

level1_depth_flag specifies whether hierarchical level 1 (see clause 6.3.1) is processed using the base depth type or the enhancement depth type. If the value of the flag is 0, hierarchical level 1 should be processed using the base depth type. If the value of the flag is 1, hierarchical level 1 should be processed using the enhancement depth type.

chroma_step_width_flag specifies whether chroma_step_width_multiplier is present or not. If the value of this flag is 1, the syntax element chroma_step_width_multiplier is present. If the value is 0, the syntax element chroma_step_width_multiplier is not present.

tile_dimensions_type specifies the resolution of the picture tiles, and it should take a value between 0 and 3 according to Table 30. The value of the type is expressed as NxM, where N is the width of the picture tile and M is height of the picture tile.

Table 30 — Resolution of the decoded picture tiles

tile_dimensions_type	Value of type
0	no tiling
1	512x256
2	1024x512
3	Custom

custom_tile_width specifies a custom width for the tile. In order to prevent incomplete TUs, as defined in 6.3.2, custom_tile_width shall be an integer multiple of the TU size (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1) for each sub-layer and for each plane within a sub-layer.

custom_tile_height specifies a custom height for the tile. In order to prevent incomplete TUs, as defined in 6.3.2, custom_tile_height shall be an integer multiple of the TU size (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1) for each sub-layer and for each plane within a sub-layer.

compression_type_entropy_enabled_per_tile_flag specifies the compression method used to encode the entropy_enabled_flag field of each picture tile, and it should take a value between 0 and 1 according to Table 31.

Table 31 — Compression method for entropy_enabled_flag of each picture tile

compression_type_entropy_enabled_per_tile_flag	Value of type
0	No compression used
1	Run length encoding

compression_type_size_per_tile specifies the compression method used to encode the size field of each picture tile, and it should take a value between 0 and 3 according to Table 32.

Table 32 — Compression method for size of each picture tile

compression_type_size_per_tile	Value of type
0	No compression used
1	Prefix Coding encoding
2	Prefix Coding encoding on differences
3	Reserved

custom_resolution_width specifies the width of a custom resolution. In order to prevent incomplete TUs, as defined in 6.3.2, when custom_resolution_width is not an integer multiple of TU size, PictureWidth is inferred to be ceiling(PictureWidth/nTbS) * nTbS, and the mechanism of conformance cropping window specified by conformance_window_flag shall be used as specified in 7.4.3.2.

custom_resolution_height specifies the height of a custom resolution. In order to prevent incomplete TUs, as defined in 6.3.2, when custom_resolution_height is not an integer multiple of TU size, PictureHeight is inferred to be ceiling(PictureHeight/nTbS) * nTbS, and the mechanism of conformance cropping window specified by conformance_window_flag shall be used as specified in 7.4.3.2.

chroma_step_width_multiplier specifies a value used to scale the step_width_sublayer2 for the chroma planes. If not specified, the default value is 64.

7.4.3.4 Data block unit picture configuration semantics

no_enhancement_bit_flag specifies that there are no enhancement data for all layerIdx < nLayers in the picture.

quant_matrix_mode specifies which quantization matrix to be used in the decoding process in accordance with Table 33. When quant_matrix_mode is not present, it is inferred to be equal to 0.

Table 33 — Quantization matrix

quant_matrix_mode	Value of type
0	each enhancement sub-layer uses the matrices used for the previous frame, unless the current picture is an IDR picture, in which case both enhancement sub-layers use default matrices
1	both enhancement sub-layers use default matrices
2	one matrix of modifiers is signalled and should be used on both enhancement sub-layers
3	one matrix of modifiers is signalled and should be used on enhancement sub-layer 2
4	one matrix of modifiers is signalled and should be used on enhancement sub-layer 1
5	two matrices of modifiers are signalled – the first one for enhancement sub-layer 2, the second for enhancement sub-layer 1
6-7	Reserved

dequant_offset_signalled_flag specifies if the offset method and the value of the offset parameter to be applied when dequantizing is signalled. If equal to 1, the method for dequantization offset and the value of the dequantization offset parameter are signalled. When dequant_offset_signalled_flag is not present, it is inferred to be equal to 0.

picture_type_bit_flag specifies whether the encoded data are sent on a frame basis (e.g., progressive mode or interlaced mode) or on a field basis (e.g., interlaced mode) in accordance with Table 34.

Table 34 — Picture type

picture_type_bit_flag	Value of type
0	frame
1	Field

field_type_bit_flag specifies, if picture_type_bit_flag is equal to 1, whether the data sent are for top or bottom field in accordance with Table 35.

Table 35 — Field type

field_type_bit_flag	Value of type
0	Top
1	Bottom

temporal_refresh_bit_flag specifies whether the temporal buffer should be refreshed for the picture. If equal to 1, the temporal buffer should be refreshed. For an IDR picture, temporal_refresh_bit_flag shall be set to 1.

temporal_signalling_present_flag specifies whether the temporal signalling coefficient group is present in the bitstream. When temporal_signalling_present_flag is not present, it is inferred to be equal to 1 if temporal_enabled_flag is equal to 1 and temporal_refresh_bit_flag is equal to 0, otherwise it is inferred to be equal to 0.

step_width_sublayer2 specifies the value of the stepwidth value to be used when decoding the encoded residuals in enhancement sub-layer 2 for the luma plane. The stepwidth value to be used when decoding the encoded residuals in enhancement sub-layer 2 for the chroma planes shall be computed as Clip3(1, 32,767, ((step_width_sublayer2 * chroma_step_width_multiplier)>>6)).

step_width_sublayer1_enabled_flag specifies whether the value of the stepwidth to be used when decoding the encoded residuals in the enhancement sub-layer 1 is a default value or is signalled. It should be either 0 (default value) or 1 (value signalled by step_width_sublayer1). The default value is 32,767. When step_width_sublayer1_enabled_flag is not present, it is inferred to be equal to 0.

dithering_control_flag specifies whether dithering should be applied. It should be either 0 (dithering disabled) or 1 (dithering enabled). When dithering_control_flag is not present, it is inferred to be equal to 0 for IDR picture, and for pictures other than the IDR picture it is inferred to be equal to the value of dithering_control_flag for the preceding picture.

step_width_sublayer1 specifies the value of the stepwidth value to be used when decoding the encoded residuals in enhancement sub-layer 1.

level1_filtering_enabled_flag specifies whether the deblocking filter in clause 8.9 should be used. It should be either 0 (filtering disabled) or 1 (filtering enabled). When level1_filtering_enabled_flag is not present, it is inferred to be equal to 0.

qm_coefficient_0[layerIdx] specifies the values of the quantization matrix scaling parameter when quant_matrix_mode is equal to 2, 3 or 5.

qm_coefficient_1[layerIdx] specifies the values of the quantization matrix scaling parameter for when quant_matrix_mode is equal to 4 or 5.

dequant_offset_mode_flag specifies the method for applying dequantization offset. If equal to 0, the default method applies, using the signalled dequant_offset as parameter. If equal to 1, the constant-offset method applies, using the signalled dequant_offset parameter.

dequant_offset specifies the value of the dequantization offset parameter to be applied. The value of the dequantization offset parameter should be between 0 and 127, inclusive.

dithering_type specifies what type of dithering is applied to the final reconstructed picture according to Table 36.

Table 36 — Dithering

dithering_type	Value of type
0	None
1	Uniform
2-3	Reserved

dithering_strength specifies a value between 0 and 31.

7.4.3.5 Data block unit encoded data semantics

surfaces[planeIdx][levelIdx][layerIdx].**entropy_enabled_flag** indicates whether a specific layer has data.

surfaces[planeIdx][levelIdx][layerIdx].**rle_only_flag** indicates whether a specific layer has only RLE or also Prefix Coding.

temporal_surfaces[planeIdx].**entropy_enabled_flag** indicates whether the temporal surface related to a specific plane has data.

temporal_surfaces[planeIdx].**rle_only_flag** indicates whether a temporal surface related to a specific plane has only RLE or also Prefix Coding.

7.4.3.6 Data block unit encoded tiled data semantics

surfaces[planeIdx][levelIdx][layerIdx].**rle_only_flag** indicates whether a specific layer had only RLE or also Prefix Coding.

surfaces[planeIdx][levelIdx][layerIdx].**tiles**[tileIdx].**entropy_enabled_flag** indicates whether a specific tile belonging to a specific surface of a specific layer has data.

temporal_surfaces[planeIdx].**rle_only_flag** indicates whether a temporal surface related to a specific plane has only RLE or also Prefix Coding.

temporal_surfaces[planeIdx].**tiles**[tileIdx].**entropy_enabled** indicates whether a specific tile belonging to a specific temporal surface has data.

entropy_enabled_per_tile_compressed_data_rle contains the RLE-encoded signalling for each picture tile and should be decoded based on 9.3.5.

compressed_size_per_tile_prefix specifies the compressed size of the encoded data for each picture tile and should be decoded based on 9.2.3.

7.4.3.7 Data block unit surface semantics

surface.size specifies the size of the entropy encoded data (see 9.1).

surface.data_rle contains the entropy encoded RLE only data when the related surface.rle_only_flag value is true (see 9.1).

surface.data_prefix_coding contains the entropy encoded Prefix Coding data when the related surface.rle_only_flag value is false (see 9.1).

7.4.3.8 Data block unit additional info semantics

additional_info_type contains additional information in the form of either SEI messages (as specified in Annex D) or VUI messages (as specified in Annex E).

payload_type specifies the payload type of an SEI message (as specified in Annex D).

7.4.3.9 Data block unit filler semantics

filler byte shall be a byte equal to 0xAA.

8 Decoding process

8.1 General decoding process

The input to this process is an LCEVC bitstream that contains an enhancement layer consisting of up to two sub-layers.

The decoding process is specified such that all decoders that conform to a specified profile and level will produce numerically identical cropped decoded output pictures when invoking the decoding process associated with that profile for a bitstream conforming to that profile and level. Any decoding process that produces identical cropped decoded output pictures to those produced by the process described herein (with the correct output order or output timing, as specified) conforms to the decoding process requirements of this document. The outputs of this process are:

— the enhancement residuals planes (sub-layer 1 residual planes) to be added to the preliminary pictures 1, obtained from the base decoder reconstructed pictures; and

— the enhancement residuals planes (sub-layer 2 residual planes) to be added to the preliminary output pictures resulting from upscaling and modifying via predicted residuals the combination of the preliminary pictures 1 and the sub-layer 1 residual planes.

The decoding process operates on data blocks and it is described as follows:

1) The decoding of payload data block units is specified in 8.2.

2) The decoding process for the picture is specified in 8.3 using syntax elements in Clause 7.

3) The decoding process for temporal prediction is specified in 8.4.

4) The decoding process for the dequantization process is specified in 8.5.

5) The decoding process for the transform is specified in 8.6.

6) The decoding process for the upscaler is specified in 8.7.

7) The decoding process for the predicted residual is specified in 8.7.4.

8) The decoding process for the residual reconstruction is specified in 8.8.

9) The decoding process for the L-1 filter is specified in 8.9.

10) The decoding process for the base encoding data extraction is specified in 8.10.

11) The decoding process for the dithering filter is specified in 8.11.

8.2 Payload data block unit process

Input to this process is the enhancement layer bitstream. The enhancement layer bitstream is encapsulated in NAL units specified in 7.3.2. A NAL unit is used to synchronize the enhancement layer information with the base decoded information.

The bitstream is organized in NAL units, with each NAL unit including one or more data blocks. For each data block, the process_block( ) syntax structure (7.3.3) is used to parse a block header (and only the block header) and invokes the relevant process_block( ) syntax element based upon the information in the block header. A NAL unit which includes encoded data comprises at least two data blocks, a picture configuration data block and an encoded (tiled) data block. The possible different data blocks are indicated in Table 19.

A sequence configuration data block should occur at least once at the beginning of the bitstream. A global configuration data block should occur at least for every IDR picture. Every encoded (tiled) data block should be preceded by a picture configuration data block. When present in a NAL unit, a global configuration data block should be the first data block in the NAL unit.

8.3 Picture enhancement decoding process

8.3.1 General enhancement decoding process

Input of this process is the portion of the bitstream following the headers decoding process described in 7.3.3. Outputs are the entropy encoded transform coefficients belonging to the picture enhancement being decoded. Every encoded picture is preceded by the picture configuration payload described in 7.3.6 and 7.4.3.4.

8.3.2 Decoding process for picture enhancement encoded data (payload_encoded_data)

Syntax of this process is described in 7.3.7. Inputs to this process are:

— A variable nPlanes containing the number of plane (7.4.3.2 depending on the value of the variable processed_planes_type_flag),

— A variable nLayers (7.4.3.2 depending on the value of transform_type),

— A constant nLevel equal to 2, since 2 enhancement sub-layers are processed.

Output of this process is the (nPlanes)x(nLevels)x(nLayers) array surfaces with elements surfaces[nPlanes][nLevels][nLayers] and, if temporal_signalling_present_flag is equal to 1, an additional temporal surface of a size nPlanes with elements temporal_surface[nPlanes].

— variable nPlanes is derived as follows:

```
if (processed_planes_type_flag == 0)
    nPlanes = 1
else
    nPlanes = 3
```

— variable nLayers is derived as follows:

```
if (transform_type == 0)
    nLayers = 4
else
    nLayers = 16
```

The encoded data is organized in chunks. The total number of chunks is calculated as follows:

total_chunk_count = nPlanes * nLevels * nLayers * (no_enhancement_bit_flag == 0) + nPlanes * (temporal_signalling_present_flag == 1) (12)

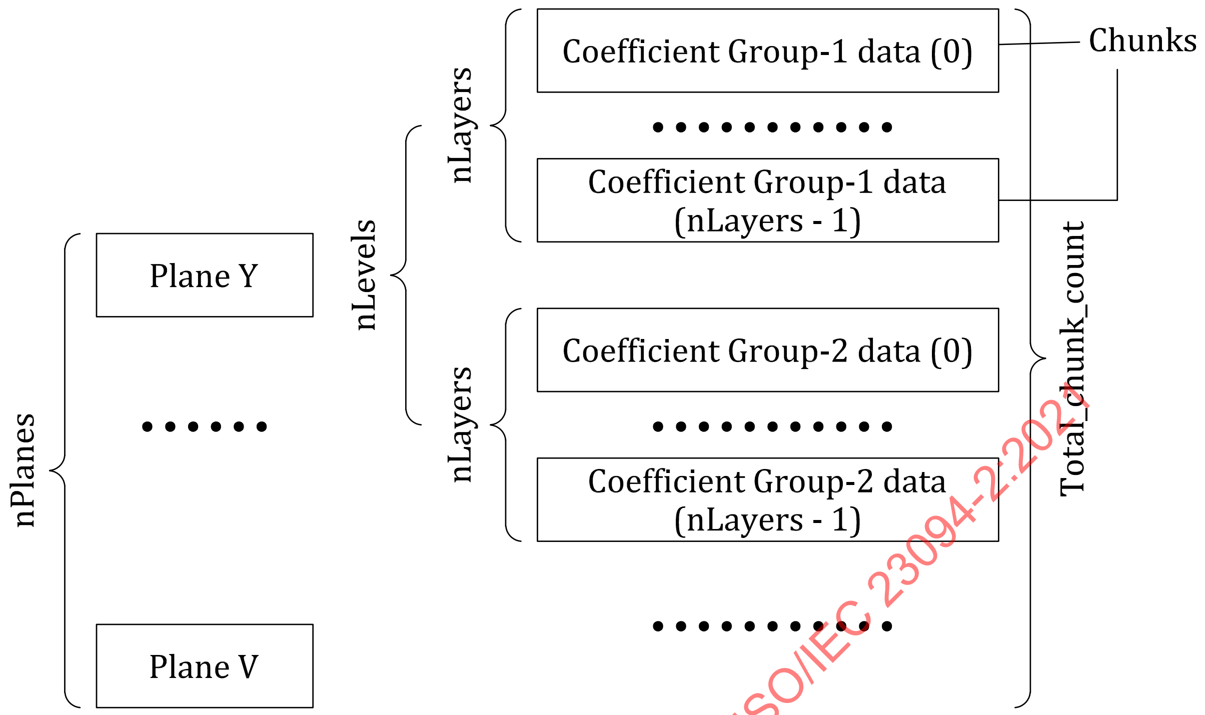

Coefficient Group-1 data (0)
Chunks
nLayers
Coefficient Group-1 data (nLayers - 1)
Plane Y
nLevels
Coefficient Group-2 data (0)
Total_chunk_count
nPlanes
nLayers
Coefficient Group-2 data (nLayers - 1)
Plane V

Figure 4 — Encoded enhancement picture data chunk structure

The enhancement picture data chunks are hierarchically organized as shown in Figure 4. For each plane, up to 2 enhancement sub-layers are extracted. For each enhancement sub-layer, up to 16 coefficient groups of transform coefficients can be extracted. Additionally, if temporal_signalling_present_flag is equal to 1, an additional chunk with temporal data for enhancement sub-layer 2 is extracted.

The variable levelIdx 1 refers to enhancement sub-layer 1 and levelIdx 2 refers to enhancement sub-layer 2.

The decoding process is the following:

The following flags surfaces[planeIdx][levelIdx][layerIdx].entropy_enabled_flag, surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag, temporal_surfaces[planeIdx].entropy_enabled_flag and temporal_surfaces[planeIdx].rle_only_flag are derived as follows:

```
shift_size = -1
for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
    if (no_enhancement_bit_flag == 0) {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
            for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                if (shift_size < 0) {
                    data = read_byte(bitstream)
                    shift_size = 8 – 1
                }
                surfaces[planeIdx][levelIdx][layerIdx].entropy_enabled_flag = ((data >> shift_size)
                & 0x1)
                surfaces[planeIdx][levelIdx][layerIdx]._rle_only_flag = ((data >> shift_size - 1))
                & 0x1
                shift_size -= 2
            }
        }
    } else {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
        for (layerIdx = 0; layer < nLayers; ++layerIdx)
```

```
            surfaces[planeIdx][levelIdx][layerIdx].entropy_enabled_flag = 0
        }
        if (temporal_signalling_present_flag == 1) {
          if (shift_size < 0) {
            data = read_byte(bitstream)
            shift_size = 8 – 1
          }
          temporal_surfaces[planeIdx].entropy_enabled_flag = ((data >> shift_size) & 0x1)
          temporal_surfaces[planeIdx].rle_only_flag = ((data >> (shift_size – 1)) & 0x1)
          shift_size −= 2
        }
      }
```

According to the values of the entropy_enabled_flag and rle_only_flag fields the content for the surfaces[planeIdx][levelIdx][layerIdx].data_rle indicating the beginning of the RLE only (or surfaces[planeIdx][levelIdx][layerIdx].data_prefix_coding indicating the beginning of the Prefix Coding and RLE) encoded transform coefficients related to the specific chunk of data are derived as follows:

```
for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
    for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
        for (layerIdx = 0; layer < nLayers; ++layerIdx) {
            if (surfaces[planeIdx][levelIdx][layerIdx].entropy_enabled_flag) {
                if (surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag) {
                    multibyte = read_multibyte(bitstream)
                    surfaces[planeIdx][levelIdx][layerIdx].size = multibyte
                    surfaces[planeIdx][levelIdx][layerIdx].data_rle = bytestream_current(bitstream)
                } else {
                    data = read_byte(bitstream)
                    multibyte = read_multibyte(bitstream)
                    surfaces[planeIdx][levelIdx][layerIdx].size = multibyte
                    surfaces[planeIdx][levelIdx][layerIdx].data_prefix_coding =
                        bytestream_current(bitstream)
                    bytestream_seek(bitstream, surfaces[planeIdx][levelIdx][layerIdx].size)
                }
            }
        }
    }
```

If temporal_signalling_present_flag is equal to 1, according to the values of the entropy_enabled_flag and rle_only_flag fields the content for the temporal_surfaces[planeIdx].data_rle indicating the beginning of the RLE only (or temporal_surfaces[planeIdx].data_prefix_coding indicating the beginning of the Prefix Coding and RLE) encoded temporal signal coefficient group related to the specific chunk of data are derived as follows:

```
  if (temporal_signalling_present_flag == 1) {
      if (temporal_surfaces[planeIdx].entropy_enabled_flag) {
          if (temporal_surfaces[planeIdx].rle_only_flag) {
              multibyte = read_multibyte(bitstream)
              temporal_surfaces[planeIdx].size = multibyte
              temporal_surfaces[planeIdx].data_rle = bytestream_current(bitstream)
          } else {
              data = read_byte(bitstream)
              multibyte = read_multibyte(bitstream)
              temporal_surfaces[planeIdx].size = multibyte
              temporal_surfaces[planeIdx].data_prefix_coding = bytestream_current(bitstream)
              bytestream_seek(bitstream, temporal_surfaces[planeIdx].size)
          }
      }
```

```
        }
    }
```

The entropy encoded transformed coefficients contained in the block of bytes of length surfaces[planeIdx][levelIdx][layerIdx].size and starting from surfaces[planeIdx][levelIdx][layerIdx].data_rle (or surfaces[planeIdx][levelIdx][layerIdx].data_prefix_coding) address are then passed to the entropy decoding process described in 9.1.1.

If temporal_signalling_present_flag is set to 1, the group of entropy encoded temporal signal values contained in the block of bytes of length temporal_surfaces[planeIdx].size and starting from temporal_surfaces[planeIdx].data_rle (or temporal_surfaces[planeIdx].data_prefix_coding) address are then passed to the entropy decoding process described in 9.1.2.

8.3.3 Decoding process for picture enhancement encoded tiled data (payload_encoded_tiled_data)

Syntax of this process is described in 7.3.8. Inputs to this process are:

— A variable nPlanes containing the number of planes (7.4.3.2 depending on the value of the variable processed_planes_type_flag),

— A variable nLayers (7.4.3.2 depending on the value of transform_type),

— A constant nLevels equal to 2, since there are 2 enhancement sub-layers,

— A variable nTilesL2, which equals to Ceil(PictureWidth / TileWidth)xCeil(PictureHeight / TileHeight) and refers to the number of tiles in L-2.

— A variable nTilesL1, which equals to:

  — (a) nTilesL2 if the variable scaling_mode_level2 is equal to 0,

  — (b) Ceil(Ceil(PictureWidth / 2) / TileWidth)xCeil(PictureHeight) / TileHeight) if the variable scaling_mode_level2 is equal to 1, and

  — (c) Ceil(Ceil(PictureWidth / 2) / TileWidth)xCeil(Ceil(PictureHeight / 2) / TileHeight) if the variable scaling_mode_level2 is equal to 2,

  and refers to the number of tiles in L-1.

Output of this process is the (nPlanes)x(nLevels)x(nLayer) array surfaces with elements surfaces[nPlanes][nLevels][nLayer] and, if temporal_signalling_present_flag is set to 1, an additional temporal surface of a size nPlanes with elements temporal_surface[nPlanes].

— variable nPlanes is derived as follows:

```
if (processed_planes_type_flag == 0)
    nPlanes = 1
else
    nPlanes = 3
```

— variable nLayers is derived as follows:

```
if (transform_type == 0)
    nLayers = 4
else
    nLayers = 16
```

The encoded data is organized in chunks. The total number of chunks is calculated as following:

total_chunk_count = nPlanes * nLayers * (nTilesL1 + nTilesL2) * (no_enhancement_bit_flag == 0) + nPlanes * nTilesL2 * (temporal_signalling_present_flag == 1) (13)

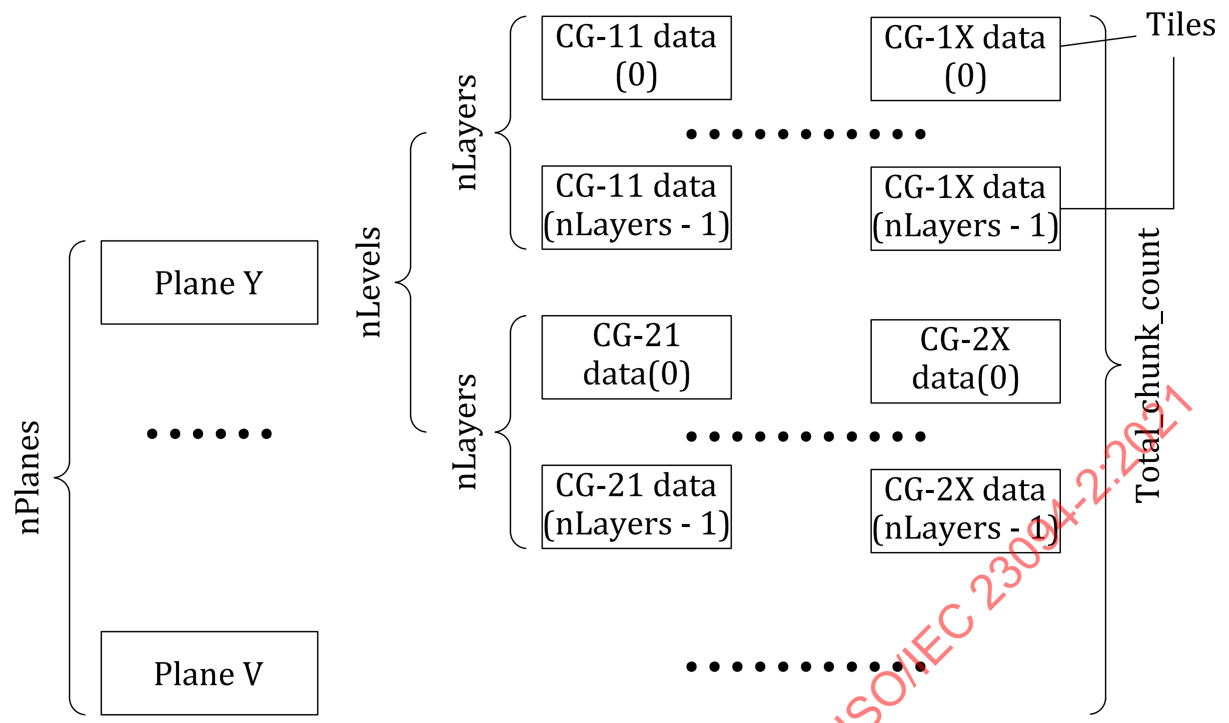


Figure 5 — Encoded chunk tile structure

The enhancement picture data chunks are hierarchically organized as shown in Figure 5. For each plane up to 2 layers of enhancement sub-layers are extracted. For each sub-layer of enhancement, up to 16 coefficient groups of transform coefficients can be extracted. Additionally, if temporal_signalling_present_flag is set to 1, an additional chunk with temporal data for enhancement sub-layer 2 is extracted.

The variable levelIdx 1 refers to enhancement sub-layer 1 and levelIdx 2 refers to enhancement sub-layer 2.

The decoding process is as follows:

The following flags surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag and, if temporal_signalling_present_flag is set to 1, temporal_surfaces[planeIdx].rle_only_flag are derived as follows:

```
shift_size = –1
for (planeIdx = 0; planeIdx < nPlanes; ++ planeIdx) {
    if (no_enhancement_bit_flag == 0) {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
            for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                if (shift_size < 0) {
                    data = read_byte(bitstream)
                    shift_size = 8 – 1
                }
                surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag = ((data >> (shift_size – 1)) &
                0x1)
                shift_size –= 1
            }
        }
    }
    if (temporal_signalling_present_flag == 1) {
        if (shift_size < 0) {
            data = read_byte(bitstream)
            shift_size = 8 – 1
            }
            temporal_surfaces[planeIdx].rle_only_flag = ((data >> (shift_size − 1)) & 0x1)
            shift_size −= 1
        }
    }
```

The following flags surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag and, if temporal_signalling_present_flag is set to 1, temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag are derived as follows:

```
if (compression_type_entropy_enabled_per_tile_flag == 0) {
    shift_size = −1
    for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
        if (no_enhancement_bit_flag == 0) {
            for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
                if (levelIdx == 1)
                    nTiles = nTilesL1
                else
                    nTiles = nTilesL2
                for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                    for (tileIdx=0; tileIdx < nTiles; tileIdx ++) {
                        if (shift_size < 0) {
                            data = read_byte(bitstream)
                            shift_size = 8 – 1
                        }
                        surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx]. entropy_enabled_flag =
                            ((data >> (shift_size – 1)) & 0x1)
                        shift_size −= 1
                    }
                }
            }
        } else {
            for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
                if (levelIdx == 1)
                    nTiles = nTilesL1
                else
                    nTiles = nTilesL2
                for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                    for (tileIdx = 0; tileIdx < nTiles; tileIdx++)
                        surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag = 0
                }
            }
        }
        if (temporal_signalling_present_flag == 1) {
            for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++) {
                if (shift_size < 0) {
                    data = read_byte(bitstream)
                    shift_size = 8 – 1
                }
                temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag =
                    ((data >> (shift_size – 1)) & 0x1)
                shift_size = 1
            }
        }
    }
} else {
    RLE decoding process as defined in 9.3.5.
}
```

According to the value of the entropy_enabled_flag and rle_only_flag fields, the content for the surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_rle indicating the beginning of the RLE only (or surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_prefix_coding indicating the beginning of the Prefix Coding and RLE) encoded coefficients related to the specific chunk of data is derived as follows:

```
if (compression_type_size_per_tile == 0) {
    for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
            if (levelIdx == 1)
                nTiles = nTilesL1
            else
                nTiles = nTilesL2
            for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                for (tileIdx = 0; tileIdx < nTiles; tileIdx++) {
                    if (surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled
                      _flag) {
                        if (surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag) {
                            multibyte = read_multibyte(bitstream)
                            surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size = multibyte
                            surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx]data_rle =
                                bytestream_current(bitstream)
                        } else {
                            data = read_byte(bitstream)
                            multibyte = read_multibyte(bitstream)
                            surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size = multibyte
                            surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_prefix_
                                coding =
                                bytestream_current(bitstream)
                            bytestream_seek(bitstream,
                                surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size)
                        }
                    }
                }
            }
        }
```

If temporal_signalling_present_flag is set to 1, according to the value of the entropy_enabled_flag and rle_only_flag fields, the content for the temporal_surfaces[planeIdx].tiles[tileIdx].data_rle indicating the beginning of the RLE only (or temporal_surfaces[planeIdx].tiles[tileIdx].data_prefix_coding Prefix Coding and RLE) encoded temporal signal coefficient group related to the specific chunk of data are derived as follows:

```
if (temporal_signalling_present_flag == 1) {
    for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++) {
        if (temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag) {
            if (temporal_surfaces[planeIdx].rle_only_flag) {
                multibyte = read_multibyte(bitstream)
                temporal_surfaces[planeIdx].tiles[tileIdx].size = multibyte
                temporal_surfaces[planeIdx].tiles[tileIdx].data_rle =
                    bytestream_current(bitstream)
            } else {
                data = read_byte(bitstream)
                multibyte = read_multibyte(bitstream)
                temporal_surfaces[planeIdx].tiles[tileIdx].size = multibyte
                temporal_surfaces[planeIdx].tiles[tileIdx].data_prefix_coding =
                    bytestream_current(bitstream)
                bytestream_seek(bitstream, temporal_surfaces[planeIdx].
                    tiles[tileIdx].size)
```

```
                        }
                    }
                }
            }
        }
    } else {
        for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
            for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
                if (levelIdx == 1)
                    nTiles = nTilesL1
                else
                    nTiles = nTilesL2
                for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                    for (tileIdx = 0; tileIdx < nTiles; tileIdx ++) {
                        if (surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag) {
                            if (surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag) {
                                Prefix Coding decoding process as defined in 9.2.3 to fill
                                        surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size
                                surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_rle =
                                        bytestream_current(bitstream)
                            } else {
                                Prefix Coding decoding process as defined in 9.2.3 to fill
                                        surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size
                                surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_prefix_
                                    coding =
                                        bytestream_current(bitstream)
                                bytestream_seek(bitstream,surfaces[planeIdx][levelIdx][layerIdx].
                                        tiles[tileIdx].size)
                            }
                        }
                    }
                }
            }
        if (temporal_signalling_present_flag == 1) {
            for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++) {
                if (temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag) {
                    if (temporal_surfaces[planeIdx].rle_only_flag) {
                        Prefix Coding decoding process as defined in 9.2.3 to fill
                            temporal_surfaces[planeIdx].tiles[tileIdx].size
                        temporal_surfaces[planeIdx].tiles[tileIdx].data_rle =
                            bytestream_current(bitstream)
                    } else {
                        Prefix Coding decoding process as defined in 9.2.3 to fill
                            temporal_surfaces[planeIdx].tiles[tileIdx].size
                        temporal_surfaces[planeIdx].tiles[tileIdx].data_prefix_coding =
                            bytestream_current(bitstream)
                        bytestream_seek(bitstream, temporal_surfaces[planeIdx].tiles[tileIdx].size)
                    }
                }
            }
        }
    }
}
```

The entropy encoded transformed coefficients contained in the block of bytes of length surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size and starting from surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_rle (or surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_prefix_coding) address are then passed to the entropy decoding process described in 9.1.1.

If temporal_signalling_enabled_flag is set to 1, the group of entropy encoded temporal signal values contained in the block of bytes of length temporal_surfaces[planeIdx].tiles[tileIdx].size and starting from temporal_surfaces[planeIdx].tiles[tileIdx].data_rle (or temporal_surfaces[planeIdx].tiles[tileIdx].data_prefix_coding) address are then passed to the entropy decoding process described in 9.1.2.

8.3.4 Decoding process for enhancement sub-layer 1 (L-1) encoded data

8.3.4.1 Derivation of dimensions of L-1 picture

The result of this process is the L-1 enhancement residual surface to be added to the preliminary intermediate picture. The L-1 dimensions of the residuals surface are the same as the preliminary intermediate picture and they are derived as follows:

— If scaling_mode_level2 (specified in 7.4.3.3) is equal to 0:

  The L-1 dimensions should be the same as the L-2 dimensions derived from resolution_type (specified in 7.4.3.3).

— If scaling_mode_level2 (specified in 7.4.3.3) is equal to 1:

  The L-1 height should be the same as the L-2 height as derived from resolution_type (specified in 7.4.3.3), whereas the L-1 width shall be computed by halving the L-2 width as derived from resolution_type (specified in 7.4.3.3).

— If scaling_mode_level2 (specified in 7.4.3.3) is equal to 2:

  The L-2 dimensions should be computed by halving the L-1 dimensions as derived from resolution_type (specified in 7.4.3.3).

8.3.4.2 General decoding process for an L-1 encoded data block

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 nTbS specifying the size of the current transform block derived in 7.4.3.2 from the value of variable transform_type (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1),

— an array TransformCoeffQ of a size (nTbS)x(nTbS) specifying L-1 entropy decoded quantized transform coefficients,

— an array recL1BaseSamples of a size (nTbS)x(nTbS) specifying the preliminary intermediate picture reconstructed samples of the current block resulting from the process described in 8.10,

— stepWidth value derived in 7.4.3.4 from the value of variable step_width_sublayer1,

— a variable IdxPlanes specifying to which plane the transform coefficients belong,

— a variable userDataEnabled derived from the value of variable user_data_enabled.

Output of this process is the (nTbS)x(nTbS) array of the residual resL1FilteredResiduals with elements resL1FilteredResiduals[x][y].

The sample location (xTbP, yTbP) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture is derived as follows:

(xTbP, yTbP) = (IdxPlanes == 0) ? (xTb0, yTb0) : (xTb0 >> ShiftWidthC, yTb0 >> ShiftHeightC) (14)

P can be related to either luma or chroma plane depending on which plane the transform coefficients belong to. ShiftWidthC and ShiftHeightC are specified in 6.2.

If no_enhancement_bit_flag is equal to 0, then the following ordered steps apply:

1) If nTbS is equal to 4, TransformCoeffQ(1)(1) is shifted to the right either by two bits (>> 2) if the variable userDataEnabled is set to 1, or by six bits (>> 6) if userDataEnabled is set to 2. And in addition, if the last bit of TransformCoeffQ(1)(1) is set to 0, TransformCoeffQ(1)(1) is shifted to the right by one bit (>> 1); otherwise (last bit of TransformCoeffQ(1)(1) is set to 1), TransformCoeffQ(1)(1) is shifted to the right by one bit (>> 1) and the value of TransformCoeffQ(1)(1) is set to be negative.

2) If nTbSs is equal to 4, TransformCoeffQ(1)(1) is shifted to the right either by two bits (>> 2) if the variable userDataEnabled is set to 1, or by six bits (>> 6) if userDataEnabled is set to 2. And in addition, if the last bit of TransformCoeffQ(1)(1) is set to 0, TransformCoeffQ(1)(1) is shifted to the right by one bit (>> 1); otherwise (last bit of TransformCoeffQ(1)(1) is set to 1), TransformCoeffQ(1)(1) is shifted to the right by one bit (>> 1) and the value of TransformCoeffQ(1)(1) is set to be negative.

3) The dequantization process as specified in 8.5 is invoked with the transform size set equal to nTbS, the array TransformCoeffQ of a size (nTbS)x(nTbS), and the variable stepWidth as inputs, and the output is an (nTbS)x(nTbS) array dequantCoeff.

4) The transformation process as specified in 8.6 is invoked with the luma location (xTbX, yTbX), the transform size set equal to nTbS, the array dequantCoeff of size (nTbS)x(nTbS) as inputs, and the output is an (nTbS)x(nTbS) array resL1Residuals.

5) The L1 filter process as specified in 8.9 is invoked with the luma location (xTbX, yTbX), the array resL1Residuals of a size (nTbS)x(nTbS) as inputs, and the output is an (nTbS)x(nTbS) array resL1FilteredResiduals.

NOTE When userDataEnabled is set to 1, particular care is given to the value of stepWidth as defined, in order to prevent saturation of the TransformCoeffQ values beyond the constraints set by the RLE decoding process, as described in 9.3, which limits the bit length of the RLE symbols to 13 bits.

If userDataEnabled is set to 1, and base_depth_type is set to 3 (i.e., the base bit depth is 14 bits per sample), then stepWidth should be greater or equal to 8.

If userDataEnabled is set to 2, and base_depth_type is set to 1 (i.e., the base bit depth is 10 bits per sample), then stepWidth should be greater or equal to 8.

If userDataEnabled is set to 2, and base_depth_type is set to 2 (i.e., the base bit depth is 12 bits per sample), then stepWidth should be greater or equal to 32.

If userDataEnabled is set to 2, and base_depth_type is set to 3 (i.e., the base bit depth is 14 bits per sample), then stepWidth should be greater or equal to 128.

If no_enhancement_bit_flag is equal to 1, the array resL1FilteredResiduals of size (nTbS)x(nTbS) is set to contain only zeros.

The picture reconstruction process for each plane as specified in 8.8.2 is invoked with the transform block location (xTb0, yTb0), the transform block size nTbS, the variable IdxPlanes, the (nTbS)x(nTbS) array resL1FilteredResiduals, and the (nTbS)x(nTbS) array recL1BaseSamples as inputs.

8.3.5 Decoding process for enhancement sub-layer 2 (L-2) encoded data

8.3.5.1 Derivation of dimensions of L-2 picture

The result of this process is the L-2 enhancement residuals plane to be added to the upscaled L-1 enhanced reconstructed picture. The dimensions of the residuals plane are the same as the value derived by the variable resolution_type described in 7.4.3.3.

8.3.5.2 General decoding process for an L-2 encoded data block

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 nTbS specifying the size of the current transform block derived in 7.4.3.2 from the value of variable transform_type (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1),

— a variable temporal_enabled_flag as derived in 7.4.3.3 and a variable temporal_refresh_bit_flag as derived in 7.4.3.4, a variable temporal_signalling_present_flag as derived in 7.4.3.4 and temporal_step_width_modifier as specified in 7.3.5,

— an array recL2ModifiedUpsampledSamples of a size (nTbS)x(nTbS) specifying the upsampled reconstructed samples resulting from process specified in 8.8.2 of the current block,

— an array TransformCoeffQ of a size (nTbS)x(nTbS) specifying L-2 entropy decoded quantized transform coefficient,

— if variable temporal_signalling_present_flag is equal to 1 and temporal_tile_intra_signalling_enabled_flag is equal to 1, a variable TransformTempSig corresponding to the value in TempSigSurface (9.3.4) at the position (xTb0 >> nTbs, yTb0 >> nTbs); and if in addition temporal_tile_intra_signalling_enabled_flag is set to 1, a variable TileTempSig corresponding to the value in TempSigSurface (9.3.4) at the position ((xTb0 % 32) * 32, (yTb0 % 32) * 32),

— stepWidth value derived in 7.4.3.4 from the value of variable step_width_sublayer2,

— a variable IdxPlanes specifying to which plane the transform coefficients belong.

Output of this process is the (nTbS)x(nTbS) array of L-2 residuals resL2Residuals with elements resL2Residuals[x][y].

The sample location (xTbP, yTbP) specifying the top-left sample of the current transform block relative to the top-left sample of the current picture is derived as follows:

(xTbP, yTbP) = (IdxPlanes == 0) ? (xTb0, yTb0) : (xTb0 >> ShiftWidthC, yTb0 >> ShiftHeightC) (15)

P can be related to either luma or chroma plane depending on to which plane the transform coefficients belong. Where ShiftWidthC and ShiftHeightC are specified in 6.2.

If no_enhancement_bit_flag is set to 0, then the following ordered steps apply:

1) If variable temporal_enabled_flag is equal to 1 and temporal_refresh_bit_flag is equal to 0, (7.4.3.4) the temporal prediction process as specified in 8.4 is invoked with the luma location (xTbY, yTbY), the transform size set equal to nTbS, a variable TransformTempSig and a variable TileTempSig as inputs and the output is an array tempPredL2Residuals of a size (nTbS)x(nTbS).

   If variable temporal_enabled_flag is equal to 1 and temporal_refresh_bit_flag is equal to 1 (7.4.3.4), the array tempPredL2Residuals of a size (nTbS)x(nTbS) is set to contain only zeros.

2) If variable temporal_enabled_flag is equal to 1 and temporal_refresh_bit_flag is equal to 0 (7.4.3.4) and TransformTempSig is equal to 0, the variable stepWidth is modified to Floor(stepWidth * (1 – (Clip3(0, 0.5, (temporal_step_width_modifier / 255))))).

3) The dequantization process as specified in 8.5 is invoked with the transform size set equal to nTbS, the array TransformCoeffQ of a size (nTbS)x(nTbS), and the variable stepWidth as inputs, and the output is an (nTbS)x(nTbS) array dequantCoeff.

4) The transformation process as specified in 8.6 is invoked with the luma location (xTbY, yTbY), the transform size set equal to nTbS, the array dequantCoeff of a size (nTbS)x(nTbS) as inputs, and the output is an (nTbS)x(nTbS) array resL2Residuals.

5) If variable temporal_enabled_flag is equal to 1, the array of tempPredL2Residuals of a size (nTbS)x(nTbS) is added to the (nTbS)x(nTbS) array resL2Residuals and resL2Residuals array is stored to the temporalBuffer at the luma location (xTbY, yTbY).

If no_enhancement_bit_flag is set to 1, then the following ordered steps apply:

1) If variable temporal_enabled_flag is equal to 1, temporal_refresh_bit_flag is equal to 0 and variable temporal_signalling_present_flag is equal to 1 (7.4.3.4), the temporal prediction process as specified in 8.4 is invoked with the luma location (xTbY, yTbY), the transform size set equal to nTbS, a variable TransformTempSig and a variable TileTempSig as inputs and the output is an array tempPredL2Residuals of a size (nTbS)x(nTbS).

   If variable temporal_enabled_flag is equal to 1, temporal_refresh_bit_flag is equal to 0 and variable temporal_signalling_present_flag is equal to 0 (7.4.3.4), the temporal prediction process as specified in 8.4 is invoked with the luma location (xTbY, yTbY), the transform size set equal to nTbS, a variable TransformTempSig set equal to 0 and a variable TileTempSig set equal to 0 as inputs and the output is an array tempPredL2Residuals of a size (nTbS)x(nTbS).

   If variable temporal_enabled_flag is equal to 1 and temporal_refresh_bit_flag is equal to 1 (7.4.3.4), the array tempPredL2Residuals of a size (nTbS)x(nTbS) is set to contain only zeros.

2) If variable temporal_enabled_flag is equal to 1, the array of tempPredL2Residuals of a size (nTbS)x(nTbS) is stored in the (nTbS)x(nTbS) array resL2Residuals and resL2Residuals array is stored to the temporalBuffer at the luma location (xTbY, yTbY). Else, the array resL2Residuals of a size (nTbS)x(nTbS) is set to contain only zeros.

The picture reconstruction process for each plane as specified in 8.8.2 is invoked with the transform block location (xTb0, yTb0), the transform block size nTbS, the variable IdxPlanes, the (nTbS)x(nTbS) array resL2Residuals, and the (xTbY)x(yTbY) recL2ModifiedUpsampledSamples as inputs.

8.4 Decoding process for the temporal prediction

8.4.1 General decoding process for temporal prediction

Inputs to this process are:

— a location (xTbP, yTbP) specifying the top-left sample of the current luma or chroma transform block relative to the top-left luma or chroma sample of the current picture. P can be related to either luma or chroma plane depending to which plane the transform coefficients belong,

— a variable nTbS specifying the size of the current transform block (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1),

— a variable TransformTempSig corresponding to the value in TempSigSurface (9.3.4) at the position (xTbP >> nTbS, yTbP >> nTbS),

— a variable TileTempSig corresponding to the value in TempSigSurface (9.3.4) at the position ((xTbP % 32) * 32, (yTbP % 32) * 32).

Output of this process is the (nTbS)x(nTbS) array of the tempPredL2Residuals with elements tempPredL2Residuals[x][y].

The following ordered steps apply:

1) If variable temporal_tile_intra_signalling is equal to 1 and xTbP >> 5 is equal to 0 and yTbP >> 5 is equal to 0 and TileTempSig is equal to 1, tiled temporal refresh process as specified in 8.4.2 is invoked with the location (xTbP, yTbP).

2) If variable TransformTempSig is equal to 0, then tempPredL2Residuals[x][y] = temporalBuffer[xTbP + x][yTbP + y] where x and y are in the range[0, (nTbS − 1)]. Otherwise, tempPredL2Residuals[x][y] are all set to 0.

8.4.2 Tiled temporal refresh

Input to this process is:

— a location (xTbP, yTbP) specifying the top-left sample of the current luma or chroma transform block relative to the top-left luma or chroma sample of the current picture. P can be related to either luma or chroma plane depending to which plane the transform coefficients belong.

Output of this process is that the samples of the area of the size 32x32 of tempPredL2Residuals at the location (xTbP, yTbP) are set to zero.

8.5 Decoding process for the dequantization

8.5.1 Decoding process for the dequantization overview

Every group of transform coefficient passed to this process belongs to a specific plane and enhancement sub-layer. They have been scaled using a uniform quantizer with deadzone. The quantizer can use a non-centred dequantization offset.

8.5.2 Scaling process for transform coefficients

Inputs to this process are:

— a variable nTbS specifying the size of the current transform block (nTbS = 2 if transform_type is equal to zero and nTbS = 4 if transform_type is equal to 1),

— an array TransformCoeffQ of size (nTbS)x(nTbS) containing entropy decoded quantized transform coefficient,

— a variable stepWidth specifying the step width value parameter,

— a variable levelIdx specifying the index of the enhancement sub-layer (with levelIdx = 1 for enhancement sub-layer 1 and levelIdx = 2 for enhancement sub-layer 2),

— a variable dQuantOffset specifying the dequantization offset (derived from process of 7.4.3.4 and variable dequant_offset),

— if quant_matrix_mode is different from 0, an array QmCoeff0 of size 1 x nTbS^2 (equal to array variable qm_coefficient_0 defined in 7.4.3.4) and further, if quant_matrix_mode is equal to 4, an array QmCoeff1 of size 1 x nTbS^2 (equal to array qm_coefficient_1 7.4.3.4),

— if nTbS == 2, an array QuantScalerDDBuffer of size (3 * nTbS)x(nTbS) containing the scaling parameters array used in the previous picture,

— if nTbS == 4, an array QuantScalerDDSBuffer of size (3 * nTbS)x(nTbS) containing the scaling parameters array used in the previous picture.

Output of this process is the (nTbS)x(nTbS) array d of dequantized transform coefficients with elements d[x][y] and the updated array QuantMatrixBuffer.

For the derivation of the scaled transform coefficients d[x][y] with x = 0...nTbS − 1, y = 0...nTbS − 1, and given matrix qm[x][y] as specified in 8.5.4, the following formula is used:

d[x][y] = TransformCoeffQ[x][y] * ((qm[x + (levelIdxSwap * nTbS)][y] + stepWidthModifier[x][y]) + appliedOffset[x][y]) (16)

8.5.3 Derivation of dequantization offset and stepwidth modifier

The variables appliedOffset[x][y] and stepWidthModifier[x][y] are derived as follows:

```
if (dequant_offset_signalled_flag == 0) {
    stepWidthModifier[x][y] = ((((Floor(−Cconst * Ln (qm[x + (levelIdxSwap * nTbS)][y]))) + Dconst) *
     (qm[x + (levelIdxSwap * nTbS)][y]2))) / 32768) >> 16
    if (TransformCoeffQ[x][y] < 0)
        appliedOffset[x][y] = (−1 * (−deadZoneWidthOffset[x][y]))
    else if (TransformCoeffQ[x][y] > 0)
        appliedOffset[x][y] = −deadZoneWidthOffset[x][y]
    else
        appliedOffset[x][y] = 0
} else if (dequant_offset_signalled_flag == 1) && (dequant_offset_mode_flag ==1) {
    stepWidthModifier[x][y] = 0
    if (TransformCoeffQ[x][y] < 0)
        appliedOffset = (−1 * (dQuantOffsetActual[x][y] − deadZoneWidthOffset[x][y]))
    else if (TransformCoeffQ[x][y] > 0)
        appliedOffset[x][y] = dQuantOffsetActual[x][y] − deadZoneWidthOffset[x][y]
    else
        appliedOffset[x][y] = 0}
} else if (dequant_offset_signalled_flag == 1) && (dequant_offset_mode_flag == 0) {
    stepWidthModifier[x][y] = (Floor((dQuantOffsetActual[x][y]) * (qm[x + (levelIdxSwap
       * nTbS)][y])) / 32768)
    if (TransformCoeffQ[x][y] < 0)
        appliedOffset = (−1 * (−deadZoneWidthOffset[x][y]))
    else if (TransformCoeffQ[x][y] > 0)
        appliedOffset[x][y] = −deadZoneWidthOffset[x][y]
    else
        appliedOffset[x][y] = 0
}
```

— Where, if stepWidth > 16, deadZoneWidthOffset is derived as follows:

deadZoneWidthOffset[x][y] = (((1 << 16) − ((Aconst * (qm[x + (levelIdxSwap * nTbs)][y] + stepWidthModifier[x][y])) + Bconst) >> 1) * (qm[x + (levelIdxSwap * nTbs)][y] + stepWidthModifier[x][y])) >> 16

— Where, if stepWidth <= 16, deadZoneWidthOffset is derived as follows:

deadZoneWidthOffset[x][y] = stepWidth >> 1

— Where:

Aconst = 39
Bconst = 126484
Cconst = 5242
Dconst = 99614

— Where dQuantOffsetActual[x][y] is computed as follows:

```
if (dequant_offset == 0)
    dQuantOffsetActual[x][y] = dQuantOffset
else {
    if (dequant_offset_mode_flag == 1)
        dQuantOffsetActual[x][y] = ((Floor(−Cconst * Ln(qm[x + (levelIdxSwap * nTbs)][y]) +
            (dQuantOffset << 9) + Floor(Cconst * Ln(StepWidth)))) * (qm[x + (levelIdxSwap * nTbs)]
          [y])) >> 16
    else if (dequant_offset_mode_flag == 0)
        dQuantOffsetActual[x][y] = ((Floor(−Cconst * Ln(qm[x + (levelIdxSwap * nTbs)][y]) +
            (dQuantOffset << 11) + Floor(Cconst * Ln(StepWidth)))) * (qm[x + (levelIdxSwap * nTbs)]
       [y])) >>16
}
```

— Where levelIdxSwap is derived as follows:

```
if (levelIdx == 2)
    levelIdxSwap = 0
else
    levelIdxSwap = 1
```

8.5.4 Derivation of quantization matrix

8.5.4.1 The quantization matrix

The quantization matrix qm[x][y] contains the actual quantization step widths to be used to decode each coefficient group.

```
if (levelIdx == 2) {
    if (scaling_mode_level2 == 1) {
        for (x = 0; x < nTbS; x++) {
            for (y = 0; y < nTbs; y++)
                qm[x][y] = qm_p[x][y]
        }
    } else {
        for (x = 0; x < nTbS; x++) {
            for (y = 0; y < nTbS; y++)
                qm[x][y] = qm_p[x + nTbS][y]
        }
    }
} else {
    for (x = 0; x < nTbS; x++) {
        for (y = 0; y < nTbs; y++)
            qm[x][y] = qm_p[x + (2 * nTbS)][y]
    }
}
```

Where qm_p[x][y] is computed as follows:

```
if (nTbs == 2) {
    for (x = 0; x < 6; x++) {
        for (y = 0; y < nTbs; y++)
            qm_p[x][y] = (Clip3 (0, (3 << 16),[(QuantScalerDDBuffer[x][y] * stepWidth) + (1 << 16)]) *
                stepWidth) >> 16
    }
} else {
    for (y = 0; y < 12; y++) {
        for (x = 0; x < nTbs; x++)
            qm_p[x][y] = (Clip3 (0, (3 << 16),[(QuantScalerDDSBuffer[x][y] * stepWidth) + (1 << 16)]) *
            stepWidth) >> 16
    }
}
```

And where QuantScalerDDBuffer[x][y] is derived in sub-clause 8.5.4.2 and QuantScalerDDSBuffer[x][y] is derived in 8.5.4.3.

8.5.4.2 Derivation of scaling parameters for 2x2 transform

If the variable nTbS is equal to 2, the default scaling parameters are as follows:

default_scaling_dd[x][y] =

```
{
{  0,    2 }
{  0,    0 }
{ 32,    3 }
{  0,   32 }
{  0,    3 }
{  0,   32 }
}                                                        (17)
```

As a first step, the array QuantScalerDDBuffer[x][y] is initialized as follows:

If the current picture is an IDR picture, QuantScalerDDBuffer[x][y] is initialized to be equal to default_scaling_dd[x][y]. If the current picture is not an IDR picture, the QuantScalerDDBuffer[x][y] matrix is left unchanged.

Following initialization, based on the value of quant_matrix_mode the array QuantScalerDDBuffer[x][y] is processed as follows:

— If the quant_matrix_mode is equal to 0 and the current picture is not an IDR picture, the QuantScalerDDBuffer[x][y] is left unchanged.

— If quant_matrix_mode is equal to 1, the QuantScalerDDBuffer[x][y] is equal to the default_scaling_dd[x][y].

— If quant_matrix_mode is equal to 2, the QuantScalerDDBuffer[x][y] is modified as follows:

```
for (MIdx = 0; MIdx < 3; MIdx++)
    for (x = 0; x < 2; x++)
        for (y = 0; y < 2; y++)
            QuantScalerDDBuffer[x + (MIdx * 2)][y] = QmCoeff0[(x * 2) + y]
```

— If quant_matrix_mode is equal to 3, the QuantScalerDDBuffer[x][y] is modified as follows:

```
for (MIdx = 0; MIdx < 2; MIdx++)
    for (x = 0; x < 2; x++)
        for (y = 0; y < 2; y++)
            QuantScalerDDBuffer[x + (MIdx * 2)][y] = QmCoeff0[(x * 2) + y]
```

— If quant_matrix_mode is equal to 4, the QuantScalerDDBuffer[x][y] is modified as follows:

```
for (x = 0; x < 2; x++)
    for (y = 0; y < 2; y++)
        QuantScalerDDBuffer[x + 4][y] = QmCoeff1[(x * 2) + y]
```

— If quant_matrix_mode is equal to 5, the QuantScalerDDBuffer is modified as follows:

```
for (MIdx = 0; MIdx < 2; MIdx ++)
    for (x = 0; x < 2; x++)
        for (y = 0; y < 2; y++)
        QuantScalerDDBuffer[x + (MIdx * 2)][y] = QmCoeff0[(x * 2) + y]
  for (x = 4, x < 6; x++)
      for (y = 0; y < 2; y++)
        QuantScalerDDBuffer[x][y] = QmCoeff1[(x * 2) + y]
```

8.5.4.3 Derivation of scaling parameters for 4x4 transform

If the variable nTbS is equal to 4, the default scaling parameters are as follows:

default_scaling_dds[x][y] =

```
{
{  13,   26,  19,   32 }
{  52,    1,  78,    9 }
{  13,   26,  19,   32 }
{ 150,   91,  91,   19 }
{  13,   26,  19,   32 }
{  52,    1,  78,    9 }
{  26,   72,   0,    3 }
{ 150,   91,  91,   19 }
{   0,    0,   0,    2 }
{  52,    1,  78,    9 }
{  26,   72,   0,    3 }
{ 150,   91,  91,   19 }
}                                                        (18)
```

As a first step, the array QuantScalerDDSBuffer[][] is initialized as follows:

— If the current picture is an IDR picture, QuantScalerDDSBuffer[x][y] is initialized to be equal to default_scaling_dds[x][y]. If the current picture is not an IDR picture, the QuantScalerDDSBuffer[x][y] matrix is left unchanged.

Following initialization, based on the value of quant_matrix_mode the array QuantScalerDDSBuffer[x][y] is processed as follows:

— If the quant_matrix_mode is equal to 0 and the current picture is not an IDR picture, the QuantScalerDDSBuffer is left unchanged.

— If quant_matrix_mode is equal to 1, the QuantScalerDDSBuffer is equal to the default_scaling_dds[x][y].

— If quant_matrix_mode is equal to 2, the QuantScalerDDSBuffer is modified as follows:

```
for (MIdx = 0; MIdx < 3; MIdx++)
  for (x = 0; x < 4; x++)
      for (y = 0; y < 4; y++)
        QuantScalerDDSBuffer[x + (MIdx * 4)][y] = QmCoeff0[(x * 4) + y]
```

— If quant_matrix_mode is equal to 3, the QuantScalerDDSBuffer is modified as follows:

```
for (MIdx = 0; MIdx < 2; MIdx++)
  for (x = 0; x < 4; x++)
      for (y = 0; y < 4; y++)
        QuantScalerDDSBuffer[x + (MIdx * 4)][y] = QmCoeff0[(x * 4) + y]
```

— If quant_matrix_mode is equal to 4, the QuantScalerDDSBuffer is modified as follows:

```
for (x = 0; x < 4; x++)
    for (y = 0; y < 4; y++)
        QuantScalerDDSBuffer[x + 8][y] = QmCoeff1[(x * 4) + y]
```

— If quant_matrix_mode is equal to 5, the QuantScalerDDSBuffer is modified as follows:

```
for (MIdx = 0; MIdx < 2; MIdx++)
    for (x = 0; x < 4; x++)
        for (y = 0; y < 4; y++)
            QuantScalerDDSBuffer[x + (MIdx * 4)][y] = QmCoeff0[(x * 4) + y]
    for (x = 8, x < 12; x++)
        for (y = 0; y < 4; y++)
            QuantScalerDDSBuffer[x][y] = qm_coefficient_1[(x * 4) + y]
```

8.6 Decoding process for the transform

8.6.1 General upscaling process description

8.6.1.1 Upscaling processes overview

Upscaling processes are applied to the decoded base picture based on the indications of scaling_mode_level1 and to the combined intermediate picture based on scaling_mode_level2.

8.6.1.2 Upscaling from decoded base picture to preliminary intermediate picture

Inputs to this process are the following:

— 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 IdxPlanes specifying the colour component of the current block,

— a variable nCurrS (as derived in 8.8.1) specifying the size of the residual block,

— an (nCurrS)x(nCurrS) array recDecodedBaseSamples specifying the decoded base samples of the current block,

— a variable srcWidth and srcHeight specifying the width and the height of the decoded base picture,

— a variable dstWidth and dstHeight specifying the with and the height of the resulting upscaled picture,

— a variable is8Bit used to select the kernel coefficients for the scaling to be applied. If the samples are 8-bit, then variable is8Bit should be equal to 0. If the samples are 16-bit, then variable is8Bit should be equal to 1.

Output of this process is the (nCurrX)x(nCurrY) array recL1ModifiedUpsampledBaseSamples of residuals with elements recL1ModifiedUpsampledBaseSamples[x][y] where nCurrX and nCurrY are derived as follows:

— If scaling_mode_level1 (specified in 7.4.3.3) is equal to 0, no upscaling is performed, and recL1ModifiedUpsampledBaseSamples[x][y] are set to be equal to recDecodedBaseSamples[x][y]

— If scaling_mode_level1 (specified in 7.4.3.3) is equal to 1:

$$nCurrX = nCurrS << 1 \quad (19)$$

$$nCurrY = nCurrS \quad (20)$$

— If scaling_mode_level1 (specified in 7.4.3.2) is equal to 2:

nCurrX = nCurrS << 1 (21)

nCurrY = nCurrS << 1 (22)

Depending on the values of the bitstream fields in global configuration as specified in 7.3.5 (Table 9) the sample bit depth for L-1 is derived as follows:

If level1_depth_flag (7.3.3) is equal to 0, the preliminary intermediate picture samples are processed at the same bit depth as they are represented for the decoded base picture, following the processes described in 8.8 and 8.9.

If level1_depth_flag (7.3.3) is equal to 1, the preliminary intermediate picture samples are converted depending on the value of a variable base_depth and a variable enhancement_depth, derived as follows:

base_depth is assigned a value between 8 and 14, depending on the value of field base_depth_type as specified in Table 23;

enhancement_depth is assigned a value between 8 and 14, depending on the value of field enhancement_depth_type as specified in Table 24;

If base_depth is equal to enhancement_depth, no further processing is required.

If enhancement_depth is greater than base_depth, the array recDecodedBaseSamples is modified as follows:

```
recDecodedBaseSamples[x][y] =
    recDecodedBaseSamples[x][y] << (enhancement_depth - base_depth)
```

The result of the process described in 8.8.2 should be processed by an upscaling filter of the type signalled in the bitstream. The type of upscaler is derived from the process described in 7.4.3.3. Depending on the value of the variable upsample_type, the following kernel types will be selected, providing recDecodedBaseSamples as input and producing recL1UpsampledBaseSamples as output:

— If upsample_type is equal to 0, the Nearest sample upscaler described in 8.7.1 shall be selected.

— If upsample_type is equal to 1, the Bilinear upscaler described in 8.7.2 is selected.

— If upsample_type is equal to 2, the Cubic upscaler described in 8.7.3 is selected.

— If upsample_type is equal to 3, the Modified Cubic upscaler described in 8.7.4 is selected.

— If upsample_type is equal to 4, the Adaptive Cubic upscaler described in 8.7.6 is selected.

The general upscaler divides the picture to upscale in 2 areas: centre area and border areas. For the Bilinear and Bicubic kernel, the border area consists of four segments, Top, Left, Right and Bottom segments, while for the Nearest kernel consists of 2 segments, Right and Bottom. These segments are defined by the border-size parameter which is usually set to 2 samples (1 sample for nearest method). Figure 6 describes the upscaler areas.

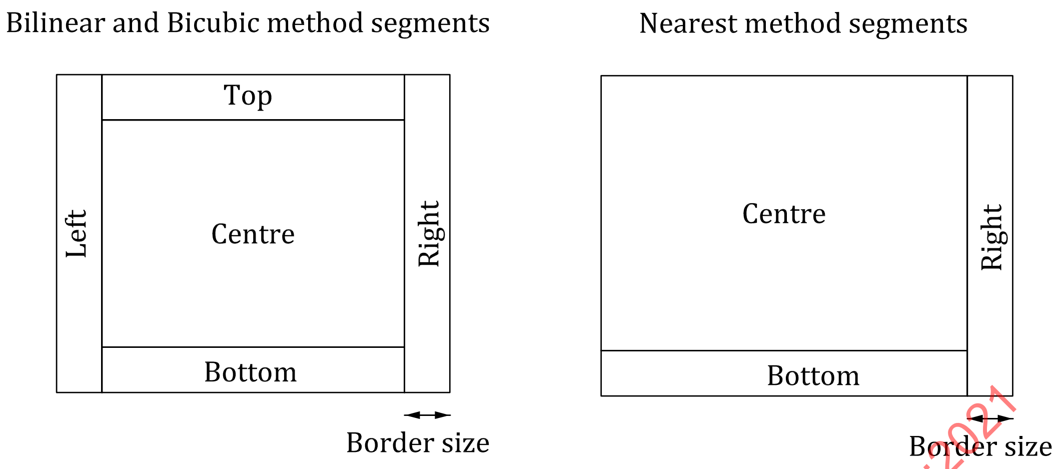


Figure 6 — Upscaler areas

Following the upscaling, if predicted_residual_mode_flag as derived in 7.4.3.3 is equal to 1, then the process described in 8.7.5 is invoked with inputs the (nCurrX)x(nCurrY) array recL1UpsampledBaseSamples and the (nCurrS)x(nCurrS) array recDecodedBaseSamples specifying the decoded base samples of the current block and output is the (nCurrX)x(nCurrY) array recL1ModifiedUpsampledBaseSamples of preliminary intermediate picture samples with elements recL1ModifiedUpsampledBaseSamples[x][y]. Otherwise, if predicted_residual_mode_flag as derived in 7.4.3.3 is equal to 0, recL1ModifiedUpsampledBaseSamples[x][y] are set to be equal to recL1UpsampledBaseSamples[x][y].

If base_depth is greater than enhancement_depth, the array recL1ModifiedUpsampledBaseSamples is modified as follows:

recL1ModifiedUpsampledBaseSamples[x][y] =

recL1ModifiedUpsampledBaseSamples[x][y] >> (base_depth - enhancement_depth)

8.6.1.3 Upscaling from combined intermediate picture to preliminary output picture

Inputs to this process are the following:

— 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 IdxPlanes specifying the colour component of the current block,

— a variable nCurrS (as derived in 8.8.1) specifying the size of the residual block,

— an (nCurrS)x(nCurrS) array recL1PictureSamples specifying the combined intermediate picture samples of the current block,

— a variable srcWidth and srcHeight specifying the width and the height of the reconstructed base picture,

— a variable dstWidth and dstHeight specifying the width and the height of the resulting upscaled picture,

— a variable is8Bit used to select the kernel coefficients for the scaling to be applied. If the samples are 8-bit, then variable is8Bit should be equal to 0. If the samples are 16-bit, then variable is8Bit should be equal to 1.

Output of this process is the (nCurrX)x(nCurrY) array recL2ModifiedUpsampledSamples of preliminary output picture samples with elements recL2ModifiedUpsampledSamples[x][y], where nCurrX and nCurrY are derived as follows:

— If scaling_mode_level2 (specified in 7.4.3.2) is equal to 0, no upscaling is performed, and recL2ModifiedUpsampledSamples[x][y] are set to be equal to recL1PictureSamples[x][y]

— If scaling_mode_level2 (specified in 7.4.3.2) is equal to 1:

nCurrX = nCurrS << 1 (23)

nCurrY = nCurrS (24)

— If scaling_mode_level2 (specified in 7.4.3.2) is equal to 2:

nCurrX = nCurrS << 1 (25)

nCurrY = nCurrS << 1 (26)

The result of the process described in 8.8.2 should be processed by an upscaling filter of the type signalled in the bitstream. The type of upscaler is derived from the process described in 7.4.3.3. Depending on the value of the variable upsample_type, the following kernel types will be selected, providing recL1PictureSamples as input and producing recL2UpsampledSamples as output:

— If upsample_type is equal to 0, the Nearest sample upscaler described in 8.7.1 is selected.

— If upsample_type is equal to 1, the Bilinear upscaler described in 8.7.2 is selected.

— If upsample_type is equal to 2, the Cubic upscaler described in 8.7.3 is selected.

— If upsample_type is equal to 3, the Modified Cubic upscaler described in 8.7.4 is selected.

— If upsample_type is equal to 4, the Adaptive Cubic upscaler described in 8.7.6 is selected.

The general upscaler divide the picture to upscale in 2 areas: centre area and border areas. For the Bilinear and Bicubic kernel, the border area consists of four segments, Top, Left, Right and Bottom segments, while for the Nearest kernel consists of 2 segments, Right and Bottom. These segments are defined by the border-size parameter which is usually set to 2 samples (1 sample for nearest method). Figure 7 describes the upscaler areas.

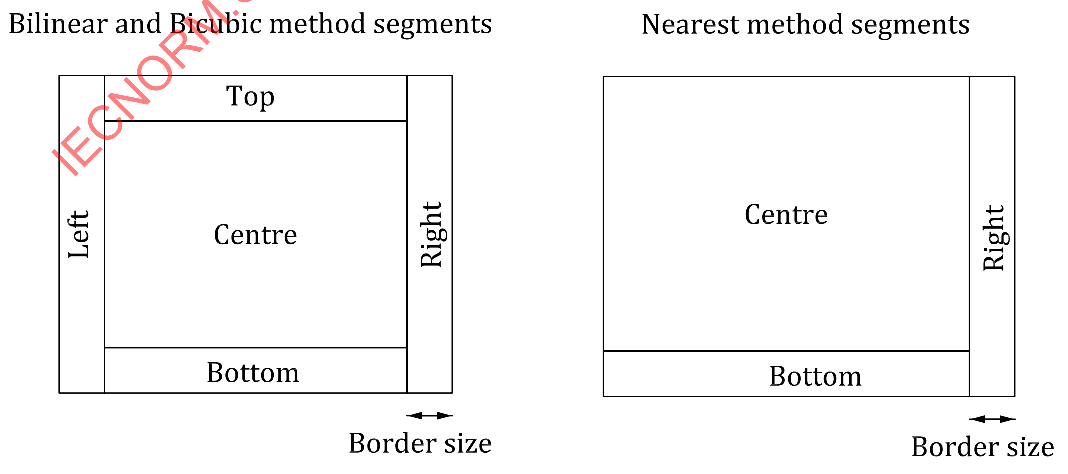


Figure 7 — Upsampler areas

Following the upscaling, if predicted_residual_mode_flag as derived in 7.4.3.3 is equal to 1, process described in 8.7.5 is invoked with inputs the (nCurrX)x(nCurrY) array recL2UpsampledSamples and the (nCurrS)x(nCurrS) array recL1PictureSamples specifying the combined intermediate picture samples of the current block and output is the (nCurrX)x(nCurrY) array recL2ModifiedUpsampledSamples of residuals with elements recModifiedUpsampledL2Samples[x][y]. Otherwise, if predicted_residual_mode_flag as derived in 7.4.3.3 is equal to 0, recL2ModifiedUpsampledSamples[x][y] are set to be equal to recL2UpsampledSamples[x][y].

Depending on the values of the bitstream fields in global configuration as specified in 7.3.5 (Table 9), the sample bit depth for L-2 is derived as follows:

If level1_depth_flag is equal to 1, the preliminary output picture samples are processed at the same bit depth as they are represented for the preliminary intermediate picture, following the processes described in 8.8 and 8.9.

If level1_depth_flag is equal to 0, the output intermediate picture samples are converted depending on the value of a variable base_depth and a variable enhancement_depth, derived as follows:

base_depth is assigned a value between 8 and 14, depending on the value of field base_depth_type as specified in Table 23;

enhancement_depth is assigned a value between 8 and 14, depending on the value of field enhancement_depth_type as specified in Table 24;

If base_depth is equal to enhancement_depth, no further processing is required.

If enhancement_depth is greater than base_depth, the array recL2ModifiedUpsampledSamples is modified as follows:

```
    recL2ModifiedUpsampledSamples[x][y] =
        recL2ModifiedUpsampledSamples[x][y] << (enhancement_depth - base_depth)
```

If base_depth is greater than enhancement_depth, the array recL2ModifiedUpsampledSamples is modified as follows:

```
    recL2ModifiedUpsampledSamples[x][y] =
        recL2ModifiedUpsampledSamples[x][y] >> (base_depth - enhancement_depth)
```

8.6.2 Transform inputs and outputs, transform types, and residual samples derivation

Inputs to this process are:

— a location (xTbP, yTbP) specifying the top-left sample of the current luma or chroma transform block relative to the top-left luma or chroma sample of the current picture. P can be related to either luma or chroma plane depending to which plane the transform coefficients belong,

— a variable nTbS specifying the size of the current transform block (nTbS = 2 if transform_type is equal to zero and nTbS = 4 if transform_type is equal to 1),

— an (nTbS)x(nTbS) array d of dequantized transform coefficients with elements d[x][y].

Output of this process is the (nTbS)x(nTbS) array R of residuals with elements R[x][y].

There are two types of transforms that can be used in the encoding process. Both leverage small kernels which are applied directly to the residuals that remain after the stage of applying Predicted Residuals (PRs). Figure 8 is a representation of how the residuals are transformed.

R_{00}	R_{01}	R_{02}	R_{03}	...
R_{10}	R_{11}	R_{12}	R_{13}	...
R_{20}	R_{21}	R_{22}	R_{23}	...
R_{30}	R_{31}	R_{32}	R_{33}	...
...	...	...	...	...

Figure 8 — Representation of how residuals are transformed

The (nTbS)x(nTbS) array R of residual samples is derived as follows:

Each (vertical) column of dequantized transform coefficients d[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 two-dimensional transformation process as specified in 8.6.3, if nTbS is equal to 2

Or

Each (vertical) column of dequantized transform coefficients d[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 two-dimensional transformation process as specified in 8.6.4, if nTbS is equal to 4.

8.6.3 2x2 directional decomposition transform

If nTbS is equal to 2, the transform has a 2x2 kernel which is applied to each 2x2 block of transform coefficients. The resulting residuals are derived as follows:

— If scaling_mode_levelX (specified in 7.4.3.3) for the corresponding enhancement sub-layer is equal to 0 or 2:

$$\{\{\{ \begin{matrix} \{R_{00}\} \\ \{R_{01}\} \\ \{R_{10}\} \\ \{R_{11}\} \end{matrix} = \begin{matrix} \{1, & 1, & 1, & 1\} \\ \{1, & -1, & 1, & -1\} \\ \{1, & 1, & -1, & -1\} \\ \{1, & -1, & -1, & 1\} \end{matrix} * \begin{matrix} \{d_{00}\} \\ \{d_{01}\} \\ \{d_{10}\} \\ \{d_{11}\} \end{matrix} \}\}\} \tag{27}$$

— If scaling_mode_levelX (specified in 7.4.3.3) for the corresponding enhancement sub-layer is equal to 1:

$$\{\{\{ \begin{matrix} \{R_{00}\} \\ \{R_{01}\} \\ \{R_{10}\} \\ \{R_{11}\} \end{matrix} = \begin{matrix} \{1, & 1, & 1, & 0\} \\ \{1, & -1, & -1, & 0\} \\ \{0, & 1, & -1, & 1\} \\ \{0, & -1, & 1, & 1\} \end{matrix} * \begin{matrix} \{d_{00}\} \\ \{d_{01}\} \\ \{d_{10}\} \\ \{d_{11}\} \end{matrix} \}\}\} \tag{28}$$

8.6.4 4x4 directional decomposition transform

If nTbS is equal to 4, the transform has a 4x4 kernel which is applied to a 4x4 block of transform coefficients. The resulting residuals are derived as follows:

— If scaling_mode_levelX (specified in 7.4.3.3) for the corresponding enhancement sub-layer is equal to 0 or 2:

$$
\begin{array}{lclcl}
\{\{\{ & & & & \\
\{R_{00}\} & & \{1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\} & & \{d_{00}\} \\
\{R_{01}\} & & \{1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1\} & & \{d_{01}\} \\
\{R_{02}\} & & \{1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1\} & & \{d_{02}\} \\
\{R_{03}\} & & \{1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1\} & & \{d_{03}\} \\
\{R_{10}\} & & \{1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1\} & & \{d_{10}\} \\
\{R_{11}\} & & \{1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1\} & & \{d_{11}\} \\
\{R_{12}\} & & \{1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1\} & & \{d_{12}\} \\
\{R_{13}\} & = & \{1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1\} & * & \{d_{13}\} \\
\{R_{20}\} & & \{1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1\} & & \{d_{20}\} \\
\{R_{21}\} & & \{1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1\} & & \{d_{21}\} \\
\{R_{22}\} & & \{1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1\} & & \{d_{22}\} \\
\{R_{23}\} & & \{1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1\} & & \{d_{23}\} \\
\{R_{30}\} & & \{1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1\} & & \{d_{30}\} \\
\{R_{31}\} & & \{1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1\} & & \{d_{31}\} \\
\{R_{32}\} & & \{1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1\} & & \{d_{32}\} \\
\{R_{33}\} & & \{1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1\} & & \{d_{33}\} \\
\}\}\} & & & &
\end{array}
\qquad (29)
$$

— If scaling_mode_levelX (specified in 7.4.3.3) for the corresponding enhancement sub-layer is equal to 1:

$$
\{\{\{
$$

$$
\begin{array}{lcl}
\{R_{00}\} & & \{1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1\} & & \{d_{00}\} \\
\{R_{01}\} & & \{1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 0,\ 0,\ 0,\ 0,\ -1,\ -1,\ -1,\ -1\} & & \{d_{01}\} \\
\{R_{02}\} & & \{1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 0,\ 0,\ 0,\ 0,\ 1,\ -1,\ 1,\ -1\} & & \{d_{02}\} \\
\{R_{03}\} & & \{1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 0,\ 0,\ 0,\ 0,\ -1,\ 1,\ -1,\ 1\} & & \{d_{03}\} \\
\{R_{10}\} & & \{0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1\} & & \{d_{10}\} \\
\{R_{11}\} & & \{0,\ 0,\ 0,\ 0,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\} & & \{d_{11}\} \\
\{R_{12}\} & & \{0,\ 0,\ 0,\ 0,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1\} & & \{d_{12}\} \\
\{R_{13}\} & = & \{0,\ 0,\ 0,\ 0,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1\} & * & \{d_{13}\} \\
\{R_{20}\} & & \{1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ -1,\ -1\} & & \{d_{20}\} \\
\{R_{21}\} & & \{1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0,\ -1,\ -1,\ 1,\ 1\} & & \{d_{21}\} \\
\{R_{22}\} & & \{1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 0,\ 0,\ 0,\ 0,\ 1,\ -1,\ -1,\ 1\} & & \{d_{22}\} \\
\{R_{23}\} & & \{1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1,\ 0,\ 0,\ 0,\ 0,\ -1,\ 1,\ 1,\ -1\} & & \{d_{23}\} \\
\{R_{30}\} & & \{0,\ 0,\ 0,\ 0,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ 1\} & & \{d_{30}\} \\
\{R_{31}\} & & \{0,\ 0,\ 0,\ 0,\ -1,\ -1,\ 1,\ 1,\ 1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1\} & & \{d_{31}\} \\
\{R_{32}\} & & \{0,\ 0,\ 0,\ 0,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1,\ -1,\ 1,\ 1,\ -1\} & & \{d_{32}\} \\
\{R_{33}\} & & \{0,\ 0,\ 0,\ 0,\ -1,\ 1,\ 1,\ -1,\ 1,\ -1,\ -1,\ 1,\ 1,\ -1,\ -1,\ 1\} & & \{d_{33}\}
\end{array}
$$

$$
\}\}\} \qquad (30)
$$

8.7 Decoding process for the upscaling

8.7.1 Nearest sample upsampler kernel description

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the input array,

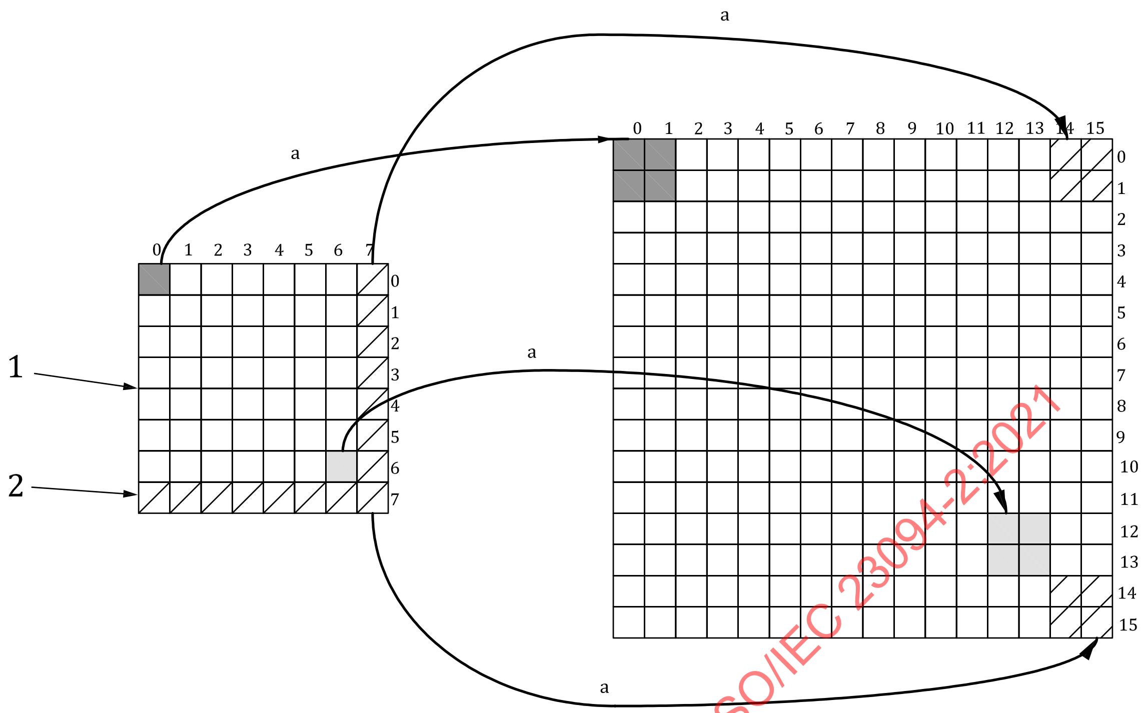
— variables dstX and dstY specifying the width and the height of the output array,

— a (srcX)x(srcY) array recInputSamples[x][y] of input samples.

Output of this process is the following:

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of output samples.

The Nearest kernel performs upscaling by copying the current source sample onto the destination 2x2 grid. This is shown in Figure 9. The destination sample positions are calculated by doubling the index of the source sample on both axes and adding +1 to extend the range to cover 4 samples as shown in Figure 9.



Key

1	centre
2	border
a	copy

Figure 9 — Nearest upsample kernel

The nearest sample kernel upscaler is applied as specified by the following ordered steps whenever (xCurr, yCurr) block belongs to the picture or to the border area as specified in Figure 9.

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 1:

```
for (ySrc = 0; ySrc < nCurrS; ++ySrc)
    yDst = ySrc
    for (xSrc = 0; xSrc < nCurrS; ++xSrc)
        xDst = xSrc << 1
        recUpsampledSamples[xDst][yDst] = recInputSamples[xSrc][ySrc]
        recUpsampledSamples[xDst + 1][yDst] = recInputSamples[xSrc][ySrc]
```

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 2:

```
for (ySrc = 0; ySrc < nCurrS; ++ySrc)
    yDst = ySrc << 1
    for (xSrc = 0; xSrc < nCurrS; ++xSrc)
        xDst = xSrc << 1
        recUpsampledSamples[xDst][yDst] = recInputSamples[xSrc][ySrc]
        recUpsampledSamples[xDst][yDst + 1] = recInputSamples[xSrc][ySrc]
        recUpsampledSamples[xDst + 1][yDst] = recInputSamples[xSrc][ySrc]
        recUpsampledSamples[xDst + 1][yDst + 1] = recInputSamples[xSrc][ySrc]
```

8.7.2 Bilinear upsampler kernel description

8.7.2.1 Bilinear upsampler kernel process inputs and outputs, process overview

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the input array,

— variables dstX and dstY specifying the width and the height of the output array,

— a (srcX)x(srcY) array recInputSamples[x][y] of input samples.

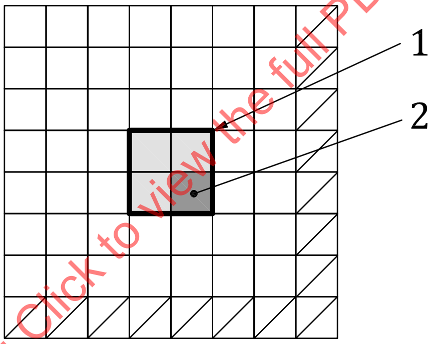
Outputs of this process are the following:

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of output samples.

The Bilinear upsampling kernel consists of three main steps. The first step involves constructing a 2x2 grid of source samples with the base sample positioned at the bottom right corner. The second step involves performing the bilinear interpolation and the third step involves writing the interpolation result to the destination samples.

8.7.2.2 Source sample grid

The bilinear method performs the upsampling by considering the values of the nearest 3 samples to the base sample. The base sample is the source sample from which the address of the destination sample is derived. Figure 10 shows the source grid used in the kernel.



Key

1 (2x2) bilinear grid

2 base pixel

Figure 10 — Example of bilinear 2x2 source grid

8.7.2.3 Bilinear interpolation

The bilinear interpolation is a weighted summation of all the samples in the source grid. The weights employed are dependent on the destination sample being derived. The algorithm applies weights which are relative to the position of the source samples with respect to the position of the destination samples. If calculating the value for the top left destination sample, then the top left source sample will receive the largest weighting coefficient while the bottom right sample (diagonally opposite) will receive the smallest weighting coefficient, and the remaining two samples will receive an intermediate weighting coefficient. This is visualized in Figure 11.

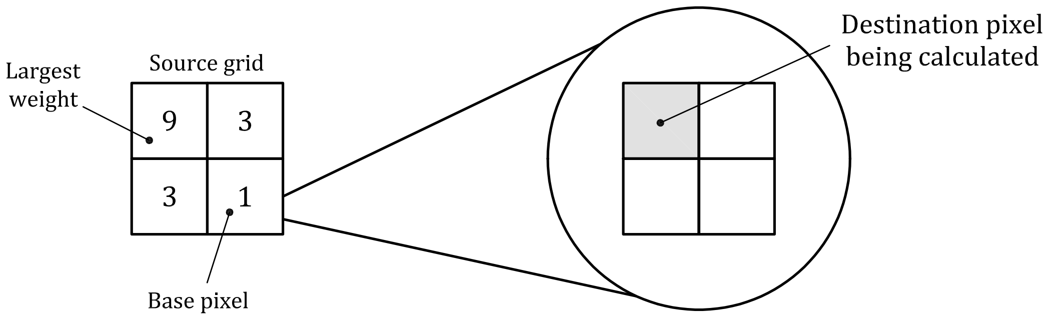


Figure 11 — Bilinear coefficient derivation

8.7.2.4 Bilinear interpolation upsampler kernel description

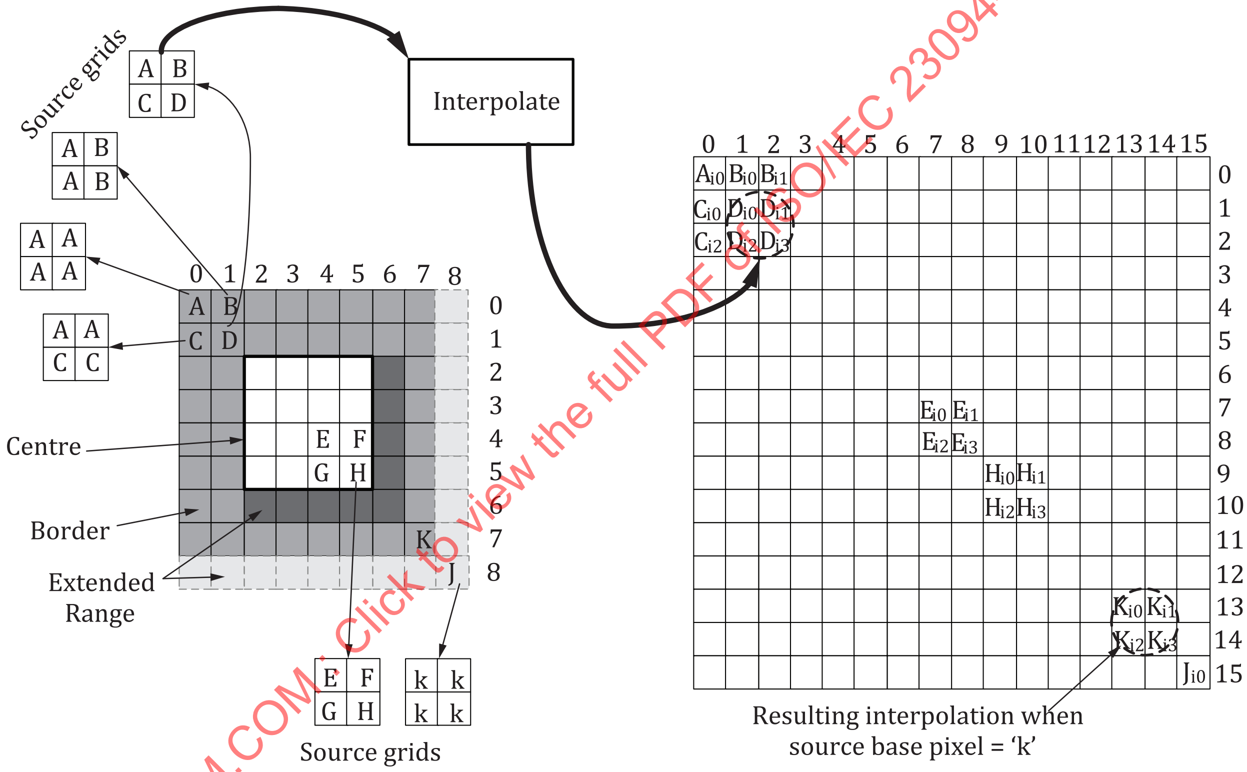


Figure 12 — Bilinear upsample kernel

The bilinear kernel upscaler, as illustrated in Figure 12, is applied as specified by the following ordered steps below when (xCurr, yCurr) block does not belong to the border area as specified in Figure 7:

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 1:

```
for (ySrc = 0; ySrc < nCurrS + 1; ++ySrc)
    for (xSrc = 0; xSrc < nCurrS + 1; ++xSrc)
        xDst = (xSrc << 1) – 1
        bilinear1D(recInputSamples[xSrc – 1][ySrc], recInputSamples[xSrc1][ySrc],
            recUpsampledSamples[xDst][ySrc], recUpsampledSamples [xDst + 1][ySrc])
```

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 2:

```
for (ySrc = 0; ySrc < nCurrS + 1; ++ySrc)
    yDst = (ySrc << 1) – 1
    for (xSrc = 0; xSrc < nCurrS + 1; ++xSrc)
```

```
        xDst = (xSrc << 1) – 1
        bilinear2D(recInputSamples[xSrc – 1][ySrc – 1], recInputSamples[xSrc][ySrc – 1],
            recInputSamples[xSrc – 1][ySrc], recInputSamples[xSrc][ySrc],
            recUpsampledSamples[xDst][ySrc], recUpsampledSamples[xDst + 1][ySrc],
            recUpsampledSamples[xDst][ySrc + 1], recUpsampledSamples[xDst + 1][ySrc + 1])
```

The bilinear kernel upscaler (described in Figure 12) is applied as specified by the following ordered steps below when (xCurr, yCurr) block belongs to the border area as specified in Figure 7:

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 1:

```
for (ySrc = 0; ySrc < nCurrS + 1; ++ySrc)
    for (xSrc = 0; xSrc < nCurrS + 1; ++xSrc)
        xDst = (xSrc << 1) – 1
        xSrc0 = Max(xSrc – 1, 0);
        xSrc1 = Min(xSrc, srcWidth – 1)
        bilinear1D(recInputSamples[xSrc0][ySrc], recInputSamples[xSrc1][ySrc], dst00, dst10)
        if (xDst >= 0)
            recUpsampledSamples[xDst][ySrc] = dst00
        if (xDst < (dstWidth–1))
            recUpsampledSamples[xDst + 1][ySrc] = dst10
```

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 2:

```
for (ySrc = 0; ySrc < nCurrS + 1; ++ySrc)
    yDst = (ySrc << 1) – 1
    ySrc0 = Max(ySrc – 1, 0)));
    ySrc1 = Min (ySrc, srcHeight – 1)))
    for (xSrc = 0; xSrc < nCurrS + 1; ++xSrc)
        xDst = (xSrc << 1) –1
        xSrc0 = Max(xSrc – 1, 0);
        xSrc1 = Min(xSrc, srcWidth – 1)
        bilinear2D(recInputSamples[xSrc0][ySrc0], recInputSamples[xSrc1][xSrc0],
            recInputSamples[xSrc0][ySrc1], recInputSamples[xSrc1][ySrc1], dst00, dst10, dst01,
            dst11)
```

The function bilinear1D (in00, in10, out00, out10) is applied as specified by the following ordered steps below:

```
in00x3 = in00 * 3
in10x3 = in10 * 3
out00 = ((in00x3 + in10 + 2) >> 2)
out10 = ((in00 + in10x3 + 2) >> 2)
```

The function bilinear2D (in00, in10, in01, in11, out00, out10, out01, out11) is applied as specified by the following ordered steps below:

```
in00x3 = in00 * 3
in10x3 = in10 * 3
in01x3 = in01 * 3
in11x3 = in11 * 3
in00x9 = in00x3 * 3
in10x9 = in10x3 * 3
in01x9 = in01x3 * 3
in11x9 = in11x3 * 3
out00 = ((in00x9 + in10x3 + in01x3 + in11 + 8) >> 4))
out10 = ((in00x3 + in10x9 + in01+ in11x3 + 8) >> 4))
out01 = ((in00x3 + in10 + in01x9 + in11x3 + 8) >> 4))
out11 = ((in00 + in10x3 + in01x3 + in11x9 + 8) >> 4))
```

8.7.3 Cubic upsampler kernel description

8.7.3.1 Cubic upsampler kernel process inputs and outputs, process overview

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the input array,

— variables dstX and dstY specifying the width and the height of the output array,

— a (srcX)x(srcY) array recInputSamples[x][y] of input samples.

Outputs of this process is:

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of output samples.

The Cubic upsampling kernel can be divided into three main steps. The first step involves constructing a 4x4 grid of source samples with the base sample positioned at the local index (2,2). The second step involves performing the bicubic interpolation and the third step involves writing the interpolation result to the destination samples.

8.7.3.2 Source sample grid

The Cubic upsampling kernel is performed by using a 4x4 source grid which is subsequently multiplied by a 4x4 kernel. During the generation of the source grid, any samples which fall outside the frame limits of the source frame are replaced with the value of the source samples at the boundary of the frame. This is visualized in Figure 13.

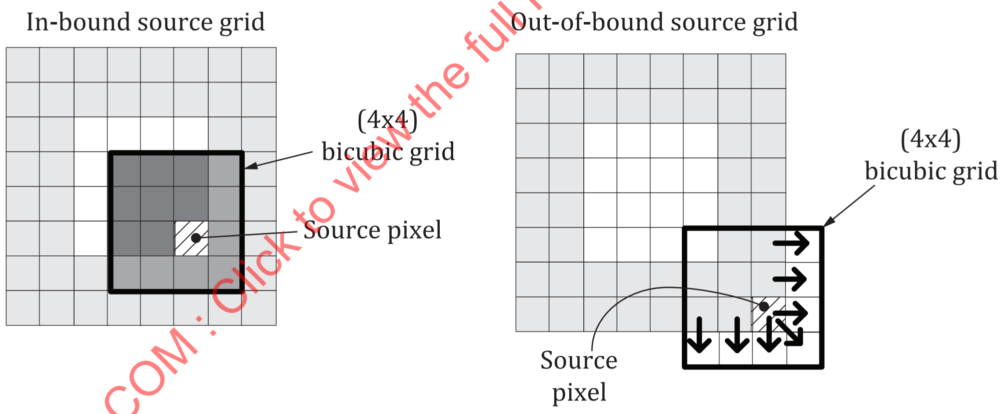


Figure 13 — Source grid for the cubic upsampler

8.7.3.3 Cubic interpolation

The kernel used for the Bicubic upsampling process typically has a 4x4 coefficient grid. However, the relative position of the destination sample with regards to the source sample will yield a different coefficient set, and since the upsampling is a factor of two, there will be 4 sets of 4x4 kernels used in the upsampling process. These sets are represented by a 4-dimensional grid of coefficients (2 x 2 x 4 x 4). The Bicubic coefficients are calculated from a fixed set of parameters; a core parameter (Bicubic parameter) of −0.6 and four spline creation parameters of [1.25, 0.25, −0.75 & −1.75]. The implementation of the filter uses fixed point math.

8.7.3.4 Cubic interpolation kernel description

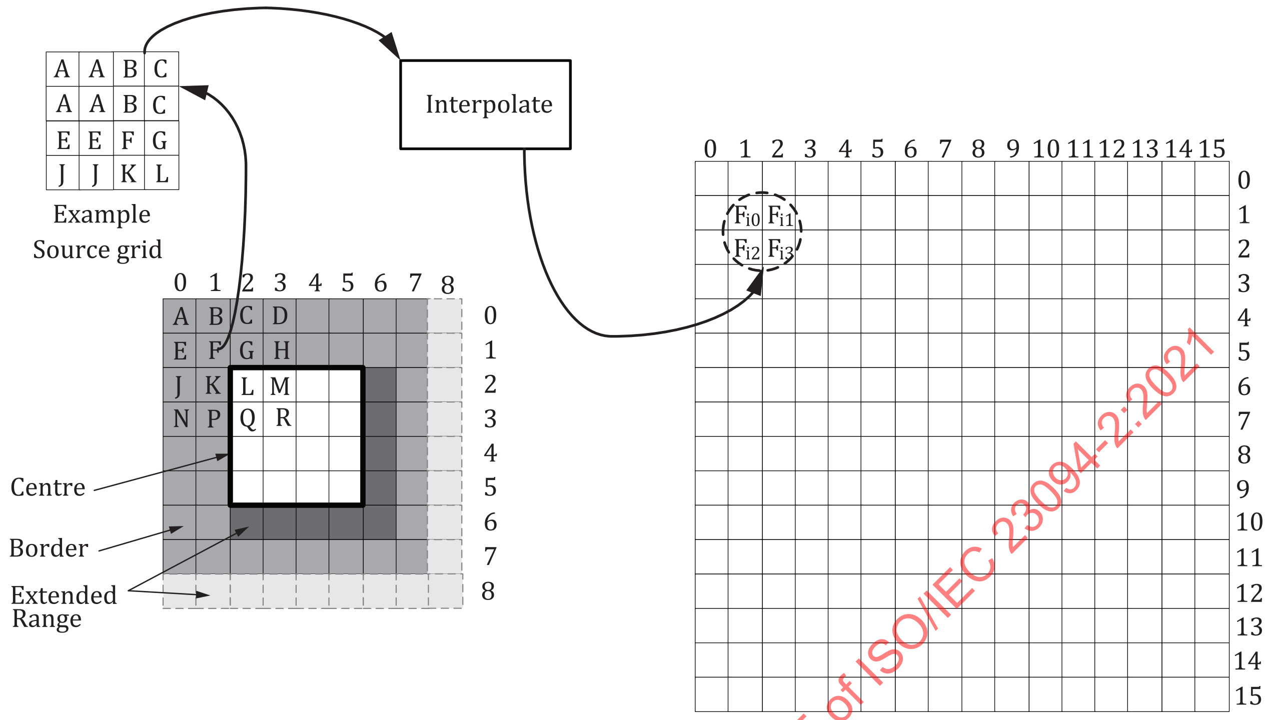


Figure 14 — Cubic upsampling algorithm

The cubic kernel upscaler (described in Figure 14) is applied on one direction (vertical and horizontal) at a time, and follows different steps if (xCurr, yCurr) block belongs to the border as specified in Figure 7.

Given a set of coefficients as follows:

kernel[y][x] =

$$\begin{array}{l} \{ \\ \{ -1382,\ 14285,\ \ 3942,\ \ -461 \ \} \\ \{ \ -461,\ \ 3942,\ 14285,\ -1382 \ \} \\ \} \end{array} \qquad (31)$$

The following variables are defined:

— kernelOffset is equal to 4.

— kernelSize is equal to 4.

```
if (Horizontal) {
    for (y = 0; y < nCurrS; y++)
        for (xSrc = 0; xSrc < nCurrS + 1; xSrc++)
            ConvolveHorizontal(recInputSamples, recUpsampledSamples, xSrc, y, kernel)
} else if (Vertical) {
    dstHeightM1 = dstHeight – 1
    for (ySrc = 0; ySrc < nCurrS + 1; ySrc++)
        yDst = (ySrc << 1) – 1
        if (border) {
            yDst0 = ((yDst > 0)&&(yDst < dstHeight)) ? yDst : −1
            yDst1 = ((yDst + 1) < dstHeightM1) ? yDst + 1 : −1
        } else {
            yDst0 = yDst
            yDst1 = (yDst + 1)
        }
        for (x = 0; x < nCurrS; x++)
            ConvolveVertical(recInputSamples, recUpsampledSamples, yDst0, yDst1, x, ySrc, kernel)
}
```

The function ConvolveHorizontal(input, output, x, y, kernel, border) is applied as specified by the following ordered steps below:

```
xDst = (x << 1) − 1;
if (border)
    dstWidthM1 = dstWidth − 1
if (xDst >= 0 && xDst < dstWidth)
    output[xDst][y] = ConvolveHorizontalKernel(kernel[0], input[x + kernelOffset][y], 14)
if (xDst < dstWidthM1)
    output[xDst + 1][y] = ConvolveHorizontalKernel(kernel[1], input[x + kernelOffset][y]), 14)
else
    output[xDst][y] = ConvolveHorizontalKernel(kernel[0], input[x + kernelOffset][y] , 14)
    output[xDst + 1][y] = ConvolveHorizontalKernel(kernel[1], input[x + kernelOffset][y], 14)
```

The function ConvolveVertical(input, output, yDst0, yDst1, x, ySrc, kernel) is applied as specified by the following ordered steps below.

The ConvolveVertical function is defined as:

```
if (border)
    dstWidthM1 = dstWidth − 1
if (yDst0 >= 0)
    output[x][yDst0] = ConvolveHorizontalKernel(kernel[0], input[x][y + kernelOffset], 14)
if (yDst0 >= 0)
    output[x][yDst] = ConvolveHorizontalKernel(kernel[1], input[x][y + kernelOffset]), 14)
else
    output[x][yDst0] = ConvolveHorizontalKernel(kernel[0], input[x + kernelOffset][y], 14)
    output[x][yDst1] = ConvolveHorizontalKernel(kernel[1], input[x + kernelOffset][y], 14)
```

The function output = ConvolveHorizontalKernel(kernel, input, shift) is applied as specified by the following ordered steps below:

```
accumulator = 0
for (int32_t x = 0; x < kernelSize; x++)
    accumulator += input[x] * kernel[x]
offset = 1 << (shift − 1)
output = ((accumulator + offset) >> shift)
```

8.7.4 Modified Cubic upsampler kernel description

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the input array,

— variables dstX and dstY specifying the width and the height of the output array,

— a (srcX)x(srcY) array recInputSamples[x][y] of input sample.

Outputs of this process are the following:

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of output samples.

The implementation of the Modified Cubic filter is using fixed point math and the process is the same described in 8.7.3.3, but with the following kernel coefficients:

kernel[y][x] =

$$\begin{Bmatrix} \{-2360, 15855, 4165, -1276\} \\ \{-1276, 4165, 15855, -2360\} \end{Bmatrix} \quad (32)$$

The following variables are defined:

— kernelOffset is equal to 4.

— kernelSize is equal to 4.

8.7.5 Predicted residual process description

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the lower resolution array,

— variables dstX and dstY specifying the width and the height of the upsampled arrays,

— a (srcX)x(srcY) array recLowerResSamples[x][y] of samples that were provided as input to the upscaling process,

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of samples that were output of the upscaling process.

Outputs of this process are the following:

— a (dstX)x(dstY) array recUpsampledModifiedSamples[x][y] of output samples.

The predicted residual process is modifying recUpsampledSamples using a 2x2 grid if scaling_mode_levelX is equal to 2 and using a 2x1 grid if scaling_mode_levelX is equal to 1. The predicted residual process is not applied if scaling_mode_levelX is equal to 0.

The predicted residual process is applied as specified by the following ordered steps whenever (xCurr, yCurr) block belongs to the picture or to the border area as specified in Figure 7:

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 1:

```
for (ySrc = 0; ySrc < srcY; ySrc++)
    yDst = ySrc
    for (xSrc = 0; xSrc < srcX; xSrc++)
        xDst = xSrc << 1
        modifier = recLowerResSamples[xSrc][ySrc] − (recUpsampledSamples[xDst][yDst] +
            recUpsampledSamples[xDst + 1][yDst]) >> 1
        recModifiedUpsampledSamples[xDst][yDst] = recUpsampledSamples[xDst][yDst] +
            modifier
        recModifiedUpsampledSamples[xDst + 1][yDst] = recUpsampledSamples[xDst + 1][yDst] +
            modifier
```

— If scaling_mode_levelX (specified in 7.4.3.3) is equal to 2:

```
for (ySrc = 0; ySrc < srcY; ySrc++)
    yDst = ySrc << 1
```

```
for (xSrc = 0; xSrc < srcX; xSrc++)
    xDst = xSrc << 1
    modifier = recLowerResSamples[xSrc][ySrc] –
        (recUpsampledSamples[xDst][yDst] +
         recUpsampledSamples[xDst + 1][yDst] +
            recUpsampledSamples[xDst][yDst + 1] +
         recUpsampledSamples[xDst + 1][yDst + 1]) >> 2
    recModifiedUpsampledSamples[xDst][yDst] = recUpsampledSamples[xDst][yDst] +
       modifier
    recModifiedUpsampledSamples[xDst][yDst + 1] = recUpsampledSamples[xDst + 1][yDst] +
       modifier
    recModifiedUpsampledSamples[xDst + 1][yDst] = recUpsampledSamples[xDst][yDst + 1] +
       modifier
    recModifiedUpsampledSamples[xDst + 1][yDst + 1] = recUpsampledSamples[xDst + 1]
       [yDst + 1] + modifier
```

8.7.6 Adaptive Cubic upsampler kernel description

Inputs to this process are the following:

— variables srcX and srcY specifying the width and the height of the input array,

— variables dstX and dstY specifying the width and the height of the output array,

— variables coeff1 to coeff4 corresponding to upsampler_coeff1 to upsampler_coeff4,

— a (srcX)x(srcY) array recInputSamples[x][y] of input sample.

Outputs of this process is the following:

— a (dstX)x(dstY) array recUpsampledSamples[x][y] of output samples.

The implementation of the Adaptive Cubic filter uses fixed point math, and the process is the same described in 8.7.3.3, but with the following kernel coefficients:

kernel[y][x] =

```
{
{ −coeff1, coeff2, coeff3, −coeff4 }
{ −coeff4, coeff3, coeff2, −coeff1 }
}
```

The following variables are defined:

— kernelOffset is equal to 4.

— kernelSize is equal to 4.

8.8 Decoding process for the residual reconstruction

8.8.1 Reconstructed residual of each block derivation

The reconstructed residual of each block is derived as follows:

The variable $nCbS_L$ is set equal to 2 if transform_type is equal to 0 or 4 if transform_type is equal to 1 (7.4.3.3). The variable $nCbS_C$ is set equal to $nCbS_L$ >> 1.

If IdxPlanes is equal to 0, the residual reconstruction process for a colour component as specified in 8.8.2 and 8.8.3 is invoked with the luma coding block location (xCb, yCb), the variable nCurrS set equal to $nCbS_L$, and the variable IdxPlanes set equal to 0 as input.

If IdxPlanes is equal to 1 or 2, the residual reconstruction process for a colour component as specified in 8.8.2 and 8.8.3 is invoked with the chroma coding block location (xCb >> ShiftWidthC, yCb >> ShiftHeightC), the variable nCurrS set equal to $nCbS_C$, and the variable IdxPlanes set equal to 1 or 2 respectively as inputs.

Where ShiftWidthC and ShiftHeightC are specified in 6.2.

8.8.2 Residual reconstruction for L-1 block

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 IdxPlanes specifying the colour component of the current block,

— a variable nCurrS (as derived in 8.8.1) specifying the size of the residual block,

— an (nCurrS)x(nCurrS) array recL1BaseSamples specifying the preliminary intermediate picture reconstructed samples of the current block as specified in 8.10,

— an (nCurrS)x(nCurrS) array resL1FilteredResiduals specifying the L-1 filtered residuals of the current block.

Output of this process is the combined intermediate picture (nCurrS)x(nCurrS) array recL1Samples with elements recL1Samples[x][y].

The (nCurrS)x(nCurrS) block of the reconstructed sample array recL1Samples at location (xCurr, yCurr) is derived as follows:

$$\text{recL1Samples}[\text{xCurr} + i][\text{yCurr} + j] = \text{recL1BaseSamples}[i][j] + \text{resL1FilteredResiduals}[i][j] \quad (33)$$

with i = 0 ... nCurrS − 1, j = 0 ... nCurrS − 1

The upscaling process for a colour component as specified in 8.7 is invoked with the location (xCurr, yCurr), the transform block size nTbS, the (nCurrS)x(nCurrS) array recL1Samples, the variables srcWidth and srcHeight specifying the size of the reconstructed base picture, the variables dstWidth and dstHeight specifying the width and the height of the upscaled resulting picture, and the variable is8Bit equal to 1 if enhancement_depth_type (7.4.3.3) is equal to 0 as inputs.

8.8.3 Residual reconstruction for L-2 block

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 IdxPlanes specifying the colour component of the current block,

— an (nCurrS)x(nCurrS) array recL2ModifiedUpscaledSamples specifying the preliminary output picture samples of the current block,

— an (nCurrS)x(nCurrS) array resL2Residuals specifying the L-2 residuals of the current block.

Output of this process is the (nCurrS)x(nCurrS) array recL2PictureSamples of combined output picture samples with elements recL2PictureSamples[x][y].

The (nCurrS)x(nCurrS) block of the reconstructed sample array recL2PictureSamples at location (xCurr, yCurr) is derived as follows:

recL2PictureSamples[xCurr + i][yCurr + j] = recL2ModifiedUpscaledSamples[i][j] + resL2Residuals[i][j] (34)

with i = 0 ... nCurrS − 1, j = 0 ... nCurrS − 1

If dithering_type as derived in 7.3.6 is not equal to 0, the process as specified in sub-clause 8.11 is invoked with the location (xCurr, yCurr) using the (nCurrS)x(nCurrS) block from the array recL2PictureSamples.

8.9 Decoding process for the L-1 filter

8.9.1 L-1 residual filter overview

One in-loop filter, namely L-1 residual filter, is applied to the L-1 residual surface block before it is added to the base reconstructed picture. The L-1 filter operates only if the variable transform_type specified in 7.4.3.2 is equal to 1. The L-1 filter operates on each 4x4 block of transformed residuals by applying a mask whose weights are structured as follows:

```
{
{ α,  β,  β,  α }
{ β,  1,  1,  β }
{ β,  1,  1,  β }
{ α,  β,  β,  α }
}                                                        (35)
```

8.9.2 Decoding process for filtering L-1 block

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,

— an array resL1Residuals of a size 4x4 specifying residuals for enhancement sub-layer 1.

Output of this process is the 4x4 array of the residual resL1FilteredResiduals with elements resL1FilteredResiduals[x][y].

In-loop filter L-1 residual filter is applied as specified by the following ordered steps:

1. A variable deblockEnabled, α and β are derived as follows (as described in 7.3.5):

```
deblockEnabled = level1_filtering_enabled_flag
if (level1_filtering_signalled_flag)
    $\alpha$ = 16 − level1_filtering_first_coefficient          (36)
    $\beta$ = 16 − level1_filtering_second_coefficient         (37)
else
    $\alpha$ = 16
    $\beta$ = 16
```

2. If deblockEnabled is true, the following steps are applied given the residual representation in Figure 8:

resL1FilteredResiduals[0][0] = (resL1Residuals[0][0] * α) >> 4
resL1FilteredResiduals[0][3] = (resL1Residuals[0][3] * α) >> 4
resL1FilteredResiduals[3][0] = (resL1Residuals[3][0] * α) >> 4
resL1FilteredResiduals[3][3] = (resL1Residuals[3][3] * α) >> 4
resL1FilteredResiduals[0][1] = (resL1Residuals[0][1] * β) >> 4
resL1FilteredResiduals[0][2] = (resL1Residuals[0][2] * β) >> 4
resL1FilteredResiduals[1][0] = (resL1Residuals[1][0] * β) >> 4
resL1FilteredResiduals[2][0] = (resL1Residuals[2][0] * β) >> 4
resL1FilteredResiduals[1][3] = (resL1Residuals[1][3] * β) >> 4
resL1FilteredResiduals[2][3] = (resL1Residuals[2][3] * β) >> 4
resL1FilteredResiduals[3][1] = (resL1Residuals[3][1] * β) >> 4
resL1FilteredResiduals[3][2] = (resL1Residuals[3][2] * β) >> 4
resL1FilteredResiduals[1][1] = resL1Residuals[1][1]
resL1FilteredResiduals[1][2] = resL1Residuals[1][2]
resL1FilteredResiduals[2][1] = resL1Residuals[2][1]
resL1FilteredResiduals[2][2] = resL1Residuals[2][2] (38)

otherwise:

resL1FilteredResiduals[0][0] = resL1Residuals[0][0]
resL1FilteredResiduals[0][3] = resL1Residuals[0][3]
resL1FilteredResiduals[3][0] = resL1Residuals[3][0]
resL1FilteredResiduals[3][3] = resL1Residuals[3][3]

resL1FilteredResiduals[0][1] = resL1Residuals[0][1]
resL1FilteredResiduals[0][2] = resL1Residuals[0][2]
resL1FilteredResiduals[1][0] = resL1Residuals[1][0]
resL1FilteredResiduals[2][0] = resL1Residuals[2][0]
resL1FilteredResiduals[1][3] = resL1Residuals[1][3]
resL1FilteredResiduals[2][3] = resL1Residuals[2][3]
resL1FilteredResiduals[3][1] = resL1Residuals[3][1]
resL1FilteredResiduals[3][2] = resL1Residuals[3][2]

resL1FilteredResiduals[1][1] = resL1Residuals[1][1]
resL1FilteredResiduals[1][2] = resL1Residuals[1][2]
resL1FilteredResiduals[2][1] = resL1Residuals[2][1]
resL1FilteredResiduals[2][2] = resL1Residuals[2][2] (39)

8.10 Decoding process for base decoder data extraction

Inputs to this process are:

— a variable nCurrS (as derived in 8.8.1) specifying the size of the residual block,

— 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 IdxBaseFrame specifying the base decoder picture buffer frame from which to read the samples,

— a variable IdxPlane specifying the colour component of the current block.

Output of this process is the (nCurrS)x(nCurrS) array recDecodedBaseSamples of picture samples with elements recDecodedBaseSamples[x][y].

The process reads the block of sample (nCurrS)x(nCurrS) from the location (xCurr, yCurr) and the frame pointed by the variable IdxBaseFrame. The blocks are read in raster order.

The sample block size nCurrX and nCurrY are derived as follows:

$$\text{nCurrX} = (\text{IdxPlane} == 0)\ ?\ \text{nCurrX} : \text{nCurrX} >> \text{ShiftWidthC} \tag{40}$$

$$\text{nCurrY} = (\text{IdxPlane} == 0)\ ?\ \text{nCurrY} : \text{nCurrY} >> \text{ShiftHeightC} \tag{41}$$

8.11 Decoding process for dither filter

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,

— an (nCurrS)x(nCurrS) array recL2PictureSamples specifying the reconstructed combined output picture samples.

Output of this process is the (nCurrS)x(nCurrS) array recL2DitheredPictureSamples of residuals with elements recL2DitheredPictureSamples[x][y].

If dithering_type is equal 1 (uniform dither), the (nCurrS)x(nCurrS) block of the reconstructed sample array recL2DitheredPictureSamples at location (xCurr, yCurr) is derived as follows:

$$\text{recL2DitheredPictureSamples}[\text{xCurr} + i][\text{yCurr} + j] = \text{recL2PictureSamples}[i][j] + \text{rand}(i,j) \tag{42}$$

with i = 0...nCurrS − 1, j = 0...nCurrS − 1. rand(i,j) is pseudo_random number in the range [−dithering_strength,+dithering_strength] with dithering_strength as derived in 7.4.3.4.

9 Parsing process

9.1 Parsing process inputs and outputs, process overview

9.1.1 Parsing process for entropy encoded transform coefficients

Inputs to this process are the bits belonging to chunks of data containing the entropy encoded transform coefficients derived from the process described in 8.3.

If tile_dimensions_type is equal to 0, for each chunk the following information is provided:

— a variable nTbS specifying the size of the current transform block (nTbS = 2 if transform_type is equal to 0 and nTbS = 4 if transform_type is equal to 1),

— a variable surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag specifying if the Prefix Coding decoder is needed,

— a variable surfaces[planeIdx][levelIdx][layerIdx].size specifying the size of the chunk of data,

— if surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag is equal to true, a variable surfaces[planeIdx][levelIdx][layerIdx].data_rle specifying the beginning of the chunk containing the entropy encoded quantized transform coefficients.

— if surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag is equal to false, a variable surfaces[planeIdx][levelIdx][layerIdx].data_prefix_coding specifying the beginning of the chunk containing the entropy encoded quantized transform coefficients,

— a variable temporal_enabled_flag as derived in 7.4.3.3.

Where planeIdx, levelIdx and layerIdx indicate the plane, enhancement sub-layer and coefficient group to which the chunk belongs.

Output of this process is a quantizedCoeffBuffer of the size of (PictureWidth / nTbS, PictureHeight / nTbS) containing entropy decoded quantized transform coefficients to be used as input for processes described in 8.3.4 and in 8.3.5, in the order described in Figure 4 and the 8.3.

If temporal_enabled_flag is equal to 1, the entropy encoded quantized transform coefficients are organized in tiles of the size of (32/nTbS)x(32/nTbS). The order of writing the entropy decoded values in quantizedCoeffBuffer is depicted in Figure 15, with a raster scan order within each (32/nTbS)x(32/nTbS) CU. When the horizontal or vertical size of the surface is not a multiple of (32/nTbS), the raster scan is limited to the actual size of the last CU, which is in the horizontal dimension (PictureWidth/nTbS) % (32/nTbS), and in the vertical dimension (PictureHeight/nTbS) % (32/nTbS).

If temporal_enabled_flag is equal to 0, the entropy encoded quantized transform coefficients are organized in PictureHeight / nTbS lines of PictureWidth / nTbS elements. The order of writing is raster scan order per line.

If tile_dimensions_type is not equal to 0, the following information is provided for each chunk:

— a variable surfaces[planeIdx][levelIdx][layerIdx].tiles pointing to the tiles of the decoded picture.

— a variable surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag specifying if the Prefix Coding decoder is needed for all tiles.

In this case, a chunk of data is further split to smaller chunks of data, which are termed as tiles. For each tile, the following information is provided:

— a variable surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size specifying the size of the chunk of tile data;

— if surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag is equal to true. a variable surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_rle specifying the beginning of the chunk containing the entropy encoded quantized transform coefficients;

— if surfaces[planeIdx][levelIdx][layerIdx].rle_only_flag is equal to false, a variable surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].data_prefix_coding specifying the beginning of the chunk containing the entropy encoded quantized transform coefficients.

Where planeIdx, levelIdx, layerIdx and tileIdx indicate the plane, enhancement sub-layer, coefficient group and tile to which the chunk belongs.

The entropy encoded quantized transform coefficients are organized in tiles corresponding to an area of 32x32 samples, and each tile is scanned in raster order as described in Figure 15. The number of coefficients corresponding to the 32x32 samples is (32 / nTbS)x(32 / nTbS).

Output of this process is a buffer containing entropy decoded quantized transform coefficients organized as in Figure 15 to be used as input for processes described in 8.3.4 and in 8.3.5, in the order described in Figure 5 and 8.3.

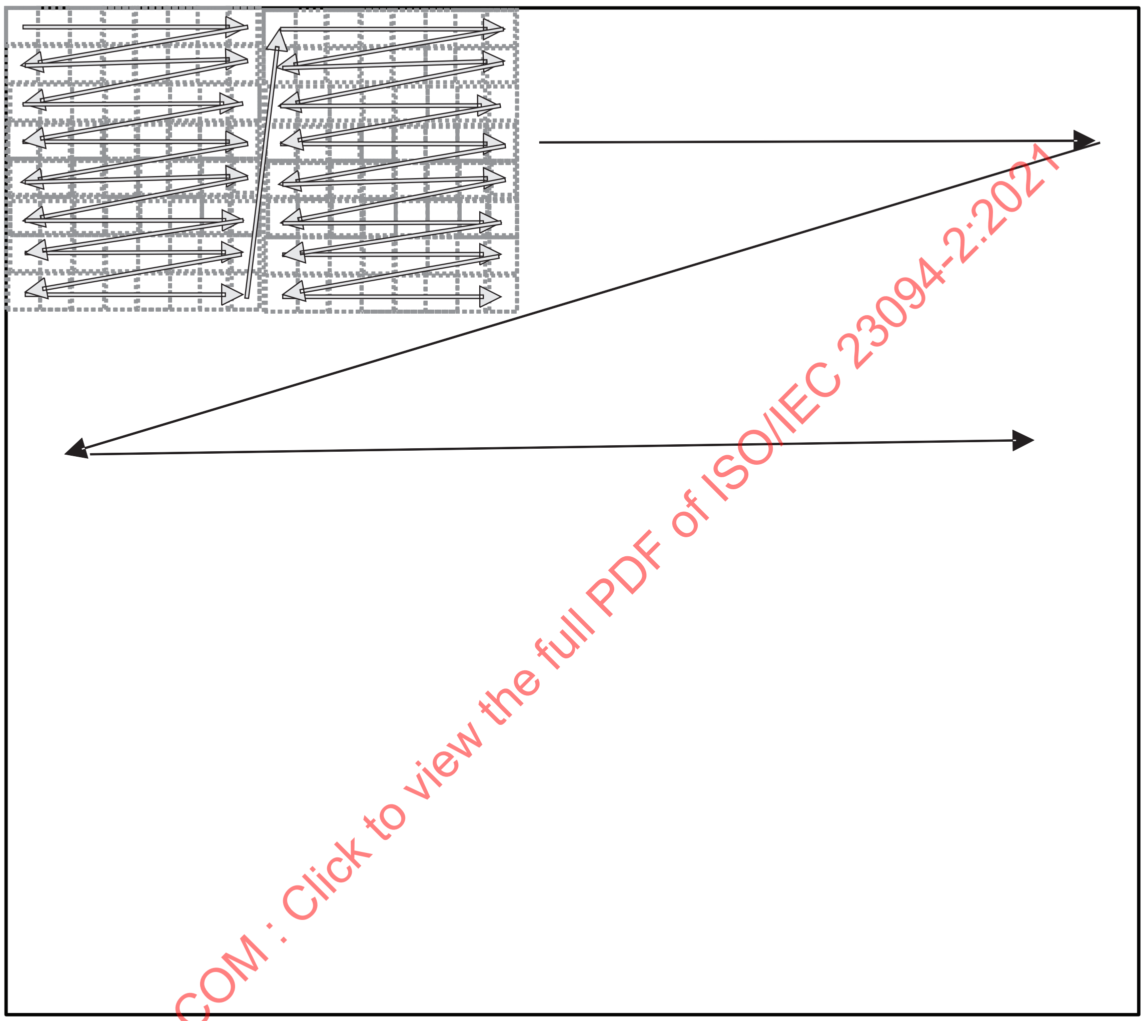


Figure 15 — Organization of the decoded transform quantized coefficients (in the example, nTbs = 4)

The entropy decoder consists of two components:

— Prefix Coding decoder described in 9.2.

— Run Length decoder described in 9.3.

The general process is described in Figure 16.

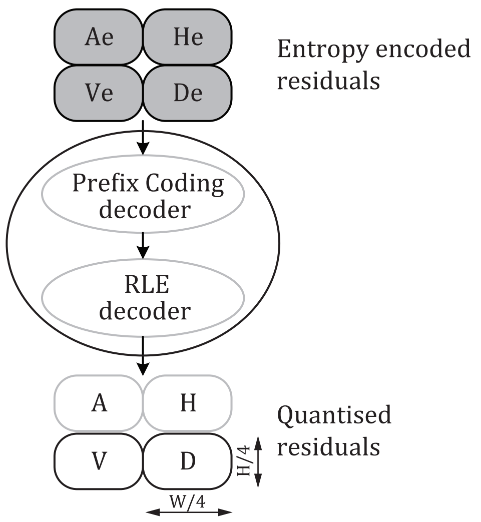


Figure 16 — Generic entropy decoder

9.1.2 Parsing process for entropy encoded temporal signal coefficient group

Inputs to this process are the bits belonging to chunks of data containing the entropy encoded temporal signal coefficient group derived from the process described in 8.3.

If tile_dimensions_type is equal to 0, for each chunk the following information are provided:

— a variable temporal_surfaces[planeIdx].rle_only_flag specifying if the Prefix Coding decoder is needed,

— a variable temporal_surfaces[planeIdx].size specifying the size of the chunk of data,

— if temporal_surfaces[planeIdx].rle_only_flag is equal to true, a variable temporal_surfaces[planeIdx].data_rle specifying the beginning of the chunk containing the entropy encoded temporal signal values,

— if temporal_surfaces[planeIdx].rle_only_flag is equal to false, a variable temporal_surfaces[planeIdx].data_prefix_coding specifying the beginning of the chunk containing the entropy encoded temporal signal values.

Where planeIdx indicates the plane to which the chunk belongs.

The output of this process is an entropy decoded temporal signal value group to be stored in TempSigSurface, as described in subsection 9.3.4.

If tile_dimensions_type is not equal to 0, the following information is provided for each chunk:

— a variable temporal_surfaces[planeIdx].tiles pointing to the tiles of the decoded picture;

— a variable temporal_surfaces[planeIdx].rle_only_flag specifying if the Prefix Coding decoder is needed for all tiles.

In this case, a chunk of data is further split to smaller chunks of data, which are termed as tiles. For each tile, the following information is provided:

— a variable temporal_surfaces[planeIdx].tiles[tileIdx].size specifying the size of the chunk of tile data;

— a variable temporal_surfaces[planeIdx].tiles[tileIdx].data specifying the beginning of the chunk containing the entropy encoded temporal signal values.

Where planeIdx and tileIdx indicate the plane and tile to which the chunk belongs.

The output of this process is an entropy decoded temporal signal coefficient group to be stored in TempSigSurface as described in 9.3.4.

The entropy decoder consists of two components:

— Prefix Coding decoder described in 9.2.

— Run Length decoder described in 9.3.

9.2 Prefix Coding decoder

9.2.1 Prefix Coding decoder description

If variable rle_only is equal to 1, the Prefix Coding decoder process is skipped, and the process described in 9.3 is invoked. If variable rle_only_flag is equal to 0, the Prefix Coding decoder is applied as specified by the following ordered steps:

1) Initialize the Prefix Coding decoder by reading the code lengths from the stream header size.

   If there are more than 31 non-zero values, the stream header is as specified in Figure 17.

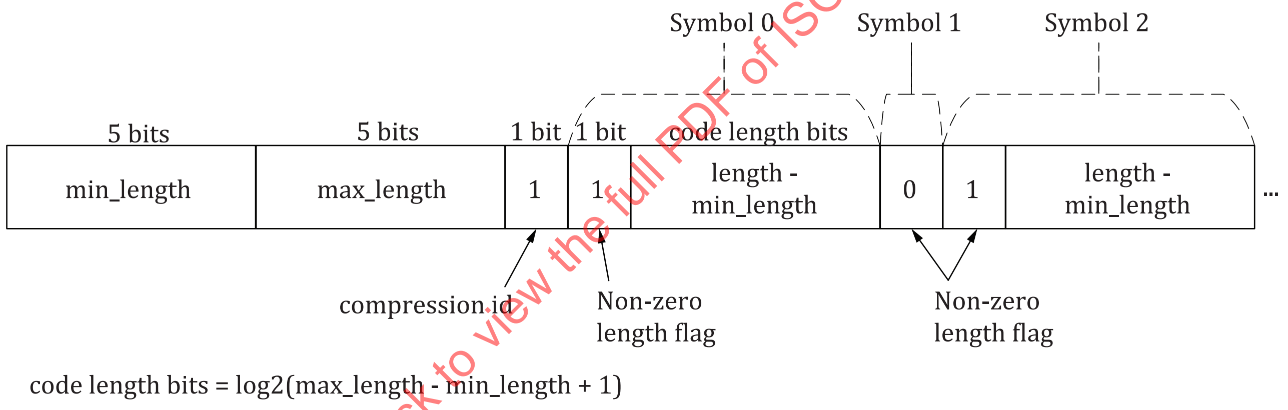


Figure 17 — Prefix coding decoder stream header for more than 31 non-zeros codes

   Otherwise the stream header is as specified in Figure 18:

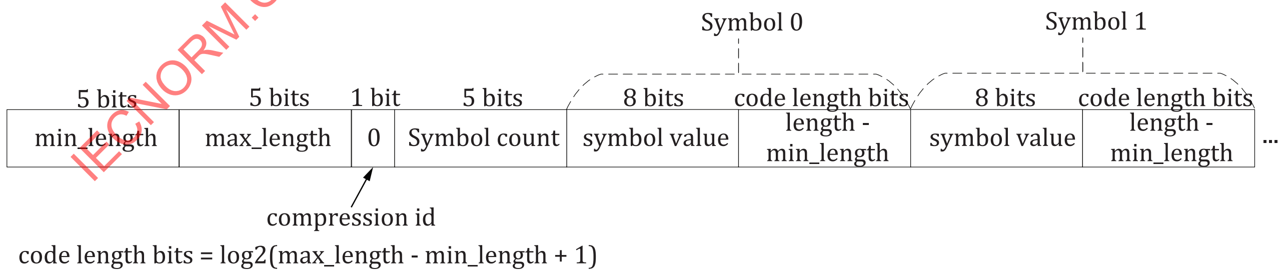


Figure 18 — Prefix coding decoder normal case

   In the special case for which the frequencies are all zero, the stream header is specified in Figure 19.

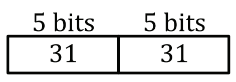


Figure 19 — Special case: Prefix Coding decoder stream header frequencies all zeros

In the special case where there is only one code in the Prefix Coding tree, the stream header is specified in Figure 20.

5 bits	5 bits	8 bits
0	0	symbol value

Figure 20 — Special case: only one code in the Prefix Coding tree

2) The code length is extracted for each symbol.

3) Assign codes to symbols from the code lengths.

4) A table is generated following the steps above in order to search the subsets of codes with identical lengths. Each element of the table records the first index of a given length and the corresponding code (firstIdx, firstCode).

9.2.2 Prefix Coding decoder table generation

To find a Prefix Coding Code for a given set of symbols a Prefix Coding tree needs to be created using the following steps:

1) First the symbols are sorted by frequency (example in Table 37). In case several symbols have the same frequency, these are ordered in descending numerical order.

Table 37 — Symbols sorted by frequency

Symbol	Frequency
A	3
B	8
C	10
D	15
E	20
F	43

2) The two lowest elements are removed from the list and made into leaves, with a parent node that has a frequency the sum of the two lower element's frequencies. The partial tree is shown in Figure 21.

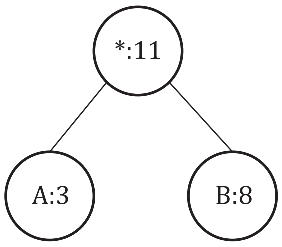


Figure 21 — Partial tree

3) After removing the lowest value in the list, the new sorted frequency table is created. Example of the new table is shown in Table 38.

Table 38 — New sorted frequency

Symbol	Frequency
C	10

Table 38 *(continued)*

Symbol	Frequency
*	11
D	15
E	20
F	43

4) The steps from 2 to 3 are repeated until only one element remains in the table (example shown in in Figure 22, Table 39, Figure 23, Table 40, Figure 24, Table 41, Figure 25 and Table 42).

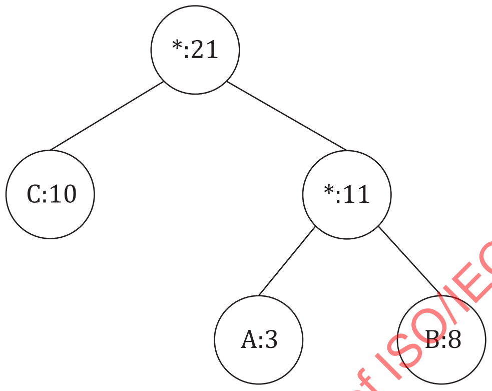


Figure 22 — Repeated loop, showing two lowest elements combined

The new list is as shown in Table 39.

Table 39 — Updated sorted frequency

Symbol	Frequency
D	15
E	20
*	21
F	43

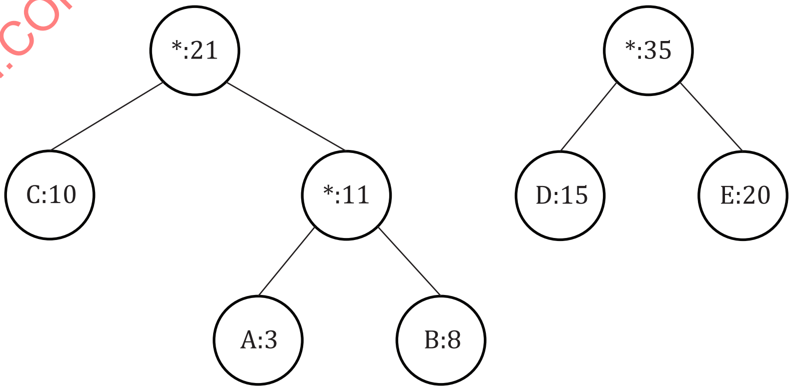


Figure 23 — Loop repeated

Table 40 — Update sorted frequency

Symbol	Frequency
*	21

Table 40 *(continued)*

Symbol	Frequency
*	35
F	43

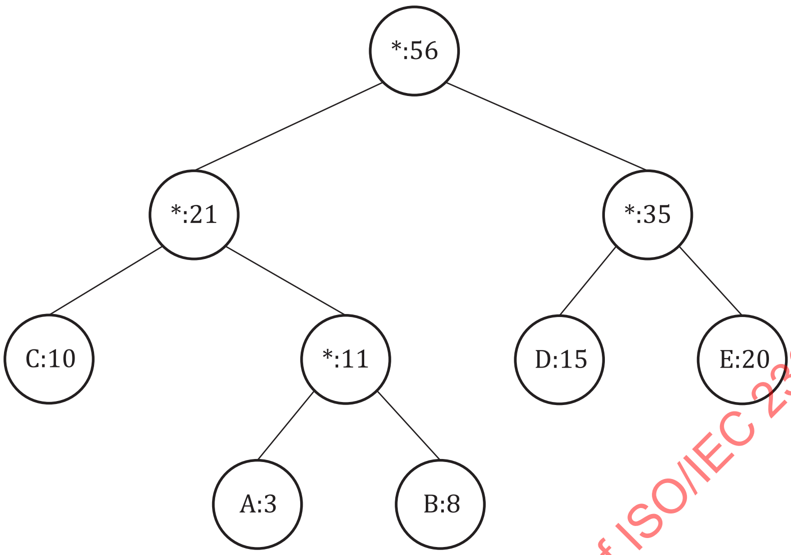


Figure 24 — Loop repeated

Table 41 — Updated sorted frequency

Symbol	Frequency
F	43
*	56

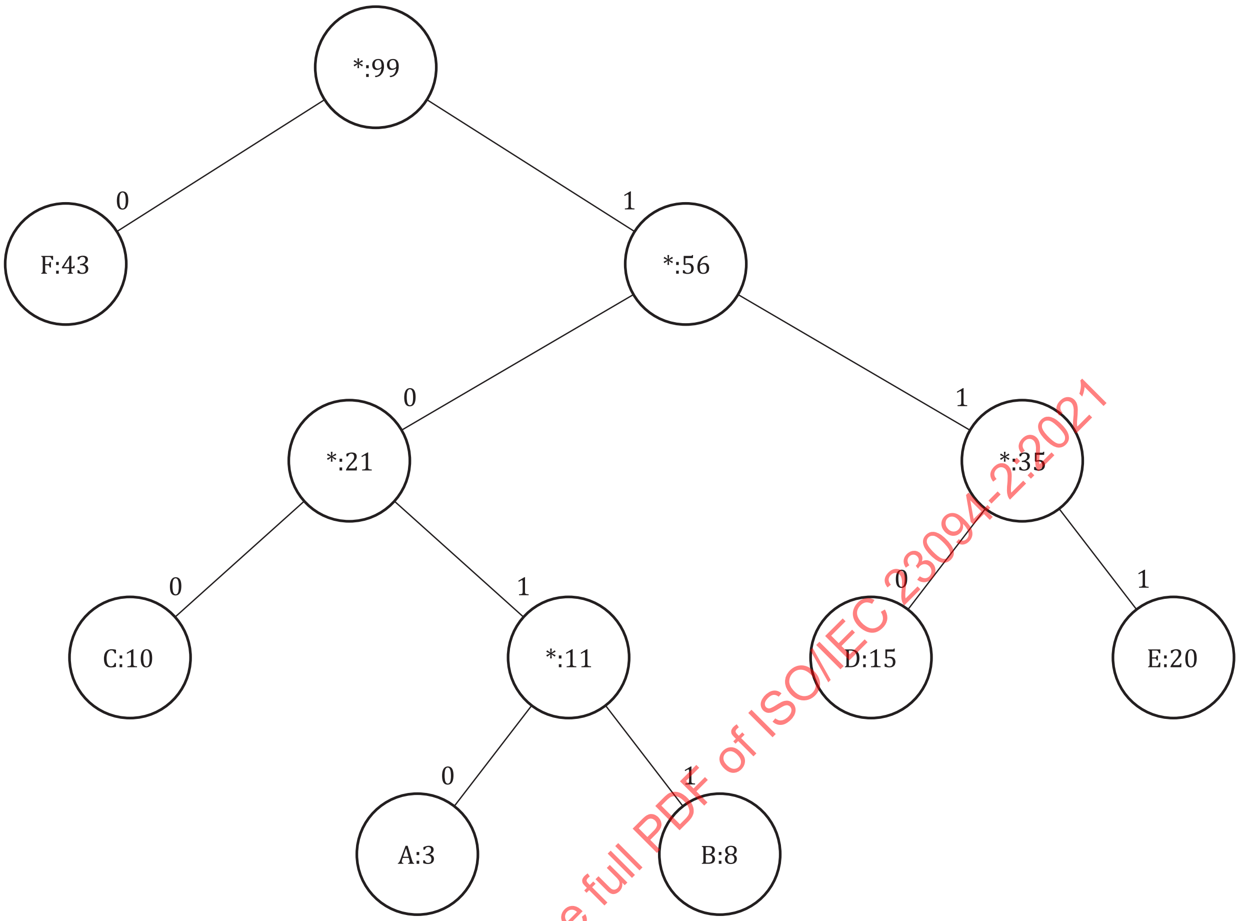


Figure 25 — Loop process completion

Table 42 — One item remaining in list

Symbol	Frequency
*	99

5) Once the tree is built, to generate the Prefix Coding code for every symbol, the tree is traversed from the root to each symbol, appending a 0 each time a left branch is taken and a 1 each time a right branch is taken. In the example above this results in the following code in Table 43.

Table 43 — Resulting Prefix Coding code

Symbol	Code	Code length
A	1010	4
B	1011	4
C	100	3
D	110	3
E	111	3
F	0	1

6) The code length of a symbol is the length of its corresponding code. To decode a Prefix Coding code, the tree is traversed beginning at the root, taking a left path if a 0 is read and a right path if a 1 is read. The symbol is found when reaching a leaf.

7) Each code related to each symbol is then passed to the process 9.3.

9.2.3 Prefix Coding decoder for tile data sizes

9.2.3.1 Prefix Coding decoder overview

The decoder reads the Prefix Coding encoded data size of each tile byte by byte. By construction, the first byte of data is guaranteed to be the least significant byte and its state the Least Significant Byte Prefix Code state. The decoder uses the state machine shown in Figure 26 to determine the state of the next byte of data. The state tells the decoder how to interpret the current byte of data as described in 9.2.3.2, and shown in Figure 26.

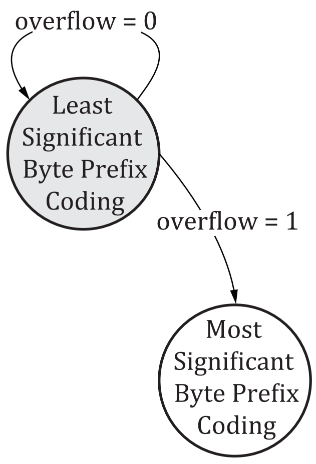

overflow = 0
Least Significant Byte Prefix Coding
overflow = 1
Most Significant Byte Prefix Coding

Figure 26 — Prefix Coding decoder state machine

9.2.3.2 Prefix Coding decoder description

The Prefix Coding for tile data sizes is described below and it has 2 states, as described is 9.2.3.1.

Least Significant Byte Prefix Coding state: This context encodes the 7 less significant bits of a non-zero value. In this state a byte is divided as shown in Figure 27.

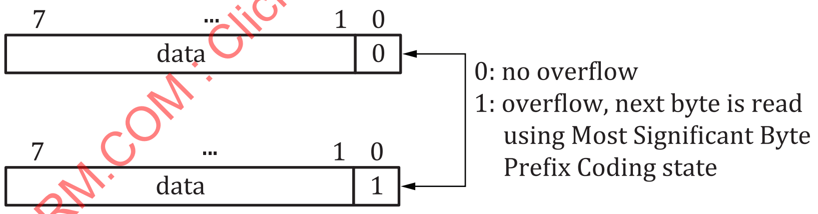

7
...
1
0
data
0
0: no overflow
1: overflow, next byte is read using Most Significant Byte Prefix Coding state
7
...
1
0
data
1

Figure 27 — Prefix Coding encoding of a byte for Least Significant Byte Prefix Coding state

The overflow bit is set if the value does not fit within 7 bits of data. When the overflow bit is set, the state of the next byte will be Most Significant Byte Prefix Coding state.

Most Significant Byte Prefix Coding state: This state encodes bits 7 to 14 of values that do not fit within 7 bits of data. Run Length encoding of a byte for Most Significant Byte Prefix Coding state is shown in Figure 28.

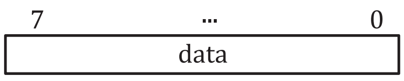


Figure 28 — Run Length encoding of a byte for Most Significant Byte Prefix Coding state

The following steps are then followed:

1) A frequency table is created for each state for use by the Prefix Coding encoder.

2) If this process is invoked with surfaces referring to entropy encoded transform coefficients, the decoded values are stored in a temporary buffer tmp_size_per_tile of size nTilesL1 or nTilesL2 (respectively, number of tiles for enhancement sub-layer 1 and sub-layer 2, as derived in 8.3.3), and are mapped to surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size as follows (planeIdx, levelIdx, layerIdx, and tileIdx have been already specified in 9.1.1):

```
if (levelIdx == 1)
    nTiles = nTilesL1
else
    nTiles = nTilesL2
if (compression_type_size_per_tile == 2) {
    for (tileIdx = 1; tileIdx < nTiles; tileIdx++) {
        tmp_size_per_tile[tileIdx] += tmp_size_per_tile[tileIdx – 1]
    }
}
for (tileIdx = 0; tileIdx < nTiles; tileIdx++) {
    surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].size = tmp_size_per_tile[tileIdx]
}
```

3) If this process is invoked with temporal_surfaces referring to an entropy encoded transform signal coefficient group, the decoded values are stored into a temporary buffer tmp_size_per_tile of size nTilesL2 (as derived in 8.3.3) and are mapped to temporal_surfaces[planeIdx].tiles[tileIdx].size as follows (planeIdx and tileIdx have been already specified in 9.1.2):

```
if (compression_type_size_per_tile == 2) {
    for (tileIdx = 1; tileIdx < nTilesL2; tileIdx++) {
        tmp_size_per_tile[tileIdx] += tmp_size_per_tile[tileIdx – 1]
    }
}
for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++) {
    temporal_surfaces[planeIdx].tiles[tileIdx].size = tmp_size_per_tile[tileIdx]
}
```

9.3 RLE decoder

9.3.1 RLE process inputs and outputs

The input of the RLE decoder is a byte stream of Prefix Coding decoded data if rle_only_flag is equal to zero or just a byte stream of raw data if rle_only_flag is equal to 1. The output of this process is a stream of quantized transform coefficients belonging to the chunk pointed to by the variables planeIdx, levelIdx and layerIdx as specified in 8.3 or the stream of temporal signals belonging to the temporal chunk.

9.3.2 RLE decoder for coefficient groups

The run length state machine is used by the Prefix Coding encoding and decoding processes to determine which Prefix Coding code to use for the current symbol or code word. The RLE decodes sequences of zeros. It also decodes the frequency tables used to build the Prefix Coding trees.

The run length decoder reads the run length encoded data byte by byte. By construction, the first byte of data is guaranteed to be the least significant byte and its state the Least Significant Byte RLE state. The decoder uses the state machine shown in Figure 29 to determine the state of the next byte of data. The state tells the decoder how to interpret the current byte of data as described in 9.3.3.

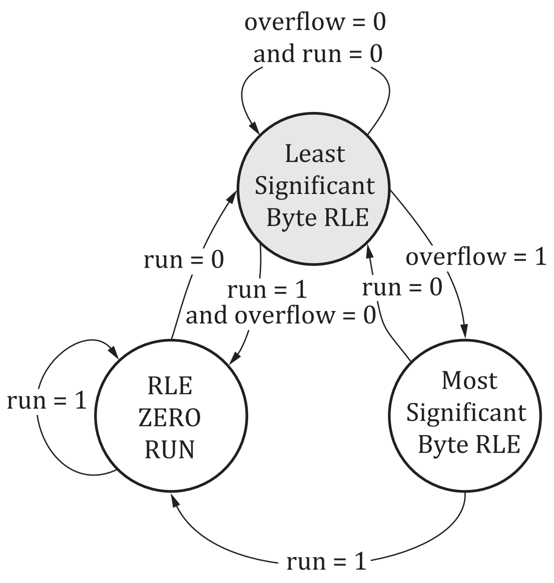


Figure 29 — RLE decoder state machine

9.3.3 RLE decoder description

The RLE consists of the following steps and has 2 states as described in 9.3.1.

Least Significant Byte RLE state: This state encodes the 6 less significant bits of a non-zero sample. In this state, a byte is divided as shown in Figure 30.

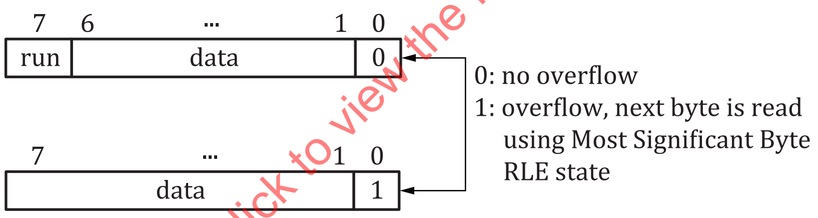

7
6
...
1
0
run
data
0
0: no overflow
1: overflow, next byte is read using Most Significant Byte RLE state
7
...
1
0
data
1

Figure 30 — Run Length encoding of a byte for Least Significant Byte RLE state

The run bit indicates that the next byte is encoding the count of a run of zeros. The overflow bit is set if the sample value does not fit within 6 bits of data. When the overflow bit is set, the state of the next byte will be Most Significant Byte RLE state. Therefore, the next state cannot be a run of zeros, and bit 7 can be used to encode data instead.

Most Significant Byte RLE state: This state encodes bits 7 to 13 of sample values that do not fit within 6 bits of data. Bit 7 encodes whether the next byte is a run of zeros. Run Length encoding of a byte for Most Significant Byte RLE state is as shown in Figure 31.

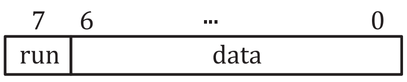


Figure 31 — Run Length encoding of a byte for Most Significant Byte RLE state

Zero Run state: This state encodes 7 bits of a zero run count. The run bit is high if more bits are needed to encode the count. Run Length encoding of a byte for Zero Run state is as shown in Figure 32.

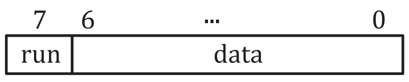


Figure 32 — Run Length encoding of a byte for Zero Run state

A frequency table is created for each state for use by the Prefix Coding encoder as described in 9.2.

For the decoder to start at a known state, the first symbol in the encoded stream always a residual.

9.3.4 RLE decoder for temporal signal coefficient group

9.3.4.1 RLE decoder overview

The Run Length state machine is used by the Prefix Coding encoding and decoding processes to determine which Prefix Coding code to use for the current symbol or code word. The RLE decodes sequences of zeros and sequences of ones. It also decodes the frequency tables used to build the Prefix Coding trees.

The Run Length Decoder reads the run length encoded data byte by byte. By construction, the first byte of data is guaranteed to be the most significant byte and its state the true value of the first symbol in the stream. The decoder uses the state machine shown in Figure 33 to determine the state of the next byte of data. The state instructs the decoder how to interpret the current byte of data as described in 9.3.4.2.

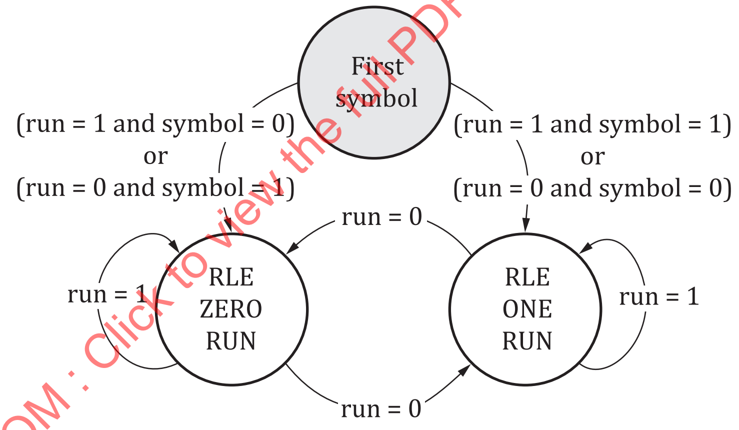

First symbol
(run = 1 and symbol = 0) or (run = 0 and symbol = 1)
(run = 1 and symbol = 1) or (run = 0 and symbol = 0)
run = 0
run = 1
RLE ZERO RUN
RLE ONE RUN
run = 1
run = 0

Figure 33 — RLE decoder state machine

9.3.4.2 RLE decoder description

The RLE consists in the following steps and has 2 states as described in 9.3.4.1.

Zero Run state: This state encodes 7 bits of a zero run count. The run bit is high if more bits are needed to encode the count. Run Length encoding of a byte for Zero Run state is as shown in Figure 34.

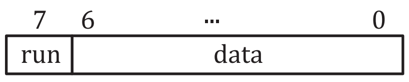


Figure 34 — Run Length encoding of a byte for Zero Run state

One Run state: This state encodes 7 bits of a one run count. The run bit is high if more bits are needed to encode the count. Run Length encoding of a byte for One Run state is as shown in Figure 35.

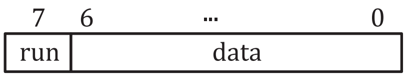


Figure 35 — Run Length encoding of a byte for One Run state

1) A frequency table is created for each state for use by the Prefix Coding encoder as described in 9.2.

2) For the decoder to start on a known state, the first symbol shall contain the real value 0 or 1.

3) The RLE decoder writes the 0 and 1 values into temporal signal surface TempSigSurface of the size (PictureWidth / nTbS, PictureHeight / nTbS) where nTbS is transform size.

4) The order of writing decoded values in the surface TempSigSurface is depicted in Figure 36, with a raster scan order within each (32 / nTbS)x(32 / nTbS) CU. When the horizontal or vertical size of the surface TempSigSurface is not a multiple of (32 / nTbS), the raster scan is limited to the actual size of the last CU that is in the horizontal dimension (PictureWidth / nTbS) % (32 / nTbS), and in the vertical dimension (PictureHeight / nTbS) % (32 / nTbS).

5) If temporal_tile_intra_signalling_enabled_flag is equal to 1 and if the value to write at the writing position (x, y) in the TempSigSurface is equal to 1 and x % (32 / nTbS) == 0 and y % (32 / nTbS) == 0, next writing position is moved to (x + 32 / nTbS, y) when (x + 32 / nTbS) < (PictureWidth / nTbs), otherwise it is moved to (0, y + 32 / nTbS), as shown in Figure 36.

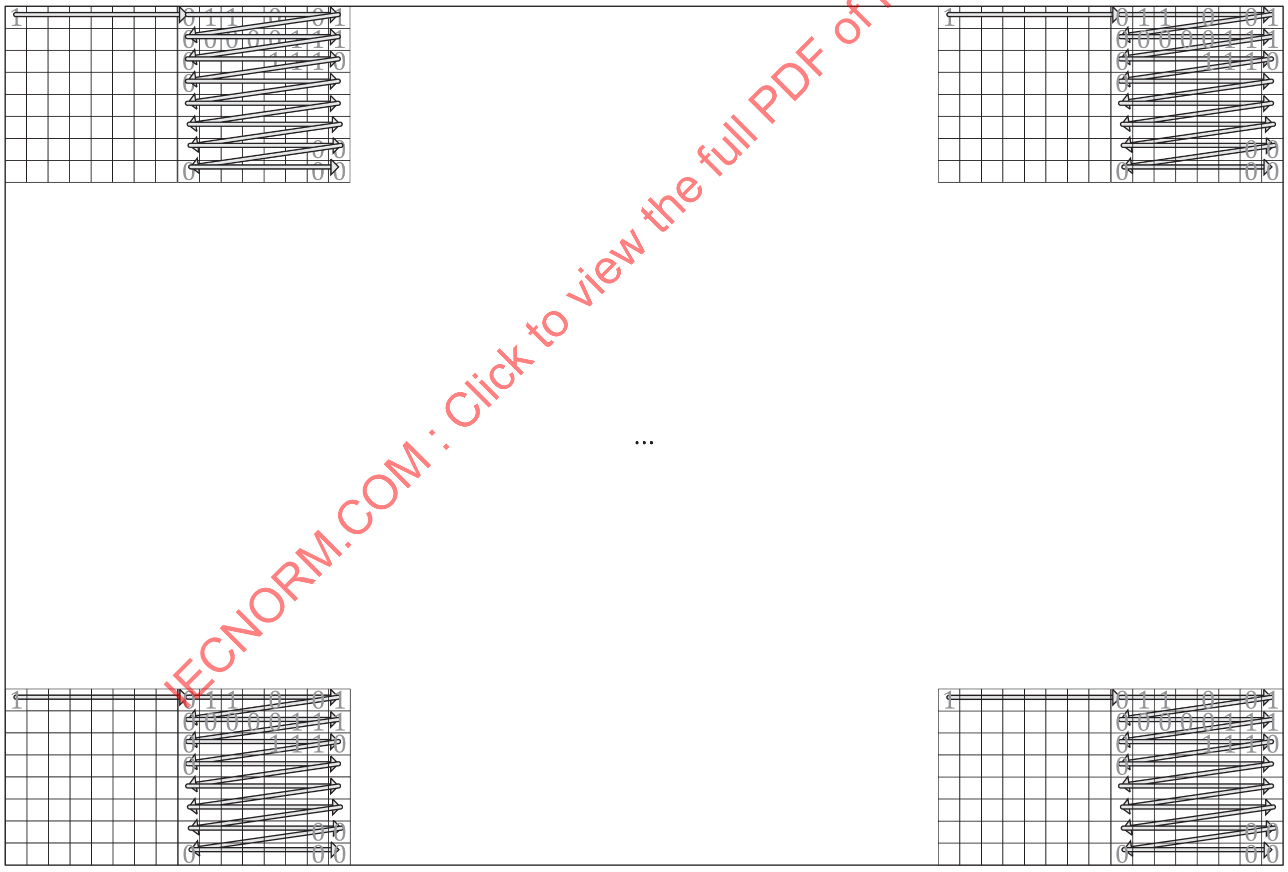


Figure 36 — Run Length Decoder writing values to the temporal signal surface (in the example, nTbs = 4)

9.3.5 RLE decoder for tile entropy_enabled_flag fields

9.3.5.1 RLE decoder overview

The Run Length state machine is used to code the entropy_enabled_flag field of each of the tiles. The RLE decodes sequences of zeros and sequences of ones.

The Run Length Decoder reads the run length encoded data byte by byte. By construction, the first byte of data is guaranteed to be the most significant byte and its state the true value of the first symbol in the stream. The decoder uses the state machine shown in Figure 37 to determine the state of the next byte of data. The state instructs the decoder how to interpret the current byte of data as described in 9.3.5.2.

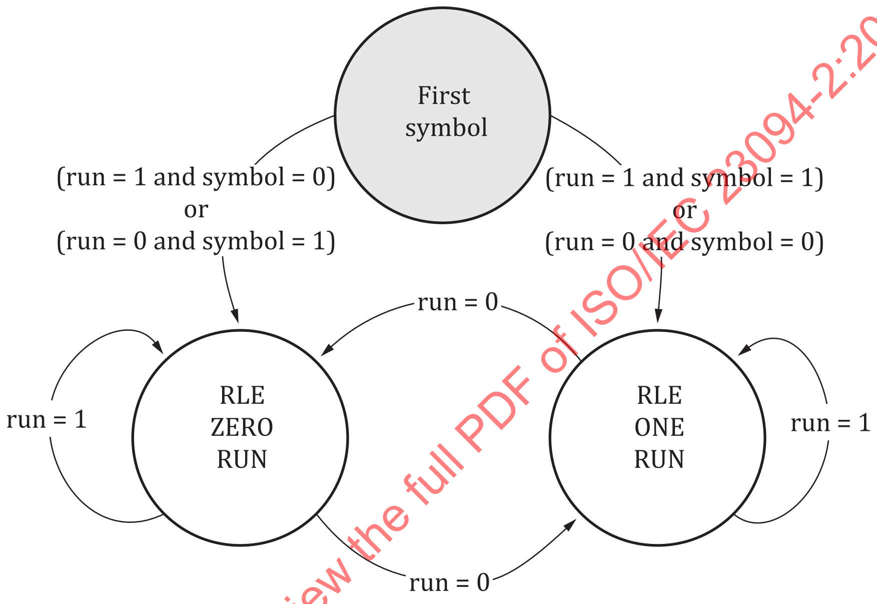


Figure 37 — RLE decoder state machine

9.3.5.2 RLE decoder description

The RLE consists in the following steps and has 2 states as described in 9.3.5.1.

Zero Run state: This state encodes 7 bits of a zero run count. The run bit is high if more bits are needed to encode the count. Run Length encoding of a byte for Zero Run state is as shown in Figure 38.

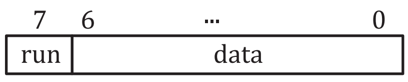


Figure 38 — Run Length encoding of a byte for Zero Run state

One Run state: This state encodes 7 bits of a one run count. The run bit is equal to 1 if more bits are needed to encode the count. Run Length encoding of a byte for One Run state is as shown in Figure 39.

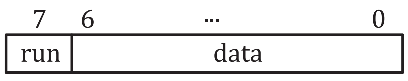


Figure 39 — Run Length encoding of a byte for One Run state

In order for the decoder to start on a known state, the first symbol shall contain the real value 0 or 1.

The RLE decoder writes the 0 and 1 values into temporary signal surface tmp_decoded_tile_entropy_enabled of size (nPlanes) x (nLevels) x (nLayers) x (nTilesL1 + nTilesL2) x (no_enhancement_bit_flag == 0) + (temporal_signalling_present_flag == 1) x (nPlanes) x (nTilesL2) and is mapped to surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag and temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag as follows:

```
for (planeIdx = 0; planeIdx < nPlanes; ++planeIdx) {
    if (no_enhancement_bit_flag == 0) {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
            if (levelIdx == 1)
                nTiles = nTilesL1
            else
                nTiles = nTilesL2
            for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                for (tileIdx = 0; tileIdx < nTiles; tileIdx++) {
                    surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag =
                        tmp_decoded_tile_entropy_enabled[tileIdx]
                }
            }
        }
    } else {
        for (levelIdx = 1; levelIdx <= nLevels; ++levelIdx) {
            if (levelIdx == 1)
                nTiles = nTilesL1
            else
                nTiles = nTilesL2
            for (layerIdx = 0; layer < nLayers; ++layerIdx) {
                for (tileIdx = 0; tileIdx < nTiles; tileIdx++)
                    surfaces[planeIdx][levelIdx][layerIdx].tiles[tileIdx].entropy_enabled_flag = 0
            }
        }
    }
    if (temporal_signalling_present_flag == 1) {
        for (tileIdx = 0; tileIdx < nTilesL2; tileIdx++) {
            temporal_surfaces[planeIdx].tiles[tileIdx].entropy_enabled_flag =
                tmp_decoded_tile_entropy_enabled[tileIdx]
        }
    }
}
```

9.4 Parsing process for 0-th order Exp-Golomb codes

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) are Exp-Golomb-coded with order 0. 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++)                                    (43)

        b = read_bits(1)
```