**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

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

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC1/SC29/WG11 N17669**

**April 2018, San Diego, CA, US**

**Title: Working Draft 1 of Versatile Video Coding**

**Source: JVET**

|  |  |
| --- | --- |
| **Joint Video Experts Team (JVET)**  **of ITU-T SG 16 WP 3 and ISO/IEC JTC 1/SC 29/WG 11**  10th Meeting: San Diego, US, 10–20 Apr. 2018 | Document: JVET-J1001-v2 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* |  | | |
| *Status:* | Output document approved by JVET | | |
| *Purpose:* | Draft text of video coding specification | | |
| *Author(s) or Contact(s):* | Benjamin Bross | Email: | firstname.lastname@hhi.fraunhofer.de |
| *Source:* | Editor | | |

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

# Abstract

Since this is an early draft, topics outside of the specific aspects that have been established by recorded meeting agreements are not included in the specification. Such aspects are to be determined by further development of the VVC project in JVET. The high-level syntax for the standard is yet to be developed. The aspects of high-level syntax in this early draft are provided only to show how certain features are likely to be controlled by some high-level syntax that may have a sequence level, a picture level, and a slice level (a picture spatial region level that includes a subset of the CTUs of the picture).

Draft 1 of Versatile Video Coding.

Ed. Notes:

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

**CONTENTS**

*Page*

Abstract i

1 Scope 1

2 Normative references 1

2.1 Identical Recommendations | International Standards 1

2.2 Paired Recommendations | International Standards equivalent in technical content 1

2.3 Additional references 1

3 Definitions 1

4 Abbreviations 6

5 Conventions 8

5.1 General 8

5.2 Arithmetic operators 8

5.3 Logical operators 8

5.4 Relational operators 8

5.5 Bit-wise operators 8

5.6 Assignment operators 9

5.7 Range notation 9

5.8 Mathematical functions 9

5.9 Order of operation precedence 11

5.10 Variables, syntax elements and tables 11

5.11 Text description of logical operations 12

5.12 Processes 13

6 Bitstream and picture formats, partitionings, scanning processes and neighbouring relationships 14

6.1 Bitstream formats 14

6.2 Source, decoded and output picture formats 14

6.3 Partitioning of pictures, slices, CTUs 16

6.3.1 Partitioning of pictures into slices 16

6.3.2 Block, quadtree and multi-type tree structures 17

6.3.3 Spatial or component-wise partitionings 17

6.4 Availability processes 17

6.4.1 Allowed binary split process 17

6.4.2 Allowed ternary split process 18

6.5 Scanning processes 19

6.5.1 CTB raster and scanning process 19

6.5.2 Up-right diagonal scan order array initialization process 19

7 Syntax and semantics 20

7.1 Method of specifying syntax in tabular form 20

7.2 Specification of syntax functions and descriptors 21

7.3 Syntax in tabular form 22

7.3.1 NAL unit syntax 22

7.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 23

7.3.3 Slice header syntax 25

7.3.4 Slice data syntax 25

7.4 Semantics 29

7.4.1 General 29

7.4.2 NAL unit semantics 29

7.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 30

7.4.4 Slice header semantics 33

7.4.5 Slice data semantics 34

# 

INTERNATIONAL STANDARD

ISO/IEC VVC

ITU-T Rec. H.VVC

ITU-T RECOMMENDATION

Versatile video coding

# Scope

This Recommendation | International Standard specifies versatile video coding.

# Normative references

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

## Identical Recommendations | International Standards

– None

## Paired Recommendations | International Standards equivalent in technical content

– None

## Additional references

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

# Definitions

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

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

* 1. **access unit**: A set of *NAL units* that are associated with each other according to a specified classification rule, are consecutive in *decoding order,* and contain exactly one *coded picture*.
  2. **AC transform coefficient**: Any *transform coefficient* for which the *frequency index* in at least one of the two dimensions is non-zero.
  3. **bin**: One bit of a *bin string*.
  4. **binarization**: A set of *bin strings* for all possible values of a *syntax element*.
  5. **binarization process**: A unique mapping process of all possible values of a *syntax element* onto a set of *bin strings*.
  6. **binary split**: A split of a rectangular MxN *block* of samples into two *blocks* where a vertical split results in a first (M / 2)xN *block* and a second (M / 2)xN *block*, and a horizontal split results in a first Mx(N / 2) *block* and a second Mx(N / 2) *block*.
  7. **bin string**: An intermediate binary representation of values of *syntax elements* from the *binarization* of the *syntax element*.
  8. **bi-predictive (B) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most two *motion vectors* and *reference indices* to *predict* the sample values of each *block*.
  9. **bitstream**: A sequence of bits, in the form of a *NAL unit stream* or a *byte stream*, that forms the representation of *coded pictures* and associated data forming one or more coded video sequences *(CVSs)*.
  10. **block**: An MxN (M-column by N-row) array of samples, or an MxN array of *transform coefficients*.
  11. **byte**: A sequence of 8 bits, within which, when written or read as a sequence of bit values, the left-most and right-most bits represent the most and least significant bits, respectively.
  12. **byte-aligned**: A position in a *bitstream* is byte-aligned when the position is an integer multiple of 8 bits from the position of the first bit in the *bitstream*, and a bit or *byte* or *syntax element* is said to be byte-aligned when the position at which it appears in a *bitstream* is byte-aligned.
  13. **byte stream**: An encapsulation of a *NAL unit stream* containing *start code prefixes* and *NAL units* as specified in Annex TBD.
  14. **can**: A term used to refer to behaviour that is allowed, but not necessarily required*.*
  15. **chroma**: An adjective, represented by the symbols Cb and Cr, specifying that a sample array or single sample is representing one of the two colour difference signals related to the primary colours.

NOTE – The term chroma is used rather than the term chrominance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term chrominance.

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

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

* 1. **intra coding**: Coding of a *coding block, slice*, or *picture* that uses *intra prediction*.
  2. **intra prediction**: A *prediction* derived from only data elements (e.g., sample values) of the same decoded *slice* without referring to a *reference picture*.
  3. **intra random access point (IRAP) access unit**: An *access unit* in which the *coded picture* is an *IRAP picture*.
  4. **intra random access point (IRAP) picture**: A *coded picture* for which each *VCL NAL unit* has nal\_unit\_type in the range of TBD, inclusive.

[Ed. (BB): IRAP NAL unit types yet to be defined (if such types will exist), pending further specification development.]

* 1. **intra (I) slice**: A *slice* that is decoded using *intra prediction* only.
  2. **leaf**: A terminating node of a tree that is a root node of a tree of depth 0.
  3. **level**: A defined set of constraints on the values that may be taken by the *syntax elements* and variables of this Specification, or the value of a *transform coefficient* prior to *scaling*.

NOTE – The same set of levels is defined for all profiles, with most aspects of the definition of each level being in common across different profiles. Individual implementations may, within the specified constraints, support a different level for each supported profile.

* 1. **list 0 (list 1) motion vector**: A *motion vector* associated with a *reference index* pointing into *reference picture list 0* (*list 1*).
  2. **list 0 (list 1) prediction**: *Inter prediction* of the content of a *slice* using a *reference index* pointing into *reference picture list 0* (*list 1*).
  3. **luma**: An adjective, represented by the symbol or subscript Y or L, specifying that a sample array or single sample is representing the monochrome signal related to the primary colours.

NOTE – The term luma is used rather than the term luminance in order to avoid the implication of the use of linear light transfer characteristics that is often associated with the term luminance. The symbol L is sometimes used instead of the symbol Y to avoid confusion with the symbol y as used for vertical location.

* 1. **may**: A term that is used to refer to behaviour that is allowed, but not necessarily required*.*

NOTE – In some places where the optional nature of the described behaviour is intended to be emphasized, the phrase "may or may not" is used to provide emphasis.

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

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

* 1. **picture parameter set (PPS)**: A *syntax structure* containing *syntax elements* that apply to zero or more entire *coded pictures* as determined by a *syntax element* found in each *slice header.*
  2. **picture order count (POC)**: A variable that is associated with each *picture*, uniquely identifies the associated *picture* among all *pictures* in the *CVS*, and, when the associated *picture* is to be output from the *decoded picture buffer*, indicates the position of the associated *picture* in *output order* relative to the *output order* positions of the other *pictures* in the same *CVS* that are to be output from the *decoded picture buffer*.
  3. **prediction**: An embodiment of the *prediction process*.
  4. **prediction process**: The use of a *predictor* to provide an estimate of the data element (e.g., sample value or motion vector) currently being decoded.
  5. **predictive (P) slice**: A *slice* that is decoded using *intra* *prediction* or using *inter prediction* with at most one *motion vector* and *reference index* to *predict* the sample values of each *block*.
  6. **predictor**: A combination of specified values or previously decoded data elements (e.g., sample value or motion vector) used in the *decoding process* of subsequent data elements.
  7. **profile**: A specified subset of the syntax of this Specification.
  8. **pulse code modulation (PCM)**: Coding of the samples of a *block* by directly representing the sample values without *prediction* or application of a transform.
  9. **quadtree**: A *tree* in which a parent node can be split into four child nodes, each of which may become parent node for another split into four child nodes.
  10. **quantization parameter**: A variable used by the *decoding process* for *scaling* of *transform coefficient levels*.
  11. **random access**: The act of starting the decoding process for a *bitstream* at a point other than the beginning of the stream.
  12. **raster scan**: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., rows of the pattern (going down) each scanned from left to right.
  13. **raw byte sequence payload (RBSP)**: A *syntax structure* containing an integer number of *bytes* that is encapsulated in a *NAL unit* and that is either empty or has the form of a *string of data bits* containing *syntax elements* followed by an *RBSP stop bit* and zero or more subsequent bits equal to 0.
  14. **raw byte sequence payload (RBSP) stop bit**: A bit equal to 1 present within a *raw byte sequence payload (RBSP)* after a *string of data bits*, for which the location of the end within an *RBSP* can be identified by searching from the end of the *RBSP* for the *RBSP stop bit*, which is the last non-zero bit in the *RBSP.*
  15. **reference index**: An index into a *reference picture list*.
  16. **reference picture**: A *picture* that is a *short-term reference picture*.

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

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

NOTE – For the decoding process of a P slice, there is one reference picture list – reference picture list 0. For the decoding process of a B slice, there are two reference picture lists – reference picture list 0 and reference picture list 1.

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

NOTE – When used to express a mandatory constraint on the values of syntax elements or on the results obtained by operation of the specified decoding process, it is the responsibility of the encoder to ensure that the constraint is fulfilled. When used in reference to operations performed by the decoding process, any decoding process that produces identical cropped decoded pictures to those output from the decoding process described in this Specification conforms to the decoding process requirements of this Specification.

* 1. **short-term reference picture**: A *picture* that is marked as "used for short-term reference".
  2. **should**: A term used to refer to behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this Specification.
  3. **slice**: An integer number of *CTUs* ordered consecutively in the *raster scan* and contained in a single *NAL unit*.
  4. **slice header**: A part of a coded *slice* containing the data elements pertaining to the first or all *CTUs* represented in the *slice*.
  5. **source**: A term used to describe the video material or some of its attributes before encoding.
  6. **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

NOTE – The location of a start code prefix can be used by a decoder to identify the beginning of a new NAL unit and the end of a previous NAL unit. Emulation of start code prefixes is prevented within NAL units by the inclusion of emulation prevention bytes.

* 1. **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit*, where the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.
  2. **syntax element**: An element of data represented in the *bitstream*.
  3. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
  4. **ternary split**: A split of a rectangular MxN *block* of samples into three *blocks* where a vertical split results in a first (M / 4)xN *block*, a second (M / 2)xN *block*, a third (M / 4)xN *block*, and a horizontal split results in a first Mx(N / 4) *block*, a second Mx(N / 2) *block*, a third Mx(N / 4) *block*.
  5. **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.
  6. **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.
  7. **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in a *transform* in the *decoding process*.
  8. **transform coefficient level**: An integer quantity representing the value associated with a particular two‑dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.
  9. **transform unit (TU)**: A *transform block* of *luma* samples and two corresponding *transform blocks* of *chroma* samples of a *picture* and *syntax structures* used to transform the *transform block* samples.
  10. **tree**: A tree is a finite set of nodes with a unique root node.
  11. **unspecified**: A term that may be used to specify some values of a particular *syntax element* to indicate that the values have no specified meaning in this Specification and will not have a specified meaning in the future as an integral part of future versions of this Specification.
  12. **video coding layer (VCL) NAL unit**: A collective term for *coded slice NAL units* and the subset of *NAL units* that have *reserved* values of nal\_unit\_type that are classified as VCL NAL units in this Specification.

# Abbreviations

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

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

B Bi-predictive

CABAC Context-based Adaptive Binary Arithmetic Coding

CB Coding Block

CBR Constant Bit Rate

CPB Coded Picture Buffer

CRC Cyclic Redundancy Check

CTB Coding Tree Block

CTU Coding Tree Unit

CU Coding Unit

CVS Coded Video Sequence

DPB Decoded Picture Buffer

EG Exponential-Golomb

EGk k-th order Exponential-Golomb

FCC Federal Communications Commission (of the United States)

FIFO First-In, First-Out

FIR Finite Impulse Response

FL Fixed-Length

GBR Green, Blue and Red

I Intra

IRAP Intra Random Access Point

LPS Least Probable Symbol

LSB Least Significant Bit

MPS Most Probable Symbol

MSB Most Significant Bit

MVP Motion Vector Prediction

NAL Network Abstraction Layer

NTSC National Television System Committee (of the United States)

P Predictive

PCM Pulse Code Modulation

POC Picture Order Count

PPS Picture Parameter Set

QP Quantization Parameter

RBSP Raw Byte Sequence Payload

RGB Same as GBR

RPS Reference Picture Set

SAR Sample Aspect Ratio

SEI Supplemental Enhancement Information

SMPTE Society of Motion Picture and Television Engineers

SODB String Of Data Bits

SPS Sequence Parameter Set

TR Truncated Rice

UCS Universal Coded Character Set

UTF UCS Transmission Format

VBR Variable Bit Rate

VCL Video Coding Layer

# Conventions

## General

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

## Arithmetic operators

The following arithmetic operators are defined as follows:

|  |  |
| --- | --- |
| + | Addition |
| − | Subtraction (as a two-argument operator) or negation (as a unary prefix operator) |
| \* | Multiplication, including matrix multiplication |
| xy | Exponentiation. Specifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation. |
| / | Integer division with truncation of the result toward zero. For example, 7 / 4 and −7 / −4 are truncated to 1 and −7 / 4 and 7 / −4 are truncated to −1. |
| ÷ | Used to denote division in mathematical equations where no truncation or rounding is intended. |
|  | Used to denote division in mathematical equations where no truncation or rounding is intended. |
|  | The summation of f( i ) with i taking all integer values from x up to and including y. |
| x % y | Modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0. |

## Logical operators

The following logical operators are defined as follows:

x && y Boolean logical "and" of x and y

x | | y Boolean logical "or" of x and y

! Boolean logical "not"

x ? y : z If x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z.

## Relational operators

The following relational operators are defined as follows:

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

= = Equal to

!= Not equal to

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

## Bit-wise operators

The following bit-wise operators are defined as follows:

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

| Bit-wise "or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

^ Bit-wise "exclusive or". When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

x >> y Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y. Bits shifted into the most significant bits (MSBs) as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

x << y Arithmetic left shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y. Bits shifted into the least significant bits (LSBs) as a result of the left shift have a value equal to 0.

## Assignment operators

The following arithmetic operators are defined as follows:

= Assignment operator

+ + Increment, i.e., *x*+ + is equivalent to *x* = *x* + 1; when used in an array index, evaluates to the value of the variable prior to the increment operation.

− − Decrement, i.e., *x*− − is equivalent to *x* = *x* − 1; when used in an array index, evaluates to the value of the variable prior to the decrement operation.

+= Increment by amount specified, i.e., x += 3 is equivalent to x = x + 3, and x += (−3) is equivalent to x = x + (−3).

−= Decrement by amount specified, i.e., x −= 3 is equivalent to x = x − 3, and x −= (−3) is equivalent to x = x − (−3).

## Range notation

The following notation is used to specify a range of values:

x = y..z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers and z being greater than y.

## Mathematical functions

The following mathematical functions are defined:

Abs( x ) = (5‑1)

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

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

Atan2( y, x ) = (5‑4)

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

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

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

Clip3( x, y, z ) = (5‑8)

Cos( x ) the trigonometric cosine function operating on an argument x in units of radians. (5‑9)

Floor( x ) the largest integer less than or equal to x. (5‑10)

GetCurrMsb( a, b, c, d ) = (5‑11)

Ln( x ) the natural logarithm of x (the base-e logarithm, where e is the natural logarithm base constant 2.718 281 828...). (5‑12)

Log2( x ) the base-2 logarithm of x. (5‑13)

Log10( x ) the base-10 logarithm of x. (5‑14)

Min( x, y ) = (5‑15)

Max( x, y ) = (5‑16)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑17)

Sign( x ) = (5‑18)

Sin( x ) the trigonometric sine function operating on an argument x in units of radians (5‑19)

Sqrt( x ) = (5‑20)

Swap( x, y ) = ( y, x ) (5‑21)

Tan( x ) the trigonometric tangent function operating on an argument x in units of radians (5‑22)

## Order of operation precedence

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

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

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

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

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

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

|  |
| --- |
| **operations (with operands x, y, and z)** |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| xy |
| "x \* y", "x / y", "x ÷ y", "", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x  <<  y", "x  >>  y" |
| "x < y", "x  <=  y", "x > y", "x  >=  y" |
| "x  = =  y", "x  !=  y" |
| "x & y" |
| "x | y" |
| "x  &&  y" |
| "x  | |  y" |
| "x ? y : z" |
| "x..y" |
| "x = y", "x  +=  y", "x  −=  y" |

## Variables, syntax elements and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower case letters with underscore characters), and one descriptor for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e., not bold) type.

In some cases the syntax tables may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower case and upper case letter and without any underscore characters. Variables starting with an upper case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower case letter are only used within the clause in which they are derived.

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in clause 7.2 and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in clause 5.8) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[ x ][ y ] or as syx. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[ x ].

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

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

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

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

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

## Text description of logical operations

In the text, a statement of logical operations as would be described mathematically in the following form:

if( condition 0 )  
 statement 0  
else if( condition 1 )  
 statement 1  
...  
else /\* informative remark on remaining condition \*/  
 statement n

may be described in the following manner:

... as follows / ... the following applies:

– If condition 0, statement 0

– Otherwise, if condition 1, statement 1

– ...

– Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".

In the text, a statement of logical operations as would be described mathematically in the following form:

if( condition 0a && condition 0b )  
 statement 0  
else if( condition 1a | | condition 1b )  
 statement 1  
...  
else  
 statement n

may be described in the following manner:

... as follows / ... the following applies:

– If all of the following conditions are true, statement 0:

– condition 0a

– condition 0b

– Otherwise, if one or more of the following conditions are true, statement 1:

– condition 1a

– condition 1b

– ...

– Otherwise, statement n

In the text, a statement of logical operations as would be described mathematically in the following form:

if( condition 0 )  
 statement 0  
if( condition 1 )  
 statement 1

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

## Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and upper case variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification may also have a lower case variable explicitly specified as input. Each process specification has explicitly specified an output. The output is a variable that can either be an upper case variable or a lower case variable.

When invoking a process, the assignment of variables is specified as follows:

– If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower case input or output variables of the process specification.

– Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

In the specification of a process, a specific coding block may be referred to by the variable name having a value equal to the address of the specific coding block.

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

## Bitstream formats

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

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

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

## Source, decoded and output picture formats

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

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

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

– Luma (Y) only (monochrome).

– Luma and two chroma (YCbCr or YCgCo).

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

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

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

The variables SubWidthC and SubHeightC are specified in Table 6‑1, depending on the chroma format sampling structure, which is specified through chroma\_format\_idc and separate\_colour\_plane\_flag. Other values of chroma\_format\_idc, SubWidthC and SubHeightC may be specified in the future by ITU‑T | ISO/IEC.

Table 6‑1 – SubWidthC and SubHeightC values derived from  
chroma\_format\_idc and separate\_colour\_plane\_flag

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **chroma\_format\_idc** | **separate\_colour\_plane\_flag** | **Chroma format** | **SubWidthC** | **SubHeightC** |
| 0 | 0 | Monochrome | 1 | 1 |
| 1 | 0 | 4:2:0 | 2 | 2 |
| 2 | 0 | 4:2:2 | 2 | 1 |
| 3 | 0 | 4:4:4 | 1 | 1 |
| 3 | 1 | 4:4:4 | 1 | 1 |

In monochrome sampling there is only one sample array, which is nominally considered the luma array.

In 4:2:0 sampling, each of the two chroma arrays has half the height and half the width of the luma array.

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

In 4:4:4 sampling, depending on the value of separate\_colour\_plane\_flag, the following applies:

– If separate\_colour\_plane\_flag is equal to 0, each of the two chroma arrays has the same height and width as the luma array.

– Otherwise (separate\_colour\_plane\_flag is equal to 1), the three colour planes are separately processed as monochrome sampled pictures.

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

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



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

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



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

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



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

## Partitioning of pictures, slices, CTUs

### Partitioning of pictures into slices

This subclause specifies how a picture is partitioned into slices. Pictures are divided into slices. A slice is a a sequence of CTUs.

For example, a picture may be divided into two slices as shown in Figure 6‑4. In this example, the first slice contains 60 CTUs and the second slice contains the remaining 39 CTUs of the picture.

When a picture is coded using three separate colour planes (separate\_colour\_plane\_flag is equal to 1), a slice contains only CTUs of one colour component being identified by the corresponding value of colour\_plane\_id, and each colour component array of a picture consists of slices having the same colour\_plane\_id value. Coded slices with different values of colour\_plane\_id within a picture may be interleaved with each other under the constraint that for each value of colour\_plane\_id, the coded slice NAL units with that value of colour\_plane\_id shall be in the order of increasing CTU address in raster scan order for the first CTU of each coded slice NAL unit.

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



Figure 6‑4 – A picture with 11 by 9 luma CTUs that is partitioned into two slices (informative)

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

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

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

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

The coding block is the root node of two trees, the prediction tree and the transform tree. The prediction tree specifies the position and size of prediction blocks. The transform tree specifies the position and size of transform blocks. The splitting information for luma and chroma is identical for the prediction tree and may or may not be identical for the transform tree.

The blocks and associated syntax structures are grouped into "unit" structures as follows:

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

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

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

### Spatial or component-wise partitionings

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

– The division of each picture into components

– The division of each component into CTBs

– The division of each picture into slices

– The division of each slice into CTUs

– The division of each CTU into CTBs

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

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

– The division of each coding unit into transform units

– The division of each coding unit into coding blocks

– The division of each coding block into transform blocks

– The division of each transform unit into transform blocks.

## Availability processes

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

### Allowed binary split process

Input to this process is a binary split mode btSplit, a coding block width cbWidth, a coding block height cbHeight, a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, a multi-type tree depth mttDepth and a partition index partIdx.

Output of this process is the variable allowBtSplit.

Table 6‑2 – Specification of parallelTtSplit, perpendicularBtSplit, cbSize, partOffsetX and partOffsetY based on btSplit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **btSplit** | **parallelTtSplit** | **perpendicularBtSplit** | **cbSize** | **partOffsetX** | **partOffsetY** |
| SPLIT\_BT\_VER | SPLIT\_TT\_VER | SPLIT\_BT\_HOR | cbWidth | 0 | cbHeight |
| SPLIT\_BT\_HOR | SPLIT\_TT\_HOR | SPLIT\_BT\_VER | cbHeight | cbWidth | 0 |

The variables parallelTtSplit, perpendicularBtSplit, cbSize, partOffsetX and partOffsetY are derived as specified in  Table 6‑2.

The variable allowBtSplit is derived as follows:

* If one or more of the following conditions are true, allowBtSplit is set equal to FALSE:
* cbSize is less than or equal to MinBtSizeY
* cbWidth is greater than MaxBtSizeY
* cbHeight is greater than MaxBtSizeY
* mttDepth is greater than or equal to MaxMttDepth

Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE:

* mttDepth is greater than 0
* partIdx is equal to 1
* MttSplitMode[ x0 ][ y0 ][ mttDepth − 1 ] is equal to parallelTtSplit
* Otherwise if all of the following conditions are true, allowBtSplit is set equal to FALSE:
* slice\_type is not equal to I
* mttDepth is equal to 1
* cbSize is greater than 1  <<  MinQtLog2SizeY
* partIdx is equal to 1
* MttSplitMode[ x0 ][ y0 ][ mttDepth − 1 ] is equal to perpendicularBtSplit
* MttSplitMode[ x0 − partOffsetX ][ y0 − partOffsetY ][ mttDepth ] is equal to btSplit

– Otherwise, allowBtSplit is set equal to TRUE.

### Allowed ternary split process

Input to this process is a ternary split mode ttSplit, a coding block width cbWidth, a coding block height cbHeight, a location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture, a multi-type tree depth mttDepth and a partition index partIdx.

Output of this process is the variable allowTtSplit.

Table 6‑3 – Specification of parallelTtSplit, perpendicularBtSplit, cbSize, partOffsetX and partOffsetY based on btSplit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ttSplit** | **parallelTtSplit** | **perpendicularBtSplit** | **cbSize** | **partOffsetX** | **partOffsetY** |
| SPLIT\_TT\_VER | SPLIT\_TT\_VER | SPLIT\_BT\_HOR | cbWidth | 0 | cbHeight |
| SPLIT\_TT\_HOR | SPLIT\_TT\_HOR | SPLIT\_BT\_VER | cbHeight | cbWidth | 0 |

The variables parallelTtSplit, perpendicularBtSplit, cbSize, partOffsetX and partOffsetY are derived as specified in  Table 6‑2.

The variable allowTtSplit is derived as follows:

* If one or more of the following conditions are true, allowSTtplit is set equal to FALSE:
* cbSize is less than or equal to 2 \* MinTtSizeY
* cbWidth is greater than MaxTtSizeY
* cbHeight is greater than MaxTtSizeY
* mttDepth is greater than or equal to MaxMttDepth
* Otherwise if all of the following conditions are true, allowTtSplit is set equal to FALSE:
* slice\_type is not equal to I
* mttDepth is greater than 0
* partIdx is equal to 1
* MttSplitMode[ x0 ][ y0 ][ mttDepth − 1 ] is equal to perpendicularBtSplit
* MttSplitMode[ x0 − partOffsetX ][ y0 − partOffsetY ][ mttDepth ] is equal to parallelTtSplit

– Otherwise, allowTtSplit is set equal to TRUE.

## Scanning processes

### CTB raster and scanning process

[Ed. (BB): Define appropriate scanning process.]

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

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

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

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

# Syntax and semantics

## Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed bitstreams. Additional constraints on the syntax may be specified, either directly or indirectly, in other clauses.

NOTE – An actual decoder should implement some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this Specification.

The following table lists examples of the syntax specification format. When **syntax\_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

|  |  |
| --- | --- |
|  | Descriptor |
| /\* A statement can be a syntax element with an associated descriptor or can be an expression used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples \*/ |  |
| **syntax\_element** | ue(v) |
| conditioning statement |  |
|  |  |
| /\* A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. \*/ |  |
| { |  |
| statement |  |
| statement |  |
| ... |  |
| } |  |
|  |  |
| /\* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true \*/ |  |
| while( condition ) |  |
| statement |  |
|  |  |
| /\* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true \*/ |  |
| do |  |
| statement |  |
| while( condition ) |  |
|  |  |
| /\* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed \*/ |  |
| if( condition ) |  |
| primary statement |  |
| else |  |
| alternative statement |  |
|  |  |
| /\* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. \*/ |  |
| for( initial statement; condition; subsequent statement ) |  |
| primary statement |  |

## Specification of syntax functions and descriptors

The functions presented here are used in the syntactical description. These functions are expressed in terms of the value of a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte\_aligned( ) is specified as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

– ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause TBD.

– b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read\_bits( 8 ).

– f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ).

– i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.

– se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause TBD.

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

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

– u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.

– ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause TBD.

## Syntax in tabular form

### NAL unit syntax

#### General NAL unit syntax

|  |  |
| --- | --- |
| nal\_unit( NumBytesInNalUnit ) { | Descriptor |
| nal\_unit\_header( ) |  |
| NumBytesInRbsp = 0 |  |
| for( i = 2; i < NumBytesInNalUnit; i++ ) |  |
| if( i + 2 < NumBytesInNalUnit && next\_bits( 24 ) = = 0x000003 ) { |  |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| i += 2 |  |
| **emulation\_prevention\_three\_byte** /\* equal to 0x03 \*/ | f(8) |
| } else |  |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| } |  |

#### NAL unit header syntax

|  |  |
| --- | --- |
| nal\_unit\_header( ) { | Descriptor |
| **forbidden\_zero\_bit** | f(1) |
| **nal\_unit\_type** | u(7) |
| } |  |

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

#### Sequence parameter set RBSP syntax

[Ed. (BB): Preliminary basic SPS, subject to further study and pending further specification development.]

|  |  |
| --- | --- |
| seq\_parameter\_set\_rbsp( ) { | Descriptor |
| **sps\_seq\_parameter\_set\_id** | ue(v) |
| **chroma\_format\_idc** | ue(v) |
| if( chroma\_format\_idc = = 3 ) |  |
| **separate\_colour\_plane\_flag** | u(1) |
| **pic\_width\_in\_luma\_samples** | ue(v) |
| **pic\_height\_in\_luma\_samples** | ue(v) |
| **bit\_depth\_luma\_minus8** | ue(v) |
| **bit\_depth\_chroma\_minus8** | ue(v) |
| **log2\_ctu\_size\_minus2** | ue(v) |
| **log2\_min\_qt\_size\_intra\_slices\_minus2** | ue(v) |
| **log2\_min\_qt\_size\_inter\_slices\_minus2** | ue(v) |
| **max\_mtt\_hierarchy\_depth\_inter\_slices** | ue(v) |
| **max\_mtt\_hierarchy\_depth\_intra\_slices** | ue(v) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Picture parameter set RBSP syntax

[Ed. (BB): Preliminary basic PPS, subject to further study and pending further specification development.]

|  |  |
| --- | --- |
| pic\_parameter\_set\_rbsp( ) { | Descriptor |
| **pps\_pic\_parameter\_set\_id** | ue(v) |
| **pps\_seq\_parameter\_set\_id** | ue(v) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Access unit delimiter RBSP syntax

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

#### End of sequence RBSP syntax

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

#### End of bitstream RBSP syntax

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

#### Filler data RBSP syntax

|  |  |
| --- | --- |
| filler\_data\_rbsp( ) { | Descriptor |
| while( next\_bits( 8 ) = = 0xFF ) |  |
| **ff\_byte** /\* equal to 0xFF \*/ | f(8) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

#### Slice layer RBSP syntax

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

#### RBSP slice trailing bits syntax

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

#### RBSP trailing bits syntax

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

#### Byte alignment syntax

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

### Slice header syntax

[Ed. (BB): Preliminary basic slice header, subject to further study and pending further specification development.]

|  |  |
| --- | --- |
| slice\_header( ) { | Descriptor |
| **slice\_pic\_parameter\_set\_id** | ue(v) |
| **slice\_address** | u(v) |
| **slice\_type** | ue(v) |
| if ( slice\_type != I ) |  |
| **log2\_diff\_ctu\_max\_bt\_size** | ue(v) |
| byte\_alignment( ) |  |
| } |  |

### Slice data syntax

#### General slice data syntax

|  |  |
| --- | --- |
| slice\_data( ) { | Descriptor |
| do { |  |
| coding\_tree\_unit( ) |  |
| **end\_of\_slice\_flag** | ae(v) |
| CtbAddrInRs++ |  |
| } while( !end\_of\_slice\_flag ) |  |
| } |  |

#### Coding tree unit syntax

|  |  |
| --- | --- |
| coding\_tree\_unit( ) { | Descriptor |
| xCtb = ( CtbAddrInRs % PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| yCtb = ( CtbAddrInRs / PicWidthInCtbsY )  <<  CtbLog2SizeY |  |
| coding\_quadtree( xCtb, yCtb, CtbLog2SizeY, 0 ) |  |
| } |  |

#### Coding quadtree syntax

|  |  |
| --- | --- |
| coding\_quadtree( x0, y0, log2CbSize, cqtDepth ) { | Descriptor |
| if( x0 + ( 1  <<  log2CbSize ) <= pic\_width\_in\_luma\_samples &&  y0 + ( 1  <<  log2CbSize ) <= pic\_height\_in\_luma\_samples &&  log2CbSize > MinQtLog2SizeY ) |  |
| **qt\_split\_cu\_flag**[ x0 ][ y0 ] | ae(v) |
| } |  |
| if( qt\_split\_cu\_flag[ x0 ][ y0 ] ) { |  |
| x1 = x0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| y1 = y0 + ( 1  <<  ( log2CbSize − 1 ) ) |  |
| coding\_quadtree( x0, y0, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples ) |  |
| coding\_quadtree( x1, y0, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_quadtree( x0, y1, log2CbSize − 1, cqtDepth + 1 ) |  |
| if( x1 < pic\_width\_in\_luma\_samples && y1 < pic\_height\_in\_luma\_samples ) |  |
| coding\_quadtree( x1, y1, log2CbSize − 1, cqtDepth + 1 ) |  |
| } else |  |
| multi\_type\_tree( x0, y0, 1  <<  log2CbSize, 1  <<  log2CbSize, 0 , 0 ) |  |
| } |  |

#### Multi-type tree syntax

|  |  |
| --- | --- |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight, mttDepth,  partIdx ) { | Descriptor |
| if( allowSplitBtVer | | allowSplitBtHor | | allowSplitTtVer | | allowSplitTtHor ) |  |
| **mtt\_split\_cu\_flag** | ae(v) |
| if( mtt\_split\_cu\_flag ) { |  |
| if( ( allowSplitBtHor | | allowSplitTtHor ) &&   ( allowSplitBtVer | | allowSplitTtVer ) ) |  |
| **mtt\_split\_cu\_vertical\_flag** | ae(v) |
| if( ( allowSplitBtVer && allowSplitTtVer && mtt\_split\_cu\_vertical\_flag ) | |   ( allowSplitBtHor && allowSplitTtHor && !mtt\_split\_cu\_vertical\_flag ) ) |  |
| **mtt\_split\_cu\_binary\_flag** | ae(v) |
| if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_VER ) { |  |
| multi\_type\_tree( x0, y0, cbWidth / 2, cbHeight, mttDepth + 1, 0 ) |  |
| multi\_type\_tree( x0 + ( cbWidth / 2 ), y0, cbWidth / 2, cbHeightY, mttDepth + 1, 1 ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_BT\_HOR ) { |  |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight / 2, mttDepth + 1, 0 ) |  |
| multi\_type\_tree( x0, y0 + ( cbHeight / 2 ), cbWidth, cbHeight / 2, mttDepth + 1, 1 ) |  |
| } else if( MttSplitMode[ x0 ][ y0 ][ mttDepth ] = = SPLIT\_TT\_VER ) { |  |
| multi\_type\_tree( x0, y0, cbWidth / 4, cbHeight, mttDepth + 1, 0 ) |  |
| multi\_type\_tree( x0 + ( cbWidth / 4 ), y0, cbWidth / 2, cbHeight, mttDepth + 1, 1 ) |  |
| multi\_type\_tree( x0 + ( 3 \* cbWidth / 4 ), y0, cbWidth / 4, cbHeight, mttDepth + 1, 2 ) |  |
| } else { /\* SPLIT\_TT\_HOR \*/ |  |
| multi\_type\_tree( x0, y0, cbWidth, cbHeight / 4, mttDepth + 1, 0 ) |  |
| multi\_type\_tree( x0, y0 + ( cbHeight / 4 ), cbWidth, cbHeight / 2, mttDepth + 1, 1 ) |  |
| multi\_type\_tree( x0, y0 + ( 3 \* cbHeight / 4 ), cbWidth, cbHeight / 4, mttDepth + 1, 2 ) |  |
| } |  |
| } else |  |
| coding\_unit( x0, y0, cbWidth, cbHeight ) |  |
| } |  |

#### Coding unit syntax

|  |  |
| --- | --- |
| coding\_unit( x0, y0, cbWidth, cbHeight ) { | Descriptor |
| if( slice\_type != I ) { |  |
| **pred\_mode\_flag** | ae(v) |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] = = MODE\_INTRA ) { |  |
| [Ed. (BB): Intra prediction yet to be added, pending further specification development.] |  |
| } else { |  |
| [Ed. (BB): Inter prediction yet to be added, pending further specification development.] |  |
| } |  |
| if( CuPredMode[ x0 ][ y0 ] != MODE\_INTRA ) |  |
| **cu\_cbf** | ae(v) |
| if( cu\_cbf ) { |  |
| transform\_tree( x0, y0, cbWidth, cbHeight ) |  |
| } |  |

#### Transform tree syntax

|  |  |
| --- | --- |
| transform\_tree( x0, y0, tbWidth, tbHeight ) { | Descriptor |
| if( tbWidth > MaxTbSizeY | | tbHeight > MaxTbSizeY ) { |  |
| trafoWidth = ( tbWidth > MaxTbSizeY ) ? (tbWidth / 2) : tbWidth |  |
| trafoHeight = ( tbHeight > MaxTbSizeY ) ? (tbHeight / 2) : tbHeight |  |
| transform\_tree( x0, y0, trafoWidth,  trafoHeight ) |  |
| if( tbWidth > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0, trafoWidth, trafoHeight ) |  |
| if( tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0, y0 + trafoHeight, trafoWidth, trafoHeight ) |  |
| if( tbWidth > MaxTbSizeY && tbHeight > MaxTbSizeY ) |  |
| transform\_tree( x0 + trafoWidth, y0 + trafoHeight, trafoWidth, trafoHeight ) |  |
| } else { |  |
| transform\_unit( x0, y0, tbWidth, tbHeight ) |  |
| } |  |
| } |  |

#### Transform unit syntax

[Ed. (BB): Current TU syntax including coded block flags for luma and chroma is just a placeholder for a transform coding specification yet to be added, pending further specification development.]

|  |  |
| --- | --- |
| transform\_unit( x0, y0, tbWidth, tbHeight ) { | Descriptor |
| **tu\_cbf\_luma**[ x0 ][ y0 ] | ae(v) |
| **tu\_cbf\_cb**[ x0 ][ y0 ] | ae(v) |
| **tu\_cbf\_cr**[ x0 ][ y0 ] | ae(v) |
| if( tu\_cbf\_luma[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, tbWidth, tbHeight, 0 ) |  |
| if( tu\_cbf\_cb[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, tbWidth / 2, tbHeight / 2, 1 ) |  |
| if( tu\_cbf\_cr[ x0 ][ y0 ] ) |  |
| residual\_coding( x0, y0, tbWidth / 2, tbHeight / 2, 2 ) |  |
| } |  |

#### Residual coding syntax

|  |  |
| --- | --- |
| residual\_coding( x0, y0,  tbWidth, tbHeight, cIdx ) { | Descriptor |
| [Ed. (BB) coefficient syntax yet to be added, pending further specification development.] |  |
|  |  |
| } |  |

## Semantics

### General

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

### NAL unit semantics

#### General NAL unit semantics

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

NOTE 1 – The video coding layer (VCL) is specified to efficiently represent the content of the video data. The NAL is specified to format that data and provide header information in a manner appropriate for conveyance on a variety of communication channels or storage media. All data are contained in NAL units, each of which contains an integer number of bytes. A NAL unit specifies a generic format for use in both packet-oriented and bitstream systems. The format of NAL units for both packet-oriented transport and byte stream is identical except that each NAL unit can be preceded by a start code prefix and extra padding bytes in the byte stream format specified in Annex TBD.

**rbsp\_byte**[ i ] is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows:

The RBSP contains an string of data bits **(**SODB) as follows:

– If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.

– Otherwise, the RBSP contains the SODB as follows:

1) The first byte of the RBSP contains the (most significant, left-most) eight bits of the SODB; the next byte of the RBSP contains the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.

2) rbsp\_trailing\_bits( ) are present after the SODB as follows:

i) The first (most significant, left-most) bits of the final RBSP byte contains the remaining bits of the SODB (if any).

ii) The next bit consists of a single rbsp\_stop\_one\_bit equal to 1.

iii) When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more rbsp\_alignment\_zero\_bit is present to result in byte alignment.

3) One or more cabac\_zero\_word 16-bit syntax elements equal to 0x0000 may be present in some RBSPs after the rbsp\_trailing\_bits( ) at the end of the RBSP.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures are carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units is as specified in **Table 7‑1**.

NOTE 2 – When the boundaries of the RBSP are known, the decoder can extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

**emulation\_prevention\_three\_byte** is a byte equal to 0x03. When an emulation\_prevention\_three\_byte is present in the NAL unit, it shall be discarded by the decoding process.

The last byte of the NAL unit shall not be equal to 0x00.

Within the NAL unit, the following three-byte sequences shall not occur at any byte-aligned position:

– 0x000000

– 0x000001

– 0x000002

Within the NAL unit, any four-byte sequence that starts with 0x000003 other than the following sequences shall not occur at any byte-aligned position:

– 0x00000300

– 0x00000301

– 0x00000302

– 0x00000303

#### NAL unit header semantics

**forbidden\_zero\_bit** shall be equal to 0.

**nal\_unit\_type** specifies the type of RBSP data structure contained in the NAL unit as specified in Table 7‑1.

[Ed. (BB): NAL unit types yet to be defined, pending further specification development.]

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

|  |  |  |  |
| --- | --- | --- | --- |
| **nal\_unit\_type** | **Name of nal\_unit\_type** | **Content of NAL unit and RBSP syntax structure** | **NAL unit type class** |
|  |  |  |  |
|  |  |  |  |

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

#### Sequence parameter set RBSP semantics

[Ed. (BB): Preliminary basic SPS, subject to further study and pending further specification development.]

**sps\_seq\_parameter\_set\_id** provides an identifier for the SPS for reference by other syntax elements. The value of sps\_seq\_parameter\_set\_id shall be in the range of 0 to 15, inclusive.

**chroma\_format\_idc** specifies the chroma sampling relative to the luma sampling as specified in clause 6.2. The value of chroma\_format\_idc shall be in the range of 0 to 3, inclusive.

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

NOTE 1 – There is no dependency in decoding processes between the colour planes having different colour\_plane\_id values. For example, the decoding process of a monochrome picture with one value of colour\_plane\_id does not use any data from monochrome pictures having different values of colour\_plane\_id for inter prediction.

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

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

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

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

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

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

BitDepthY = 8 + bit\_depth\_luma\_minus8 (7‑1)

QpBdOffsetY = 6 \* bit\_depth\_luma\_minus8 (7‑2)

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

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

BitDepthC = 8 + bit\_depth\_chroma\_minus8 (7‑3)

QpBdOffsetC = 6 \* bit\_depth\_chroma\_minus8 (7‑4)

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

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

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

CtbLog2SizeY = log2\_ctu\_size\_minus2 + 2 (7‑5)

CtbSizeY = 1  <<  CtbLog2SizeY (7‑6)

MinCbLog2SizeY = 2 (7‑7)

MinCbSizeY = 1  <<  MinCbLog2SizeY (7‑8)

MinTbSizeY = 4 (7‑9)

MaxTbSizeY = 64 (7‑10)

PicWidthInCtbsY = Ceil( pic\_width\_in\_luma\_samples ÷ CtbSizeY ) (7‑11)

PicHeightInCtbsY = Ceil( pic\_height\_in\_luma\_samples ÷ CtbSizeY ) (7‑12)

PicSizeInCtbsY = PicWidthInCtbsY \* PicHeightInCtbsY (7‑13)

PicWidthInMinCbsY = pic\_width\_in\_luma\_samples / MinCbSizeY (7‑14)

PicHeightInMinCbsY = pic\_height\_in\_luma\_samples / MinCbSizeY (7‑15)

PicSizeInMinCbsY = PicWidthInMinCbsY \* PicHeightInMinCbsY (7‑16)

PicSizeInSamplesY = pic\_width\_in\_luma\_samples \* pic\_height\_in\_luma\_samples (7‑17)

PicWidthInSamplesC = pic\_width\_in\_luma\_samples / SubWidthC (7‑18)

PicHeightInSamplesC = pic\_height\_in\_luma\_samples / SubHeightC (7‑19)

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

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

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

– Otherwise, CtbWidthC and CtbHeightC are derived as follows:

CtbWidthC = CtbSizeY / SubWidthC (7‑20)

CtbHeightC = CtbSizeY / SubHeightC (7‑21)

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

**log2\_min\_qt\_size\_intra\_slices\_minus2** plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 2 (I). The value of log2\_min\_qt\_size\_intra\_slices\_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeIntraY = log2\_min\_qt\_size\_intra\_slices\_minus2 + 2 (7‑22)

[Ed. (BB): The leaf of a quadtree can either be a coding unit or the root of a nested multi-type tree.]

**log2\_min\_qt\_size\_inter\_slices\_minus2** plus 2 specifies the minimum luma size of a leaf block resulting from quadtree splitting of a CTU in slices with slice\_type equal to 0 (B) or 1 (P). The value of log2\_min\_qt\_size\_inter\_slices\_minus2 shall be in the range of 0 to CtbLog2SizeY − 2, inclusive.

MinQtLog2SizeInterY = log2\_min\_qt\_size\_inter\_slices\_minus2 + 2 (7‑23)

**max\_mtt\_hierarchy\_depth\_inter\_slices** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 0 (B) or 1 (P). The value of max\_mtt\_hierarchy\_depth\_inter\_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

**max\_mtt\_hierarchy\_depth\_intra\_slices** specifies the maximum hierarchy depth for coding units resulting from multi-type tree splitting of a quadtree leaf in slices with slice\_type equal to 2 (I). The value of max\_mtt\_hierarchy\_depth\_intra\_slices shall be in the range of 0 to CtbLog2SizeY − MinTbLog2SizeY, inclusive.

#### Picture parameter set RBSP semantics

[Ed. (BB): Preliminary basic PPS, subject to further study and pending further specification development.]

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

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

#### Access unit delimiter RBSP semantics

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

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

Table 7‑2 – Interpretation of pic\_type

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

#### End of sequence RBSP semantics

When included in a NAL unit with nuh\_layer\_id equal to 0, the end of sequence RBSP specifies that the current access unit is the last access unit in the coded video sequence in decoding order and the next subsequent access unit in the bitstream in decoding order (if any) is an IRAP access unit. The syntax content of the SODB and RBSP for the end of sequence RBSP are empty.

#### End of bitstream RBSP semantics

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

#### Filler data RBSP semantics

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

**ff\_byte** is a byte equal to 0xFF.

#### Slice layer RBSP semantics

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

#### RBSP slice trailing bits semantics

**cabac\_zero\_word** is a byte-aligned sequence of two bytes equal to 0x0000.

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

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

Let the variable RawMinCuBits be derived as follows:

RawMinCuBits = MinCbSizeY \* MinCbSizeY \*  
 ( BitDepthY + 2 \* BitDepthC / ( SubWidthC \* SubHeightC ) ) (7‑24)

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

NOTE – The constraint on the maximum number of bins resulting from decoding the contents of the coded slice NAL units can be met by inserting a number of cabac\_zero\_word syntax elements to increase the value of NumBytesInVclNalUnits. Each cabac\_zero\_word is represented in a NAL unit by the three-byte sequence 0x000003 (as a result of the constraints on NAL unit contents that result in requiring inclusion of an emulation\_prevention\_three\_byte for each cabac\_zero\_word).

#### RBSP trailing bits semantics

**rbsp\_stop\_one\_bit** shall be equal to 1.

**rbsp\_alignment\_zero\_bit** shall be equal to 0.

#### Byte alignment semantics

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

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

### Slice header semantics

[Ed. (BB): Preliminary basic slice header, subject to further study and pending further specification development.]

When present, the value of the slice header syntax element slice\_pic\_parameter\_set\_id shall be the same in all slice headers of a coded picture.

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

**slice\_address** specifies the address of the first CTB in the slice, in CTB raster scan of a picture. The length of the slice\_address syntax element is Ceil( Log2( PicSizeInCtbsY ) ) bits. The value of slice\_address shall be in the range of 0 to PicSizeInCtbsY − 1, inclusive, and the value of slice\_address shall not be equal to the value of slice\_address of any other coded slice NAL unit of the same coded picture.

The variable CtbAddrInRs, specifying a CTB address in CTB raster scan of a picture, is set equal to slice\_address.

**slice\_type** specifies the coding type of the slice according to Table 7‑3.

Table 7‑3 – Name association to slice\_type

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

When nal\_unit\_type has a value in the range of TBD, inclusive, i.e., the picture is an IRAP picture, slice\_type shall be equal to 2.

[Ed. (BB): IRAP slice types to be defined (if such types will exist), pending further specification development.]

**log2\_diff\_ctu\_max\_bt\_size** specifies the difference between the luma CTB size and the maximum luma size (width or height) of a coding block that can be split using a binary split. The value of log2\_diff\_ctu\_max\_bt\_size shall be in the range of 0 to CtbLog2SizeY − MinCbLog2SizeY, inclusive.

When log2\_diff\_ctu\_max\_bt\_size is not present, the value of log2\_diff\_ctu\_max\_bt\_size is inferred to be equal to 2.

The variables MinQtLog2SizeY, MaxBtLog2SizeY, MinBtLog2SizeY, MaxTtLog2SizeY, MinTtLog2SizeY, MaxBtSizeY, MinBtSizeY, MaxTtSizeY, MinTtSizeY and MaxMttDepth are derived as follows:

MinQtLog2SizeY = ( slice\_type = = I ) ? MinQtLog2SizeIntraY : MinQtLog2SizeInterY (7‑25)

MaxBtLog2SizeY = CtbLog2SizeY − log2\_diff\_ctu\_max\_bt\_size (7‑26)

MinBtLog2SizeY = MinCbLog2SizeY (7‑27)

MaxTtLog2SizeY = ( slice\_type = = I ) ? 5 : 7 (7‑28)

MinTtLog2SizeY = MinCbLog2SizeY (7‑29)

MaxBtSizeY = 1  <<  MaxBtLog2SizeY (7‑30)

MinBtSizeY = 1  <<  MinBtLog2SizeY (7‑31)

MaxTtSizeY = 1  <<  MaxTtLog2SizeY (7‑32)

MinTtSizeY = 1  <<  MinTtLog2SizeY (7‑33)

MaxMttDepth = ( slice\_type = = I ) ? max\_mtt\_hierarchy\_depth\_intra\_slices :   
 max\_mtt\_hierarchy\_depth\_inter\_slices (7‑34)

[Ed. (BB): Currently the maximum TT size is fixed (32x32 luma samples and corresponding chroma samples for I-slices and 128x128 luma samples and corresponding chroma samples for P/B-slices ) as well as the maximum BT size for I-slices (CtbLog2SizeY − 2, e.g. 32x32 luma samples and corresponding chroma samples for a CTU size of 128x128 luma samples).]

### Slice data semantics

#### General slice data semantics

**end\_of\_slice\_flag** equal to 0 specifies that another CTU is following in the slice. end\_of\_slice\_flag equal to 1 specifies the end of the slice, i.e., that no further CTU follows in the slice.

#### Coding tree unit semantics

The CTU is the root node of the coding quadtree structure.

#### Coding quadtree semantics

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

When qt\_split\_cu\_flag[ x0 ][ y0 ] is not present, the following applies:

* If one or more of the following conditions are true, the value of qt\_split\_cu\_flag[ x0 ][ y0 ] is inferred to be equal to 1:
* x0 + ( 1  <<  log2CbSize ) is greater than pic\_width\_in\_luma\_samples.
* y0 + ( 1  <<  log2CbSize ) is greater than pic\_height\_in\_luma\_samples.

– Otherwise, the value of qt\_split\_cu\_flag[ x0 ][ y0 ] is inferred to be equal to 0.

#### Multi-type tree semantics

The variables allowSplitBtVer, allowSplitBtHor, allowSplitTtVer allowSplitTtHor are derived as follows:

* The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT\_BT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current partition index partIdx as input, and the output is assigned to allowSplitBtVer.
* The allowed binary split process as specified in clause 6.4.1 is invoked with the binary split mode SPLIT\_BT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current partition index partIdx as input, and the output is assigned to allowSplitBtHor.
* The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT\_TT\_VER, the coding block width cbWidth, the coding block height cbHeight, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current partition index partIdx as input, and the output is assigned to allowSplitTtVer.
* The allowed ternary split process as specified in clause 6.4.2 is invoked with the ternary split mode SPLIT\_TT\_HOR, the coding block height cbHeight, the coding block width cbWidth, the location ( x0, y0 ), the current multi-type tree depth mttDepth, the current partition index partIdx as input, and the output is assigned to allowSplitTtHor.

**mtt\_split\_cu\_flag** equal to 0 specifies that a coding unit is not split. mtt\_split\_cu\_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split or into three coding units using a ternary split as indicated by the syntax element mtt\_split\_cu\_binary\_flag. The binary or ternary split can be either vertical or horizontal as indicated by the syntax element mtt\_split\_cu\_vertical\_flag.

When mtt\_split\_cu\_flag is not present, the value of mtt\_split\_cu\_flag is inferred to be equal to 0.

**mtt\_split\_cu\_vertical\_flag** equal to 0 specifies that a coding unit is split horizontally. mtt\_split\_cu\_vertical\_flag equal to 1 specifies that a coding unit is split vertically

When mtt\_split\_cu\_vertical\_flag is not present, it is inferred as follows:

* If allowSplitBtHor is equal to TRUE or allowSplitTtHor is equal to TRUE, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 0.
* Otherwise, the value of mtt\_split\_cu\_vertical\_flag is inferred to be equal to 1

**mtt\_split\_cu\_binary\_flag** equal to 0 specifies that a coding unit is split into three coding units using a ternary split. mtt\_split\_cu\_binary\_flag equal to 1 specifies that a coding unit is split into two coding units using a binary split.

When mtt\_split\_cu\_binary\_flag is not present, it is inferred as follows:

* If allowSplitBtVer is equal to FALSE and allowSplitBtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to 0.
* Otherwise if allowSplitTtVer is equal to FALSE and allowSplitTtHor is equal to FALSE, the value of mtt\_split\_cu\_binary\_flag is inferred as to be equal to 1.
* Otherwise if allowSplitBtHor is equal to TRUE and allowSplitTtVer is equal to TRUE, the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to !mtt\_split\_cu\_vertical\_flag.
* Otherwise (allowSplitBtVer is equal to TRUE and allowSplitTtHor is equal to TRUE), the value of mtt\_split\_cu\_binary\_flag is inferred to be equal to mtt\_split\_cu\_vertical\_flag.

The variable MttSplitMode[ x ][ y ][ mttDepth ] is derived from the value of mtt\_split\_cu\_vertical\_flag and from the value of mtt\_split\_cu\_binary\_flag as defined in Table 7‑4 for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.



Figure 7‑1 – Multi-type tree spliting modes indicated by MttSplitMode (informative)

MttSplitMode[ x0 ][ y0 ][ mttDepth ] represents horizontal and vertical binary and ternary splittings of a coding unit within the multi-type tree as illustrated in Figure 7‑1. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered coding block relative to the top-left luma sample of the picture.

Table 7‑4 – Specification of MttSplitMode[ x ][ y ][ mttDepth ] for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1

|  |  |  |
| --- | --- | --- |
| **MttSplitMode[ x0 ][ y0 ][ mttDepth ]** | **mtt\_split\_cu\_vertical\_flag** | **mtt\_split\_cu\_binary\_flag** |
| SPLIT\_TT\_HOR | 0 | 0 |
| SPLIT\_BT\_HOR | 0 | 1 |
| SPLIT\_TT\_VER | 1 | 0 |
| SPLIT\_BT\_VER | 1 | 1 |

#### Coding unit semantics

**pred\_mode\_flag** equal to 0 specifies that the current coding unit is coded in inter prediction mode. pred\_mode\_flag equal to 1 specifies that the current coding unit is coded in intra prediction mode. The variable CuPredMode[ x ][ y ] is derived as follows for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1:

* If pred\_mode\_flag is equal to 0, CuPredMode[ x ][ y ] is set equal to MODE\_INTER.
* Otherwise (pred\_mode\_flag is equal to 1), CuPredMode[ x ][ y ] is set equal to MODE\_INTRA.

When pred\_mode\_flag is not present, the variable CuPredMode[ x ][ y ] is is inferred to be equal to MODE\_INTRA for x = x0..x0 + cbWidth − 1 and y = y0..y0 + cbHeight − 1.

**cu\_cbf** equal to 1 specifies that the transform\_tree( ) syntax structure is present for the current coding unit. cu\_cbf equal to 0 specifies that the transform\_tree( ) syntax structure is not present for the current coding unit.

When cu\_cbf is not present, its value is inferred to be equal to 1.

#### Transform tree semantics

[Ed. (BB): The transform scheme does not have any syntax for spliting a CU into TUs. However, if the height or width of a CU is larger than the current maximum transform length of 64 luma samples or the corresponding chroma sample length, the CU will be implicitly split to divide it into TUs.]

#### Transform unit semantics

[Ed. (BB): Current TU semantics including coded block flags for luma and chroma are just a placeholder for a transform coding scheme yet to be added, pending further specification development.]

**tu\_cbf\_luma**[ x0 ][ y0 ] equal to 1 specifies that the luma transform block contains one or more transform coefficient levels not equal to 0. The array indices x0, y0 specify the location ( x0, y0 ) of the top-left luma sample of the considered transform block relative to the top-left luma sample of the picture.

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

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

#### Residual coding semantics

[Ed. (BB): The transform coefficient coding syntax is yet to be added, pending further specification development.]