Independent Submission Y. Lim
Internet-Draft M. Park
Intended status: Informational M. Budagavi
Expires: 12 February 2026 R. Joshi
K. Choi
Samsung Electronics
11 August 2025

# Advanced Professional Video
draft-lim-apv-06

## Abstract

This document describes bitstream format of Advanced Professional Video (APV) and decoding process of it. APV is a professional video codec providing visually lossless compression mainly for recording and post production. APV is designed and developed to be open public standard with no licensing and royalty is required for use.

## Status of This Memo



## Copyright Notice

Table of Contents

## 1. Introduction

This document defines the bitstream formats and decoding process for Advanced Professional Video (APV) Codec. APV codec is a professional video codec that was developed in response to the need for professional level high quality video recording and post production. The primary purpose of APV codec is for use in professional video recording and editing workflows for various types of content. APV has been designed and developed to be made freely available to the industry for open public use without any licensing. This specification is neither the product of the IETF nor a consensus view of the community.

The APV codec supports the following features:

* Perceptually lossless video quality that is close to the original, uncompressed quality

* Low complexity and high throughput intra frame only coding without inter frame coding.

* Intra frame coding without prediction between pixel values but between transformed values for low delay encoding

* Support for high bit-rates up to a few Gbps for 2K, 4K and 8K resolution content, enabled by a lightweight entropy coding scheme

* Frame tiling for immersive content and for enabling parallel encoding and decoding

* Support for various chroma sampling formats from 4:0:0 to 4:4:4:4, and bit depths from 10 to 16 (Note: The profiles supporting 10 bit and 12 bit are currently defined)

* Support for multiple decoding and re-encoding without severe visual quality degradation

# 2. Terms

## 2.1. Terms and definitions

* access unit (AU): a collection of primitive bitstream units including various types of frames, metadata, filler, and access unit information, associated with a specific time

* band: a defined set of constraints on the value of the maximum coded data rate of each level

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

* byte-aligned: a position in a bitstream that is an integer multiple of 8 bits from the position of the first bit in the bitstream

* chroma: a sample array or single sample representing one of the two color difference signals related to the primary colors, represented by the symbols Cb and Cr in 4:2:2 or 4:4:4 color format

* coded frame: a coded representation of a frame containing all macroblocks of the frame

* coded representation: a data element as represented in its coded form

* component: an array or a single sample from one of the three arrays (luma and two chroma) that compose a frame in 4:2:2, or 4:4:4 color format, or an array or a single sample from an array that compose a frame in 4:0:0 color format, or an array or a single sample from one of the four arrays that compose a frame in 4:4:4:4 color format.

* decoded frame: a frame derived by decoding a coded frame

* decoder: an embodiment of a decoding process

* decoding process: a process specified that reads a bitstream and derives decoded frames from it

* encoder: an embodiment of an encoding process

* encoding process: a process that produces a bitstream conforming to this document

* flag: a variable or single-bit syntax element that can take one of the two possible values: 0 and 1

* frame: an array of luma samples and two corresponding arrays of chroma samples in 4:2:2, and 4:4:4 color format, or an array of samples in 4:0:0 color format, or four arrays of samples in 4:4:4:4 color format

* level: a defined set of constraints on the values that are taken by the syntax elements and variables of this document, or the value of a transform coefficient prior to scaling

* luma: a sample array or single sample representing the monochrome signal related to the primary colors, represented by the symbol or subscript Y or L

* macroblock (MB): a square block of luma samples and two corresponding blocks of chroma samples of a frame in 4:2:2 or 4:4:4 color format, or a sqaure block of samples of a frame in 4:0:0 color format, or four square blocks of samples of a frame in 4:4:4:4 color format

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

* prediction: an embodiment of the prediction process

* prediction process: use of a predictor to provide an estimate of the data element currently being decoded

* predictor: a combination of specified values or previously decoded data elements used in the decoding process of subsequent data elements

* primitive bitstream unit (PBU): a data structure to construct an access unit with frame and metadata

* profile: a specified subset of the syntax of this document

* quantization parameter (QP): a variable used by the decoding process for scaling of transform coefficient levels

* 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 top row of the two-dimensional pattern scanned from left to right, followed by the second, third, etc., rows of the pattern each scanned from left to right

* raw bitstream: an encapsulation of a sequence of access units where a field indicating the size of an access unit precedes each access unit as defined in Section 12.1

* source: a term used to describe the video material or some of its attributes before encoding process

* syntax element: an element of data represented in the bitstream

* syntax structure: zero or more syntax elements present together in the bitstream in a specified order

* tile: a rectangular region of MBs within a particular tile column and a particular tile row in a frame

* tile column: a rectangular region of MBs having a height equal to the height of the frame and width specified by syntax elements in the frame header

* tile row: a rectangular region of MBs having a height specified by syntax elements in the frame header and a width equal to the width of the frame

* tile scan: a specific sequential ordering of MBs partitioning a frame in which the MBs are ordered consecutively in MB raster scan in a tile and the tiles in a frame are ordered consecutively in a raster scan of the tiles of the frame

* transform coefficient: a scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional index

## 2.2. Abbreviated terms

* I: intra

* LSB: least significant bit

* MSB: most significant bit

* RGB: Red, Green and Blue

# 3. Conventions used in this document

## 3.1. General

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

## 3.2. Operators

The operators and the order of precedence are the same as used in the C programming language [ISO9899], with the exception of the operators described in the Section 3.2.1 and Section 3.2.2

### 3.2.1. Arithmetic operators

* // : an integer division with rounding of the result toward zero. For example, 7//4 and -7//-4 are rounded to 1 and -7//4 and 7//-4 are rounded to -1

* / or div(x,y) : a division in mathematical equations where no truncation or rounding is intended

* % : a modulus. x % y is a remainder of x divided by y

* min(x,y) : the minimum value of the values x and y

* max(x,y) : the maximum value of the values x and y

* ceil(x) : the smallest integer value that is larger than or equal to x

* clip(x,y,z) : clip(x,y,z)=max(x,min(z,y))

* sum (i=x, y, f(i)) : a summation of f(i) with i taking all integer values from x up to and including y

* log2(x) : the base-2 lograithm of x

#### 3.2.2. Bitwise operators

* & (bit-wise "and") : When operating on integer arguments, operates on a two’s complement representation of the integer value. When operating on arguments with unequal bit depths, the bit depths are equalized by adding zeros in significant positions to the argument with lower bit depth.

* | (bit-wise "or") : When operating on integer arguments, operates on a two’s complement representation of the integer value. When operating on arguments with unequal bit depths, the bit depths are equalized by adding zeros in significant positions to the argument with lower bit depth.

* 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.

### 3.3. Range notation

* x = y..z

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

#### 3.3.1. Order of operations precedence

When order of precedence is not indicated explicitly by use of parentheses, operations are evaluated in the following order.

* Operations of a higher precedence are evaluated before any operation of a lower precedence. Table 1 specifies the precedence of operations from highest to lowest; operations closer to the top of the table indicates a higher precedence.

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

<table>
<tr><th>operations (with operands x, y, and z)</th></tr>
<tr><td>"x++", "x--"</td></tr>
<tr><td>"!x", "-x" (as a unary prefix operator)</td></tr>
<tr><td>x^y (power)</td></tr>
<tr><td>"x * y", "x / y", "x // y", "x % y"</td></tr>
<tr><td>"x + y", "x - y", "sum (i=x, y, f(i))"</td></tr>
<tr><td>"x << y", "x >> y"</td></tr>
<tr><td>"x < y", "x <= y", "x > y", "x >= y"</td></tr>
<tr><td>"x == y", "x != y"</td></tr>
<tr><td>"x & y"</td></tr>
<tr><td>"x | y"</td></tr>
<tr><td>"x && y"</td></tr>
<tr><td>"x || y"</td></tr>
<tr><td>"x ? y : z"</td></tr>
<tr><td>"x..y"</td></tr>
<tr><td>"x = y", "x += y", "x -= y"</td></tr>
</table>

Table 1: Operation precedence from highest (top of the table) to lowest (bottom of the table)

## 3.4. Variables, syntax elements and tables

Each syntax element is described by its name in all lowercase letters and its type is provided next to the syntax code in each row. Each syntax element and multi-byte integers are written in big endian format. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements.

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 uppercase letters and without any underscore characters. Variables with names starting with an uppercase letter are derived for the decoding of the current syntax structure and all dependent syntax structures. Variables with names starting with an uppercase letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables with names starting with a lowercase letter are only used within the section in which they are derived.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in Section 5.2 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.

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. Square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first square bracket is used as a column (horizontal) index and the second square bracket is used as a row (vertical) index.

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.

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

### 3.5. Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. 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), the assignment is implied.

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

## 4. Formats and processes used in this document

### 4.1. Bitstream formats

This section specifies the bitstream of the Advanced Professional Video (APV) Codec.

The raw bitstream format is a format consist with a sequence of AUs where the field indicating the size of access units precedes each of them. The raw bitstream format is specified in Section 12.1.

### 4.2. Source, decoded and output frame formats

This section specifies the relationship between the source and the decoded frames that are the results of the decoding process.

The video source that is represented by the bitstream is a sequence of frames.

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

* Monochrome (for example, Luma only)

* Luma and two chroma (for example, YCbCr or YCgCo).

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

* Arrays representing other unspecified tri-stimulus color samplings (for example, YZX, also known as XYZ).

* Arrays representing other unspecified four color samplings

For the convenience of notation and terminology in this document, the variables and terms associated with these arrays can be referred to as luma and chroma regardless of the actual color representation method in use.

The variables SubWidthC, SubHeightC and NumComps are specified in Table 2, depending on the chroma format sampling structure, which is specified through chroma_format_idc. Other values of chroma_format_idc, SubWidthC, SubHeightC and NumComps may be specified in the future.

| chroma_format_idc | Chroma format | SubWidthC | SubHeightC | NumComps |
|---|---|---|---|---|
| 0 | 4:0:0 | 1 | 1 | 1 |
| 1 | reserved | reserved | reserved | reserved |
| 2 | 4:2:2 | 2 | 1 | 3 |
| 3 | 4:4:4 | 1 | 1 | 3 |
| 4 | 4:4:4:4 | 1 | 1 | 4 |
| 5..7 | reserved | reserved | reserved | reserved |

Table 2: SubWidthC, SubHeightC and NumComps values derived from chroma_format_idc

In 4:0:0 sampling, there is only one sample array that can be considered as 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 and 4:4:4:4 sampling, all the sample arrays have the same height and width as the luma array.

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 10 to 16, inclusive.

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 frame are as shown in Figure 1.

```
& * & * & * & * & * ...

& * & * & * & * & * ...

& * & * & * & * & * ...

& * & * & * & * & * ...

          ...

& - location where both luma and chroma sample exist

* - location where only luma sample exist
```

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

When the value of chroma_format_idc is equal to 3 or 4, for each frame, all the array samples are co-sited and the nominal locations in a frame are as shown in Figure 2.

```
& & & & & & & & & & ...

& & & & & & & & & & ...

& & & & & & & & & & ...

& & & & & & & & & & ...

          ...

& - location where both luma and chroma sample exist
```

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

The samples are processed in units of MBs. The variables MbWidth and MbHeight, which specify the width and height of the luma arrays for each MB, are defined as follows:

* MbWidth = 16

* MbHeight = 16

The variables MbWidthC and MbHeightC, which specify the width and height of the chroma arrays for each MB, are derived as follows:

* MbWidthC = MbWidth // SubWidthC

* MbHeightC = MbHeight // SubHeightC

## 4.3. Partitioning of a frame

### 4.3.1. Partitioning of a frame into tiles

This section specifies how a frame is partitioned into tiles.

A frame is divided into tiles. A tile is a group of MBs that cover a rectangular region of a frame and is processed independently of other tiles. Every tile has the same width and height, except possibly tiles at the right or bottom frame boundary when the frame width or height is not a multiple of the tile width or height, respectively. The tiles in a frame are scanned in raster order. Within a tile, the MBs are scanned in raster order. Each MB is comprised of one (MbWidth) x (MbHeight) luma array and zero, two, or three corresponding chroma sample arrays.

For example, a frame is divided into 6 tiles (3 tile columns and 2 tile rows) as shown in Figure 3: Frame with 10 by 8 MBs that is partitioned into 6 tiles. In this example, the tile size is defined as 4 column MBs and 4 row MBs. In case of the third and sixth tiles (in raster order), the tile size is 2 column MBs and 4 row MBs since the frame width is not multiple of the tile width.

```
        +==================+==================+=========+
        #    |    |    |    # MB | MB | MB | MB # MB | MB #
        +------------------+------------------+---------+
        #    |    |    |    # MB | MB | MB | MB # MB | MB #
        +-----   tile  -----+------------------+---------+
        #    |    |    |    # MB | MB | MB | MB # MB | MB #
        +------------------+------------------+---------+
        #    |    |    |    # MB | MB | MB | MB # MB | MB #
        +==================+==================+=========+
        # MB | MB | MB | MB # MB | MB | MB | MB # MB | MB #
        +------------------+------------------+---------+
        # MB | MB | MB | MB # MB | MB | MB | MB # MB | MB #
        +------------------+------------------+---------+
        # MB | MB | MB | MB # MB | MB | MB | MB # MB | MB #
        +------------------+------------------+---------+
        # MB | MB | MB | MB # MB | MB | MB | MB # MB | MB #
        +==================+==================+=========+

                    #,=  tile boundary

                    |,-  MB boundary
```

Figure 3: Frame with 10 by 8 MBs that is partitioned into 6 tiles

### 4.3.2. Spatial or component-wise partitioning

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

* the division of each frame into components;
* the division of each frame into tile columns;
* the division of each frame into tile rows;
* the division of each tile column into tiles;
* the division of each tile row into tiles;
* the division of each tile into color components;
* the division of each tile into MBs;
* the division of each MB into blocks.

## 4.4. Scanning processes

### 4.4.1. Zig-zag scan

This process converts a two dimensional array into an one dimensional array. The process starts at the top-left position position in the block then moves digaonally, changing direction at the edges of the block until it reaches the bottom-right position. Figure 4 shows an example of scanning order for 4x4 size block.

```
+===================+
# 00 | 01 | 05 | 06 #
+-------------------+
# 02 | 04 | 07 | 12 #
+-------------------+
# 03 | 08 | 11 | 13 #
+-------------------+
# 09 | 10 | 14 | 15 #
+===================+
```

Figure 4: Example of zig-zag scanning order for 4x4 block

Inputs to this process are:

* a variable blkWidth specifying the width of a block, and

* a variable blkHeight specifying the height of a block.

Output of this process is the array zigZagScan[sPos].

The array index sPos specifies the scan position ranging from 0 to (blkWidth * blkHeight)-1. Depending on the value of blkWidth and blkHeight, the array zigZagScan is derived as follows:

```
pos = 0
zigZagScan[pos] = 0
pos++
for(line = 1; line < (blkWidth + blkHeight - 1); line++){
  if(line % 2){
    x = min(line, blkWidth - 1)
    y = max(0, line - (blkWidth - 1))
    while(x >=0 && y < blkHeight){
      zigZagScan[pos] = y * blkWidth + x
      pos++
      x--
      y++
    }
  }
  else{
    y = min(line, blkHeight - 1)
    x = max(0, line - (blkHeight - 1))
    while(y >= 0 && x < blkWidth){
      zigZagScan[pos] = y * blkWidth + x
      pos++
      x++
      y--
    }
  }
}
```

Figure 5: Pseudo-code for zig-zag scan

#### 4.4.2. Inverse scan

Inputs to this process are:

* a variable blkWidth specifying the width of a block, and

* a variable blkHeight specifying the height of a block.

Output of this process is the array inverseScan[rPos].

The array index rPos specifies the raster scan position ranging from 0 to (blkWidth * blkHeight)-1. Depending on the value of blkWidth and blkHeight, the array inverseScan is derived as follows:

* The variable forwardScan is derived by invoking zig-zag scan order 1D array initialization process as specified in Section 4.4.1 with input parameters blkWidth and blkHeight.

* The output variable inverseScan is derived as follows:

```
for(pos = 0; pos < blkWidth * blkHeight; pos++){
  inverseScan[forwardScan[pos]] = pos
}
```

Figure 6: Pseudo-code for inverse zig-zag scan

# 5. Syntax and semantics

## 5.1. Method of specifying syntax

The syntax tables specify a superset of the syntax of all allowed bitstreams. Note that an actual decoder MUST implement some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this document.

The APV bitstream is described in this document using syntax code based on the C programming language [ISO9899] and uses its if/else, while, and for keywords as well as functions defined within this document.

The syntax table in syntax code is presented in a two-column format such as shown in Figure 7. In this form, the type column provides a type referenced in that same line of syntax code by using syntax elements processing function defined in Section 5.2.5.

```
syntax code                                                     | type
----------------------------------------------------------------|-----
ExampleSyntaxCode() {                                           |
       operations                                               |
       syntax_element                                           | u(n)
}                                                               |
```

Figure 7: A depiction of type-labeled syntax code for syntax description in this document

## 5.2. Syntax functions and descriptors

The functions presented in this document 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.

### 5.2.1. byte_aligned()

* If the current position in the bitstream is on the last bit of a byte, 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.

### 5.2.2. more_data_in_tile()

* If the current position in the i-th tile() syntax structure is less than TileSize[i] in bytes from the beginning of the tile_header() syntax structure of the i-th tile, the return value of more_data_in_tile() is equal to TRUE.

* Otherwise, the return value of more_data_in_tile() is equal to FALSE.

### 5.2.3. next_bits(n)

This function provides the next n bits in the bitstream for comparison purposes, without advancing the bitstream pointer.

### 5.2.4. read_bits(n)

This function indicates to read 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.

### 5.2.5. Syntax element processing functions

* 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, i.e. big endian format. The parsing process for this descriptor is specified by the return value of the function read_bits(n).

* u(n): unsigned integer using n bits. 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.

* h(v): variable-length entropy coded syntax element with the left bit first, i.e. big endian format. The parsing process for this descriptor is specified in Section 7.1.

### 5.3. List of syntax and semantics

#### 5.3.1. Access unit

```
syntax code                                                     | type
----------------------------------------------------------------|-----
access_unit(au_size){                                           |
    signature                                                   | f(32)
    currReadSize = 4                                            |
    do(){                                                       |
        pbu_size                                                | u(32)
        currReadSize += 4                                       |
        pbu()                                                   |
        currReadSize += pbu_size                                |
    } while (au_size > currReadSize)                            |
}                                                               |
```

Figure 8: access unit syntax code

* signature

  A four-character code that identifies the bitstream as an APV AU. The value MUST be ’aPv1’ (0x61507631).

* pbu_size

  indicates the size of a primitive bitstream unit in bytes. The value of 0 for pbu_size is prohibited and the value of 0xFFFFFFFF for pbu_size is reserved for future use.

Note: An AU consists of one primary frame, zero or more non-primary frames such as a frame for additional view, zero or more alpha frames, zero or more depth frames, zero or more preview frame such as a frame with smaller resolution, zero or more metadatas, and zero or more fillers.

#### 5.3.2. Primitive bitstream unit

```
syntax code                                                    | type
---------------------------------------------------------------|-----
pbu(){                                                         |
    pbu_header()                                               |
    if((1 <= pbu_type && pbu_type <=2) ||                      |
      (25 <= pbu_type && pbu_type <= 27))                      |
      frame()                                                  |
    else if(pbu_type == 65)                                    |
      au_info()                                                |
    else if(pbu_type == 66)                                    |
      metadata()                                               |
    else if (pbu_type == 67)                                   |
      filler()                                                 |
}                                                              |
```

Figure 9: primitive bitstream unit syntax code

#### 5.3.3. Primitive bitstream unit header

```
syntax code                                                    | type
---------------------------------------------------------------|-----
pbu_header(){                                                  |
    pbu_type                                                   | u(8)
    group_id                                                   | u(16)
    reserved_zero_8bits                                        | u(8)
}                                                              |
```

Figure 10: primitive bitstream unit header syntax code

* pbu_type

  indicates the type of data in a PBU listed in Table 3. Other values of pbu_type are reserved for future use.

| pbu_type | meaning | notes |
|---|---|---|
| 0 | reserved | |
| 1 | primary frame | |
| 2 | non-primary frame | |
| 3...24 | reserved | |
| 25 | preview frame | |
| 26 | depth frame | |
| 27 | alpha frame | |
| 28...64 | reserved | |
| 65 | access unit information | |
| 66 | metadata | |
| 67 | filler | |
| 68...255 | reserved | |

Table 3: List of PBU types

Note: A PBU with pbu_type equal to 65 (access unit information) may happen in an AU. If it exists, it MUST be the first PBU in an AU, and it can be ignored by a decoder.

* group_id

  indicates the identifier to associate coded frame with metadata. More than two frame can have the same group_id in a single AU. A primary frame and a non-primary frame MUST have different group_id value and two non-primary frames MUST have different group_id values. When the value of group_id is equal to 0, the value of pbu_type MUST be greater than 64. The value of 0xFFFF for group_id is reserved for future use.

* reserved_zero_8bits

MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

#### 5.3.4. Frame

```
syntax code                                                    | type
---------------------------------------------------------------|-----
frame(){                                                       |
    frame_header()                                             |
    for(i = 0; i < NumTiles; i++){                             |
        tile_size[i]                                           | u(32)
        tile(i)                                                |
    }                                                          |
    filler()                                                   |
}                                                              |
```

Figure 11: frame() syntax code

* tile_size[i]

  indicates the size in bytes of i-th tile data (i.e., tile(i)) in raster order in a frame. The value of 0 for tile_size[i] is reserved for future use.

  The variable TileSize[i] is set equal to tile_size[i].

#### 5.3.5. Frame header

```
syntax code                                                   | type
--------------------------------------------------------------|-----
frame_header(){                                               |
  frame_info()                                                |
  reserved_zero_8bits                                         | u(8)
  color_description_present_flag                              | u(1)
  if(color_description_present_flag){                         |
    color_primaries                                           | u(8)
    transfer_characteristics                                  | u(8)
    matrix_coefficients                                       | u(8)
    full_range_flag                                           | u(1)
  }                                                           |
  use_q_matrix                                                | u(1)
  if(use_q_matrix){                                           |
    quantization_matrix()                                     |
  }                                                           |
  tile_info()                                                 |
  reserved_zero_8bits                                         | u(8)
  byte_alignment()                                            |
}                                                             |
```

Figure 12: frame_header() syntax code

* reserved_zero_8bits

  MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

* color_description_present_flag equal to 1

  specifies that color_primaries, transfer_characteristics and matrix_coefficients are present. color_description_present_flag equal to 0 specifies that color_primaries, transfer_characteristics and matrix_coefficients are not present.

* color_primaries

  MUST have the semantics of ColourPrimaries as specified in [H273]. When the color_primaries syntax element is not present, the value of color_primaries is inferred to be equal to 2.

* transfer_characteristics

MUST have the semantics of TransferCharacteristics as specified in [H273]. When the transfer_characteristics syntax element is not present, the value of transfer_characteristics is inferred to be equal to 2.

* matrix_coefficients

  MUST have the semantics of MatrixCoefficients as specified in [H273]. When the matrix_coefficients syntax element is not present, the value of matrix_coefficients is inferred to be equal to 2.

* full_range_flag

  MUST have the semantics of VideoFullRangeFlag as specified in [H273]. When the full_range_flag syntax element is not present, the value of full_range_flag is inferred to be equal to 0.

* use_q_matrix

  equal to 1 specifies that the quantization matrices are present. use_q_matrix equal to 0 specifies that the quantization matrices are not present.

* reserved_zero_8bits

  MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

### 5.3.6. Frame information

```
syntax code                                                     | type
----------------------------------------------------------------|-----
frame_info(){                                                   |
  profile_idc                                                   | u(8)
  level_idc                                                     | u(8)
  band_idc                                                      | u(3)
  reserved_zero_5bits                                           | u(5)
  frame_width                                                   | u(24)
  frame_height                                                  | u(24)
  chroma_format_idc                                             | u(4)
  bit_depth_minus8                                              | u(4)
  capture_time_distance                                         | u(8)
  reserved_zero_8bits                                           | u(8)
}                                                               |
```

Figure 13: frame_info() syntax code

* profile_idc

  indicates a profile to which the coded frame conforms to as specified in Section 9. Bitstreams SHALL NOT contain values of profiles_idc other than those specified in Section 9. Other values of profile_idc are reserved for future use.

* level_idc

  indicates a level to which the coded frame conforms to as specified in Section 9. Bitstreams SHALL NOT contain values of level_idc other than those specified in Section 9. Other values of level_idc are reserved for future use.

* band_idc

  specifies a maximum coded data rate of level_idc as specified in Section 9. Bitstreams SHALL NOT contain values of band_idc other than those specified in Section 9. The value of band_idc MUST be in the range of 0 to 3. Other values of band_idc are reserved for future use.

* reserved_zero_5bits

  shall be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_5bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9. MUST ignore PBU with values of reserved_zero_5bits greater than 0.

* frame_width

  specifies the width of frame in units of luma samples. frame_width MUST be a multiple of 2 when chroma_format_idc has a value of 2. The value of 0 for frame_width is reserved for future use.

* frame_height

  specifies the height of frame in units of luma samples. The value of 0 for frame_height is reserved for future use.

* The variables FrameWidthInMbsY, FrameHeightInMbsY, FrameWidthInSamplesY, FrameHeightInSamplesY, FrameWidthInSamplesC, FrameHeightInSamplesC, FrameSizeInMbsY, and FrameSizeInSamplesY are derived as follows:

- FrameWidthInSamplesY = frame_width
- FrameHeightInSamplesY = frame_height
- FrameWidthInMbsY = ceil(FrameWidthInSamplesY / MbWidth)
- FrameHeightInMbsY = ceil(FrameHeightInSamplesY / MbHeight)
- FrameWidthInSamplesC = FrameWidthInSamplesY // SubWidthC
- FrameHeightInSamplesC = FrameHeightInSamplesY // SubHeightC
- FrameSizeInMbsY = FrameWidthInMbsY * FrameHeightInMbsY
- FrameSizeInSamplesY = FrameWidthInSamplesY * FrameHeightInSamplesY

* chroma_format_idc

  specifies the chroma sampling relative to the luma sampling as specified in Table 2 The value of chroma_format_idc MUST be 0, 2, 3, or 4. Other values of chroma_format_idc are reserved for future use.

* bit_depth_minus8

  specifies the bit depth of the samples. The variables BitDepth and QpBdOffset are derived as follows:

  o BitDepth = bit_depth_minus8 + 8

  o QpBdOffset = bit_depth_minus8 * 6

  bit_depth_minus8 MUST be in the range of 2 to 8, inclusive. Other values of bit_depth_minus8 are reserved for future use.

* capture_time_distance

  indicates time difference between the capture time of the frames in the previous access unit and frames in the current access unit in milliseconds if there has been any access unit preceding the access unit this frame belongs to.

* reserved_zero_8bits

MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

#### 5.3.7. Quantization matrix

```
syntax code                                                   | type
--------------------------------------------------------------|-----
quantization_matrix(){                                        |
  for(i = 0; i < NumComps; i++){                              |
    for(y = 0; y < 8; y++){                                   |
      for(x = 0; x < 8; x++){                                 |
        q_matrix[i][x][y]                                     | u(8)
      }                                                       |
    }                                                         |
  }                                                           |
}                                                             |
```

Figure 14: quantization_matrix() syntax code

* q_matrix[i][x][y]

  specifies a scaling value in the quantization matrices. When q_matrix[i][x][y] is not present, it is inferred to be equal to 16. The array index i specifies an indicator for the color component; when chroma_format_idc is equal to 2 or 3, 0 for Y, 1 for Cb and 2 for Cr. The value of 0 for q_matrix[i][x][y] is reserved for future use.

  The quantization matrix, QMatrix[i][x][y], is derived as follows:

  QMatrix[i][x][y] = q_matrix[i][x][y]

#### 5.3.8. Tile info

```
syntax code                                                    | type
---------------------------------------------------------------|-----
tile_info(){                                                   |
  tile_width_in_mbs                                            | u(20)
  tile_height_in_mbs                                           | u(20)
  startMb=0                                                    |
  for(i = 0; startMb < FrameWidthInMbsY; i++){                 |
    ColStarts[i] = startMb * MbWidth                           |
    startMb += tile_width_in_mbs                               |
  }                                                            |
  ColStarts[i] = FrameWidthInMbsY*MbWidth                      |
  TileCols = i                                                 |
  startMb = 0                                                  |
  for(i = 0; startMb < FrameHeightInMbsY; i++){                  |
    RowStarts[i] = startMb * MbHeight                          |
    startMb += tile_height_in_mbs                              |
  }                                                            |
  RowStarts[i] = FrameHeightInMbsY*MbHeight                      |
  TileRows = i                                                 |
  NumTiles = TileCols * TileRows                               |
  tile_size_present_in_fh_flag                                 | u(1)
  if(tile_size_present_in_fh_flag){                            |
    for(i = 0; i < NumTiles; i++){                             |
      tile_size_in_fh[i]                                       | u(32)
    }                                                          |
  }                                                            |
}                                                              |
```

Figure 15: tile_info() syntax code

* tile_width_in_mbs

  specifies the width of a tile in units of MBs.

* tile_height_in_mbs

  specifies the height of a tile in units of MBs.

* tile_size_present_in_fh_flag

  equal to 1 specifies that tile_size_in_fh[i] is present in Frame header. tile_size_present_in_fh_flag equal to 0 specifies that tile_size_in_fh[i] is not present in Frame header.

* tile_size_in_fh[i]

indicates the size in bytes of i-th tile data in raster order in a frame. The value of tile_size_in_fh[i] MUST have the same value with tile_size[i]. When it is not present, the value of tile_size_in_fh[i] is inferred to be equal to tile_size[i]. The value of 0 for tile_size_in_fh[i] is reserved for future use.

### 5.3.9. Access unit information

```
syntax code                                                   | type
--------------------------------------------------------------|-----
au_info(){                                                    |
  num_frames                                                  | u(16)
  for(i = 0; i < num_frames; i++){                            |
    pbu_type                                                  | u(8)
    group_id                                                  | u(16)
    reserved_zero_8bits                                       | u(8)
    frame_info()                                              |
  }                                                           |
  reserved_zero_8bits                                         | u(8)
  byte_alignment()                                            |
  filler()                                                    |
}                                                             |
```

Figure 16: au_info() syntax code

* num_frames

  indicates the number of frames contained in the current AU.

* pbu_type

  has the same semantics as pbu_type in the pbu_header() syntax. Note: The value of pbu_type MUST be 1, 2, 25, 26, or 27 in bitstreams conforming to this version of document.

* group_id

  has the same semantics as group_id in the pbu_header() syntax.

* reserved_zero_8bits

  MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

### 5.3.10. Metadata

```
syntax code                                                   | type
--------------------------------------------------------------|-----
metadata(){                                                   |
  metadata_size                                               | u(32)
  currReadSize = 0                                            |
  do{                                                         |
    payloadType = 0                                           |
    while(next_bits(8) == 0xFF){                              |
      ff_byte                                                 | f(8)
      payloadType += ff_byte                                  |
      currReadSize++                                          |
    }                                                         |
    metadata_payload_type                                     | u(8)
    payloadType += metadata_payload_type                      |
    currReadSize++                                            |
                                                              |
    payloadSize = 0                                           |
    while(next_bits(8) == 0xFF){                              |
      ff_byte                                                 | f(8)
      payloadSize += ff_byte                                  |
      currReadSize++                                          |
    }                                                         |
    metadata_payload_size                                     | u(8)
    payloadSize += metadata_payload_size                      |
    currReadSize++                                            |
                                                              |
    metadata_payload(payloadType, payloadSize)                |
    currReadSize += payloadSize                               |
  } while(metadata_size > currReadSize)                       |
  filler()                                                    |
}                                                             |
```

Figure 17: metadata() syntax code

* metadata_size

  specifies the size of metadata before filler() in the current PBU.

* ff_byte

  is a byte equal to 0xFF.

* metadata_payload_type

  specifies the last byte of the payload type of a metadata

* metadata_payload_size

  specifies the last byte of the payload size of a metadata

Syntax and semantics of metadata_payload() are specified in Section 8.

### 5.3.11. Filler

```
syntax code                                                    | type
---------------------------------------------------------------|-----
filler(){                                                      |
  while(next_bits(8) == 0xFF)                                  |
    ff_byte                                                    | f(8)
}                                                              |
```

Figure 18: filler() syntax code

* ff_byte

  is a byte equal to 0xFF.

### 5.3.12. Tile

```
syntax code                                                    | type
---------------------------------------------------------------|-----
tile(tileIdx){                                                 |
  tile_header(tileIdx)                                         |
  for(i = 0; i < NumComps; i++){                               |
    tile_data(tileIdx, i)                                      |
  }                                                            |
  while(more_data_in_tile()){                                  |
    tile_dummy_byte                                            | b(8)
  }                                                            |
}                                                              |
```

Figure 19: tile() syntax code

* tile_dummy_byte

  has any pattern of 8-bit string.

### 5.3.13. Tile header

```
syntax code                                                     | type
----------------------------------------------------------------|-----
tile_header(tileIdx){                                           |
  tile_header_size                                              | u(16)
  tile_index                                                    | u(16)
  for(i = 0; i < NumComps; i++){                                |
    tile_data_size[i]                                           | u(32)
  }                                                             |
  for(i = 0; i < NumComps; i++){                                |
    tile_qp[i]                                                  | u(8)
  }                                                             |
  reserved_zero_8bits                                           | u(8)
  byte_alignment()                                              |
}                                                               |
```

Figure 20: tile_header() syntax code

* tile_header_size

  indicates the size of the tile header in bytes.

* tile_index

  specifies the tile index in raster order in a frame. tile_index MUST have the same value with tileIdx.

* tile_data_size[i]

  indicates the size of i-th color component data in a tile in bytes. The array index i specifies an indicator for the color component; when chroma_format_idc is equal to 2 or 3, 0 for Y, 1 for Cb and 2 for Cr. The value of 0 for tile_data_size[i] is reserved for future use.

* tile_qp[i]

  specify the quantization parameter value for i-th color component. The array index i specifies an indicator for the color component; when chroma_format_idc is equal to 2 or 3, 0 for Y, 1 for Cb and 2 for Cr. Qp[i] to be used for the MBs in the tile are derived as follows

  o Qp[i] = tile_qp[i] - QpBdOffset

  o Qp[i] MUST be in the range of -QpBdOffset to 51, inclusive.

* reserved_zero_8bits

MUST be equal to 0 in bitstreams conforming to this version of document. Values of reserved_zero_8bits greater than 0 are reserved for future use. Decoders conforming to a profile specified in Section 9 MUST ignore PBU with values of reserved_zero_8bits greater than 0.

### 5.3.14. Tile data

```
syntax code                                                    | type
---------------------------------------------------------------|-----
tile_data(tileIdx, cIdx){                                      |
  x0 = ColStarts[tileIdx % TileCols]                           |
  y0 = RowStarts[tileIdx // TileCols]                          |
  numMbColsInTile = (ColStarts[tileIdx % TileCols + 1] -       |
          ColStarts[tileIdx % TileCols]) // MbWidth            |
  numMbRowsInTile = (RowStarts[tileIdx // TileCols + 1] -      |
          RowStarts[tileIdx // TileCols]) // MbHeight          |
  numMbsInTile = numMbColsInTile * numMbRowsInTile             |
  PrevDC = 0                                                   |
  PrevDcDiff = 20                                              |
  Prev1stAcLevel = 0                                           |
  for(i = 0; i < numMbsInTile; i++){                           |
    xMb = x0 + ((i % numMbColsInTile) * MbWidth)               |
    yMb = y0 + ((i // numMbColsInTile) * MbHeight)             |
    macroblock_layer(xMb, yMb, cIdx)                           |
  }                                                            |
  byte_alignment()                                             |
}                                                              |
```

Figure 21: tile_data() syntax code

### 5.3.15. Macroblock layer

```
syntax code                                                    | type
---------------------------------------------------------------|-----
macroblock_layer(xMb, yMb, cIdx){                              |
  subW = (cIdx == 0)? 1 : SubWidthC                            |
  subH = (cIdx == 0)? 1 : SubHeightC                           |
  blkWidth = (cIdx == 0)? MbWidth : MbWidthC                   |
  blkHeight = (cIdx == 0)? MbHeight : MbHeightC                |
  TrSize = 8                                                   |
  for(y = 0; y < blkHeight; y += TrSize){                      |
    for(x = 0; x < blkWidth; x += TrSize){                     |
      abs_dc_coeff_diff                                        | h(v)
      if(abs_dc_coeff_diff)                                    |
        sign_dc_coeff_diff                                     | u(1)
      TransCoeff[cIdx][xMb // subW + x][yMb // subH + y] =     |
            PrevDC + abs_dc_coeff_diff *                       |
            (1 - 2*sign_dc_coeff_diff)                         |
      PrevDC =                                                 |
        TransCoeff[cIdx][xMb // subW + x][yMb // subH + y]     |
      PrevDcDiff = abs_dc_coeff_diff                           |
      ac_coeff_coding(xMb // subW + x, yMb // subH + y,        |
            log2(TrSize), log2(TrSize), cIdx)                  |
    }                                                          |
  }                                                            |
}                                                              |
```

Figure 22: macroblock_layer() syntax code

* abs_dc_coeff_diff

  specifies the absolute value of the difference between the current DC transform coefficient level and PrevDC.

* sign_dc_coeff_diff

  specifies the sign of the difference between the current DC transform coefficient level and PrevDC. sign_dc_coeff_diff equal to 0 specifies that the difference has a positive value. sign_dc_coeff_diff equal to 1 specifies that the difference has a negative value.

The transform coefficients are represented by the arrays TransCoeff[cIdx][x0][y0]. The array indices x0, y0 specify the location (x0, y0) relative to the top-left sample for each component of the frame. The array index cIdx specifies an indicator for the color component; when chroma_format_idc is equal to 2 or 3, 0 for Y, 1 for Cb and 2 for Cr. The value of TransCoeff[cIdx][x0][y0] MUST be in the range of -32768 to 32767, inclusive.

### 5.3.16.  AC coefficient coding

```
syntax code                                                    | type
---------------------------------------------------------------|-----
ac_coeff_coding(x0, y0, log2BlkWidth, log2BlkHeight, cIdx){    |
  scanPos = 1                                                  |
  firstAC = 1                                                  |
  PrevLevel = Prev1stAcLevel                                   |
  PrevRun = 0                                                  |
  do{                                                          |
    coeff_zero_run                                             | h(v)
    for(i = 0; i < coeff_zero_run; i++){                       |
      blkPos = ScanOrder[scanPos]                              |
      xC = blkPos & ((1 << log2BlkWidth) - 1)                  |
      yC = blkPos >> log2BlkWidth                              |
      TransCoeff[cIdx][x0+xC][y0 + yC] = 0                     |
      scanPos++                                                |
    }                                                          |
    PrevRun = coeff_zero_run                                   |
    if(scanPos < (1 << (log2BlkWidth + log2BlkHeight))){       |
      abs_ac_coeff_minus1                                      | h(v)
      sign_ac_coeff                                            | u(1)
      level = (abs_ac_coeff_minus1 + 1) *                      |
        (1 - 2 * sign_ac_coeff)                                |
      blkPos = ScanOrder[scanPos]                              |
      xC = blkPos & ((1 << log2BlkWidth) - 1)                  |
      yC = blkPos >> log2BlkWidth                              |
      TransCoeff[cIdx][x0 + xC][y0 + yC] = level               |
      scanPos++                                                |
      PrevLevel = abs_ac_coeff_minus1 + 1                      |
      if(firstAC == 1){                                        |
        firstAC = 0                                            |
        Prev1stAcLevel = PrevLevel                             |
      }                                                        |
    }                                                          |
  } while(scanPos < (1 << (log2BlkWidth + log2BlkHeight)))     |
}                                                              |
```

Figure 23: ac_coeff_coding() syntax code

*  coeff_zero_run

   specifies the number of zero-valued transform coefficient levels that are located before the position of the next non-zero transform coefficient level in a scan of transform coefficient levels.

*  abs_ac_coeff_minus1

plus 1 specifies the absolute value of an AC transform coefficient level at the given scanning position.

* sign_ac_coeff

  specifies the sign of an AC transform coefficient level for the given scanning position. sign_ac_coeff equal to 0 specifies that the corresponding AC transform coefficient level has a positive value. sign_ac_coeff equal to 1 specifies that the corresponding AC transform coefficient level has a negative value.

The array ScanOrder[sPos] specifies the mapping of the zig-zag scan position sPos, ranging from 0 to (1 << log2BlkWidth) * (1 << log2BlkHeight) - 1, inclusive, to a raster scan position rPos. ScanOrder is derived by invoking Section 4.4.1 with input parameters blkWidth equal to (1 << log2BlkWidth) and blkHeight equal to (1 << log2BlkHeight).

### 5.3.17. Byte alignment

```
syntax code                                                    | type
---------------------------------------------------------------|-----
byte_alignment(){                                              |
  while(!byte_aligned())                                       |
    alignment_bit_equal_to_zero                                | f(1)
}                                                              |
```

Figure 24: byte_alignment() syntax code

* alignment_bit_equal_to_zero

  MUST be equal to 0.

## 6. Decoding process

This process is invoked to obtain a decoded frame from a bitstream. Input to this process is a bitstream of a coded frame. Output of this process is a decoded frame.

The decoding process operates as follows for the current frame:

* The syntax structure for a coded frame is parsed to obtain the parsed syntax structures.

* The processes in Section 6.1, Section 6.2 and Section 6.3 specify the decoding processes using syntax elements in all syntax structures. It is the requirement of bitstream conformance that the coded tiles of the frame MUST contain tile data for every MB

of the frame, such that the division of the frame into tiles and the division of the tiles into MBs each forms a partitioning of the frame.

* After all the tiles in the current frame have been decoded, the decoded frame is cropped using the cropping rectangle if FrameWidthInSamplesY is not equal to FrameWidthInMbY * MbWidth or FrameHeightInSamplesY is not equal to FrameHeightInMbsY * MbHeight.

* The cropping rectangle, which specifies the samples of a frame that are output, is derived as follows.

  - The cropping rectangle contains the luma samples with horizontal frame coordinates from 0 to FrameWidthInSampleY - 1 and vertical frame coordinates from 0 to FrameHeightInSamplesY - 1, inclusive.

  - The cropping rectangle contains the two chroma arrays having frame coordinates (x//SubWidthC, y//SubHeightC), where (x,y) are the frame coordinates of the specified luma samples.

## 6.1. MB decoding process

This process is invoked for each MB.

Input to this process is a luma location (xMb, yMb) specifying the top-left sample of the current luma MB relative to the top left luma sample of the current frame. Outputs of this process are the reconstructed samples of all the NumComps color components (when chorma_format_idc is equal to 2 or 3, Y, Cb, and Cr) for the current MB.

The following steps applies:

* Let recSamples[0] be a (MbWidth)x(MbHeight) array of the reconstructed samples of the first color component (when chroma_format_idc is equal to 2 or 3, Y).

* The block reconstruction process as specified in Section 6.2 is invoked with the luma location (xMb, yMb), the variable nBlkW set equal to MbWidth, the variable nBlkH set equal to MbHeight, the variable cIdx set equal to 0, and the (MbWidth)x(MbHeight) array recSamples[0] as inputs, the output is a modified version of the (MbWidth)x(MbHeight) array resSamples[0], which is the reconstructed samples of the first color component for the current MB.

* When chroma_format_idc is not equal to 0, Let recSamples[1] be a (MbWidthC)x(MbHeightC) array of the reconstructed samples of the second color component (when chroma_format_idc is equal to 2 or 3, Cb).

* When chroma_format_idc is not equal to 0, The block reconstruction process as specified in Section 6.2 is invoked with the luma location (xMb, yMb), the variable nBlkW set equal to MbWidthC, the variable nBlkH set equal to MbHeightC, the variable cIdx set equal to 1, and the (MbWidthC)x(MbHeightC) array recSamples[1] as inputs, the output is a modified version of the (MbWidthC)x(MbHeightC) array recSamples[1], which is the reconstructed samples of the second color component for the current MB.

* When chroma_format_idc is not equal to 0, Let recSamples[2] be a (MbWidthC)x(MbHeightC) array of the reconstructed samples of the third color component(when chroma_format_idc is equal to 2 or 3, Cr).

* When chroma_format_idc is not equal to 0, The block reconstruction process as specified in Section 6.2 is invoked with the luma location (xMb, yMb), the variable nBlkW set equal to MbWidthC, the variable nBlkH set equal to MbHeightC, the variable cIdx set equal to 2, and the (MbWidthC)x(MbHeightC) array recSamples[2] as inputs, the output is a modified version of the (MbWidthC)x(MbHeightC) array recSamples[2], which is the reconstructed samples of the third color component for the current MB.

* When chroma_format_idc is equal to 4, let recSamples[3] be a (MbWidthC)x(MbHeightC) array of the reconstructed samples of the fourth color component.

* When chroma_format_idc is equal to 4, the block reconstruction process as specified in Section 6.2 is invoked with the luma location (xMb, yMb), the variable nBlkW set equal to MbWidthC, the variable nBlkH set equal to MbHeightC, the variable cIdx set equal to 3, and the (MbWidthC)x(MbHeightC) array recSamples[3] as inputs, the output is a modified version of the (MbWidthC)x(MbHeightC) array recSamples[3], which is the reconstructed samples of the fourth color component for the current MB.

## 6.2. Block reconstruction process

Inputs to this process are:

* a luma location (xMb, yMb) specifying the top-left sample of the current MB relative to the top left luma sample of the current frame,

* two variables nBlkW and nBlkH specifying the width and the height of the current block,

* a variable cIdx specifying the color component of the current block, and

* an (nBlkW)x(nBlkH) array recSamples of reconstructed block.

Output of this process is a modified version of the (nBlkW)x(nBlkH) array recSamples of reconstructed samples.

The following applies:

* The variables numBlkX and numBlkY are derived as follows:

  o numBlkX = nBlkW // TrSize

  o numBlkY = nBlkH // TrSize

* For yIdx = 0..numBlkY - 1, the following applies:

  o For xIdx = 0..numBlkX - 1, the following applies:

The variables xBlk and yBlk are derived as follows:

  o xBlk = xMb // (cIdx==0? 1: SubWidthC) + xIdx*TrSize

  o yBlk = yMb // (cIdx==0? 1: SubHeightC) + yIdx*TrSize

* The scaling and transformation process as specified in Section 6.3 is invoked with the location (xBlk, yBlk), the variable cIdx set equal to cIdx, the transform width nBlkW set equal to TrSize and the transform height nBlkH set equal to TrSize as inputs, and the output is a (TrSize)x(TrSize) array r of reconstructed block.

* The (TrSize)x(TrSize) array recSamples is modified as follows:

```
recSamples[(xIdx * TrSize) + i, (yIdx * TrSize) + j] = r[i,j],
   with i=0..TrSize-1, j=0..TrSize-1
```

## 6.3. Scaling and transformation process

Inputs to this process are:

* a location (xBlkY, yBlkY) of the current color component specifying the top-left sample of the current block relative to the top-left sample of the current frame,

* a variable cIdx specifying the color component of the current block,

* a variable nBlkW specifying the width of the current block, and

* a variable nBlkH specifying the height of the current block.

Output of this process is the (nBlkW)x(nBlkH) array of reconstructed samples r with elements r[x][y].

The quantization parameter qP is derived as follows:

```
qP = Qp[cIdx] + QpBdOffset
```

The (nBlKW)x(nBlkH) array of reconstructed samples r is derived as follows:

* The scaling process for transform coefficients as specified in Section 6.3.1 is invoked with the block location (xBlkY, yBlkY), the block width nBlkW and the block height nBlkH, the color component variable cIdx, and the quantization parameter qP as inputs, and the output is an (nBlkW)x(nBlkH) array of scaled transform coefficients d.

* The transformation process for scaled transform coefficients as specified in Section 6.3.2 is invoked with the block location (xBlkY, yBlkY), the block width nBlkW and the block height nBlkH, the color component variable cIdx, and the (nBlkW)x(nBlkH) array of scaled transform coefficients d as inputs, and the output is an (nBlkW)x(nBlkH) array of reconstructed samples r.

* The variable bdShift is derived as follows:

  ```
  bdShift = 20 - BitDepth
  ```

* The reconstructed sample values r[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1 are modified as follows:

  ```
  r[x][y] = clip(0, (1 << BitDepth)-1, ((r[x][y]+(1 << (bdShift-
     1)))>>bdShift) + (1 << (BitDepth-1)))
  ```

### 6.3.1. Scaling process for transform coefficients

Inputs to this process are:

* a location (xBlkY, yBlkY) of the current color component specifying the top-left sample of the current block relative to the top-left sample of the current frame,

* a variable nBlkW specifying the width of the current block,

* a variable nBlkH specifying the height of the current block,

* a variable cIdx specifying the color component of the current block, and

* a variable qP specifying the quantization parameter.

Output of this process is the (nBlkW)x(nBlkH) array d of scaled transform coefficients with elements d[x][y].

The variable bdShift is derived as follows:

```
bdShift = BitDepth + ((log2(nBlkW) + log2(nBlkH)) // 2) - 5
```

The list levelScale[] is specified as follows:

```
levelScale[k] = {40, 45, 51, 57, 64, 71} with k = 0..5.
```

For the derivation of the scaled transform coefficients d[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1, the following applies:

* The scaled transform coefficient d[x][y] is derived as follows:

```
d[x][y] = clip(-32768, 32767, ((TransCoeff[cIdx][xBlkY][yBlkY]
   * QMatrix[cIdx][x][y] * levelScale[qP % 6] << (qP//6)) + (1 <<
   (bdShift-1)) >> bdShift))
```

### 6.3.2. Process for scaled transform coefficients

#### 6.3.2.1. General

Inputs to this process are:

* a location (xBlkY, yBlkY) of the current color component specifying the top-left sample of the current block relative to the top-left sample of the current frame,

* a variable nBlkW specifying the width of the current block,

* a variable nBlkH specifying the height of the current block, and

* an (nBlkW)x(nBlkH) array d of scaled transform coefficients with elements d[x][y].

Output of this process is the (nBlkW)x(nBlkH) array r of reconstructed samples with elements r[x][y].

The (nBlkW)x(nBlkH) array r of reconstructed samples is derived as follows:

* Each (vertical) column of scaled transform coefficients d[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1 is transformed to e[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1 by invoking the one-dimensional transformation process as specified in Section 6.3.2.2 for each column x = 0..nBlkW - 1 with the size of the transform block nBlkH, and the list d[x][y] with y = 0..nBlkH - 1 as inputs, and the output is the list e[x][y] with y = 0..nBlkH - 1.

* The following applies:

  g[x][y] = (e[x][y] + 64) >> 7

* Each (horizontal) row of the resulting array g[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1 is transformed to r[x][y] with x = 0..nBlkW - 1, y = 0..nBlkH - 1 by invoking the one-dimensional transformation process as specified in Section 6.3.2.2 for each row y = 0..nBlkH - 1 with the size of the transform block nBlkW, and the list g[x][y] with x = 0..nBlkW - 1 as inputs, and the output is the list r[x][y] with x = 0..nBlkW - 1.

#### 6.3.2.2. Transformation process

Inputs to this process are:

* a variable nTbS specifying the sample size of scaled transform coefficients, and

* a list of scaled transform coefficients x with elements x[j], with j = 0..(nTbS - 1).

* Output of this process is the list of transformed samples y with elements y[i], with i = 0..(nTbS - 1).

* The transformation matrix derivation process as specified in Section 6.3.2.3. invoked with the transform size nTbS as input, and the transformation matrix transMatrix as output.

* The list of transformed samples y[i] with i = 0..(nTbS - 1) is derived as follows:

```
y[i] = sum(j = 0, nTbS - 1, transMatrix[i][j] * x[j])
```

#### 6.3.2.3. Transformation matrix derivation process

Input to this process is a variable nTbS specifying the horizontal sample size of scaled transform coefficients.

Output of this process is the transformation matrix transMatrix.

The transformation matrix transMatrix is derived based on nTbs as follows:

* If nTbS is equal to 8, the following applies:

```
transMatrix[m][n] =
   {
    {  64,  64,  64,  64,  64,  64,  64,  64 }
    {  89,  75,  50,  18, -18, -50, -75, -89 }
    {  84,  35, -35, -84, -84, -35,  35,  84 }
    {  75, -18, -89, -50,  50,  89,  18, -75 }
    {  64, -64, -64,  64,  64, -64, -64,  64 }
    {  50, -89,  18,  75, -75, -18,  89, -50 }
    {  35, -84,  84, -35, -35,  84, -84,  35 }
    {  18, -50,  75, -89,  89, -75,  50, -18 }
   }
```

Figure 25: Transform matrix for nTbS == 8

# 7. Parsing process

## 7.1. Process for syntax element type h(v)

This process is invoked for the parsing of syntax elements with descriptor h(v) in Section 5.3.15 and Section 5.3.16.

### 7.1.1. Process for abs_dc_coeff_diff

Inputs to this process are bits for the abs_dc_coeff_diff syntax element. Output of this process is a value of the abs_dc_coeff_diff syntax element. The variable kParam is derived as follows:

```
kParam = clip(0, 5, PrevDcDiff >> 1)
```

The value of syntax element abs_dc_coeff_diff is obtained by invoking the parsing process for variable length codes as specified in Section 7.1.4 with kParam.

#### 7.1.2. Process for coeff_zero_run

Inputs to this process are bits for the coeff_zero_run syntax element.

Output of this process is a value of the coeff_zero_run syntax element.

The variable kParam is derived as follows:

```
kParam = clip(0, 2, PrevRun >> 2)
```

The value of syntax element coeff_zero_run is obtained by invoking the parsing process for variable length codes as specified in Section 7.1.4 with kParam.

#### 7.1.3. Process for abs_ac_coeff_minus1

Inputs to this process are bits for the abs_ac_coeff_minus1 syntax element.

Output of this process is a value of the abs_ac_coeff_minus1 syntax element.

The variable kParam is derived as follows:

```
kParam = clip(0, 4, PrevLevel >> 2)
```

The value of syntax element abs_ac_coeff_minus1 is obtained by invoking the parsing process for variable length codes as specified in Section 7.1.4 with kParam.

#### 7.1.4. Process for variable length codes

Input to this process is kParam.

Output of this process is a value, symbolValue, of a syntax element.

The symbolValue is derived as follows:

```
symbolValue = 0
parseExpGolomb = 1
k = kParam
stopLoop = 0

if(read_bits(1) == 1){
  parseExpGolomb = 0
}
else{
  if(read_bits (1) == 0){
    symbolValue += (1 << k)
    parseExpGolomb = 0
  }
  else{
    symbolValue += (2 << k)
    parseExpGolomb = 1
  }
}

if(parseExpGolomb){
  do{
    if(read_bits(1) == 1){
      stopLoop = 1
    }
    else{
      symbolValue += (1 << k)
      k++
    }
  } while(!stopLoop)
}

if(k > 0)
  symbolValue += read_bits(k)
```

Figure 26: Parsing process of symbolValue

where the value returned from read_bits(n) is interpreted as a binary representation of a n-bit unsigned integer with most significant bit written first.

## 7.2. Codeword generation process for h(v) (informative)

This process specifies the code generation process for syntax elements with descriptor h(v).

#### 7.2.1. Process for abs_dc_coeff_diff

Input to this process is a symbol value of the abs_dc_coeff_diff syntax element.

Output of this process is a codeword of the abs_dc_coeff_diff syntax element.

The variable kParam is derived as follows:

```
kParam = clip(0, 5, PrevDcDiff >> 1)
```

The codeword of syntax element abs_dc_coeff_diff is obtained by invoking the generation process for variable length codes as specified in Section 7.2.4 with the symbol value symbolValue and kParam.

#### 7.2.2. Process for coeff_zero_run

Input to this process is a symbol value of the coeff_zero_run syntax element.

Output of this process is a codeword of the coeff_zero_run syntax element.

The variable kParam is derived as follows:

```
kParam = clip(0, 2, PrevRun >> 2)
```

The codeword of syntax element coeff_zero_run is obtained by invoking the generation process for variable length codes as specified in Section 7.2.4 with the symbol value symbolValue and kParam.

#### 7.2.3. Process for abs_ac_coeff_minus1

Input to this process is a symbol value of the abs_ac_coeff_minus1 syntax element.

Output of this process is a codeword of the abs_ac_coeff_minus1 syntax element.

The variable kParam is derived as follows:

```
kParam = clip(0, 4, PrevLevel >> 2)
```

The codeword of syntax element abs_ac_coeff_minus1 is obtained by invoking the generation for variable length codes as specified in Section 7.2.4 with the symbol value symbolValue and kParam.

#### 7.2.4. Process for variable length codes

Inputs to this process are symbolVal and kParam

Output of this process is a codeword of a syntax element.

The codeword is derived as follows:

```
PrefixVLCTable[3][2] = {{1, 0}, {0, 0}, {0, 1}}

symbolValue = symbolVal
valPrefixVLC = clip(0, 2, symbolVal >> kParam)
bitCount = 0
k = kParam

while(symbolValue >= (1 << k)){
  symbolValue -= (1 << k)
  if(bitCount < 2)
    put_bits(PrefixVLCTable[valPrefixVLC][bitCount], 1)
  else
    put_bits(0, 1)
  if(bitCount >= 2)
    k++
  bitCount++
}

if(bitCount < 2)
  put_bits(PrefixVLCTable[valPrefixVLC][bitCount], 1)
else
  put_bits(1, 1)

if(k > 0)
  put_bits(symbolValue, k)
```

Figure 27: Generating bits from symbolValue

where a codeword generated from put_bits(v, n) is interpreted as a binary representation of an n-bit unsigned integer value v with most significant bit written first.

## 8. Metadata information

### 8.1. Metadata payload

```
syntax code                                                   | type
--------------------------------------------------------------|-----
metadata_payload(payloadType, payloadSize){                   |
  if(payloadType == 4){                                       |
    metadata_itu_t_t35(payloadSize)                           |
  }                                                           |
  else if(payloadType == 5){                                  |
    metadata_mdcv(payloadSize)                                |
  }                                                           |
  else if(payloadType == 6){                                  |
    metadata_cll(payloadSize)                                 |
  }                                                           |
  else if(payloadType == 10){                                 |
    metadata_filler(payloadSize)                              |
  }                                                           |
  else if(payloadType == 170){                                |
    metadata_user_defined(payloadSize)                        |
  }                                                           |
  else{                                                       |
    metadata_undefined(payloadSize)                           |
  }                                                           |
  byte_alignment()                                            |
}                                                             |
```

Figure 28: metadata_payload() syntax code

## 8.2. List of metadata syntax and semantics

### 8.2.1. Filler metadata

```
syntax code                                                   | type
--------------------------------------------------------------|-----
metadata_filler(payloadSize){                                 |
  for(i = 0; i < payloadSize; i++){                           |
    ff_byte                                                   | f(8)
  }                                                           |
}                                                             |
```

* ff_byte

  is a byte equal to 0xFF.

### 8.2.2. Recommendation ITU-T T.35 metadata

This metadata contains information registered as specified in [ITUT-T35].

```
syntax code                                                   | type
--------------------------------------------------------------|-----
metadata_itu_t_t35(payloadSize){                              |
  itu_t_t35_country_code                                      | b(8)
  readSize = payloadSize - 1                                  |
                                                              |
  if(itu_t_t35_country_code == 0xFF){                         |
    itu_t_t35_country_code_extension                          | b(8)
    readSize--                                                |
  }                                                           |
                                                              |
  while (readSize > 0){                                       |
    itu_t_t35_payload                                         | b(8)
    readSize--                                                |
  }                                                           |
}                                                             |
```

Figure 29: metadata_itu_t_t35() syntax code

* itu_t_t35_country_code

  MUST be a byte having the semantics of country code as specified in Annex A of [ITUT-T35].

* itu_t_t35_country_code_extension

  MUST be a byte having the semantics of country code as specified in Annex B of [ITUT-T35].

* itu_t_t35_payload

  MUST be bytes having the semantics of data registered as specified in [ITUT-T35].

The terminal provider code and terminal provider oriented code as specified in [ITUT-T35] MUST be contained in the first one or more bytes of the itu_t_t35_payload. Any remaining bytes in itu_t_t35_payload data MUST be data having syntax and semantics as specified by the entity identified by the [ITUT-T35] country code and terminal provider code. Note that the registration of any metadata to be carried with this type of payload must be registered through either national administrator, Alliance for Telecommuncations Industry Solutions (ATIS) or ITUT-T Telecommnunication Standardization Bureau (TSB) as specified in [ITUT-T35].

### 8.2.3. Mastering display color volume metadata

```
syntax code                                                    | type
---------------------------------------------------------------|-----
metadata_mdcv(payloadSize){                                    |
  for(i = 0; i < 3; i+ + ) {                                   |
    primary_chromaticity_x[i]                                  | u(16)
    primary_chromaticity_y[i]                                  | u(16)
  }                                                            |
  white_point_chromaticity_x                                   | u(16)
  white_point_chromaticity_y                                   | u(16)
  max_mastering_luminance                                      | u(32)
  min_mastering_luminance                                      | u(32)
}                                                              |
```

Figure 30: metadata_mdcv() syntax code

* primary_chromaticity_x[i]

  specifies a 0.16 fixed-point format of X chromaticity coordinate of mastering display as defined by CIE 1931, where i = 0, 1, 2 specifies Red, Green, Blue respectively.

* primary_chromaticity_y[i]

  specifies a 0.16 fixed-point format of Y chromaticity coordinate of mastering display as defined by CIE 1931, where i = 0, 1, 2 specifies Red, Green, Blue respectively.

* white_point_chromaticity_x

  specifies a 0.16 fixed-point format of white point X chromaticity coordinate of mastering display as defined by CIE 1931.

* white_point_chromaticity_y

  specifies a 0.16 fixed-point format of white point Y chromaticity coordinate as mastering display defined by CIE 1931.

* max_mastering_luminance

  is a 24.8 fixed-point format of maximum display mastering luminance, represented in candelas per square meter.

* min_mastering_luminance

  is a 18.14 fixed-point format of minimum display mastering luminance, represented in candelas per square meter.

#### 8.2.4. Content light level information metadata

```
syntax code                                                   | type
--------------------------------------------------------------|-----
metadata_cll(payloadSize){                                    |
  max_cll                                                     | u(16)
  max_fall                                                    | u(16)
}                                                             |
```

Figure 31: metadata_cll() syntax code

* max_cll

  specifies the maximum content light level information as specified in [CEA-861.3], Appendix A.

* max_fall

  specifies the maximum frame-average light level information as specified in [CEA-861.3], Appendix A.

#### 8.2.5. User defined metadata

This metadata has user data identified by a universal unique identifier as specifies in [ISO11578], the contents of which are not specified in this document.

```
syntax code                                                 | type
------------------------------------------------------------|-----
metadata_user_defined(payloadSize){                         |
  uuid                                                      | u(128)
  for(i = 0; i < (payloadSize - 16); i++)                   |
    user_defined_data_payload                               | b(8)
}                                                           |
```

Figure 32: metadata_user_defined() syntax code

* uuid

  MUST be a 128-bit value specified as a generated UUID according to the procedures of [ISO11578] Annex A.

* user_defined_data_payload

  MUST be a byte having user defined syntax and semantics as specified by the UUID generator.

#### 8.2.6. Undefined metadata

```
syntax code                                                  | type
-------------------------------------------------------------|-----
metadata_undefined(payloadSize){                             |
  for(i = 0; i < payloadSize; i++){                          |
    undefined_metadata_payload_byte                          | b(8)
  }                                                          |
}                                                            |
```

Figure 33: metadata_undefined() syntax code

* undefined_metadata_payload_byte

  is a byte reserved for future case.

## 9. Profiles, levels, and bands

### 9.1. Overview of profiles, levels, and bands

Profiles, levels and bands specify restrictions on a coded frame and hence limits on the capabilities needed to decode the coded frame. Profiles, levels and bands are also used to indicate interoperability points between individual decoder implementations.

> NOTE: This document does not include individually selectable "options" at the decoder, as this would increase interoperability difficulties. Each profile specifies a subset of algorithmic features and limits that MUST be supported by all decoders conforming to that profile.

> NOTE: Encoders are not required to make use of any particular subset of features supported in a profile.

Each level with a band specifies a set of limits on the values that may be taken by the syntax elements of this document. The same set of level and band definitions is used with all profiles, but individual implementations may support a different level for each supported profile. For any given profile, a level with a band generally corresponds to a particular decoder processing load and memory capability.

### 9.2. Requirements on video decoder capability

Capabilities of video decoders conforming to this document are specified in terms of the ability to decode video streams conforming to the constraints of profiles, levels and bands specified in this section. When expressing the capabilities of a decoder for a specified profile, the level and the band supported for that profile MUST also be expressed.

Specific values are specified in this section for the syntax elements profile_idc, level_idc and band_idc. All other values of profile_idc, level_idc and band_idc are reserved for future use.

> NOTE: Decoders SHALL NOT infer that a reserved value of profile_idc between the values specified in this document indicates intermediate capabilities between the specified profiles, as there are no restrictions on the method to be chosen for the use of such future reserved values. However, decoders MUST infer that a reserved value of level_idc and a reserved value of band_idc between the values specified in this document indicates intermediate capabilities between the specified levels.

### 9.3. Profiles

#### 9.3.1. General

All constraints for a coded frame that are specified are constraints for the coded frame that are activated when the bitstream of the access unit is decoded.

#### 9.3.2. 422-10 profile

Conformance of a coded frame to the 422-10 profile is indicated by profile_idc equal to 33.

Coded frames conforming to the 422-10 profile MUST obey the following constraints:

* chroma_format_idc MUST be equal to 2.

* bit_depth_minus8 MUST be equal to 2.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 422-10 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 422-10 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

### 9.3.3. 422-12 profile

Conformance of a coded frame to the 422-12 profile is indicated by profile_idc equal to 44.

Coded frames conforming to the 422-12 profile MUST obey the following constraints:

* chroma_format_idc MUST be equal to 2.

* bit_depth_minus8 MUST be in the range of 2 to 4.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 422-12 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 422-12 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 422-12 profile or the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

#### 9.3.4. 444-10 profile

Conformance of a coded frame to the 444-10 profile is indicated by profile_idc equal to 55.

Coded frames conforming to the 444-10 profile MUST obey the following constraints:

* chroma_format_idc MUST be in the range of 2 to 3.

* bit_depth_minus8 MUST be equal to 2.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 444-10 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 444-10 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 444-10 profile or the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

#### 9.3.5. 444-12 profile

Conformance of a coded frame to the 444-12 profile is indicated by profile_idc equal to 66.

Coded frames conforming to the 444-12 profile MUST obey the following constraints:

* chroma_format_idc MUST be in the range of 2 to 3.

* bit_depth_minus8 MUST be in the range of 2 to 4.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 444-12 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 444-12 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 444-12 profile, the 444-10 profile, the 422-12 profile, or the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

### 9.3.6. 4444-10 profile

Conformance of a coded frame to the 4444-10 profile is indicated by profile_idc equal to 77.

Coded frames conforming to the 4444-10 profile MUST obey the following constraints:

* chroma_format_idc MUST be in the range of 2 to 4.

* bit_depth_minus8 MUST be equal to 2.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 4444-10 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 4444-10 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 4444-10 profile, the 444-10 profile or the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

### 9.3.7. 4444-12 profile

Conformance of a coded frame to the 4444-12 profile is indicated by profile_idc equal to 88.

Coded frames conforming to the 4444-12 profile MUST obey the following constraints:

* chroma_format_idc MUST be in the range of 2 to 4.

* bit_depth_minus8 MUST be in the range of 2 to 4.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 4444-12 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 4444-12 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 4444-12 profile, the 4444-10 profile, the 444-12 profile, the 444-10 profile, the 422-12 profile or the 422-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

### 9.3.8. 400-10 profile

Conformance of a coded frame to the 400-10 profile is indicated by profile_idc equal to 99.

Coded frames conforming to the 400-10 profile MUST obey the following constraints:

* chroma_format_idc MUST be equal to 0.

* bit_depth_minus8 MUST be equal to 2.

* pbu_type MUST be equal to 1

The level and the band constraints specified for the 400-10 profile in Section 9.4 MUST be fulfilled. Decoders conforming to the 400-10 profile at a specific level (identified by a specific value of L) and a specific band (identified by a specific value of B) MUST be capable of decoding all coded frames for which all of the following conditions apply:

* The coded frame is indicated to conform to the 400-10 profile.

* The coded frame is indicated to conform to a level (by a specific value of level_idc) that is lower than or equal to level L.

* The coded frame is indicated to conform to a band (by a specific value of band_idc) that is lower than or equal to level B.

## 9.4. Levels and bands

### 9.4.1. General

For purposes of comparison of level capabilities, a particular level of each band is considered to be a lower level than some other level when the value of the level_idc of the particular level of each band is less than that of the other level.

* The luma sample rate (luma samples per second) MUST be less than or equal to "Max luma sample rate".

* The coded data rate (bits per second) MUST be less than or equal to "Max luma sample rate".

* The value of tile_width_in_mbs MUST be greater than or equal to 16.

* The value of tile_height_in_mbs MUST be greater than or equal to 8.

* The value of TileCols MUST be less than or equal to 20.

* The value of TileRows MUST be less than or equal to 20.

### 9.4.2. Limits of levels and bands

Table 4 specifies the limits for each level of each band. A level to which a coded frame conforms is indicated by the syntax elements level_idc and band_idc as follows:

* level_idc MUST be set equal to a value of 30 times the level number specified in Table 4.

| level | Max luma sample rate (sample/sec) | Max coded data rate (Mbits/sec) band_idc==0 | Max coded data rate (Mbits/sec) band_idc==1 | Max coded data rate (Mbits/sec) band_idc==2 | Max coded data rate (Mbits/sec) band_idc==3 |
|---|---|---|---|---|---|
| 1 | 3,041,280 | 8 | 11 | 15 | 23 |
| 1.1 | 6,082,560 | 16 | 21 | 30 | 45 |
| 2 | 15,667,200 | 39 | 54 | 76 | 114 |
| 2.1 | 31,334,400 | 78 | 108 | 152 | 227 |
| 3 | 66,846,720 | 114 | 159 | 222 | 333 |
| 3.1 | 133,693,440 | 227 | 317 | 444 | 666 |
| 4 | 265,420,800 | 455 | 637 | 892 | 1,338 |
| 4.1 | 530,841,600 | 910 | 1,274 | 1,784 | 2,675 |
| 5 | 1,061,683,200 | 1,820 | 2,548 | 3,567 | 5,350 |
| 5.1 | 2,123,366,400 | 3,639 | 5,095 | 7,133 | 10,699 |
| 6 | 4,777,574,400 | 7,278 | 10,189 | 14,265 | 21,397 |
| 6.1 | 8,493,465,600 | 14,556 | 20,378 | 28,529 | 42,793 |
| 7 | 16,986,931,200 | 29,111 | 40,756 | 57,058 | 85,586 |
| 7.1 | 33,973,862,400 | 58,222 | 81,511 | 114,115 | 171,172 |

Table 4: General level limits

Table 5 shows widely used typical configurations of resolution and frame rate of video and corresponding levels for them.

| use case | resolution | frame per second | luma sample per second | level |
|---|---|---|---|---|
| 720p | 1280 x 720 | 30 | 27,648,000 | 2.1 |
| FHD | 1920 x 1080 | 30 | 62,208,000 | 3 |
| UHD 4K | 3840 x 2160 | 60 | 497,664,000 | 4.1 |
| UHD 4K | 3840 x 2160 | 120 | 995,328,000 | 5 |
| UHD 8K | 7680 x 4320 | 60 | 1,990,656,000 | 5.1 |
| UHD 8K | 7680 x 4320 | 120 | 3,981,312,000 | 6 |

Table 5: Example of typical video configurations and corresponding levels (informative)

## 10. Security considerations

APV decoder MUST take appropriate security considerations into account.

Like any other audio or video codec, APV should not be used with insecure ciphers or cipher modes that are vulnerable to known plaintext attacks. Some of the header bits as well as the padding are easily predictable.

Implementations of the APV codec need to take appropriate security considerations into account. Those related to denial of service are outlined in Section 2.1 of [RFC4732]. A decoder MUST be robust against any non-compliant or malicious payloads. Malicious payloads MUST NOT cause the decoder to overrun its allocated memory or to take an excessive amount of resources to decode. An overrun in allocated memory could lead to arbitrary code execution by an attacker. The same applies to the encoder, even though problems in encoders are typically rarer. Malicious video streams MUST NOT cause the encoder to misbehave because this would allow an attacker to attack transcoding gateways. A frequent security problem in image and video codecs is failure to check for integer overflows. An example is allocating "frame_width * frame_height" in pixel count computations without considering that the multiplication result may have overflowed the range of the arithmetic type. The implementation MUST ensure that no read outside allocated and initialized memory occurs.

A decoder MUST NOT try to process the metadata whose type is not recognized by the implementation. Failure of processing of any metadata exactly according to the syntax structure specificed MAY put a decoder in unknown status.

None of the content carried in APV is intended to be executable.

## 11. IANA considerations

This document has no actions for IANA.

## 12. Appendix

### 12.1. Raw bitstream format

```
syntax code                                                    | type
---------------------------------------------------------------|-----
raw_bitstream_access_unit(){                                   |
    au_size                                                    | u(32)
    access_unit(au_size)                                       |
}                                                              |
```

Figure 34: raw_bitstream_access_unit() syntax code

* au_size

  indicates the size of access unit in bytes. 0 is prohibited and 0xFFFFFFFF is reserved.

### 12.2. APV implementations

#### 12.2.1. OpenAPV open source project

The Academy Software Foundation (ASWF) [ASWF] jointly found by Academy of Motion Picture Arts and Science (AMPAS) [AMPAS] and the Linux Foundation has created an open source software development project conformant to this document [OpenAPV]. The project also provides various test vectors for verification of the implementations.

#### 12.2.2. Android open source project

The Android open source project (AOSP) has implemented Advanced Professional Video (APV) conformant to this document [AOSP16APV].

### 12.2.3. FFmpeg open source project

The FFmpeg project is developing APV decoder [FFmpegAPVdec] and APV encdoer [FFmpegAPVenc] conformant to this document.

## 13. References

### 13.1. Normative References

[CEA-861.3]
"CEA-861.3, HDR Static Metadata Extension", January 2015.

[H273] "Recommendation ITU-T H.273 (V4) | ISO/IEC 23091-2:2025, Coding-independent code points - Part 2 Video", July 2024, <https://www.itu.int/rec/T-REC-H.273>.

[ISO11578] "ISO/IEC 11578:1996, Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)", December 1996, <https://www.iso.org/standard/2229.html>.

[ISO9899] "ISO/IEC 9899:2018, Information technology - Programming languages - C", June 2018, <https://www.iso.org/standard/74528.html>.

[ITUT-T35] "Recommendation ITU-T T.35, Procedure for the allocation of ITU-T defined codes for non-standard facilities", February 2000, <https://www.itu.int/rec/T-REC-T.35>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4732] Handley, M., Ed., Rescorla, E., Ed., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, DOI 10.17487/RFC4732, December 2006, <https://www.rfc-editor.org/rfc/rfc4732>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

### 13.2. Informative References

[AMPAS] "Academy of Motion Picture Arts and Science", n.d., <https://www.oscars.org/>.

[AOSP16APV]
"Android open source project version 16", n.d., <https://developer.android.com/about/versions/16/features#apv>.

[ASWF] "The Academy Software Foundation", n.d., <https://www.aswf.io/>.

[FFmpegAPVdec]
"FFmpeg implementation of APV decoder", n.d., <https://git.ffmpeg.org/gitweb/ffmpeg.git/commit/483cadf8d77d3260eec8781f5f18c50f27e468f8>.

[FFmpegAPVenc]
"FFmpeg implementation of APV encoder", n.d., <https://git.ffmpeg.org/gitweb/ffmpeg.git/commit/fab691edaf53bbf10429ef3448f1f274e5078395>.

[OpenAPV] "OpenAPV", n.d., <https://github.com/AcademySoftwareFoundation/openapv>.

## Authors' Addresses

Youngkwon Lim
Samsung Electronics
6105 Tennyson Pkwy, Ste 300
Plano, TX, 75024
United States of America
Email: yklwhite@gmail.com

Minwoo Park
Samsung Electronics
34, Seongchon-gil, Seocho-gu
Seoul
3573
Republic of Korea
Email: m.w.park@samsung.com

Madhukar Budagavi
Samsung Electronics
6105 Tennyson Pkwy, Ste 300
Plano, TX, 75024
United States of America
Email: m.budagavi@samsung.com

Rajan Joshi
Samsung Electronics
11488 Tree Hollow Ln
San Diego, CA, 92128
United States of America
Email: rajan_joshi@ieee.org

Kwang Pyo Choi
Samsung Electronics
34 Seongchon-gil Seocho-gu
Seoul
3573
Republic of Korea
Email: kwangpyo.choi@gmail.com