HD Photo

Photographic Still Image File Format

This file format is also known under the name Windows Media™ Photo

Bitstream Specification

Version	1.0
Status	Release

Microsoft Corporation Technical Documentation License Agreement for the specification "HD Photo Bitstream"

READ THIS! THIS IS A LEGAL AGREEMENT BETWEEN MICROSOFT CORPORATION ("MICROSOFT") AND THE RECIPIENT OF THE ABOVE REFERENCED MATERIALS, WHETHER AN INDIVIDUAL OR AN ENTITY ("YOU"). IF YOU HAVE ACCESSED THIS AGREEMENT IN THE PROCESS OF DOWNLOADING THESE MATERIALS ("MATERIALS") FROM A MICROSOFT WEB SITE, BY CLICKING "I ACCEPT", DOWNLOADING, USING OR PROVIDING FEEDBACK ON THE MATERIALS, YOU AGREE TO THESE TERMS. IF THIS AGREEMENT IS ATTACHED TO MATERIALS, BY ACCESSING, USING OR PROVIDING FEEDBACK ON THE ATTACHED MATERIALS, YOU AGREE TO THESE TERMS. IF YOU DO NOT AGREE TO THESE TERMS, YOU ARE NOT AUTHORIZED TO ACCESS, DOWNLOAD, USE OR REVIEW THE MATERIALS.

For good and valuable consideration, the receipt and sufficiency of which are acknowledged, You and Microsoft agree as follows:

1. You may review these Materials only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with a Microsoft product, specification, service or technology ("Microsoft Product") as described in these Materials; and (b) to provide feedback on these Materials to Microsoft. All other rights are retained by Microsoft; this Agreement does not give You rights under any Microsoft patents. You may not (i) duplicate any part of these Materials, (ii) remove this Agreement or any notices from these Materials, or (iii) give any part of these Materials, or assign or otherwise provide Your rights under this Agreement, to anyone else.

2. These Materials may contain preliminary information or inaccuracies, and may not correctly represent any associated Microsoft Product as commercially released. All Materials are provided entirely "AS IS." To the extent permitted by law, MICROSOFT MAKES NO WARRANTY OF ANY KIND, DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, AND ASSUMES NO LIABILITY TO YOU FOR ANY DAMAGES OF ANY TYPE IN CONNECTION WITH THESE MATERIALS OR ANY INTELLECTUAL PROPERTY IN THEM.

3. If You are an entity and (a) merge into another entity or (b) a controlling ownership interest in You changes, Your right to use these Materials automatically terminates and You must destroy them.

4. You have no obligation to give Microsoft any suggestions, comments or other feedback ("Feedback") relating to these Materials. However, any Feedback you voluntarily provide may be used in Microsoft Products and related specifications or other documentation (collectively, "Microsoft Offerings") which in turn may be relied upon by other third parties to develop their own products, services or technology ("Third Party Products"). Accordingly, if You do give Microsoft Feedback on any version of these Materials or the Microsoft Offerings to which they apply, You agree: (a) Microsoft may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any Microsoft Offering; (b) You also grant third parties, without charge, only those patent rights necessary to enable Third Party Products to use, implement or interface with any specific parts of a Microsoft Product that incorporate Your Feedback; and (c) You will not give Microsoft any Feedback (i) that You have reason to believe is subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any Microsoft Offering incorporating or derived from such Feedback, or other Microsoft intellectual property, to be licensed to or otherwise shared with any third party.

5. Microsoft has no obligation to maintain the confidentiality of any Microsoft Offering, or the confidentiality of Your Feedback, including Your identity as the source of such Feedback.

6. This Agreement is governed by the laws of the State of Washington. Any dispute involving it must be brought in the federal or state superior courts located in King County, Washington, and You waive any defenses allowing the dispute to be litigated elsewhere. If there is litigation, the losing party must pay the other party's reasonable attorneys' fees, costs and other expenses. If any part of this Agreement is unenforceable, it will be considered modified to the extent necessary to make it enforceable, and the remainder shall continue in effect. This Agreement is the entire agreement between You and Microsoft concerning these Materials; it may be changed only by a written document signed by both You and Microsoft.

Contents

Preface

About This Specification

HD Photo is a file format and associated codec specifically designed to for use with all types of continuous tone photographic content. This document describes the features and capabilities of the HD Photo Version 1.0 release. This version is compatible with the implementation of HD Photo in the released versions of Windows Vista, Windows Presentation Foundation (WPF) and Windows Imaging Component (WIC).

The information contained in this specification is subject to change. Every effort has been made to ensure accuracy at the time of publication.

This bitstream specification is written for device manufacturers implementing embedded support for the HD Photo file format encoding and/or decoding and should be used in conjunction with the reference source code and other information provided in the HD Photo Device Porting Kit.

Additional information covering the HD Photo file container format and metadata tags can be found in the HD Photo Feature Specification. This separate document also contains information required by Windows software developers to implement support for HD Photo encoding and/or decoding.

Contact Information

For questions, comments or requests for additional information, you may contact the owners of this specification at **hdphoto @ microsoft.com**. Additional information, best practices, tools, utilities, sample code, sample image content, links to additional resources and community discussion can currently be found at **http://blogs.msdn.com/billcrow**.

Licensing Notes

This document is part of the HD Photo Device Porting Kit, and may only be used under the terms of the associated license. Information on licensing this DPK, including for use in XPS, is currently available at www.microsoft.com/windows/windowsmedia/wmphoto.

Trademark Notice

Windows Media™ is a registered trademark of Microsoft Corporation. All rights are reserved.

Formatting Conventions

This specification uses the following formatting conventions:

Terms are formatted like *this*.

Important comments, typically highlighting unimplemented or preliminary features look like this.

```
Code looks like this.
```

Raw text and editorial notes look like this.

Conformance Notation

Documents consist of normative, and optionally informative text. Normative text is that text which describes elements of the design that are indispensable or contains the conformance language keywords: "shall", "should" or "may".

Informative text is text that is potentially helpful to the user, but not indispensable and can be removed, changed or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in the document is normative except: the Introduction, any section explicitly labeled as "Informative", or individual paragraphs that start with "Note:".

Normative references are those external documents referenced in normative text and are indispensable to the user.

Bibliographic references are those references made from informative text or are otherwise not indispensable to the user.

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.

The keywords, "should" and "should not" indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate a course of action permissible within the limits of the document.

The keyword, "reserved" indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword, "forbidden" indicates "reserved" and in addition indicates that the provision shall never be defined in the future.

A conformant implementation is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement optional provisions ("may") and need not implement them as described.

Chapter 1. OVERVIEW

1.1 Objectives for Introducing a New Still Image Format

Today's file formats for continuous tone images present many limitations in maintaining the highest image quality or delivering the most optimal system performance. HD Photo was designed to remove these limitations. The design objectives include:

- High performance, embedded system friendly compression
    - Small memory footprint
    - Simple, integer-only operations (no divides)
- Industry-leading compression quality
- Lossless or lossy compression using the same algorithm
- Support a very wide range of pixel formats:
    - Monochrome, RGB, CMYK or n-Channel image representation
    - 8 or 16-bit unsigned integer
    - 16 or 32-bit signed integer
    - 16 or 32-bit floating point
    - Several packed bit formats
        - 1bpc monochrome
        - 5 or 10bpc RGB
        - RGBE Radiance
- Simple, extensible TIFF-like container structure
- Planar or interleaved alpha channel
- Embedded ICC Profile
- EXIF and XMP metadata

HD Photo is the only format that offers high dynamic range image encoding, lossless or lossy compression, multiple color formats, and performance that enables practical in-device implementation.

1.2 Format Identification

Throughout the development of the HD Photo still image file format, the project was first known (internal to Microsoft) by its code name "Photon". Througout it's Beta release, and when publicly introduced at WinHEC 2006, the file format was named "Windows Media™ Photo." With the introduction of the released version of the Device Porting Kit, the format name has been officially changed to "HD Photo."

"Windows Media™ Photo" is still used as the name for the Windows implementation of the "HD Photo" format, in both Windows Vista and the Windows Image Component (WIC) and .NET Frameworks 3.0 redistibutable packages for down-level versions of Windows. "Windows Media™ Photo" is still used internally with this release of the Device Porting Kit.

Windows Media™ Photo and HD Photo refer to exactly the same file format. There are no features or changes that differentiate one from the other, and they share the same bit stream. They are two names for the same format.

We have extended the specification to allow two different file extensions – either .wdp or .hdp – to be used to identify an HD Photo (or Windows Media™ Photo) file. With the 1.0 release of Windows Vista, only the .wdp extension is recognized. This will be updated at some point in the future. Applications that support the HD Photo file format should recognize either extension, and ideally offer the option to create files with either extension.

While we recognize the potential confusion and issues that this name change can create, we believe the new name, initiated in part based on feedback from our partners, provides a better identification for broad, cross-platform support for this new still image file format.

1.3 Compression Algorithm Overview

HD Photo employs a new, state-of-the-art compression algorithm optimized for the digital photography market. HD Photo offers image quality comparable to JPEG-2000 with computational and memory performance more closely comparable to JPEG. HD Photo delivers a lossy compressed image of better perceptive quality than JPEG at less than half the file size. The same compression algorithm can also deliver mathematically lossless compressed images that are typically 2.5 times smaller than the original uncompressed data.

HD Photo uses a very high performance reversible color space conversion, a reversible lapped biorthogonal transform and an advanced non-arithmentic entropy coding scheme. The transform requires at most 3 non-trivial (multiply plus addition) and 7 trivial (addition or shift) operations per pixel (with no divisions) at the highest quality level. In the highest performance mode, only 1 non-trivial and 4 trivial operations per pixel are required. The image is processed in 16x16 macro blocks, allowing a minimal memory footprint for embedded implementations.

HD Photo provides native support for both RGB and CMYK, providing a reversible color transform for each of these color formats to an internal luminance-dominant format used for optimal compression efficiency. In addition HD Photo supports YUV, monochrome and arbitrary n-channel color formats.

Because the transforms employed are fully reversible, the codec supports both lossless and lossy operation using a single algorithm. This significantly simplifies the implementation for embedded applications.

HD Photo supports a wide range of popular numerical encodings at multiple bit depths. 8-bit and 16-bit formats, as well as some specialized packed bit formats are supported for both lossy and lossless compression. 32-bit formats are only supported using lossy compression as only 24 bits are typically retained through the various transforms designed to achieve maximum compression efficiency. While HD Photo uses integer arithmetic exclusively for it's internal processing, an innovative color transform process provides lossless encoding support for both fixed and floating point image information. This also enables extremely efficient conversion between different color formats as part of the encode/decode process.

A HD Photo file is defined by a container and an elementary bitstream. This nomenclature is consistent with container and elementary bitstream definitions for audio and video content. The container is a higher level structure that contains one or more elementary bitstreams, plus descriptive metadata. Each elementary bitstream defines an image. The container further contains information about each image, annotation, metadata, audio data, and definition of relationship between images. The HD Photo container is fully described in the "HD Photo Preliminary Specification."

This document describes the data format of the elementary bitstream. The elementary bitstream is generally created by an encoder that operates on an input image. Further, the elementary bitstream is intended for consumption by a decoder, which reconstructs the input image or an approximation thereof. Although the bitstream is fully described by describing the decoder, this specification defines both the encoder and decoder operations. It is understood that the encoder may choose among multiple possibilities in encoding an image, and each elementary bitstream may reconstruct to a different approximation of the input.

Chapter 2. Background

2.1 Definitions

2.1.1 Alpha channel

The *alpha channel* is a planar structure defining the transparency of the corresponding grayscale or color image. Typically, and with HD Photo, the *alpha channel* has the same dimensions as the luminance channel.

2.1.2 Block

A *block* is defined in HD Photo to be a 4×4 group of pixels. *Blocks* are aligned to 4-pixel offsets from the left and top of the image. A *block* may also refer to the transform coefficients or other intermediate stage of processing of the 4×4 pixel group. Typically, a *block* refers only to one color channel. The term *block* may be used to describe other sized areas as well when indicated, e.g. "8×8 block".

2.1.3 Macroblock

A *macroblock* is defined in HD Photo to be a 16×16 group of pixels. Typically, a *macroblock* refers to the collection of co-located groups of 16×16 pixels across all color channels, and possibly the alpha channel. Further, for YUV420/YUV422 color formats, a *macroblock* refers to 16×16 pixel groups in the luminance channel, and 8×8/16×8 pixel groups in the chrominance channels. A *macroblock* is composed of integer blocks of pixels.

2.1.4 Luminance

The *luminance* channel or plane of an image roughly corresponds to a panchromatic or grayscale rendering of the image. The *luminance* channel is internally generated from the RGB (red/green/blue) input by the process of color conversion. Alternatively, the input may be a native YUV image in which case the *luminance* channel refers to the Y color plane. When the input is a grayscale image, the *luminance* channel is the original image itself. For an n-channel input image, the first channel is regarded as the *luminance* channel.

2.1.5 Chrominance

The *chrominance* channels or planes of an image correspond to difference planes that contain the color information of the image. Typically, there are two *chrominance* channels, except for the case of grayscale, CMYK, Bayer and n-channel images. In these cases, there are zero, two, two and n-1 *chrominance* channels. Further, *chrominance* channels are smaller in size than luminance channels for YUV 4:2:0 and YUV 4:2:2 images.

2.1.6 Macroblock alignment

HD Photo compresses groups of Macroblocks. Images are not always sized in terms of multiples of 16. When an image width or height is not a multiple of 16, the image is extended by some means (such as padding with boundary pixels) to the left and bottom such that the extended image is the nearest higher multiple of 16. This is referred to as *Macroblock alignment*. *Macroblock alignment* only happens at the right and bottom of the image.

2.1.7 Windowing

Windowing refers to the coding of an image with a redundant margin outside of the image. This is usually due to lossless cropping of a sub-image shifted by a non-multiple of 16 offset from a larger image, which results in an unusable area around the image caused by misalignment with macroblock boundaries. The windowing margin may be between 0 and 31 pixels, independently specified on each image side.

Chapter 3. Image and Bitstream Structures

3.1 Image Structure Hierarchy

An image is as follows:

An image is composed of multiple color planes (or a single plane). The first color plane is referred to as luminance and roughly corresponds to a monochrome representation of the image. The remaining color planes are referred to as chrominance. Generally, luminance and chrominance planes are of the same size.

Grayscale images have no chrominance planes.

> For the special case of YUV4:2:2, the chrominance planes are half the width of the luminance plane. The luminance plane width is even.
>
> For the special case of YUV4:4:4, the chrominance planes are half the width and half the height of the luminance plane. The luminance plane width and height are even.

In addition to the luminance and chrominance planes, an image may carry an alpha plane of the same size as the luminance. This carries transparency information.

Each image is composed of non-overlapping 4x4 blocks.

Blocks form a regular pattern on the plane.

Blocks cover the entire image, and may spill over the image boundaries. In this case, pixel values outside of the image and within the block are arbitrarily defined (such as by extension or extrapolation).

4x4 blocks are equally applicable to luminance, chrominance and alpha planes regardless of color format.

> For color formats other than grayscale, YUV 4:2:0 and YUV 4:2:2, blocks of all colorplanes are collocated. Blocks of YUV 4:2:0 and YUV 4:2:2 chrominance planes are collocated.

Further, blocks are grouped into non-overlapping 4x4 clusters, known as macroblocks. Each macroblock further contains blocks of all colorplanes.

Macroblocks form a regular pattern on the image.

> Macroblocks of grayscale images are composed of 16 blocks.
>
> Macroblocks of YUV 4:2:0 images are composed of 16 luminance blocks, and 4 each of U and V plane blocks in a 2x2 pattern collocated with the 16 luminance blocks.
>
> Macroblocks of YUV 4:2:2 images are composed of 16 luminance blocks, and 8 each of U and V plane blocks in a 4x2 pattern collocated with the 16 luminance blocks.
>
> Macroblocks of YUV 4:4:4 images are composed of 48 blocks, 16 for each plane in collocated 4x4 patterns, etc.

Macroblocks cover the entire image, and may spill over the image boundaries. In this case, pixel values outside of the image and within the macroblock are arbitrarily defined (such as by extension or extrapolation).

Macroblocks are grouped into regular structures called tiles.

Tiles form a regular pattern on the image – in other words, tiles in a horizontal row are of the same height and aligned; tiles in a vertical column are of the same width and aligned.

Subject to the above, tiles may be of arbitrary size which is a multiple of 16 and macroblock aligned.

An image may contain between 1 and 256 columns of tiles in the horizontal direction and between 1 and 256 rows of tiles in the vertical direction. Thus, an image may contain between 1 and 65536 tiles.

When an image contains one tile, it is said to be "untiled". If the number of tiles is greater than 1, the image is said to be "tiled".

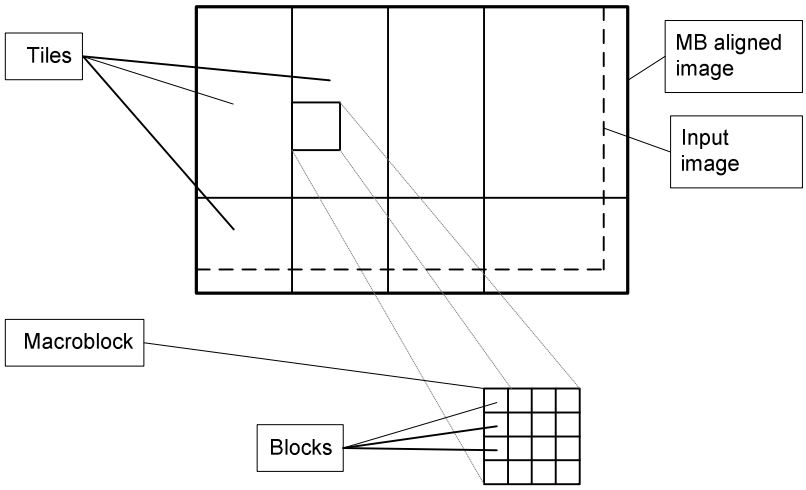


Figure 1: Image structure hierarchy: (i) macroblock aligned extrapolated image (bold rectangle) showing (ii) original image edges on left and bottom (dashed lines), (iii) 2x4 regular tiling pattern, (iv) macroblock in tile (1,2) and (v) blocks within macroblock (expanded subfigure). Color planes are not explicitly shown.

3.2 Bitstream Structure

There are two fundamental modes of operation of HD Photo affecting the structure of the bitstream – Spatial and Frequency. In both the modes, the bitstream is laid out as a header, followed by a sequence of tiles as shown in Figure 2.

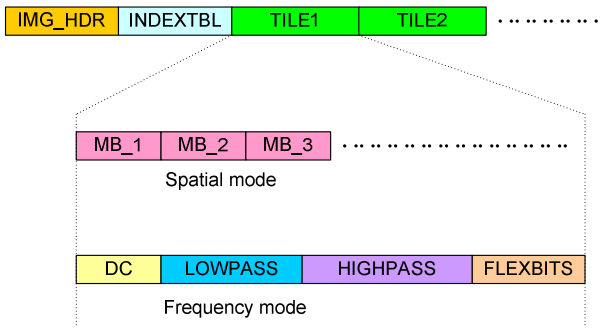


Figure 2: Layout of HD Photo bitstream. Image header is followed by a sequence of tiles which are in Spatial or Frequency mode.

In the Spatial Mode, the bitstream of each tile is laid out in macroblock order. The compressed bits pertinent to each macroblock are located together. Macroblock data is laid out in raster scan order, i.e. scanning left to right, top to bottom.

In the Frequency Mode, the bitstream of each tile is laid out as a hierarchy of bands. The first band is referred to as DC. The DC band carries information of the DC value of each macroblock, in raster scan order. The second band is referred to as Lowpass. This band carries information of the lowpass coefficients which are fifteen in number in each macroblock (for each color plane, with some exceptions). The third band is called the AC band and carries information of the remaining 240 coefficients of each macroblock colorplane (with some exceptions). Finally, the fourth band called Flexbits is an optional layer that carries information regarding the low order bits of the AC coefficients particularly for lossless and low loss cases. (For higher QPs, Flexbits is often null.)

3.3 Precision and Word Length

The HD Photo encoder and decoder perform integer operations. Further, HD Photo supports lossless encoding and decoding. Therefore, the primary machine precision required by HD Photo is integer.

However, integer operations defined in HD Photo lead to rounding errors for lossy coding. These errors are small by design, however they cause drops in the rate distortion curve. For the sake of improved coding performance by the reduction of rounding errors, HD Photo defines a secondary machine precision. In this mode, the input is pre multiplied by 8 (i.e. left shifted by 3 bits) and the final output is divided by 8 with rounding (i.e. right shifted by 3 bits). These operations are carried out at the front end of the encoder and the rear end of the decoder, and are largely invisible to the rest of the processes. Further, the quantization levels are scaled accordingly such that a stream created with the primary machine precision and decoded using the secondary machine precision (and vice versa) produces an acceptable image.

The secondary machine precision cannot be used when lossless compression is desired. The machine precision used in creating a compressed file is explicitly marked in the header.

The secondary machine precision is equivalent to using scaled arithmetic in the codec, and hence this mode is referred to as Scaled. The primary machine precision is referred to as Unscaled.

HD Photo is designed to provide good encoding and decoding speed. A design goal of HD Photo is that the data values on both encoder and decoder do not exceed 16 signed bits for an 8 bit input. (However, intermediate operation within a transform stage may exceed this figure.) This holds true for both modes of machine precision.

Conversely, when the primary machine precision is chosen, the range expansion of the intermediate values is by 8 bits. Since the secondary machine precision performs a pre-multipliation by 8, its range expansion is 8 + 3 = 11 bits.

HD Photo uses two different word lengths for intermediate values. These word lengths are 16 and 32 bits.

Chapter 4. Processing Steps

4.1 Overview

4.1.1 Encoder Processing Steps

The HD Photo encoder carries out the following steps:

1. Pre-scaling
2. Color conversion
3. Transform
    a. Outer pre-filter
    b. Outer HD Photo Core Transform or PCT
    c. Inner pre-filter
    d. Inner PCT
4. Quantization
5. Coefficient scanning
6. Entropy coding
7. Interleaving

These steps are explained in the sections below.

4.1.2 Decoder Processing Overview

The HD Photo decoder performs the inverse operations as shown below:

1. De-interleaving
2. Entropy decoding
3. Coefficient retrieval
4. Dequantization
5. Inverse transform
    a. Inner PCT
    b. Inner post-filter
    c. Outer PCT
    d. Outer post-filter
6. Inverse color conversion
7. Post-scaling

These steps are explained in the sections below.

4.2 Pre/post Scaling

The pre/post scaling steps are optional and are used only when the input data range is greater than 27/24 bits. On the encoder side, the input data is right-shifted by some number *m* bits such that the data range is reduced to 27/24 bits or below. On the decoder side, the output is left shifted by *m* bits. The 27 bit limit is used when data is scaled, and the 24 bit limit applies when the data is unscaled. For the most common cases such as for 16 bit data, the pre/post scaling steps are omitted although they may still be used. The number of shift bits *m* is sent in the field of "SHIFT_BITS" in the IMAGE_PLANE_HEADER.

4.3 Color Conversion

4.3.1 Overview

HD Photo uses a reversible color conversion to convert between the external and internal color formats. Several external color formats are supported, including:

1. Grayscale
2. RGB/BGR
3. CMYK
4. YUV 4:2:0
5. YUV 4:2:2
6. YUV 4:4:4
7. Bayer patterns

Of these, the YUV formats are already in an efficient representation used internally. The color conversion operation on the encoder side multiplies the data by 8 (and divides the data by 8 on the decoder side) if the scaled mode is used.

For the other external formats, color conversion is applied to convert between external and internal formats. The internal formats are:

1. Grayscale
2. YUV 4:2:0
3. YUV 4:2:2
4. YUV 4:4:4
5. YUVK, a four color format

In order to convert from RGB to/from YUV 4:2:X, downsampling/upsampling must be performed. Refer to the code in strenc.c and strdec.c. Grayscale conversions may be treated as YUV 4:4:4 conversions with the understanding that the U, V channels may be deleted on the encoder side, and replaced with zeros on the decoder side.

The remaining color conversions are described below. In general, the conversions are lossless for integer input/output. They are implemented using "lifting" steps in the code. The reference to "YUV" below is an abuse of notation since the internal color space does not really map to the traditional CCIR/SMPTE YUV color space. However, the Y channel roughly corresponds to luminance or a grayscale representation of the original. Likewise, the U and V channels are low entropy chrominance or color difference channels. Alternately, these are referred to as Co and Cg channels.

A bias is conditionally added to the Y term to zero center its range.

In the code, a re-arrangement of the data is being performed as well. This is for optimization purposes.

When the scaling mode is used, the color values are shifted left prior to encoder color conversion. On the decoder side, the values are rounded down after color conversion. Further, for lossy compression, the color converted values on the decoders are clamped to fit the appropriate range as indicated in the bitstream.

4.3.2 RGB ↔ YUV 4:4:4

The encoder implements the following:

$$V = B - R$$

$$U = R - G + \left\lceil \frac{V}{2} \right\rceil$$

$$Y = G + \left\lfloor \frac{U}{2} \right\rfloor$$

The decoder implements the following:

$$G = Y - \left\lfloor \frac{U}{2} \right\rfloor$$

$$R = U + G - \left\lceil \frac{V}{2} \right\rceil$$

$$B = V + R$$

4.3.3 cmyk ↔ YUVK

The encoder implements the following:

$$V = c - y$$

$$U = c - m - \left\lfloor \frac{V}{2} \right\rfloor$$

$$Y = k - m - \left\lfloor \frac{U}{2} \right\rfloor$$

$$K = k - \left\lfloor \frac{Y}{2} \right\rfloor$$

The decoder implements the following:

$$k = K + \left\lfloor \frac{Y}{2} \right\rfloor$$

$$m = k - Y - \left\lfloor \frac{U}{2} \right\rfloor$$

$$c = U + m + \left\lfloor \frac{V}{2} \right\rfloor$$

$$y = c - V$$

4.4 Transform

4.4.1 Encoder Transform Overview

HD Photo uses a two level lapped transform, which is further broken up as follows on the encoder side:

- A pre filter operation is optionally applied to 4x4 areas evenly straddling blocks in two dimensions. Further, a pre filter is applied to boundary areas which are 2x4 or 4x2 in size. The four 2x2 corners on each color plane are left untouched.
- A block transform called the HD Photo Core Transform (PCT) is applied to all 4x4 blocks. This completes the first transform stage.
- DC coefficients of the 4x4 blocks are grouped together into a planar structure. 16 such coefficients exist for each macroblock color plane, and these are structured as 4x4 blocks in the DC plane.
- A second level pre filter operation is optionally applied to 4x4 areas evenly straddling blocks in two dimensions in the DC plane. Further, a pre filter is applied to boundary 2x4 and 4x2 areas, and the four 2x2 corner areas are left untouched.
- The PCT is applied to each 4x4 block corresponding to DC coefficients in a macroblock color plane.

These operations are repeated for all color planes. For the special cases of YUV 4:2:2 and YUV 4:2:0 chrominance channels, appropriately modified transforms are applied (for instance a 2x2 transform is used as the block transform of the chrominance channel DC planes of YUV 4:2:0).

This process is shown in Figure 3.

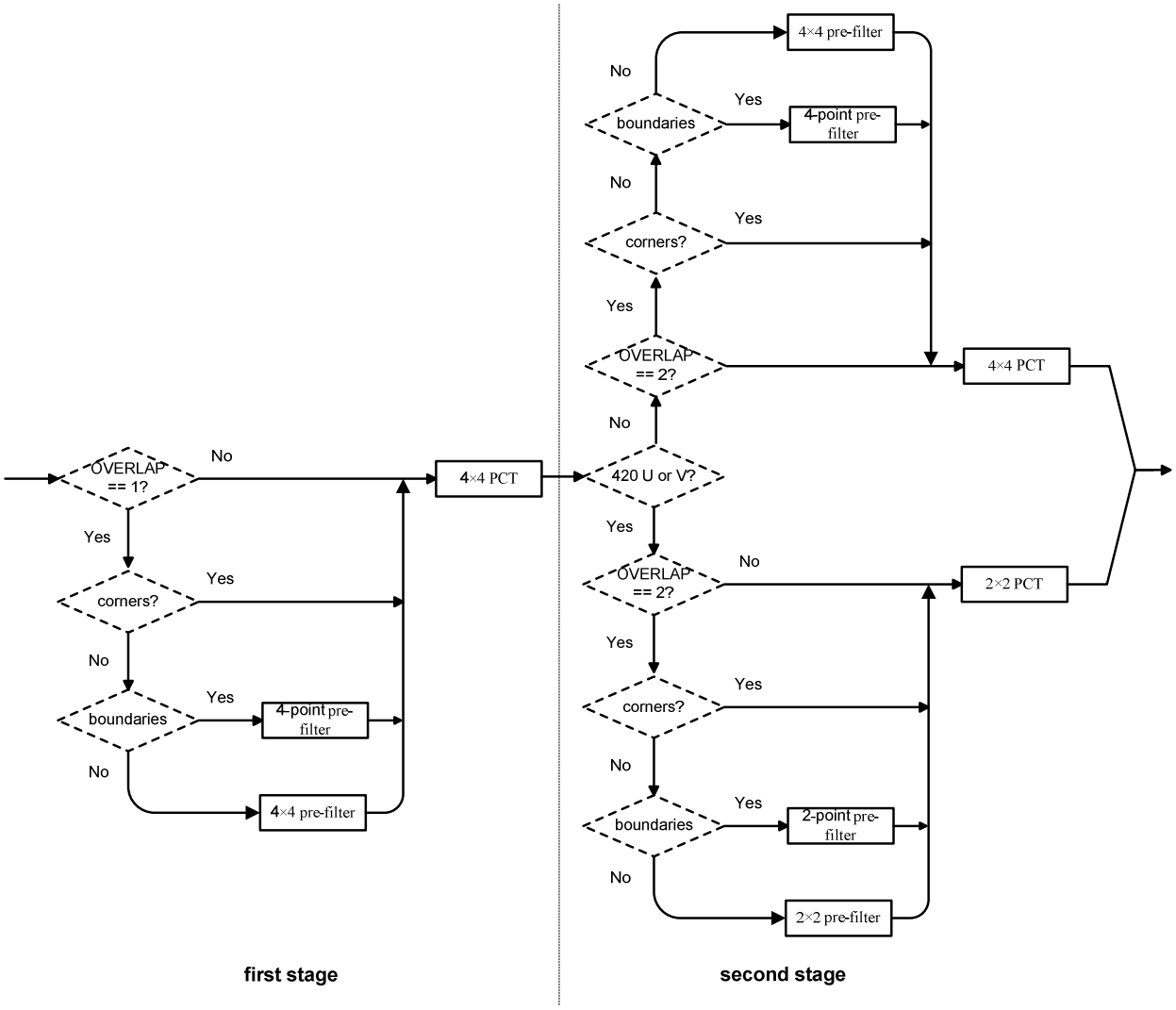


Figure 3: Encoder side transform block diagram

4.4.2 Decoder Transform Overview

On the decoder side, the steps are as follows:

- The inverse PCT is applied to each 4x4 block corresponding to decoded and dequantized DC coefficients arranged in a planar array known as the DC plane.
- A post filter operation is optionally applied to 4x4 areas evenly straddling blocks in the DC plane. Further, a post filter is applied to boundary 2x4 and 4x2 areas, and the four 2x2 corner areas are left untouched.
- The resulting array contains DC coefficients of the 4x4 blocks corresponding to the first level transform. The DC coefficients are (figuratively) copied into a larger array, and the remaining coefficients (lowpass and AC terms) decoded, dequantized and populated into the remaining positions.
- The inverse PCT is applied to each 4x4 block.
- A post filter operation is optionally applied to 4x4 areas evenly straddling blocks in the DC plane. Further, a post filter is applied to boundary 2x4 and 4x2 areas, and the four 2x2 corner areas are left untouched.

This process is shown in Figure 4.

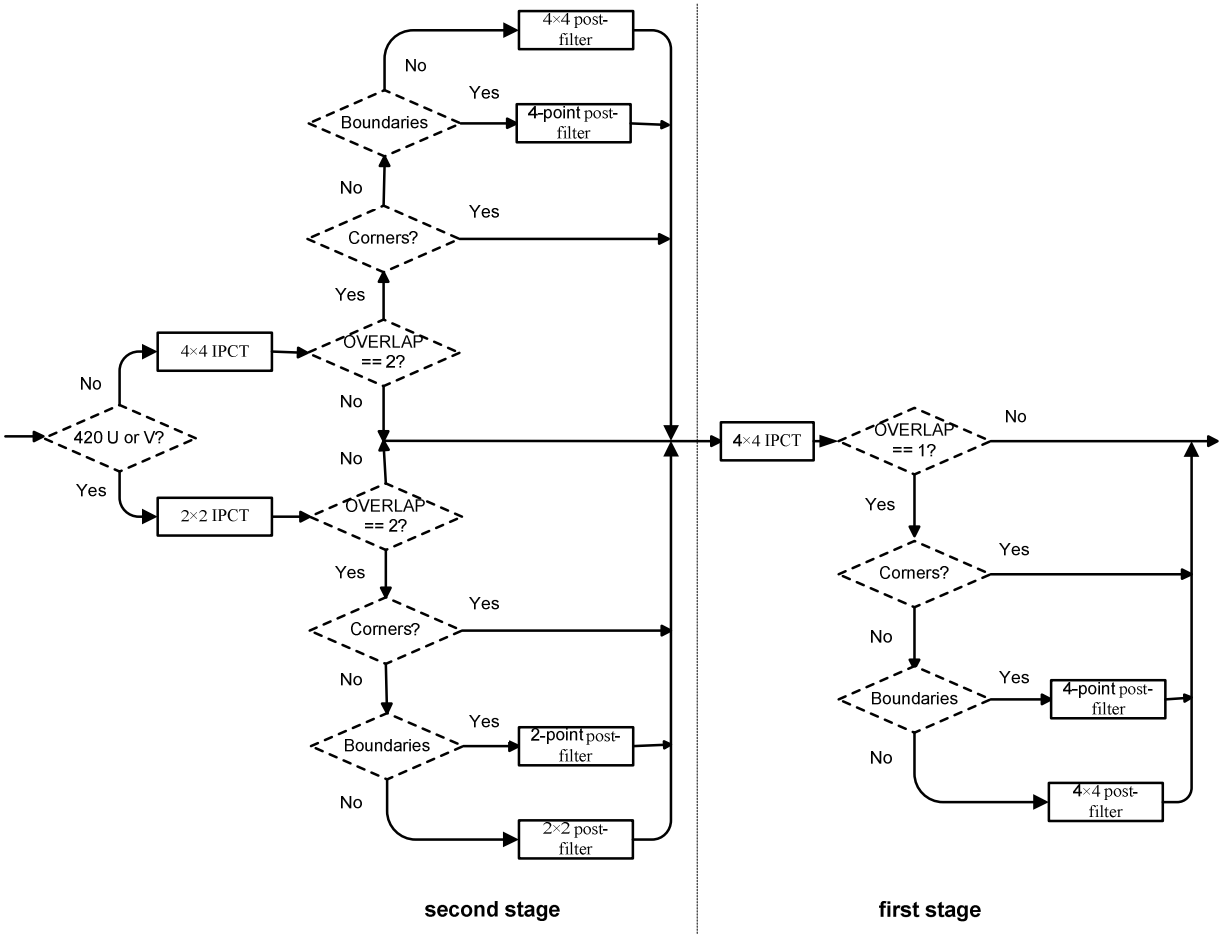


Figure 4: Decoder side transform block diagram

The application of pre filters and post filters is governed by the OVERLAP syntax element. OVERLAP may take on three values as described in section:

If OVERLAP = 0, no pre filtering and post filtering shall be performed.

If OVERLAP = 1, only the outer pre filtering (i.e. step 1 of the encoder) and outer post filtering (i.e. step 5 of the decoder) shall be performed.

If OVERLAP = 2, both inner and outer prefiltering (i.e. steps 1 & 3 of the encoder), and both inner and outer post filtering (i.e. steps 3 & 5 of the decoder) shall be performed.

Lossless coding is possible with all overlap modes, although OVERLAP = 0 is usually best or sufficient for lossless coding. OVERLAP = 1 produces the shortest bitstream for a large class of images and quantization levels. OVERLAP = 2 is recommended for high quantization levels but with slightly higher complexity. OVERLAP = 0 is recommended for lowest complexity encoding / decoding; however this mode implements a hierarchical block transform and potentially introduces blocking at low bitrates.

4.4.3 HD Photo Core Transform

The HD Photo Core Transform (PCT) is inspired by the 4×4 DCT, yet it is fundamentally different. The first key difference is that the DCT is linear whereas the PCT is nonlinear. The second key difference is that due to the fact that it is defined on real numbers, the DCT is not a lossless operation in the integer to integer space. The PCT is defined on integers and is lossless in this space. The third key difference is that the 2D DCT is a separable operation. The PCT is non-separable by design.

The entire transform process can be written as the cascade of three elementary 2×2 transform operations, which are the 2×2 Hadamard transform T_h, and the following:

1D rotate: T_odd

2D rotate: T_odd_odd

These transforms are implemented as a non-separable operations and are described first, followed by the description of the entire PCT.

4.4.3.1 2D 2x2 Hadamard transform

HD Photo implements the 2D 2x2 Hadamard transform T_h as shown in Table 1. R is a rounding factor which may take on the value 0 or 1 only. T_h is involutory (i.e. two applications of T_h on a data vector [a b c d] succeed in recovering the original values of [a b c d], provided R is unchanged between the applications).

```
T_h (int &a, int &b, int &c, int &d, int R) {
    a += d;
    b –= c;
    int t1 = (a – b + R) >> 1;
    int t2 = c;
    c = t1 – d;
    d = t1 – t2;
    a –= d;
    b += c;
}
```

Table 1. Code for function T_h

4.4.3.2 1D rotate T_odd

T_odd is defined by the C++ code in Table 2.

```
T_odd (int &a, int &b, int &c, int &d) {
    b –= c;
    a += d;
    c += (b + 1) >> 1;
    d = ((a + 1) >> 1) – d;

    b –= (3*a + 4) >> 3;
    a += (3*b + 4) >> 3;
    d –= (3*c + 4) >> 3;
    c += (3*d + 4) >> 3;

    d += b >> 1;
    c –= (a + 1) >> 1;
    b –= d;
    a += c;
}
```

Table 2. Code for function T_odd

The lossless inverse of T_odd is derived by reversing the steps of T_odd with attention to sign, and is defined in the code shown in Table 3.

```
InvT_odd (int &a, int &b, int &c, int &d) {
    b += d;
    a –= c;
    d –= b >> 1;
    c += (a + 1) >> 1;

    a –= (3*b + 4) >> 3;
    b += (3*a + 4) >> 3;
    c –= (3*d + 4) >> 3;
    d += (3*c + 4) >> 3;

    c –= (b + 1) >> 1;
    d = ((a + 1) >> 1) – d;
    b += c;
    a –= d;
}
```

Table 3. Code for function InvT_odd

4.4.3.3 2D rotate T_odd_odd

T_odd_odd and its inverse InvT_odd_odd are defined by the C++ code in Table 4 and Table 5 respectively.

```
T_odd_odd (int &a, int &b, int &c, int &d) {
    int t1, t2;
    b = –b;
    c = –c;

    d += a;
    c –= b;
    a –= (t1 = d >> 1);
    b += (t2 = c >> 1);

    a += (b * 3 + 4) >> 3;
    b –= (a * 3 + 3) >> 2;
    a += (b * 3 + 3) >> 3;

    b –= t2;
    a += t1;
    c += b;
    d –= a;
}
```

Table 4. Code for function T_odd_odd

```
InvT_odd_odd (int &a, int &b, int &c, int &d) {
    int t1, t2;
    d += a;
    c -= b;
    a -= (t1 = d >> 1);
    b += (t2 = c >> 1);

    a -= (b * 3 + 3) >> 3;
    b += (a * 3 + 3) >> 2;
    a -= (b * 3 + 4) >> 3;

    b -= t2;
    a += t1;
    c += b;
    d -= a;

    b = -b
    c = -c
}
```

Table 5. Code for function InvT_odd_odd

4.4.3.4 PCT Operations

The correspondence between 2x2 data and the previously listed C++ code is shown in Figure 5. Color coding using four gray levels to indicate the four data points is introduced here, to facilitate the HD Photo transform description in the next section.

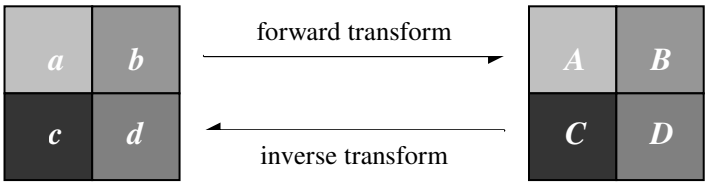


Figure 5: Transform and inverse transform operation

The 2D 4×4 point PCT is built using T_h, T_odd and T_odd_odd. The PCT is composed of two stages, which are shown in Figure 6 and Figure 7 respectively. Each stage consists of four 2x2 transforms which may be done in any arbitrary sequence, or concurrently, within the stage.

For the 2D 4x4 point IPCT, the stages are reversed in order, and the steps within each stage of transformation use the inverse of the steps in the forward transform process. Since T_h is its own inverse, the second stage of the inverse HD Photo transform is merely the first stage of the forward HD Photo transform, as shown in Figure 6. The first stage of the inverse HD Photo transform is shown in Figure 8.

The resulting PCT transform coefficients are ordered as shown in Figure 9, which is the same ordering in which transform coefficients are input to the inverse of the PCT.

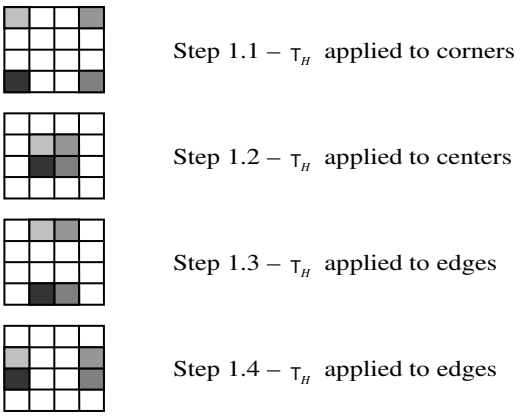


Figure 6: First stage of PCT (second stage of IPCT)

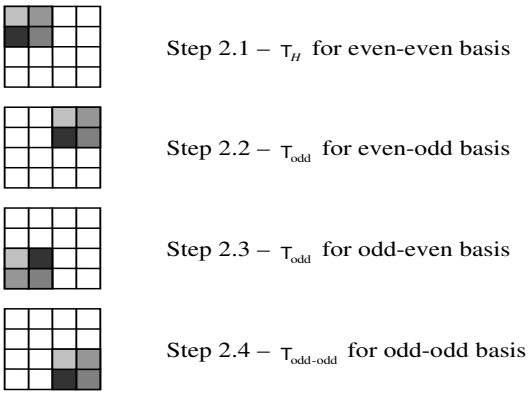


Figure 7: Second stage of PCT

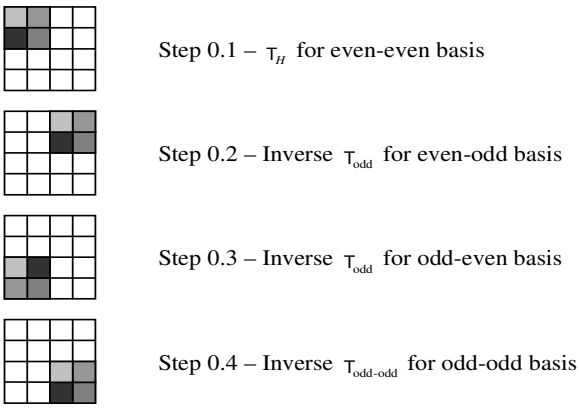


Figure 8: First stage of IPCT

0	2	1	3
8	10	9	11
4	6	5	7
12	14	13	15

Figure 9: PCT/IPCT transform coefficient ordering

4.4.3.5 HD Photo transform for chrominance DC plane in YUV 4:2:0

The chrominance DC plane in a YUV 4:2:0 image is divided into 2×2 blocks corresponding to the DC values of 4×4 blocks within a macroblock. The two point DCT is implemented by the 2×2 Hadamard transform T_h.

4.4.3.6 HD Photo transform for chrominance DC plane in YUV 4:2:2

This is similar to the YUV 4:2:0 transform, with an additional two point Hadamard transform step. Refer to the code for details.

4.5 Pre/post filtering

4.5.1 Overview

Four operators determine the pre and post filters used in the HD Photo transform. These are:

1. 4x4 pre/post filter
2. 4 point pre/post filter
3. 2x2 pre/post filter
4. 2 point pre/post filter

The pre-filter makes use of the operators T_h, T_odd_odd, fwdScale and fwdRotate. fwdRotate and fwdScale are defined in Table 6 and Table 7 respectively.

```
fwdRotate (Pixell &a, Pixell &b) {
    a += ((b) * 3 + 8) >> 4;
    b -= ((a) * 3 + 4) >> 3;
    a += ((b) * 3 + 8) >> 4;
}
```

Table 6. Code for function fwdRotate

```
fwdScale (Pixell &a, Pixell &b) {
    b += a;
    a -= (b + 1) >> 1;
    b -= (*a * 3 + 4) >> 3;
    a -= (*b * 3 + 8) >> 4;
    b -= (*a * 3 + 0) >> 3;
    a += (b + 1) >> 1;
    b -= a;
}
```

Table 7. Pseudocode for function fwdScale

Likewise, the post-filter uses T_h, InvT_odd_odd, invScale and invRotate. invRotate and invScale are defined in Table 8 and Table 9 respectively.

```
invRotate (Pixell &a, Pixell &b) {
    a -= ((b) * 3 + 8) >> 4;
    b += ((a) * 3 + 4) >> 3;
    a -= ((b) * 3 + 8) >> 4;
}
```

Table 8. Code for function invRotate

```
invScale (Pixell &a, Pixell &b) {
    b += a;
    a -= (b + 1) >> 1;
    b += (*a * 3 + 0) >> 3;
    a += (*b * 3 + 8) >> 4;
    b += (*a * 3 + 4) >> 3;
    a += (b + 1) >> 1;
    b -= a;
}
```

Table 9. Code for function invScale

4.5.2 4×4 pre-filter

Primarily, the 4x4 pre-filter is applied to all block junctions (areas straddling 4 blocks evenly) in all color planes when OVERLAP is 1 or 2. Also, the 4x4 filter is applied to all block junctions in the DC plane for all planes when OVERLAP is 2, and for only the luminance plane when OVERLAP is 2 and color format is either YUV 4:2:0 or YUV 4:2:2.

If the input data block is $\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$, the 4×4 pre-filter,

4x4PreFilter (a, b, c, d, e, f, g, h, i, j, l, l, m, n, o, p), is defined in Table 10.

```
4x4PreFilter (Pixell &a ... Pixell &p) {
    T_h(a, d, m, p, 0);
    T_h (b, c, n, o, 0);
    T_h (e, h, i, l, 0);
    T_h (f, g, j, k, 0);

    fwdScale (a, p);
    fwdScale (b, l);
    fwdScale (e, o);
    fwdScale (f, k);

    fwdRotate (n, m);
    fwdRotate (j, i);
    fwdRotate (h, d);
    fwdRotate (g, c);
    T_odd_odd (k, l, o, p);

    T_h (a, m, d, p, 0);
    T_h (b, n, c, o, 0);
    T_h (e, i, h, l, 0);
    T_h (f, j, g, k, 0);
}
```

Table 10. Code for function 4x4PreFilter

4.5.3 4×4 post-filter

During decoding, the post-filter is applied to the same areas as the pre-filter was during encoding. The post-filter operator, 4x4PostFilter, is defined in Table 11.

```
4x4PostFilter (Pixell &a ... Pixell &p) {
    T_h(a, d, m, p, 0);
    T_h (b, c, n, o, 0);
    T_h (e, h, i, l, 0);
    T_h (f, g, j, k, 0);

    invRotate (n, m);
    invRotate (j, i);
    invRotate (h, d);
    invRotate (g, c);
    InvT_odd_odd (k, l, o, p);

    invScale (a, p);
    invScale (b, l);
    invScale (e, o);
    invScale (f, k);

    T_h (a, m, d, p, 0);
    T_h (b, n, c, o, 0);
    T_h (e, i, h, l, 0);
    T_h (f, j, g, k, 0);
}
```

Table 11. Code for function 4x4PostFilter

4.5.4 4-point pre-filter

Linear 4-point filters are applied to edge straddling 2x4 and 4x2 areas on the boundary of the image. If the input data is [*a b c d*] the 4-point pre-filter, 4PreFilter(a, b, c, d), is defined in Table 12.

```
4PreFilter (PixelI &a ... PixelI &d) {
    a –= (d * 3 + 16) >> 5;
    b –= (c * 3 + 16) >> 5;
    d –= (a * 3 + 8) >> 4;
    c –= (b * 3 + 8) >> 4;
    a += d – ((d * 3 + 16) >> 5);
    b += c – ((c * 3 + 16) >> 5);
    d –= (a + 1) >> 1;
    c –= (b + 1) >> 1;

    fwdRotate(c, d);

    d += (a + 1) >> 1;
    c += (b + 1) >> 1;
    a –= d;
    b –= c;
}
```

Table 12. Code for function 4PreFilter

4.5.5 4-point post-filter

Accordingly, if the input coefficients are [*a b c d*], the 4-point post-filter, 4PostFilter(a, b, c, d), is defined in Table 13.

```
4PostFilter (PixelI &a ... PixelI &d) {
    a += d;
    b += c;
    d –= (a + 1) >> 1;
    c –= (b + 1) >> 1;

    invRotate(c, d);

    d += (a + 1) >> 1;
    c += (b + 1) >> 1;
    a –= d – ((d * 3 + 16) >> 5);
    b –= c – ((c * 3 + 16) >> 5);
    d += (a * 3 + 8) >> 4;
    c += (b * 3 + 8) >> 4;
    a += (d * 3 + 16) >> 5;
    b += (c * 3 + 16) >> 5;
}
```

Table 13. Code for function 4PostFilter

4.5.6 2×2 pre-filter

The 2x2 pre-filter is applied to areas straddling blocks in the DC plane for the chrominance channels of YUV 4:2:0 and YUV 4:2:2 data. If the input data is $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the 2×2 pre-filter 2x2PreFilter (a, b, c, d), is defined in Table 14.

```
2x2PreFilter (Pixell &a ... Pixell &d) {
    a += d;
    b += c;
    d -= (a + 1) >> 1;
    c -= (b + 1) >> 1;

    b -= ((a + 2) >> 2);
    a -= ((b + 1) >> 1);
    b -= ((a + 2) >> 2);

    d += (a + 1) >> 1;
    c += (b + 1) >> 1;
    a -= d;
    b -= c;
}
```

Table 14. Code for function 2x2PreFilter

4.5.7 2x2 post-filter

On the decoder side, the 2×2 post-filter is applied to areas where the 2×2 pre-filter was applied during encoding. Accordingly, if the input coefficient is $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the 2×2 post-filter 2x2PostFilter (a, b, c, d), is defined in Table 15.

```
2x2PostFilter (Pixell &a ... Pixell &d) {
    a += d;
    b += c;
    d -= (a + 1) >> 1;
    c -= (b + 1) >> 1;

    b += ((a + 2) >> 2);
    a += ((b + 1) >> 1);
    b += ((a + 2) >> 2);

    d += (a + 1) >> 1;
    c += (b + 1) >> 1;
    a -= d;
    b -= c;
}
```

Table 15. Code for function 2x2PostFilter

4.5.8 2-point pre-filter

As for the 4-point filters, the 2-point pre-filter is applied to boundary 2×1 and 1×2 pixels that straddle blocks. The 2-point pre-filter, 2PreFilter (a, b) is defined in Table 16.

```
2PreFilter (Pixell &a, Pixell &b) {
    b −= (a + 4) >> 3;
    a −= (b + 2) >> 2;
    b −= (a + 4) >> 3;
}
```

Table 16. Code for function 2PreFilter

4.5.9 2-point post-filter

Likewise, the 2-point post filter is applied to boundary 2×1 and 1×2 pixels that straddle blocks. The 2-point post-filter, 2PostFilter (a, b) is defined in Table 17.

```
2PostFilter (Pixell &a, Pixell &b) {
    b += (a + 4) >> 3;
    a += (b + 2) >> 2;
    b += (a + 4) >> 3;
}
```

Table 17. Code for function 2PostFilter

4.6 Quantization

Quantization is a process whereby the transform coefficients are essentially divided and rounded to an integer value, called the *quantized value*. Dequantization is the process where essentially coefficients are reconstructed from their quantized values by multiplying them. The divisor in quantization and multiplier in dequantization are usually identical and referred to as the *quantization parameter* or *QP*.

In the lossless coding mode of HD Photo, QP = 1.

For lossy coding, QP > 1. In HD Photo, QP is chosen to be an integer chosen from a harmonic scale. This is determined by another integer *QPIndex* as per the rule:

QP = QPIndex	for QPIndex ≤ 16
QP = ((QPIndex % 16) + 16) << ((QPIndex >> 4) – 1)	otherwise

The symbol "%" denotes remainder of integer division or the mod function. When QPIndex = 255, QP is given by ((255 % 16) + 16) << (15 – 1) = 507904.

On the decoder side, each entropy decoded transform coefficient is dequantized by multiplying the coefficient with QP. On the encoder side, the specific rounding factor in the process of quantization is implementation specific, and is not covered in this document. In the lossless mode, coefficients are passed through (i.e. multiplied or divided by 1) on both the encoder and decoder.

QP is allowed to differ across AC, lowpass and DC bands. The DC QP within a tile is fixed. The DC QP across tiles may vary. The AC and lowpass QPs within a tile may take on either the same value, or one of a multiple value set. This may changed at every macroblock and is signaled in the bitstream.

4.7 DCAC prediction

When tiling is used, each tile is deemed to be a separate image for the purpose of DCAC prediction, to ensure independent decoding of tiles.

4.7.1 DC prediction

Four modes are allowed for the prediction of the DC coefficient of a macroblock. These modes are:

1. Predict from left (*predictor* = *DC* [*left_MB*])
2. Predict from top (*predictor* = *DC* [*top_MB*])
3. Predict from left and top (*predictor* = (*DC* [*left_MB*] + *DC* [*top_MB*]) >> 1)
4. Null predict (*predictor* = 0)

Which mode to pick is determined from the position of the macroblock, as well as the DC values to the left, top and top-left of the macroblock (shown in Figure 10). Further, if the image has color channels, the corresponding values of the chroma channels are also used.

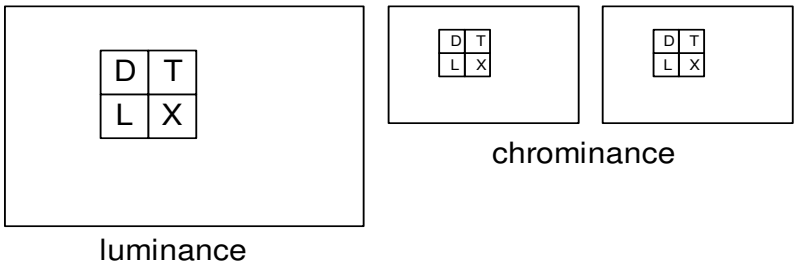


Figure 10: DC values to the left (L), top (T) and top-left (D, for diagonal) of the current DC value (X). Luminance channels are used when meaningful (for YUV – 420/422/444 – and CMYK cases)

The DC prediction mode (dc_mode) is determined by the following pseudocode (where [mx, my] is the macroblock index of the current macroblock in the X and Y directions in an image (or an image tile if tiling is enabled), starting from [0,0]):

```
If (mx == 0 and my == 0)
        dc_mode = Null predict
Else if (mx == 0)
        dc_mode = Predict from top
Else if (my == 0)
        dc_mode = Predict from left
Else {
        diff_h = abs (D – L)   // luminance
        diff_v = abs (D – T)   // luminance
        If (chrominance channels are available) {
                diff_h = diff_h * scale + sum_over_chrominance_channels { abs (D – L) }
                diff_v = diff_v * scale + sum_over_chrominance_channels { abs (D – T) }
        }

        If (diff_h * orient_weight < diff_v) {
                dc_mode = Predict from top
        }
        Else if (diff_v * orient_weight < diff_h) {
                dc_mode = Predict from left
        }
        Else {
```

```
                dc_mode = Predict from left and top
        }
}

Where
scale = 8 for YUV 420, 4 for YUV 422, 2 otherwise
orient_weight = 4
```

For interior macroblocks, directional DC prediction (i.e. prediction from top or left) happens only when one of the directions is dominant as compared to the other. This is captured in the last set of if..else statements, where directional difference metrics are checked to see if they are skewed by a factor of 4 or higher.

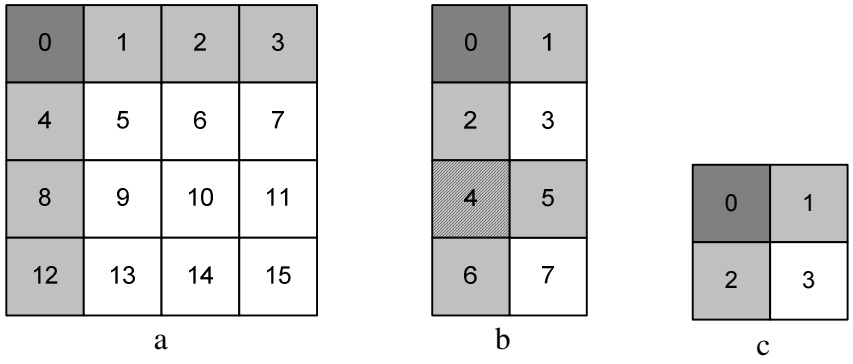


Figure 11: (a) DC and DCAC coefficients in a 4x4, (b) 422 chroma lowpass, and (c) 420 chroma lowpass block. DC value of block shown in dary gray is at position 0, DCAC values are shown in light gray.

4.7.2 DCAC prediction of lowpass band

Three modes are allowed for the prediction of the DCAC coefficient of the inner transform of a macroblock. These modes are:

1. Predict from left (*predictor* = *DCAC* [*left_MB*])
2. Predict from top (*predictor* = *DCAC* [*top_MB*])
3. Null predict (*predictor* = 0)

The lowpass DCAC prediction mode (lowpass_DCAC_mode) is determined by the DC prediction mode, together with the quantizer indices of the current block and block from which DC is predicted, as shown in the pseudocode below:

```
lowpass_DCAC_mode = Null predict
If ((DC_mode == Predict from top) && (quantizer_index (current MB) == quantizer_index (top MB)))
{
        lowpass_DCAC_mode = Predict from top
}
Else if ((DC_mode == Predict from left) && (quantizer_index (current MB) == quantizer_index (left
MB))) {
        lowpass_DCAC_mode = Predict from left
}
```

This rule ensures that prediction of inner transform DCAC terms does not take place across macroblocks with different quantizers. Further DCAC is predicted only if one direction is dominant – this property is derived from the DC prediction mode.

A special case is for 422, position 5 is predicted from position 1 in Figure 11 if *DC_mode = Predict from top* regardless what *lowpass_DCAC_mode* is.

4.7.3 DCAC prediction of highpass band

Information in the lowpass transform coefficients is used to determine the *orientation* of prediction, based on a simple metric associated with each macroblock. Three modes are allowed for the prediction of the DCAC coefficients of the outer transform. The same mode is used for all blocks within a macroblock for which in-macroblock prediction is possible. For blocks that have no valid reference within the macroblock, null prediction is used. The three modes are:

1. Predict from left (predictor = DCAC [left_block within macroblock])
2. Predict from top (predictor = DCAC [top_block within macroblock])
3. Null predict (*predictor* = 0)

Prediction from left is shown in Figure 12. Prediction from top is similar, with the pattern of arrows transposed to point downwards. DCAC terms of the first column of blocks is not predicted.

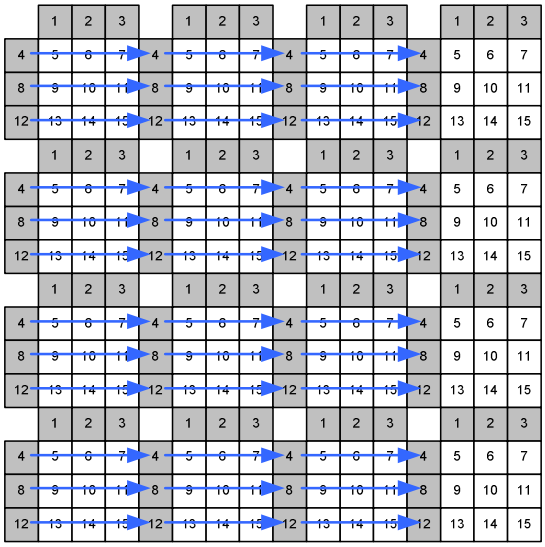


Figure 12: Highpass DCAC prediction from left shown. Prediction from top is similar.

The highpass DCAC prediction mode (highpass_DCAC_mode) is determined by the pseudocode shown below.

```
diff_h = abs(lowpass(4)) + abs(lowpass(8)) + abs(lowpass(12))
diff_v = abs(lowpass(1)) + abs(lowpass(2)) + abs(lowpass(3))

If (chrominance is present) {
        If (YUV 420) {
                diff_h = diff_h + abs(lowpass_U(2)) + abs(lowpass_V(2))
                diff_v = diff_v + abs(lowpass_U(1)) + abs(lowpass_V(1))
        }
        Else if (YUV 422) {
                diff_h = diff_ h + abs(lowpass_U(2)) + abs(lowpass_V(2)) + abs(lowpass_U(6)) +
abs(lowpass_V(6))
                diff_v = diff_v + abs(lowpass_U(1)) + abs(lowpass_V(1)) + abs(lowpass_U(5)) +
abs(lowpass_V(5))
        }
        Else {
                diff_h = diff_h + abs(lowpass_U(4)) + abs(lowpass_V(4))
                diff_v = diff_v + abs(lowpass_U(1)) + abs(lowpass_V(1))
        }
}

If (diff_h * orient_weight < diff_v) {
        highpass_DCAC_mode = Predict from top
}
Else if (diff_v * orient_weight < diff_h) {
        highpass_DCAC_mode = Predict from left
}
Else {
        highpass_DCAC_mode = Null predict
}
```

orient_weight = 4 as defined earlier. *lowpass* is the quantized inner transform numbered in left-to-right, top-to-bottom order. *lowpass* coefficients normally go from 0…15. For the chrominance channels (represented by *lowpass_U* and *lowpass_V*), they run from 0…3 for YUV 420, 0…7 for YUV 422 and 0…15 for YUV 444/CMYK.

In the implementation of a HD Photo codec (encoder or decoder), the only information that needs to be available for future use is 1 DC + 6 DCAC = 7 coefficients per macroblock channel (fewer for YUV 420 / YUV 422 chrominance). Therefore, at most for YUV 444, 21 coefficients need to be cached per macroblock. Further, the coefficients used for prediction from left can be discarded after the next macroblock is coded/decoded. For YUV 444, therefore, it is necessary to only cache 12 coefficients per macroblock for use in the next row of macroblocks.

4.8 Coefficient scanning

Coefficient scanning is the process of reordering the array of transform coefficients into a linear array, and vice versa. This is also commonly referred to as "zig-zag scan" although this nomenclature is inaccurate for HD Photo and is therefore avoided.

1. Within each image, tiles are scanned in raster order.
2. Within each tile, macroblocks are scanned in raster order.
3. Within each macroblock, DC coefficients are scanned across all color planes in sequence.
4. Within each macroblock, lowpass coefficients are scanned in the following order:
    a. Each color plane is scanned in sequence.
    b. Within each color plane, lowpass transform coefficients are scanned in an adaptive order described in 4.8.1. For the special cases of YUV 4:2:0 and YUV 4:2:2, the luminance color plane is followed by a composite of the chrominance color plane coefficients.

5. Within each macroblock, highpass coefficients are scanned in the following order:
   a. Each color plane is scanned in sequence.
   b. Within each color plane, each *sub-macroblock* is scanned in raster order. A sub-macroblock is defined as an 8x8 region or a 2x2 aggregation of blocks.
   c. Within each sub-macroblock, blocks are scanned in raster order.
   d. Within each block, highpass transform coefficients are scanned in an adaptive order described in 4.8.1.

4.8.1 Adaptive scan order

Within a block of data, coefficients are scanned in a deterministic (but possibly varying) manner as shown in the example in Figure 13. This is similar to the traditional "zig-zag scan" applied to a 4×4 block. In this example, the first coefficient in the linear array is the top left entry (DC) of the transform coefficient matrtix. The second coefficient in the linear array is the transform coefficient marked "1" and so on. Thus, the representation of Figure 13 uniquely determines a scan pattern.

Scan patterns in HD Photo are not required to be continuous as in the above example. Also, the scan patterns in HD Photo do not include the DC term (i.e. the top left cell in the figure). Further, scan patterns in HD Photo are allowed to change over the course of the tile. The adaptation rules are defined below.

Three scan patterns are used in HD Photo. These are referred to as the "lowpass", "highpass horizontal" and "highpass vertical" scan patterns. The lowpass scan pattern is used to encode/decode the lowpass transform coefficients in a macroblock. The highpass horizontal and vertical scan patterns are used to encode/decode the highpass transform coefficients in a macroblock. Macroblocks that are signaled as dominant horizontal use the highpass horizontal scan pattern, and likewise for vertical. Macroblocks showing no dominance of orientation also use the highpass horizontal scan pattern. The scan pattern is derived from the macroblock DC prediction pattern as defined in 4.7.1.

The three scan patterns are initialized to a specific ordering at the start (top left macroblock) of each tile. The lowpass scan pattern and the highpass horizontal scan pattern are initialized to the pattern shown in Figure 14(a), and the highpass vertical scan pattern is initialized to the pattern shown in Figure 14(b). All color channels within a macroblock use the same corresponding scan pattern.

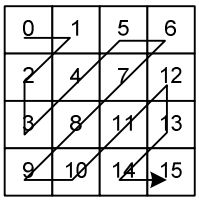


Figure 13: Example coefficient scan order, using a continuous zigzag scan pattern.

X	0	3	7
1	2	5	10
4	6	9	12
8	11	13	14

(a)

X	3	7	9
0	2	6	11
1	5	12	14
4	8	13	10

(b)

Figure 14: Initial DC scan patterns – (a) lowpass and highpass horizontal scan pattern, and (b) highpass vertical scan pattern.

The scan pattern can be defined as a 1D array. The array elements refer to the transform coefficient index (from 0 through 15, in raster order for the 4×4 transform block), in order of their scan. Thus, the scan pattern shown in Figure 14(a) can be written as:

Order[] = { 1,4,5, 2,8,6,9, 3,12,10,7, 13,11,14,15 }

For each of the three scan patterns, another array (labeled "Totals") is created. This is initialized with descending values as shown:

Totals[] = { 28,26,24, 22,20,18,16, 14,12,10, 6,4,2,0 }

The array Totals tallies the incidence of occurrence of the particular coefficient. During the process of encoding or decoding, if the n^{th} element of the scan is nonzero (i.e. transform coefficient with index Order[n] is nonzero), then the n^{th} element of Totals is incremented by 1 (i.e. Totals[n] := Totals[n] + 1). If after incrementing, it is found that Totals[n] > Totals[n – 1] (for n ≥ 1), an *exchange* operation is applied to the scan order. During the exchange step, the scan orders and corresponding Totals of n and n – 1 are switched as follows:

Temp := Order[n]

Order[n] := Order[n – 1]

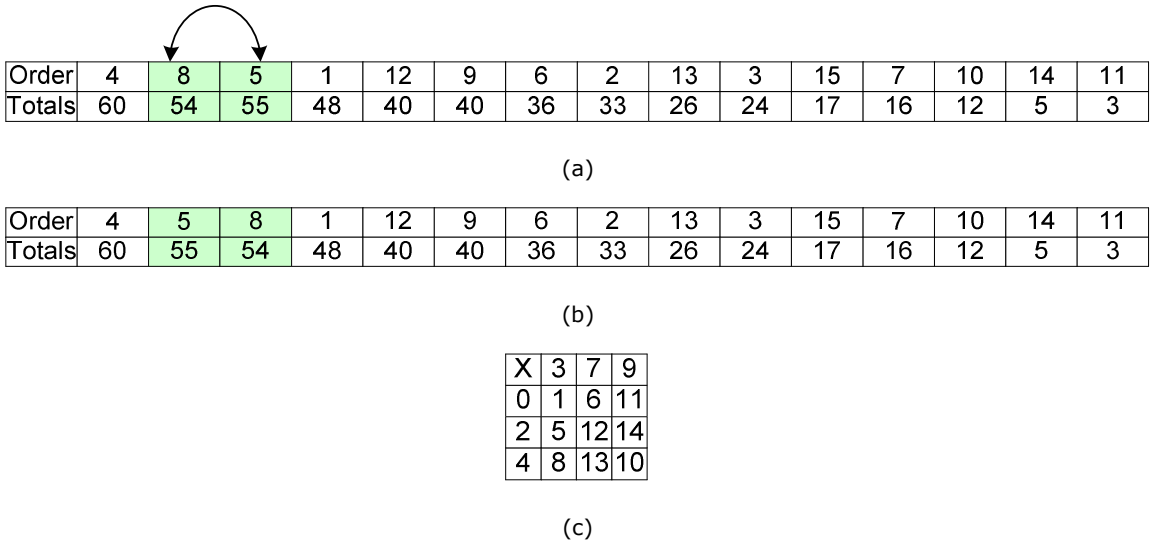
Order[n – 1] := Temp

Temp := Totals[n]

Totals[n] := Totals[n – 1]

Totals[n – 1] := Temp

As a result of the exchange operation, the coefficient indexed by Order[n] (prior to exchange) is now scanned prior to the coefficient indexed by Order[n – 1] (prior to exchange). Figure 15(a) shows a situation where Totals[n] > Totals[n – 1], indicated by shaded elements. The arrows show the elements that need to be exchanged. Figure 15(b) shows the Order and Totals arrays subsequent to the exchange, and Figure 15(c) shows the corresponding scan order indices on the 4x4 block. Exchange, when triggered, occurs subsequent to encoding or decoding. Therefore, the adaptation is causal.



Order	4	8	5	1	12	9	6	2	13	3	15	7	10	14	11
Totals	60	54	55	48	40	40	36	33	26	24	17	16	12	5	3

(a)

Order	4	5	8	1	12	9	6	2	13	3	15	7	10	14	11
Totals	60	55	54	48	40	40	36	33	26	24	17	16	12	5	3

(b)

X	3	7	9
0	1	6	11
2	5	12	14
4	8	13	10

(c)

Figure 15: Inverse scan order (a)just prior to an exchange operation occurring between shaded coefficients, (b) subsequent to exchange operation, and (c) scan pattern corresponding to (b)

HD Photo resets the Totals array to its initial value every 8 macroblocks, beginning with the leftmost macroblock within the tile. The maximum value of any element of the Totals array is restricted to 8 (macroblocks) x 16 (coefficients of a certain frequency in a macroblock) x 16 (colorplanes in a macroblock) = 2048, plus the maximum initialization value of 28. 13 bits are sufficient for the Totals array in this case (9 bits are sufficient for the usual case of 3 channel images). The Order array is set to its initial value only at the start (top left) of every tile.

4.8.2 Scan order for YUV 4:2:2 and YUV 4:2:0 chroma highpass coefficients

Scan orders are defined for YUV 4:2:2 and YUV 4:2:0 chroma highpass coefficients. Refer to the code for details.

4.9 Adaptive Coefficient Normalization

Wide dynamic range input data leads to even wider dynamic range transform coefficients during the process of encoding an image. With small or unity quantization factors, the range of quantized transform coefficients is also large. *Adaptive coefficient normalization* is a step in the HD Photo encoder which processes the transform coefficients so as to render it suitable for encoding using efficient joint symbol entropy coding, and likewise in the HD Photo decoder.

Adaptive coefficient normalization proceeds by tracking a statistical measure of variance of transform coefficients. Based on this measure, the current transform coefficient is regrouped into dyadic bins. Instead of encoding the transform coefficient, its bin id is sent using an efficient entropy code. Subsequently, an in-bin address locating the transform coefficient within its respective bin is sent. This index is sent with a fixed length code. Sign information is also sent when necessary. On the decoder side, the same statistical measure is accumulated. Based on this measure, the binning rule is determined which allows for the both the bin id and in-bin address to be decoded. These quantities uniquely determine the quantized transform coefficient.

Further, adaptive coefficient normalization allows 'layering'. This is the bitstream feature that allows a lossy reconstruction even when the fixed length in-bin addresses are missing. In order for layering to work correctly, it is required that the statistical measure of variance of transform coefficients is determined entirely by the bin ids.

Adaptive coefficient normalization is used for all frequency bands, DC, lowpass and highpass. Layering is enabled only for the highpass band since it is most meaningful only for this band. The fixed length in-bin indices of the highpass band are referred to as FlexBits. The Adaptive Coefficient Normalization process is shown in Figure 16.

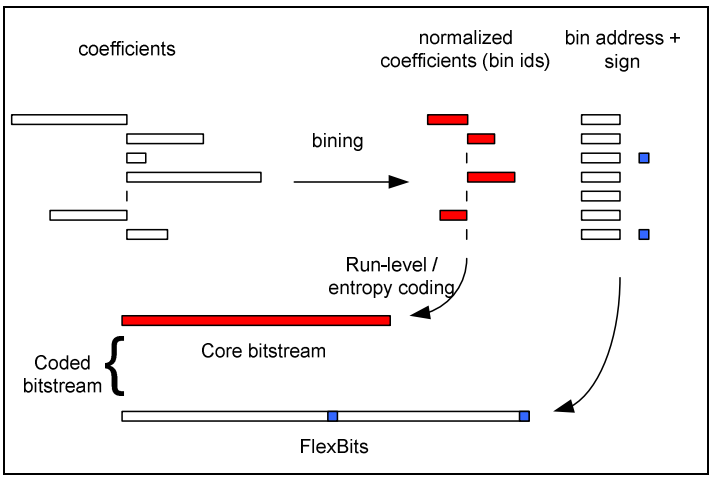


Figure 16: Representation of the grouping process and layering of output bitstream into core and FlexBits.

Adaptation of the normalization factor or bin size is performed by the function UpdateModelMB.

4.9.1.1 UpdateModelMB

The function UpdateModelMB is defined in Table 18.

```
UpdateModelMB (Lm[], Model) {
    // Cf is the color format of the image
    // Band is the frequency band (DC = 0, LP = 1, HP = 2)
    int aWeight0[3] = { 240/*DC*/, 12/*LP*/, 1 }
    int aWeight1[3][MAX_CHANNELS] = {
       { 0,240,120,80, 60,48,40,34, 30,27,24,22, 20,18,17,16 },
       { 0,12,6,4,     3,2,2,2,     2,1,1,1,     1,1,1,1 },
       { 0,16,8,5,     4,3,3,2,     2,2,2,1,     1,1,1,1 }
    }
    int aWeight2[6] = { 120,37,2,/*420*/ 120,18,1/*422*/ }
    Lm[0] *= aWeight0[Band]
    if (Cf == YUV_420) {
       Lm[1] *= aWeight2[Band]
    }
    else if (Cf == YUV_422) {
       Lm[1] *= aWeight2[3 + Band]
    }
    else {
       Lm[1] *= aWeight1[Band][NumChannels - 1]
       if (Model.m_band == HP)
          Lm[1] >>= 4
    }
    for (int j = 0; j < 2; j++) {
       int iMS = Model.State[j]
       int iDelta = (Lm[j] - MODELWEIGHT) >> 2
```

```
        if (iDelta <= -8) {
            iDelta += 4
            if (iDelta < -16)
                iDelta = -16
            iMS += iDelta
            if (iMS < -8) {
                if (Model.Bits[j] == 0)
                    iMS = -8
                else {
                    iMS = 0
                    Model.Bits[j]--
                }
            }
        }
        else if (iDelta >= 8) {
            iDelta -= 4
            if (iDelta > 15)
                iDelta = 15
            iMS += iDelta
            if (iMS > 8) {
                if (Model.Bits[j] >= 15) {
                    Model.Bits[j] = 15
                    iMS = 8
                }
                else {
                    iMS = 0
                    Model.Bits[j]++
                }
            }
        }
        Model.State[j] = iMS
        if (Cf == Y_ONLY)
            break
    }
}
```

Table 18. Pseudocode for function UpdateModelMB

4.10 Inverse Color Conversion

1. For 16-bit unsigned int (BD_16), 16-bit signed integer(BD_16S), 32-bit unsigned int (BD_32) and 32-bit signed integer(BD_32S): left shift SHIFT_BITS.

2. For 32-bit float (BD_32F): the decoder finds the sign bit S, exponent E, and mantissa M for a 32-bit float using the bits of the output integer X, a specified mantissa length MANTISSA (l_m), and an exponent bias EXPBIAS (c).

S is 1 if X is negative. Otherwise, S is 0.

The l_m MSBs of the mantissa M is set as the l_m bits in the LSBs of X. This may involve a truncation or left-shift, as necessary.

The exponent E is set by the exponent bias c and a shift value appropriate for a 32-bit float (e.g., 127). For example, E = (X >> l_m) – c + 127.

The output float F is bit composited as: F = [S (1 bit) | E (8 bits) | M (23 bits)].

Chapter 5. Bitstream Layout

This chapter describes the bitstream layout of the HD Photo format. The bitstream is hierarchical and is comprised of the following layers: Image, Tile, Macroblock and Block. Further, the Macroblock and Block layers are laid out differently for the Spatial and Frequency modes of the bitstream. The ordering of these layers and the layout of bits in each layer is defined below.

5.1 Glossary and special syntax

The bitstream layout is defined by a set of tables. Each row of the table describes one step of pseudo-code, or the decoding of one syntax element. In the latter case, the syntax element is spelt out in the first column. Syntax elements are labeled by upper case entries, wheras functions are labeled in mixed case. The number of bits required to decode this element is specified in the 'Num bits' column. The 'Reference' column carries more information about the interpretation and semantics of the syntax element, and the decoding process when corresponding 'Num bits' is labeled as 'Variable'.

'Num bits' labeled as VLWESC is described in 0 below.

The 'Descriptor' column of syntax element entries contain one of three terms: 'bool', 'uimsbf' or 'struct'. The descriptor 'bool' indicates that the syntax element is a Boolean variable represented with one bit. The 'true' value corresponds to the bit being set (i.e. equal to 1), and 'false' is otherwise. The descriptor 'uimsbf' expands to unsigned integer, most significant bit first. This refers to the decoding of bits, starting with the most significant bit being decoded first. The value of the pattern of *k* bits (*k* being specified in the 'Num bits' column, or accompanying text) is interpreted as an unsigned integer and assigned to the variable in the first column. The descriptor 'struct' means that the syntax element is a structure of other component syntax elements. Such elements are further broken up in other referenced tables.

5.1.1 Flush to byte operator (FLUSHBYTE)

FLUSHBYTE is a variable length syntax elements which accounts for the flush-to-byte operation. The length of FLUSHBYTE is therefore between 0 and 7 bits, and upon reading FLUSHBYTE the bitstream 'pointer' is byte aligned. The value of FLUSHBYTE is discarded and therefore does not have any interpretation.

A double horizontal line in the table denotes a byte-aligned location. Every FLUSHBYTE syntax element in the following tables is followed by a double horizontal line indicating that the bitstream is byte-aligned subsequent to the FLUSHBYTE.

5.1.2 Decode Variable Length Word operator (VLWESC)

VLWESC is a variable length code that reads an integer number of bytes and returns an unsigned integer value, or an escape symbol. This code is used in multiple instances, and is defined in Table 19.

VLWESC( ) {	*Number of bits*	*Descriptor*	*Reference*
FIRST_BYTE	8	uimsbf	
if (FIRST_BYTE < 0xfb) {			
SECOND_BYTE	8	uimsbf	
Value = FIRST_BYTE * 256 + SECOND_BYTE			
}			
else if (FIRST_BYTE == 0x0fb) {			
FOUR_BYTES	32	uimsbf	
Value = FOUR_BYTES			
}			
else if (FIRST_BYTE == 0x0fc) {			

EIGHT_BYTES	64	uimsbf	
Value = EIGHT_BYTES			
}			
else { // FIRST_BYTE == 0xfd||0xfe||0xff			
Value = 0 // Escape Mode			
}			
return Value			
}			

Table 19. Pseudocode defining function *VLWESC*

5.2 Image (IMAGE)(Variable size)

IMAGE (){	*Num bits*	*Descriptor*	*Reference*
IMAGE_HEADER	variable	struct	Table 21
alphaPlane = false IMAGE_PLANE_HEADER	variable	struct	Table 25
If (ALPHACHANNEL_PRESENT) { alphaPlane = true IMAGE_PLANE_HEADER	variable	struct	Table 25
} INDEX_TABLE	variable	struct	Table 33
for (n = 0; n < NumberIndexTableEntries; n++) { TILE(n)	variable	struct	5.2.2 5.3
}			
}			

Table 20. Image layer

5.2.1 Image Header (IMAGE_HEADER)

The syntax of image header shall be defined by Table 21.

IMAGE_HEADER (){	*Num bits*	*Descriptor*	*Reference*
GDISIGNATURE	64	uimsbf	5.2.1.1
VERSION	4	uimsbf	5.2.1.2
SUBVERSION	4	uimsbf	5.2.1.3
TILING_FLAG	1	bool	5.2.1.4
BITSTREAMFORMAT	1	uimsbf	5.2.1.5
ORIENTATION	3	uimsbf	5.2.1.6
INDEXTABLE_PRESENT_FLAG	1	bool	5.2.1.7
OVERLAP	2	uimsbf	5.2.1.8
SHORT_HEADER_FLAG	1	bool	5.2.1.9
LONG_WORD_FLAG	1	bool	5.2.1.10
WINDOWING_FLAG	1	bool	5.2.1.11
TRIM_FLEXBITS_FLAG	1	bool	5.2.1.12
TILE_STRETCH_FLAG	1	bool	5.2.1.13
RESERVED	2	uimsbf	5.2.1.14
ALPHACHANNEL_FLAG	1	bool	5.2.1.15
SOURCE_CLR_FMT	4	uimsbf	5.2.1.16
SOURCE_BITDEPTH	4	uimsbf	5.2.1.17
if (SHORT_HEADER_FLAG) { WIDTH_MINUS1	16	uimsbf	5.2.1.18
HEIGHT_MINUS1	16	uimsbf	5.2.1.19
}			
else {			

WIDTH_MINUS1	32	uimsbf	5.2.1.18
HEIGHT_MINUS1	32	uimsbf	5.2.1.19
}			
if (TILING_FLAG) {			
NUM_VERT_TILES_MINUS1	12	uimsbf	5.2.1.20
NUM_HORIZ_TILES_MINUS1	12	uimsbf	5.2.1.21
}			
for (n = 0; n < NUM_VERT_TILES_MINUS1; n++) {			
if (SHORT_HEADER_FLAG)			
WIDTH_IN_MB_OF_TILEI[n]	8	uimsbf	5.2.1.22
else			
WIDTH_IN_MB_OF_TILEI[n]	16	uimsbf	5.2.1.22
}			
for (n = 0; n < NUM_HORIZ_TILES_MINUS1; n++)			
if (SHORT_HEADER_FLAG)			
HEIGHT_IN_MB_OF_TILEI[n]	8	uimsbf	5.2.1.23
else			
HEIGHT_IN_MB_OF_TILEI[n]	16	uimsbf	5.2.1.23
}			
if (TILE_STRETCH_FLAG) {			
for (n = 0; n < NumSpatialTiles; n++)			5.2.1.21
TILE_STRETCH[n]	8	uimsbf	5.2.1.24
}			
if (WINDOWING_FLAG) {			
NUM_TOP_EXTRAPIXELS	6	uimsbf	5.2.1.25
NUM_LEFT_EXTRAPIXELS	6	uimsbf	5.2.1.26
NUM_BOTTOM_EXTRAPIXELS	6	uimsbf	5.2.1.27
NUM_RIGHT_EXTRAPIXELS	6	uimsbf	5.2.1.28
}			

Table 21. Image Header layer

5.2.1.1 GDI Signature (GDISIGNATURE)(64 bits)

GDISIGNATURE is a 64-bit syntax element that identifies the bitstream. It shall have the value 0x574D50484F544F00.

Note: This signature corresponds to “WMPHOTO” in ASCII.

5.2.1.2 Codec Version (VERSION)(4 bits)

VERSION is a 4-bit syntax element that specifies the version of the bitstream. It shall have the value 1. All other values are Reserved.

5.2.1.3 Codec Sub-version (SUBVERSION)(4 bits)

SUBVERSION is a 4-bit syntax element that specifies the sub-version of the bitstream. This value shall be ignored by the decoder.

5.2.1.4 Tiling Flag (TILING_FLAG)(1 bit)

TILING_FLAG is a Boolean syntax element. TILING_FLAG == true specifies that tiles are present in the bitstream, and vice versa.

5.2.1.5 Bitstream Format (BITSTREAM_FORMAT)(1 bit)

BITSTREAM_FORMAT is a 1-bit syntax element, and may take the value 0 or 1. If BITSTREAM_FORMAT == 0, the bitstream shall be laid out in the Spatial mode. If BITSTREAM_FORMAT == 1, the bitstream shall be laid out in the Frequency Mode.

5.2.1.6 Orientation (ORIENTATION)(3 bits)

ORIENTATION is a 3-bit syntax element and specifies the orientation of the image as defined in Table 22.

Note: The orientation syntax element at the container layer shall override the ORIENTATION syntax element. Therefore, this syntax element may be ignored by the decoder. Its purpose is to provide information regarding the desired orientation of the image, and it is desirable for this field to match the orientation specification at the container level.

CLR_FMT	*Meaning of Color Format*
0	NONE
1	FLIP VERTICAL
2	FLIP HORIZONTAL
3	FLIP VERTICAL and FLIP HORIZONTAL
4	ROTATE CLOCKWISE
5	ROTATE CLOCKWISE and FLIP VERTICAL
6	ROTATE CLOCKWISE and FLIP HORIZONTAL
7	ROTATE CLOCKWISE, FLIP VERTICAL and FLIP HORIZONTAL

Table 22. Orientation values

5.2.1.7 Index Table Present Flag (INDEXTABLE_PRESENT_FLAG)(1 bit)

INDEXTABLE_PRESENT_FLAG is a Boolean syntax element that specifies if index table is present in the bitstream. If INDEXTABLE_PRESENT_FLAG == true, the index table shall be present in the bitstream. Further, if BITSTREAM_FORMAT == 'Frequency Mode, or NUM_VERTTILES_MINUS1 > 0, or NUM_HORIZTILES_MINUS1 > 0, INDEXTABLE_PRESENT_FLAG shall be set to true and the index table shall be present in the bitstream. Else, the index table shall not be present in the bitstream. See 5.2.4.

5.2.1.8 Overlap (OVERLAP)(2 bits)

OVERLAP is a 2-bit syntax element that specifies whether overlap is present.

If OVERLAP = 0, no post filtering shall be performed. If OVERLAP = 1, only the outer post filtering (i.e. step 5 of the decoder) shall be performed. If OVERLAP = 2, both inner and outer post filtering (i.e. steps 3 & 5 of the decoder) shall be performed. The value 3 is Reserved.

5.2.1.9 Short Header Flag (SHORT_HEADER_FLAG)(1 bit)

SHORT_HEADER_FLAG is a Boolean syntax element that specifies the bitsizes of the syntax elements that represent width and height of the image and the tiles. If SHORT_HEADER_FLAG == true, smaller bit sizes are used. If SHORT_HEADER_FLAG == false, longer bit sizes are used.

5.2.1.10 Long Word Flag (LONG_WORD_FLAG)(1 bit)

LONG_WORD_FLAG is a Boolean syntax element and specifies whether 16 bit integers may be used for transform computations. If LONG_WORD_FLAG == 0, 16 bit integer numbers and arrays may be used for the outer stage of transform computations. Intermediate operations within the transform (such as (3*a+1)>>1) shall be performed with higher accuracy. If LONG_WORD_FLAG == true, 32 bit integer numbers and arrays shall be used for transform computations.

Note: 32 bit arithmetic may be safely used to decode an image regardless of the value of LONG_WORD_FLAG.

5.2.1.11 Windowing Flag (WINDOWING_FLAG)(1 bit)

WINDOWING_FLAG is a Boolean syntax element that specifies if windowing is present in the bitstream. If WINDOWING_FLAG == true, windowing may be present in the bitstream. If WINDOWING_FLAG == false, windowing shall not be present in the bitstream.

5.2.1.12 Trim FlexBits Flag (TRIM_FLEXBITS_FLAG)(1 bit)

TRIM_FLEXBITS_FLAG is a Boolean syntax element that specifies if Flexbits, if present in the bitstream, are retained in less than full precision.

5.2.1.13 Tile Stretch Flag (TILE_STRETCH_FLAG)(1 bit)

TILE_STRETCH_FLAG is a Boolean syntax element that specifies if tile stretching parameters are present in the bitstream. The value of TILE_STRETCH_FLAG shall be equal to 0. The value 1 is Reserved.

5.2.1.14 Reserved Flag (RESERVED_FLAG)(2 bits)

RESERVED_FLAG is a 2-bit syntax element. The value of RESERVED_FLAG shall be equal to 0. Other values are Reserved.

5.2.1.15 Alpha Channel Flag (ALPHACHANNEL_FLAG)(1 bit)

ALPHACHANNEL_FLAG is a Boolean syntax element that specifies whether interleaved alpha channel is present in the bitstream. If ALPHACHANNEL_FLAG == 1, alpha channel shall be present and interleaved. If ALPHACHANNEL_FLAG == 0, alpha channel shall be absent in the bitstream, or shall be carried as a separate image within the container.

5.2.1.16 Source Color Format (SOURCE_CLR_FMT)(4 bits)

SOURCE_CLR_FMT is a 4-bit syntax element and specifies the color format of the source image as defined in Table 23.

SOURCE_CLR_FMT	Source Color Format
0	Y_ONLY
1	YUV_420
2	YUV_422
3	YUV_444
4	CMYK
5	BAYER
6	N-CHANNEL
7	RGB
8	RGBE
9-15	Reserved

Table 23. Color Format of Source Image

If (iPlane == 1), the value of SOURCE_CLR_FMT shall be equal to 0.

5.2.1.17 Source Bit Depth (SOURCE_BITDEPTH)(4 bits)

SOURCE_BITDEPTH is a 4-bit syntax element and specifies the bitdepth of the source image, as defined in Table 24. BD_1, BD_8, BD_16, BD_32, BD_5 and BD_10 are unsigned integer, corresponding to 1, 8, 16, 32, 5 and 10 bits per channel respectively. BD_16S and BD_32S are signed integer corresponding to 16 and 32 bits per channel respectively. BD_16F is 16-bit Half. BD_32F is 32-bit float. BD_565 corresponds to RGB 5:6:5.

SOURCE_BITDEPTH	Source Bit Depth
0	BD_1, white foreground
1	BD_8
2	BD_16
3	BD_16S
4	BD_16F
5	BD_32
6	BD_32S
7	BD_32F
8	BD_5
9	BD_10
10	BD_565
11-14	Reserved
15	BD_1, black foreground

Table 24. Bit Depth of Source Image

5.2.1.18 Width (WIDTH_MINUS1)(16 bits or 32 bits)

WIDTH_MINUS1 is a syntax element that specifies the width of the coded area, minus 1. If SHORT_HEADER == true, WIDTH_MINUS1 is a 16-bit syntax element. Otherwise, WIDTH_MINUS1 is a 32-bit syntax element.

The width of the coded area shall be defined as

Width = WIDTH_MINUS1 + 1

5.2.1.19 Height (HEIGHT_MINUS1)(16 bits or 32 bits)

HEIGHT_MINUS1 is a syntax element that specifies the height of the coded area, minus 1. If SHORT_HEADER == true, HEIGHT_MINUS1 is a 16-bit syntax element. Otherwise, HEIGHT_MINUS1 is a 32-bit syntax element.

The height of the coded area shall be defined as

Height = HEIGHT_MINUS1 + 1

5.2.1.20 Number of Vertical Tiles (NUM_VERT_TILES_MINUS1)(12 bits)

NUM_VERT_TILES_MINUS1 is a 12-bit syntax element that shall be present only if TILING_FLAG == true, and specifies the number of tiles in a row, minus one. When NUM_VERT_TILES_MINUS1 is not present, its value shall be inferred to be zero.

Note: "Vertical" indicates that the partitioning of the image corresponding to these tiles runs in the vertical direction.

5.2.1.21 Number of Horizontal Tiles (NUM_HORIZ_TILES_MINUS1)(12 bits)

NUM_HORIZ_TILES_MINUS1 is a 12-bit syntax element that shall be present only if TILING_FLAG == 1, and specifies the number of tiles in a row, minus one. When NUM_HORIZ_TILES_MINUS1 is not present, its value shall be inferred to be zero.

Note: "Horizontal" indicates that the partitioning of the image corresponding to these tiles runs in the horizontal direction.

The number of image tiles shall be defined as:

NumSpatialTiles = (NUM_VERT_TILES_MINUS1 + 1) * (NUM_HORIZ_TILES_MINUS1 + 1)

5.2.1.22 Width in MB of Tile n (WIDTH_IN_MB_OF_TILEI[n])(8 or 16 bits)

WIDTH_IN_MB_OF_TILEI[n] is a syntax element that specifies the width (in macroblock units) of the nth tile along the horizontal direction, minus 1. If SHORT_HEADER == true, WIDTH_IN_MB_OF_TILEI[n] is a 8-bit syntax element. Otherwise, it is a 16-bit syntax element.

The width of the last tile in macroblock units is derived by subtracting the transmitted widths (plus one) from the width of the coded area in macroblock units ⌈Width/16⌉.

The position of the left boundary of the nth tile, LeftBoundaryInMBOfTile[n], shall be derived as follows:

```
LeftBoundaryInMBOfTile[0] = 0;

for (n = 0; n <  NUM_VERT_TILES_MINUS1; n++)

   LeftBoundaryInMBOfTile[n+1] = LeftBoundaryInMBOfTile[n] + WIDTH_IN_MB_OF_TILE[n] + 1
```

5.2.1.23 Height in MB of Tile n (HEIGHT_IN_MB_OF_TILEI[n])(8 or 16 bits)

HEIGHT_IN_MB_OF_TILEI[n] is a syntax element that specifies the height (in macroblock units) of the nth tile along the vertical direction, minus 1. If SHORT_HEADER == trye, HEIGHT_IN_MB_OF_TILEI[n] is a 8-bit syntax element. Otherwise, it is a 16-bit syntax element.

The height of the last tile in macroblock units is derived by subtracting the transmitted heights (plus one) from the height of the coded area in macroblock units ⌈Height/16⌉.

The position of the top boundary of the nth tile, TopBoundaryInMBOfTile[n], shall be derived as follows:

```
TopBoundaryInMBOfTile[0] = 0;

for (n = 0; n <  NUM_HORIZ_TILES_MINUS1; n++)

   TopBoundaryInMBOfTile[n+1] = TopBoundaryInMBOfTile[n] + HEIGHT_IN_MB_OF_TILE[n] + 1
```

The number of macroblocks in a tile, NumMBInTile[n], shall be derived as follows:

```
NumMBInTile[n] =   (WIDTH_IN_MB_OF_TILE[n] + 1) * (HEIGHT_IN_MB_OF_TILE[n] + 1)
```

5.2.1.24 Tile Stretch of Tile n (TILE_STRETCH[n])(8 bits)

TILE_STRETCH[n] is an 8-bit syntax element whose value shall be ignored by the decoder.

5.2.1.25 Image Offset from Top (NUM_TOP_EXTRAPIXELS)(6 bits)

NUM_TOP_EXTRAPIXELS is a 6-bit syntax element that shall be present if WINDOWING_FLAG == true, and shall specify the vertical offset of the top of the image from the coded area. When this syntax element is not present, its value shall be set to be zero.

5.2.1.26 Image Offset from Left (NUM_LEFT_EXTRAPIXELS)(6 bits)

NUM_LEFT_EXTRAPIXELS is a 6-bit syntax element that shall be present only if WINDOWING_FLAG == true, and shall specify the horizontal offset of the left boundary of the image from the coded area. When this syntax element is not present, its value shall be set to be zero.

5.2.1.27 Image Offset from Bottom (NUM_BOTTOM_EXTRAPIXELS)(6 bits)

NUM_BOTTOM_EXTRAPIXELS is a 6-bit syntax element that shall be present only if WINDOWING_FLAG == true, and shall specify the vertical offset of the bottom of the image from the coded area. When this syntax element is not present, its value shall be set to be zero.

5.2.1.28 Image Offset from Right (NUM_RIGHT_EXTRAPIXELS)(6 bits)

NUM_RIGHT_EXTRAPIXELS is a 6-bit syntax element that shall be present only if WINDOWING_FLAG == true, and shall specify the horizontal offset of the right boundary of the image from the coded area. When this syntax element is not present, its value shall be set to be zero.

The height and width of the "viewable" image area shall be derived as

ImageWidth = Width - NUM_LEFT_EXTRAPIXELS - NUM_RIGHT_EXTRAPIXELS

ImageHeight = Height - NUM_TOP_EXTRAPIXELS - NUM_BOTTOM_EXTRAPIXELS

5.2.2 Image Plane Header (IMAGE_PLANE_HEADER)(Variable size)

IMAGE_PLANE_HEADER (){	*Num bits*	*Descriptor*	*Reference*
CLR_FMT	3	uimsbf	5.2.2.1
NO_SCALED_FLAG	1	bool	5.2.2.2
BANDS_PRESENT	4	uimsbf	5.2.2.3
if (CLR_FMT == 'YUV_444') {			
CHROMA_CENTERING	4	uimsbf	5.2.2.4
COLOR_INTERPRETATION	4	uimsbf	5.2.2.5
} else if (CLR_FMT == 'N_CHANNEL') {			
NUM_CHANNELS_MINUS1	4	uimsbf	5.2.2.6
COLOR_INTERPRETATION	4	uimsbf	5.2.2.5
}			
if (SOURCE_CLR_FMT == 'BAYER') {			
BAYER_PATTERN	2	uimsbf	5.2.2.7
CHROMA_CENTERING_BAYER	2	uimsbf	5.2.2.8
COLOR_INTERPRETATION	4	uimsbf	5.2.2.5
}			
if (SOURCE_BITDEPTH ∈ {BD_16,BD_16S,BD_32,BD_32S}) {			
SHIFT_BITS	8	uimsbf	5.2.2.9
}			
if (SOURCE_BITEPTH == 'BD_32F') {			
MANTISSA	8	uimsbf	5.2.2.10
EXPBIAS	8	uimsbf	5.2.2.11
}			
DC_FRAME_UNIFORM	1	bool	5.2.2.12
if (DC_FRAME_UNIFORM) {			
DC_QUANTIZER()	variable	struct	Table 30
}			
if (BANDS_PRESENT != 'SB_DC_ONLY') {			
USE_DC_QUANTIZER	1	bool	5.2.2.13
if (USE_DC_QUANTIZER == false) {			
LP_FRAME_UNIFORM	1	bool	5.2.2.14
if (LP_FRAME_UNIFORM) {			
NUM_LP_QUANTIZERS = 1			5.2.2.18
LP_QUANTIZER()	variable	struct	Table 31
}			
}			
if (BANDS_PRESENT != 'SB_NO_HIGHPASS') {			
USE_LP_QUANTIZER	1	bool	5.2.2.15
if (USE_LP_QUANTIZER == false) {			
HP_FRAME_UNIFORM	1	bool	5.2.2.16
if (HP_FRAME_UNIFORM) {			

	NUM_HP_QUANTIZERS = 1			5.2.2.19
	HP_QUANTIZER()	variable	struct	Table 32
	}			
	}			
	}			
}				
FLUSHBYTE		variable		5.1.1
}				

Table 25. Image Plane Header layer

5.2.2.1 Color Format (CLR_FMT)(3 bits)

CLR_FMT is a 3-bit syntax element that shall specify the internal color format of the coded image as defined in Table 26.

CLR_FMT	Color Format
0	Y_ONLY
1	YUV_420
2	YUV_422
3	YUV_444
4	CMYK
5	BAYER
6	N-CHANNEL
7	Reserved

Table 26. Color Format

5.2.2.2 No Scaled Arithmetic Flag (NO_SCALED_FLAG)(1 bit)

NO_SCALED_FLAG is a Boolean syntax element that specifies if the transform uses scaling. If NO_SCALED_FLAG == true, scaling shall not be used. If NO_SCALED_FLAG == false, scaling shall be used. In this case, scaling shall be performed by appropriately rounding down the output of the final stage (color conversion) by 3 bits.

NO_SCALED_FLAG shall be set to true if lossless coding is desired, even if lossless coding is used for only a subregion of an image. Lossy coding may use either mode.

Note: The rate-distortion performance for lossy coding is superior when scaling is used (i.e. NO_SCALED_FLAG == false), especially at low quantization values.

5.2.2.3 Bands Present (BANDS_PRESENT)(4 bits)

BANDS_PRESENT is a 4-bit syntax element that indicates whether the various frequency bands are present in the bitstream. The presence of the bands shall be specified by BANDS_PRESENT syntax element as defined in Table 26.

Note: SB_ISOLATED indicates that the bitstream cannot be decoded without external information. This mode may be used to transmit isolated frequency bands in the frequency domain, which may be decoded in conjunction with the lower frequency bands already known to the decoder.

BANDS_PRESENT	Interpretation
0	SB_ALL (All subbands are present)
1	SB_NO_FLEXBITS (Flexbits is not present)
2	SB_NO_HIGHPASS (Flexbits and Highpass are not present)
3	SB_DC_ONLY (Only DC is present.)
4	SB_ISOLATED
5-15	Reserved

Table 27. Bands Present

The Number of bands present in the bitstream, NumSubBandsPresent, shall be defined as follows:

```
if (BANDS_PRESENT == 'SB_ALL')

    NumBandsPresent = 4;

else if (BANDS_PRESENT == 'SB_NO_FLEXBITS')

    NumBandsPresent = 3;

else if (BANDS_PRESENT == 'SB_NO_HIGHPASS')

    NumBandsPresent = 2;

else if (BANDS_PRESENT == 'SB_DC_ONLY')

     NumBandsPresent = 1;

else  // Reserved or SB_ISOLATED

    NumBandsPresent = 0;
```

5.2.2.4 Chroma Centering (CHROMA_CENTERING)(4 bits)

CHROMA_CENTERING is a 4-bit syntax element and shall be present only if CLR_FMT == 'YUV_444'. When CHROMA_CENTERING is not present, its value shall be inferred to be zero. This value shall be ignored by the decoder.

5.2.2.5 Color Interpretation (COLOR_INTERPRETATION)(4 bits)

COLOR_INTERPRETATION is a 4-bit syntax element that shall be present only if CLR_FMT $\in$ {'YUV_444', 'N_CHANNEL'} or SOURCE_CLR_FMT == 'BAYER'. When COLOR_INTERPRETATION is not present, its value shall be inferred to be zero. This value shall be ignored by the decoder.

5.2.2.6 Number of Channels (NUM_CHANNELS_MINUS1)(4 bits)

NUM_CHANNELS_MINUS1 is a 4-bit syntax element that shall be present only if CLR_FMT == "N_CHANNEL'.

The number of channels, NChannels, shall be defined as

```
if (CLR_FMT == 'N_CHANNEL')

          NChannels = NUM_CHANNELS_MINUS1 + 1

else if (CLR_FMT== Y_ONLY)

          NChannels = 1
```

```
else if (CLR_FMT == YUV_420 || CLR_FMT == YUV_422 || CLR_FMT == YUV_422)
        NChannels = 3
else if (CLR_FMT == CMYK || CLR_FMT == BAYER)
        NChannels = 4
```

5.2.2.7 Bayer Pattern (BAYER_PATTERN)(2 bits)

BAYER_PATTERN is a 2-bit syntax element and shall be present only if SOURCE_CLR_FMT == 'BAYER'. This syntax element shall specify the source Bayer pattern as defined in Table 28. This value shall be ignored by the decoder.

BAYER_PATTERN	Source Bayer Pattern
0	GR / BG
1	GB / RG
2	BG / GR
3	RG / GB

Table 28. Bayer Pattern interpretation

5.2.2.8 Chroma Centering Bayer (CHROMA_CENTERING_BAYER)(2 bits)

CHROMA_CENTERING_BAYER is a 2-bit syntax element and shall be present only if SOURCE_CLR_FMT == 'BAYER'. This value shall be ignored by the decoder.

5.2.2.9 Pre/post Shift Bits (SHIFT_BITS)(8 bits)

SHIFT_BITS is an 8-bit syntax element that shall be present only if SOURCE_BITDEPTH $\in$ {'BD_16', 'BD_16S', 'BD_32', 'BD_32S'} and specifies the number of bits by which to left-shift the reconstructed data.

5.2.2.10 Length of Mantissa (MANTISSA)(8 bits)

MANTISSA is a 8-bit syntax element that shall be present only if SOURCE_BITDEPTH == 'BD_32F'. It specifies the number of mantissa bits in the encoding of floating point data.

5.2.2.11 Exponent Bias (EXPBIAS)(8 bits)

EXPBIAS is a 8-bit syntax element that shall be present only if SOURCE_BITDEPTH == 'BD_32F'. This element specifies the bias of the exponent in the encoding of floating point data.

5.2.2.12 DC Frame Uniform (DC_FRAME_UNIFORM)(1 bit)

DC_FRAME_UNIFORM is a Boolean syntax element that signals whether a single quantizer is used for the DC band. If DC_FRAME_UNIFORM == true, a single quantizer shall be used for the DC bands of each color plane in the image, and this quantizer shall be signaled in the image plane header. If DC_FRAME_UNIFORM == 0, multiple quantizers may be used for the DC bands of each color plane in the image, and these quantizers shall be signaled in the tile header.

5.2.2.13 Use DC Quantizer (USE_DC_QUANTIZER)(1 bit)

USE_DC_QUANTIZER is a Boolean syntax element which signals whether the low pass band uses the same quantizer as the DC band. If USE_DC_QUANTIZER == true, value of the low pass quantizer shall be set to that of the DC band quantizer. If USE_DC_QUANTIZER == false, the value of the low pass quantizer shall be explicitly signaled in the bitstream.

5.2.2.14 Low Pass Frame Uniform Flag (LP_FRAME_UNIFORM)(1 bit)

LP_FRAME_UNIFORM is a Boolean syntax element that signals whether a single quantizer is used for the low pass band. If LP_FRAME_UNIFORM == true, a single quantizer shall be used for all the low pass bands of each color plane in the image, and this quantizer shall be signaled in the image plane header. If LP_FRAME_UNIFORM == false, multiple quantizers may be used for the low pass bands of each color plane in the image, and these quantizers shall be signaled in the tile header.

5.2.2.15 Use Low Pass Quantizer (USE_LP_QUANTIZER)(1 bit)

USE_LP_QUANTIZER is a Boolean syntax element that signals whether the high pass band uses the same quantizer as the low pass band. If USE_LP_QUANTIZER == true, the value of the high pass quantizer shall be set to that of the corresponding low pass band quantizer. If USE_LP_QUANTIZER == false, the value of the high pass quantizer shall be explicitly signaled in the bitstream.

5.2.2.16 High Pass Frame Uniform Flag (HP_FRAME_UNIFORM)(1 bit)

HP_FRAME_UNIFORM is a Boolean syntax element that signals whether a single quantizer is used for the high pass band. If HP_FRAME_UNIFORM == true, a single quantizer shall be used for all the high pass bands of each color plane in the image, and this quantizer shall be signaled in the image plane header. If HP_FRAME_UNIFORM == false, multiple quantizers may be used for the high pass bands of each color plane in the image, and these quantizers shall be signaled in the tile header.

5.2.2.17 Channel Mode (CH_MODE)(2 bits)

CH_MODE is a 2-bit syntax element that is present if NChannels > 1. This element signals whether the color planes share a quantizer. The meaning of this syntax element shall be as defined in Table 29.

Value	*Channel Mode*
0	CH_UNIFORM
1	CH_SEPARATE
2	CH_INDEPENDENT
3	Reserved

Table 29. Interpretation of Channel Mode

If NChannels == 1, the value of CH_MODE shall be set to be 'CH_UNIFORM'.

5.2.2.18 Number of LP Quantizers (NUM_LP_QUANTIZERS)(4 bits)

NUM_LP_QUANTIZERS is a 4 bit syntax element that is present if LP_FRAME_UNIFORM == false. This syntax element signals the number of low pass band quantizers per color plane in the tile. If LP_FRAME_UNIFORM == true, the value of this syntax element shall be set to be 1.

5.2.2.19 Number of HP Quantizers (NUM_HP_QUANTIZERS)(4 bits)

NUM_HP_QUANTIZERS is a 4 bit syntax element that is present if HP_FRAME_UNIFORM == false. This syntax element signals the number of high pass band quantizers per color plane in the tile. If HP_FRAME_UNIFORM == true, the value of this syntax element shall be set to be 1.

5.2.3 Quantizers

DC_QUANTIZER () {	Num bits	Descriptor	Reference
if (NChannels == 1)			
CH_MODE = CH_UNIFORM			5.2.2.17
else			
CH_MODE	2	uimsbf	5.2.2.17
if (CH_MODE == CH_UNIFORM)			
DC_QUANT	8	uimsbf	5.2.3.1
else if (CH_MODE == CH_SEPARATE) {			
DC_QUANT_Y	8	uimsbf	5.2.3.2
DC_QUANT_UV	8	uimsbf	5.2.3.3
}			
else if (CH_MODE == CH_INDEPENDENT) {			
for (i = 0; i < NChannels; i++)			
DC_QUANT_CH[i]	8	uimsbf	5.2.3.4
}			
}			

Table 30. DC_QUANTIZER

LP_QUANTIZER () {	Num bits	Descriptor	Reference
for (q = 0; q < NUM_LP_QUANTIZERS; q++) {			
if (NChannels == 1)			
CH_MODE = CH_UNIFORM			5.2.2.17
else			
CH_MODE	2	uimsbf	5.2.2.17
if (CH_MODE == CH_UNIFORM)			
LP_QUANT[q]	8	uimsbf	5.2.3.5
else if (CH_MODE == CH_SEPARATE) {			
LP_QUANT_Y[q]	8	uimsbf	5.2.3.6
LP_QUANT_UV[q]	8	uimsbf	5.2.3.7
}			
else if (CH_MODE == CH_INDEPENDENT)			
for (i = 0; i < NChannels; i++)			
LP_QUANT_CH[i][q]	8	uimsbf	5.2.3.8
}			
}			

Table 31. LP_QUANTIZER

HP_QUANTIZER () {	Num bits	Descriptor	Reference
for (q = 0; q < NUM_HP_QUANTIZERS; q++) {			
if (NChannels == 1)			
CH_MODE = CH_UNIFORM			5.2.2.17
else			
CH_MODE	2	uimsbf	5.2.2.17
if (CH_MODE == CH_UNIFORM)			
HP_QUANT[q]	8	uimsbf	5.2.3.9
else if (CH_MODE == CH_SEPARATE) {			
HP_QUANT_Y[q]	8	uimsbf	5.2.3.10

	HP_QUANT_UV[q]	8	uimsbf	5.2.3.11
	}			
	else if (CH_MODE == CH_INDEPENDENT)			
	for (i = 0; i < NChannels; i++)			
	HP_QUANT_CH[i][q]	8	uimsbf	5.2.3.12
}				
}				

Table 32. HP_QUANTIZER

5.2.3.1 DC Quant (DC_QUANT)(8 bits)

DC_QUANT is an 8-bit syntax element that is present if CH_MODE == CH_UNIFORM. In this case, the value of the DC quantizer for all the color planes shall be set to DC_QUANT.

5.2.3.2 DC Quant Y (DC_QUANT_Y)(8 bits)

DC_QUANT_Y is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the DC quantizer for the luma plane shall be set to DC_QUANT_Y.

5.2.3.3 DC Quant UV (DC_QUANT_UV)(8 bits)

DC_QUANT_UV is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the DC quantizer for the chroma planes shall be set to DC_QUANT_UV.

5.2.3.4 DC Quant Channel (DC_QUANT_CH[i])(8 bits)

DC_QUANT_CH[i] is an 8-bit syntax element that is present if CH_MODE == CH_INDEPENDENT. In this case, the value of the DC quantizer for the i^{th} color plane shall be set to DC_QUANT_CH[i].

5.2.3.5 Low Pass Quant (LP_QUANT[q])(8 bits)

LP_QUANT[q] is an 8-bit syntax element that is present if CH_MODE == CH_UNIFORM. In this case, the value of the q^{th} low pass quantizer for all the color planes shall be set to LP_QUANT[q].

5.2.3.6 Low Pass Quant Y (LP_QUANT_Y[q])(8 bits)

LP_QUANT_Y[q] is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the q^{th} low pass quantizer for the luma plane shall be set to LP_QUANT_Y[q].

5.2.3.7 Low Pass Quant UV (LP_QUANT_UV[q])(8 bits)

LP_QUANT_UV[q] is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the q^{th} low pass quantizer for the chroma planes shall be set to LP_QUANT_UV[q].

5.2.3.8 Low Pass Quant Channel (LP_QUANT_CH[i][q])(8 bits)

LP_QUANT_CH[i][q] is an 8-bit syntax element that is present if CH_MODE == CH_INDEPENDENT. In this case, the value of the q^{th} low pass quantizer for the i^{th} color plane shall be set to LP_QUANT_CH[i][q].

5.2.3.9 High Pass Quant (HP_QUANT[q])(8 bits)

HP_QUANT[q] is an 8-bit syntax element that is present if CH_MODE == CH_UNIFORM. In this case, the value of the q^{th} high pass quantizer for all the color planes shall be set to HP_QUANT[q].

5.2.3.10 High Pass Quant Y (HP_QUANT_Y[q])(8 bits)

HP_QUANT_Y[q] is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the q^{th} high pass quantizer for the luma plane shall be set to HP_QUANT_Y[q].

5.2.3.11 High Pass Quant UV (HP_QUANT_UV[q])(8 bits)

HP_QUANT_UV[q] is an 8-bit syntax element that is present if CH_MODE == CH_SEPARATE. In this case, the value of the q^{th} high pass quantizer for the chroma planes shall be set to HP_QUANT_UV[q].

5.2.3.12 High Pass Quant Channel (HP_QUANT_CH[i][q])(8 bits)

HP_QUANT_CH[i][q] is an 8-bit syntax element that is present if CH_MODE == CH_INDEPENDENT. In this case, the value of the q^{th} high pass quantizer for the i^{th} color plane shall be set to HP_QUANT_CH[i][q].

5.2.4 Index Table (INDEX_TABLE)(Variable size)

INDEX_TABLE (){	*Num bits*	*Descriptor*	*Reference*
if (INDEXTABLE_PRESENT_FLAG) {			
INDEXTABLE_STARTCODE	16	uimsbf	0
for (n = 0; n < NumberIndexTableEntries; n++) {			5.2.2
INDEX_OFFSET_TILE[n]	VLWESC		0
}			
}			
SKIP_BYTES	VLWESC		5.2.4.3
PADDING_DATA	SKIP_BYTES * 8		5.2.4.4
}			

Table 33. INDEX_TABLE syntax table

The number of index table entries, NumberIndexTableEntries, shall be defined as:

```
if (INDEXTABLE_PRESENT_FLAG == false)

        NumberIndexTableEntries = 0

else {

        if (BITSTREAM_FORMAT == 'Spatial Mode')

            NumberIndexTableEntries =  NumSpatialTiles

        else // BITSTREAM_FORMAT == 'Frequency Mode'

            NumberIndexTableEntries = NumSpatialTiles * NumSubBands;

 }
```

NumSubBands = 4 – BANDS_PRESENT for values of BANDS_PRESENT from 0 through 3. When BANDS_PRESENT is SB_ISOLATED, NumSubBands shall be set to 4.

5.2.4.1 Index Table Start Code (INDEXTABLE_STARTCODE)

INDEXTABLE_STARTCODE is a 16 bit length syntax element which indicates the start of the Index Table. This element shall have the value 0x0001. Other values of INDEXTABLE_STARTCODE are Reserved.

5.2.4.2 Index Offset of Tile n (INDEX_OFFSET_TILE[n])

INDEX_OFFSET_TILE[n] is a syntax element which specifies the offset of the n^{th} tile-frequency packet from the start of the coded image data. The size and value of this syntax element shall be decoded via VLWESC.

The ordering of this information shall be as follows: Index Offset elements corresponding to each tile shall be consecutively ordered in low-to-high order of the frequency, i.e. DC followed by Lowpass, Highpass and Flexbits. Index Offset elements of tiles shall be ordered in the raster scan order of the respective tiles, i.e. top-to-bottom left-to-right. Raster order of tiles shall precede any re-orientation of the image described by the ORIENTATION flag, or similar structure in the container layer.

Note: For images in Spatial mode of operation, only one Index Offset element is sent per tile. Likewise, for images with missing sub-bands (such as when BANDS_PRESENT = 3), Index Offset elements are sent only for the existing sub-bands, except for the case of SB_ISOLATED in which an escape symbol is sent for missing sub-band tiles. An example of this syntax element for an image with 4 spatial tiles and two frequency bands (DC and Lowpass, i.e. BANDS_PRESENT = SB_NO_HIGHPASS) is given below. Here, pD_n and pL_n are the index offset elements of the DC and Lowpass bands of tile n:

$pD_0\ pL_0\ pD_1\ pL_1\ pD_2\ pL_2\ pD_3\ pL_3$

The tile-frequency packets within an image shall be ordered by the rule: higher frequencies of a tile shall not be sent prior to lower frequencies of the same tile.

In the Spatial mode when the number of tiles is 1, the index offset of the only packet is zero.

5.2.4.3 Number of Skipped Bytes (SKIP_BYTES)

SKIP_BYTES is a syntax element which specifies the number of bytes till the start of the coded image data. The size and value of this syntax element is decoded via VLWESC.

5.2.4.4 Padding Data (PADDING_DATA)(SKIP_BYTES)

PADDING_DATA is a stream of SKIP_BYTES number of bytes between the end of the SKIP_BYTES symbol and the start of the coded image data. This data shall be ignored.

Note: This stream of bytes is a placeholder for future bitstream extensions.

5.3 Tile (TILE)(Variable size)

TILE {	Num bits	Descriptor	Reference
TILE_STARTCODE	24	uimsbf	5.3.1.1
TILE_LOCATION_HASH	5	uimsbf	5.3.1.2
TILE_TYPE	3	uimsbf	5.3.1.3
if (TILE_TYPE == 'Spatial' || TILE_TYPE == 'Flexbits')			
if (TRIM_FLEXBITS_FLAG)			5.2.1.12
TRIM_FLEXBITS	4	uimsbf	5.3.1.4
}			
if (TILE_TYPE == 'Spatial') {			
TILE_SPATIAL	variable	struct	Table 35
}			
else if (TILE_TYPE == 'DC') {			
TILE_DC	variable	struct	Table 36
}			
else if (TILE_TYPE == 'Lowpass') {			
TILE_LOWPASS	variable	struct	Table 38
}			
else if (TILE_TYPE == 'Highpass') {			
TILE_HIGHPASS	variable	struct	Table 40
}			
else if (TILE_TYPE == 'Flexbits') {			
TILE_FLEXBITS	variable	struct	Table 42
}			
FLUSHBYTE	variable		5.1.1
}			

Table 34. Tile layer

TILE_SPATIAL {	Num bits	Descriptor	Reference
TILE_HEADER_DC	variable	struct	Table 37
if (BANDS_PRESENT != 'SB_DC_ONLY') {			
TILE_HEADER_LOWPASS	variable	struct	Table 39
if (BANDS_PRESENT != 'SB_NO_HIGHPASS') {			
TILE_HEADER_HIGHPASS	variable	struct	Table 41
}			
}			
for (n = 0; n < NumMBInTile; n++) {			
if (BANDS_PRESENT != 'SB_DC_ONLY' && NUM_LP_QUANTIZERS > 1 && USE_DC_QUANTIZER == false) {			
LP_QUANTIZER_INDEX	variable	uimsbf	5.3.1.5
if (BANDS_PRESENT != 'SB_NO_HIGHPASS' && NUM_HP_QUANTIZERS > 1 && USE_LP_QUANTIZER == false)			
HP_QUANTIZER_INDEX	variable	uimsbf	5.3.1.6
}			
MB_DC(n)	variable	struct	Table 44
if (BANDS_PRESENT != 'SB_DC_ONLY') {			
MB_LP(n)	variable	struct	Table 49
if (BANDS_PRESENT != 'SB_NO_HIGHPASS') {			
MB_CBP(n)	variable	struct	Table 53
for (m = 0; m < number of blocks in macroblock; m++) {			
BLOCK_AC(m)	variable	struct	Table 57
BLOCK_FLEXBITS(m)	variable	struct	Table 70
}			
}			
}			
}			
}			

Table 35. Syntax of Spatial mode Tile (TILE_SPATIAL)

TILE_DC {	Num bits	Descriptor	Reference
TILE_HEADER_DC	variable	struct	Table 37
for (n = 0; n < NumMBInTile; n++) {			
MB_DC(n)	variable	struct	Table 44
}			
}			

Table 36. Syntax of DC Tile (TILE_DC)

TILE_HEADER_DC {	Num bits	Descriptor	Reference
if (DC_FRAME_UNIFORM == FALSE)			
DC_QUANTIZER	variable	struct	Table 30
}			

Table 37. Syntax of DC Tile Header (TILE_HEADER_DC)

TILE_LOWPASS {	Num bits	Descriptor	Reference
TILE_HEADER_LOWPASS	variable	struct	Table 39
for (n = 0; n < NumMBInTile; n++) {			
if (NUM_LP_QUANTIZERS > 1 && USE_DC_QUANTIZER == false) {			
LP_QUANTIZER_INDEX	variable	uimsbf	5.3.1.4
}			
MB_LP(n)	variable	struct	Table 49
}			
}			

Table 38. Syntax of Lowpass Tiles (TILE_LOWPASS)

TILE_HEADER_LOWPASS {	Num bits	Descriptor	Reference
if (LP_FRAME_UNIFORM == false) {			
USE_DC_QUANTIZER	1	bool	5.2.2.13
if (USE_DC_QUANTIZER == false) {			
NUM_LP_QUANTIZERS	4	uimsbf	5.2.2.18
LP_QUANTIZER	variable	struct	Table 31
}			
}			
}			

Table 39. Syntax of Lowpass Tile Header (TILE_HEADER_LOWPASS)

TILE_HIGHPASS {	Num bits	Descriptor	Reference
TILE_HEADER_HIGHPASS	variable	struct	Table 41
for (n = 0; n < NumMBInTile; n++) {			
if (NUM_HP_QUANTIZERS > 1 && USE_LP_QUANTIZER == false)			
HP_QUANTIZER_INDEX	variable	uimsbf	5.3.1.6
MB_CBP(n)	variable	struct	Table 53
MB_AC(n)	variable	struct	Table 57
}			
}			

Table 40. Syntax of Highpass Tiles (TILE_HIGHPASS)

TILE_HEADER_HIGHPASS {	Num bits	Descriptor	Reference
if (HP_FRAME_UNIFORM == false) {			
USE_LP_QUANTIZER	1	bool	5.2.2.15
if (USE_LP_QUANTIZER == false) {			
NUM_HP_QUANTIZERS	4	uimsbf	5.2.2.19
HP_QUANTIZER ()	variable	struct	Table 32
}			
}			
}			

Table 41. Syntax of Highpass Tile Header (TILE_HEADER_HIGHPASS)

TILE_FLEXBITS {	Num bits	Descriptor	Reference
for (n = 0; n < NumMBInTile; n++) {			
MB_FLEXBITS(n)	variable	struct	Table 70
}			
}			

Table 42. Syntax of Flexbits Tiles (TILE_FLEXBITS)

5.3.1.1 Start Code of Tile (TILE_STARTCODE)(24 bits)

TILE_STARTCODE is a 24 bit syntax element that indicates the start of a tile. If the TILE_STARTCODE == 1, the tile shall be considered to be a valid tile and decodable. If TILE_STARTCODE != 1 and TILE_TYPE != 'Flexbits', the tile shall be considered to an invalid tile. If TILE_STARTCODE != 1 and TILE_TYPE == 'Flexbits', the value of TRIM_FLEXBITS shall be set to 15, and all the data bits of this tile shall be set to zero.

Note: There is no guarantee that a byte-aligned 24 bit pattern evaluating to 0x000001 will not occur at any other location in the bitstream. Therefore, TILE_STARTCODE can only be used to reconfirm the start of a tile in conjunction with the index table entries and not as a guaranteed indicator of the start of a tile.

5.3.1.2 Hash of Tile Location (TILE_LOCATION_HASH)(5 bits)

TILE_LOCATION_HASH is a 5 bit syntax element. The value of TILE_LOCATION shall be equal to (VerticalTileIndex * NumberOfHorizontalTiles + HorizontalTileIndex) % 32, except in the case of 'Flexbits' TILE_TYPE for which TILE_LOCATION_HASH can take any arbitrary value.

This value shall match with the corresponding index table entry.

5.3.1.3 Tile Type (TILE_TYPE)(3 bits)

TILE_TYPE is a 3 bit syntax element that shall describe the type of data contained in this tile according to Table 43. This value shall match with the corresponding index table entry.

TILIE_TYPE	Tile Type
0	Spatial tile
1	DC tile
2	Lowpass tile
3	Highpass tile
4	Flexbits tile
5-7	Reserved

Table 43. Tile type

5.3.1.4 Trim Flexbits (TRIM_FLEXBITS)(4 bits)

TRIM_FLEXBITS is a 4 bit syntax element that is present if TILE_TYPE == 'Spatial' or 'Flexbits', and TRIM_FLEXBITS_FLAG == true. Else, TRIM_FLEXBITS shall be set to zero.

5.3.1.5 LP Quantizer Index (LP_QUANTIZER_INDEX)(4 bits)

LP_QUANTIZER_INDEX is a variable length syntax element that is present if NUM_LP_QUANTIZERS > 1, and signals the quantizer index used for the low pass band of each macroblock. The low pass band quantizer for each color plane shall be derived from the q^{th} quantization parameter set where LP_QUANTIZER_INDEX takes the value q.

5.3.1.6 HP Quantizer Index (HP_QUANTIZER_INDEX)(4 bits)

HP_QUANTIZER_INDEX is a variable length syntax element that is present if NUM_HP_QUANTIZERS > 1, and signals the quantizer index for the high pass band of each macroblock. The high pass band quantizer for each color plane shall be derived from the q^{th} quantization parameter set where HP_QUANTIZER_INDEX takes the value q.

5.4 Macroblock

5.4.1 Macroblock DC (MB_DC)

MB_DC {	*Number of bits*	*Descriptor*	*Reference*
Lm[2] = {0, 0}			
if (CLR_FMT ∈ {Y_ONLY, CMYK, N_CHANNEL}) {			
for (n=0; n < NChannels; n++) {			
iDC[n] = 0			
IS_DC_CH	1	bool	5.4.1.1
m = (n != 0)			
if (IS_DC_CH) {			
iDC[n] = DEC_DC (iModelDC[m], VLDC[1])	variable	uimsbf	Table 45
Lm[m] = Lm[m] + 1			
}			
}			
}			
else { // CLR_FMT ∉ {Y_ONLY, CMYK, N_CHANNEL}			
IS_DC_YUV // decode with VLDC[0]	variable	uimsbf	5.4.1.2
iDCY = iDCU = iDCV = 0			
if (IS_DC_YUV & 4) {			
iDCY = DEC_DC (iModelDC[0], VLDC[1])	variable	uimsbf	Table 45
Lm[0] = Lm[0] + 1			
}			
if (IS_DC_YUV & 2) {			
iDCU = DEC_DC (iModelDC[1], VLDC[2])	variable	uimsbf	Table 45
Lm[1] = Lm[1] + 1			
}			
if (IS_DC_YUV & 1) {			
iDCV = DEC_DC (iModelDC[1], VLDC[2])	variable	uimsbf	Table 45
Lm[1] = Lm[1] + 1			
}			
}			
UpdateModelMB (Lm, ModelDC)			Table 18
if (ResetContext) {			Table 72
for (k = 0; k <= 2; k++)			
Adapt (VLDC[k])			
}			
}			

Table 44. Syntax of Macroblock DC Layer

DEC_DC (iModelBits, VLDC[k]) {	Number of bits	Descriptor	Reference
DC = DECODE_ABS_LEVEL (VLDC[k]) – 1	variable	uimsbf	Table 46
if (iModelBits) {			
DC_REF	iModelBits	uimsbf	5.4.1.8
DC = (DC<< iModelBits) | DC_REF			
}			
SIGN	1	bool	5.4.1.9
if (SIGN == true)			
DC = –DC			
Return DC			
}			

Table 45. Syntax of DC coefficient decode DEC_DC

DECODE_ABS_LEVEL (VLDC[k]) {	Number of bits	Descriptor	Reference
aRemap[] = {2,3,4,6,10,14}			
aFixedLen[]={0,0,1,2,2,2};			
ABSLEVEL_INDEX // decode with VLDC[k]	variable	uimsbf	5.4.1.3
if (ABSLEVEL_INDEX < 6) {			
iFixed = aFixedLen[ABSLEVEL_INDEX]			
iLevel = aRemap[ABSLEVEL_INDEX]			
if (iFixed > 0) {			
LEVEL_REF	iFixed	uimsbf	5.4.1.5
iLevel += LEVEL_REF			
}			
}			
else { // Escape mode			
FIXED_NUM	4	uimsbf	5.4.1.4
iFixed = FIXED_NUM + 4;			
if (iFixed == 19) {			
FIXED_NUM_EXT	2	uimsbf	5.4.1.5
iFixed += FIXED_NUM_EXT			
if (iFixed == 22) {			
FIXED_NUM_EXT2	3	uimsbf	5.4.1.6
iFixed += FIXED_NUM_EXT2			
}			
}			
LEVEL_REF	iFixed	uimsbf	5.4.1.7
iLevel = 2 + (1 << iFixed) + LEVEL_REF			
}			
Return iLevel			
}			

Table 46. Syntax of DECODE_ABS_LEVEL

5.4.1.1 Is DC Channel (IS_DC_CH)(1 bit)

IS_DC_CH is a Boolean syntax element that is present if CLR_FMT $\in$ {Y_ONLY, CMYK, N_CHANNEL}. If IS_DC_CH is true, the DC coefficient of the corresponding color plane shall be signaled in the bitstream. If IS_DC_CH is false, the DC coefficient of the corresponding color plane shall be set to zero.

5.4.1.2 Is DC YUV (IS_DC_YUV)(Variable)

IS_DC_YUV is a variable length syntax element that is present if CLR_FMT $\notin$ {Y_ONLY, CMYK, N_CHANNEL}. IS_DC_CH jointly signals the zero/non-zero status of the DC coefficients of the Y, U and V planes. The variable length code VLDC[0] of alphabet size 8 shall be used in decoding IS_DC_YUV. This code is defined in Table 47.

IS_DC_YUV	Code
0	10
1	001
2	0000 1
3	0001
4	11
5	010
6	0000 0
7	011

Table 47. Code tables for IS_DC_YUV

5.4.1.3 Absolute Level Index (ABSLEVEL_INDEX)(Adaptive Variable size)

ABSLEVEL_INDEX is a variable length syntax element that takes on values between 0 and 6. If ABSLEVEL_INDEX < 6, it jointly signals the coarse level, and the number of bits needed to signal the level refinement. Otherwise, it signals the escape mode, where the number of bits needed to signal the level is signaled followed by the level itself. The variable length code VLDC[k] of alphabet size 7, where k=1 or 2, shall be used in decoding ABSLEVEL_INDEX.

The coding of this symbol shall use one of two tables, adaptively determined as specified by the code. The codes are defined in Table 48.

ABSLEVEL_INDEX	Code 0	Code 1
0	01	1
1	10	01
2	11	001
3	001	0001
4	0001	0000 1
5	0000 0	0000 00
6	0000 1	0000 01

Table 48. Code tables for ABSLEVEL_INDEX

5.4.1.4 Fixed Number (FIXED_NUM)(4)

FIXED_NUM is a 4-bit syntax element that shall be present if ABSLEVEL_INDEX ≥ 6 (in the escape mode). It signals the number of bits needed to represent the coefficient level.

5.4.1.5 Fixed Number Extension (FIXED_NUM_EXT)(2)

FIXED_NUM_EXT is a 2-bit syntax element that shall be present if FIXED_NUM == 15. It signals the number of extension bits needed to represent the coefficient level.

5.4.1.6 Fixed Number Extension 2 (FIXED_NUM_EXT2)(3)

FIXED_NUM_EXT is a 2-bit syntax element that shall be present if FIXED_NUM == 15 and FIELD_NUM_EXT == 3. It signals the number of additional extension bits needed to represent the coefficient level.

5.4.1.7 Level Refinement (LEVEL_REF)(iFixed)

LEVEL_REF is a syntax element which signals the refinement in level value. The number of bits, iFixed, needed to signal this syntax element shall be computed from either ABSLEVEL_INDEX if ABSLEVEL_INDEX < 6, or from FIXED_NUM and FIXED_NUM_EXT if ABSLEVEL_INDEX ≥ 6.

5.4.1.8 DC Refinement (DC_REF)(iModelBits)

DC_REF is a syntax element which signals the refinement in the DC value. DC_REF shall be sent with iModelBits number of bits.

5.4.1.9 Sign (SIGN)(1 Bit)

SIGN is a Boolean syntax element which signals the sign of a coefficient. If SIGN is true, the coefficient shall be negative, otherwise the coefficient shall be positive.

5.4.2 Macroblock Lowpass (MB_LP)(Variable size)

MB_LP() {	*Num bits*	*Descriptor*	*Reference*
Lm[2] = {0,0}			
iQIndex = 0			
if (ResetTotals) {			Table 73
for (k = 1; k < 16; k++)			
giTotalsLP[k] = 34 – k*2			
}			
if (CLR_FMT ∈ {YUV_420, YUV_422, YUV_444}) {			
iFullPlanes = 2 + (CLR_FMT == YUV_444)			
iMax = (1 << iFullPlanes) – 1; // Max value of CBP			
if (giCntZero <= 0 || giCntMax < 0) {			
CBP_YUV_LP1	variable	uimsbf	5.4.2.1
if (giCntMax < giCntZero)			
iCBPLP = iMax – CBP_YUV_LP1			
else			
iCBPLP = CBP_YUV_LP1			
}			
else {			
CBP_YUV_LP2	iFullPlanes	uimsbf	5.4.2.2
iCBPLP = CBP_YUV_LP2			
}			
giCntZero += 1 – (4 * (iCBP == 0))			
giCntZero = max(-8, min(7,g_iCntZero))			
giCntMax += 1 – (4 * (iCBP == iMax))			
giCntMax = max(-8, min(7,g_iCntMax))			
}			
else {			
iCBPLP = 0			
for(n=0; n < NChannels; n++) {			
CBP_CH_LP	1	bool	5.4.2.3
iCBPLP |= (CBP_CH_LP << n)			
}			
}			
for (n = 0; n < FullPlanes; n++) {			
if ((iCBP>>n)&1) {			
iNumNonZero = DECODE_BLOCK(iChannel > 0, aRLCoeffs[], VLLP[], 5, 1+9*((CLR_FMT==YUV_420)&& (n==1))+((CLR_FMT==YUV_422)&&(n==1)));	variable	struct	Table 59
if ((CLR_FMT == YUV_420 || CLR_FMT == YUV_422) && n) {			
aTemp[14] = {0}			
aRemap[ ] = {4, 1, 2, 3, 5, 6, 7};			
pRemap = aRemap + (CLR_FMT == YUV_420)			
iCountChr = CLR_FMT == YUV_422 ? 14: 6			
i = 0			
for (k=0; k< iNumNonZero; k+) {			
i += aRLCoeffs[k*2]			
aTemp[i] = aRLCoeffs[k*2+1]			
i++			
}			
for (k = 0; k < iCount; k++)			
aDC[k &1 + 1][pRemap[k>>1]] = aTemp[k]			

```
            }
         }
         else {
            i = 1
            for (k=0; k< iNumNonZero; k+) {
               i += aRLCoeffs[k*2]
               pCoeffs[pScan[i]] = aRLCoeffs[k*2+1]
               pTotals[i]++
               if (pTotals[i] > pTotals[i-1]) {
                  Swap(pTotals[i], pTotals[i–1])  // exchanges values
                  Swap(pScan[i], pScan[i–1])  // exchanges values
               }
               i++
            }
         }
         iIndx = n > 0
         iModelBits = ModelLP.Bits[iIndx]
         Lm[iIndx] += iNumNonZero
         if (iModelBits) {
         if (CLR_FMT==YUV_420 ||CLR_FMT==YUV_422 &&n) {
            for (k=0; k<(CLR_FMT==YUV_420 ? 4 :8); k++) {
               DC[1][k]= REFINE_LP(DC[1][k], iModelBits);          variable    struct    Table 50
               DC[2][k] = REFINE_LP(DC[2][k], iModelBits);         variable    struct    Table 50
            }
         }
         else {
            for (k = 0; k< 16; k++)
               pCoeffs[k]  = REFINE_LP(pCoeffs[k], iModelBits);   variable    struct    Table 50
         }
      }
   }
   UpdateModelMB (Lm, ModelLP)                                                          Table 18
   if (ResetContext) {                                                                  Table 72
      for (k = 0; k < 8; k++)
         Adapt(VLLP[k]);
   }
}
```

Table 49. MB LowPass layer

REFINE_LP(Coeff, iModelBits) {	Number of bits	Descriptor	Reference
COEFF_REF	iModelBits	uimsbf	5.4.2.3
if (Coeff > 0) {			
Coeff <<= iModelBits;			
Coeff += COEFF_REF;			
}			
else if (Coeff < 0) {			
Coeff <<= iModelBits;			
Coeff -= COEFF_REF;			
}			
else {			
Coeff = COEFF_REF;			
if (Coeff) {			
SIGN	1	bool	5.4.1.9
if (SIGN)			
Coeff = -Coeff;			
}			
}			
return Coeff;			
}			

Table 50. Refine in MB LowPass layer

5.4.2.1 CBP Low Pass YUV1 (CBP_LP_YUV1)(Variable Size)

CBP_LP_YUV1 is a syntax element that shall be present if CLR_FMT ∈ {YUV_420, YUV_422, YUV_444} and (giCntZero[iCxt] <= 0 || giCntMax[iCxt] < 0). It jointly signals the status of the low pass band of the Y, U and V color planes.

If CLR_FMT is YUV_444, the decoding of CBP_LP_YUV1 shall be defined by Table 51. If CLR_FMT is YUV_420 or YUV_422, the decoding of CBP_LP_YUV1 shall be defined by Table 52.

CBP_LP_YUV1	Meaning
0	0
100	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Table 51. Decoding of CBP_LP_YUV1 if CLR_FMT = YUV_444

Value	Tile Type
0	0
10	1
110	2
111	3

Table 52. Decoding of CBP_LP_YUV1 if CLR_FMT ∈ {YUV_420, YUV_422}

5.4.2.2 CBP Low Pass YUV2 (CBP_LP_YUV2)(iFullPlanes)

CBP_LP_YUV2 is a syntax element that shall be present if CLR_FMT ∈ {YUV_420, YUV_422, YUV_444} and (giCntZero[iCxt] > 0 && giCntMax[iCxt] >= 0). The size of this syntax element is given by iFullPlanes.

5.4.2.3 CBP Low Pass Channel (CBP_LP_CH)(1)

CBP_LP_CH is a Boolean syntax element that shall be present for each color channel in an image, if CLR_FMT ∉ {YUV_422, YUV_420, YUV_444}. It signals the status of the low pass band of the corresponding color plane. If CBP_LP_CH is false, the low pass band for this macroblock of the corresponding color plane shall be set to zero. If CBP_LP_CH is true, the low pass band for this macroblock shall be non-zero.

5.4.2.4 Coeff Refinement (COEFF_REF)(iModelBits)

COEFF_REF is a syntax element that refines the value of the low pass coefficient. The size of this syntax element is given by iModelBits.

5.4.3 Coded Block Pattern (CBP)(Variable size)

MB_CBP(Context) {	*Number of bits*	*Descriptor*	*Reference*
gFLC [] = {0, 2, 1, 2, 2, 0}			
gOff[] = {0, 4, 2, 8, 12, 1}			
gOut[] ={0, 15, 3, 12, 1, 2, 4, 8, 5, 6, 9, 10, 7, 11, 13, 14}			
if(CLR_FMT == CMYK || CLR_FMT == N_CHANNEL)			
iChannel = NChannels;			
else			
iChannel = 1;			
for (i = 0; i <iChannel; i++) {			
NUM_CBP	variable	uimsbf	5.4.3.1
iCBP = REFINE_CBP (NUM_CBP)	variable	struct	Table 54
for (iBlock = 0; iBlock < 4; iBlock++) {			
if (iCBP &(1<<iBlock)) {			
NUM_BLKCBP	variable	uimsbf	5.4.3.2
iVal = NUM_BLKCBP+1			
iBlkCBPCY = 0			
if (iVal>=6) { //Is chroma			
CHR_CBP	variable	uimsbf	Table 56
iBlkCBPCY = 0x10 * (CHR_CBP +1)			
if (iVal > =9) {			
VAL_INC	variable	uimsbf	Table 56
iVal += VAL_INC			
}			
iVal -= 6			
}			
iCode = gOff[iVal]			
if (gFLC[iCode]) {			
CODE_INC	gFLC[iCode]	uimsbf	5.4.3.3
iCode += CODE_INC			
}			
iBlkCBPCY += gOut[iCode]			
if (CLR_FMT == YUV_444) {			
iDiffCBP[0] |= (iBlkCBPCY & 0xf) << (iBlock * 4)			
for (k = 0; k < 2; k++) {			
if ((iBlkCBPCY >> (k+4)) & 0x01) {			
NUM_CH_SUBBLK	variable	uimsbf	5.4.3.4
iCBPChr = REFINE_CBP (NUM_CH_SUBBLK)	variable	struct	Table 54
iDiffCBP[k+1] |= (iCBPChr << (iBlock * 4));			
}			
}			
} //CLR_FMT == YUV_444			
else if (CLR_FMT == YUV_422) {			

iDiffCBP[0] |= (iBlkCBPCY & 0xf) << (iBlock * 4)
for (k = 0; k < 2; k++)
if((iBlkCBPCY >> (k+4)) & 0x01) {
iShift[4] = {0, 1, 4, 5}
CBP_CH_SBLK
	variable	uimbsf	Table 56

iCBPChr = iShift[CBP_CH_SBLK+ 1]
iDiffCBP[k+1] |= (iCBPChr << iShift[iBlock])
}
}
} //CLR_FMT == YUV_422
else if (CLR_FMT == YUV_420) {
iDiffCBP[0] = ((iBlkCBPCY & 0xf) << (iBlock * 4)
iDiffCBP[1]= ((iBlkCBPCY >> 4) & 0x1) << iBlock)
iDiffCBP[2]= ((iBlkCBPCY >> 5) & 0x1) << iBlock)
}
else { // Default
iDiffCBP [i] = ((iBlkCBPCY) << (iBlock * 4))
}
} // if(iCBP…)
} // iBlock
} // i
}			

Table 53. Macroblock Coded Block Pattern

REFINE_CBP (iNum) {	*Number of bits*	*Descriptor*	*Reference*
if (iNum == 2) { REF_CBP1	variable	uimsbf	Table 55

}
else if (iNum == 1) {
REF_CBP
	2	uimsbf	5.4.3.5

iNum = (1<<REF_CBP)
}
else if (iNum == 3) {
REF_CBP
	2	uimsbf	5.4.3.5

iNum = (0xf ^ (1<<REF_CBP))
}
else if (iNum == 4)
iNum = 0xf;
Return iNum
}			

Table 54. Refinement of Coded Block Pattern (REFINE_CBP)

Value	*iNum*
00	3
01	5
100	6
101	9
110	10
111	12

Table 55. Decoding of REF_CBP1

5.4.3.1 Number CBP (NUM_CBP)(Adaptive Variable Size)

NUM_CBP is a variable syntax element that signals the number of coded blocks in a macroblock. The VLC that signals NUM_CBP is adaptive.

Note: The identity of blocks that are coded is obtained by calling the RefineCBP ( ).

5.4.3.2 Number Block CBP (NUM_BLKCBP)(Adaptive Variable Size)

NUM_BLKCBP is a variable length syntax element that signals the number of coded sub blocks in a block. The VLC that signals NUM_CBP is adaptive.

Bit pattern	Value
1	0
01	1
00	2

Table 56. Decoding of CHR_CBP, VAL_INC and CBP_CH_SBLK

5.4.3.3 Code Increment (CODE_INC)(g_FLC[iCode])

CODE_INC is a syntax element that signals the number of coded sub blocks in a block. The size of this syntax element shall be defined by g_FLC[iCode], where g_FLC[] and iCode are defined in Table 53.

5.4.3.4 Code Increment (NUM_CH_SBLK)(Adaptive Variable Size)

NUM_CH_SBLK is a syntax element that signals the number of coded chroma subblocks in a block if CLR_FMT = YUV_444. The VLC that defines the decoding of NUM_CH_SUBBLK is adaptive.

Note: The identity of chroma sub blocks that are coded is obtained by calling the RefCBP ( ).

5.4.3.5 Refine CBP (REF_CBP)(2)

REF_CBP is a fixed length syntax element that refines the coded status of the blocks or subblocks. The size of REF_CBP shall be 2 bits.

5.5 AC

5.5.1 Macroblock AC (MB_AC)(Variable size)

MB_AC () {	Num bits	Descriptor	Reference
if (ResetTotals) {			Table 73
TotalsH[0] = TotalsV[0] = 0x7FFF			
for (k = 1; k < 16; k++) {			
TotalsH[k] = TotalsV[k] = 34 – k*2			
}			
}			
int Lm[2] = {0, 0}			
iOrient = g_iOrientation			
if (iOrient == 1) {			
pScan = ScanV			
pTotals = TotalsV			
}			
else {			
pScan = ScanH			
pTotals = TotalsH			
}			
for (i = 0; i < NChannels; i++) {			
iIndx = i > 0			
bChroma = iIndx			
iModelBits = ModelAC.Bits[iIndx]			
iNBlocks = 4			
if (iIndx && CLR_FMT == YUV_420)			

iNBlocks = 1			
else if (iIndx && CLR_FMT == YUV_422)			
iNBlocks = 2			
iCBP = giCBP[i]			
for (iBlock = 0; iBlock < iNBlocks; iBlock++, iCBP >>= 1) {			
iNumNonZero = DECODE_BLOCK_ADAPTIVE (iCBP &1, bChroma, giHuff, pCoeffs, pScan, pTotals, iModelBits, iQP, pOrder)	struct	uimsbf	Table 58
iLapMean[iIndx] += iNumNonZero			
}			
}			
UpdateModel (Lm, ModelAC)			Table 18
}			

Table 57. Syntax of Block AC

DECODE_BLOCK_ADAPTIVE(bNoSkip, bChroma, VLD[], pCoeffs pScan[], pTotals[], iModelBits, Int pOrder[]) {	Number of bits	Descriptor	Reference
aLocalCoeff [16] = {0}			
k = iLocation			
if (bNoSkip) {			
iQP1 = (iQP << iModelBits)			
iNumNonZero = DECODE_BLOCK (bChroma, aLocalCoeff, VLD, 13, 1)	struct	uimsbf	Table 59
for (kk = 0; kk < iNumNonZero; kk++) {			
k += aLocalCoeff[kk * 2]			
pCoeffs[pScan[k]] = aLocalCoeff[kk*2+1] * iQP1			
pTotals[k]++			
if (pTotals[k] > pTotals[k-1]) {			
Swap (pTotals[k], pTotals[k–1]) // exchange values			
Swap (pScan[k], pScan[k–1]) // exchange values			
}			
k++			
}			
}			
return iNumNonZero			
}			

Table 58. DecodeBlockAdaptive

DECODE_BLOCK(bChroma, Coeff[], VLC[], iCxtOffset, iLocation) {	Number of bits	Descriptor	Reference
iNumNZ = 1			
iHuffIndx = bChroma * 3+iCxtOffset;			
FIRST_INDEX // variable length decode with VLC[iHuffIndx]	variable	uimsbf	5.5.1.1
SIGN	1	bool	
iSR = FIRST_INDEX & 1			
iSRn = FIRST_INDEX >> 2			
iCont = iSR & iSRn			
if (FIRST_INDEX & 2)			
Coeff[1] = DECODE_ABS_LEVEL (VLC[6+iCxtOffset+iCont])	variable	uimsbf	Table 46
else			
Coeff[1] = 1			
if (SIGN)			
Coeff[1] = –Coeff[1]			
Coeff[0] = 0			
if (iSR == 0)			
Coeff[0] = DECODE_RUN (15 – iLocation)	struct	uimsbf	Table 60
iLocation += Coeff[0]			
while (iSRn != 0) {			
iSR = iSRn & 1			
Coeff[iNumNZ * 2] = 0			
if (iSR == 0)			
Coeff[iNumNZ * 2] = DECODE_RUN (15 – iLocation)	struct	uimsbf	Table 60
iLocation += Coeff[iNumNZ * 2]			
iIndex = DECODE_INDEX (iLocation, VLC[iHuffIndex+iCont+1])	variable	uimsbf	Table 61

iSRn = iIndex >> 1			
iCont &= iSRn			
SIGN	1	bool	5.4.1.9
if (iIndex & 1)			
Coeff[iNumNZ*2+1] = DECODE_ABS_LEVEL (VLC[6+iCxtOffset+iCont])	struct	uimsbf	Table 46
Else			
Coeff[iNumNZ*2+1] = 1			
if (SIGN)			
Coeff[iNumNZ*2+1] = –Coeff[iNumNZ * 2+1]			
iNumNZ++			
}			
Return iNumNZ			
}			

Table 59. DecodeBlock

DECODE_RUN (iMaxRun) {	*Number of bits*	*Descriptor*	*Reference*
aRemap[ ] = { 1,2,3,5,7, 1,2,3,5,7, 1,2,3,4,5};			
aRunBin[ ] ={-1,-1,-1,-1, 2,2,2, 1,1,1,1, 0,0,0,0}s			
aRunFixedLength[ ] = { 0, 0, 1, 1,3, 0,0, 1,1,2, 0,0,0,0,1 };			
if (iMaxRun < 5) {			
iRun = 1;			
if (iMaxRun != 1)			
RUN	variable	uimsbf	5.5.1.5
iRun = RUN;			
}			
else {			
RUN_INDEX	variable	uimsbf	5.5.1.6
iIndex = RUN_INDEX + 5* aRunBin[RUN_INDEX]			
iFixed = aRunFixedLength[iIndex]			
iRun = aRemap[iIndex]			
if (iFixed) {			
RUN_REF	iFixed	uimsbf	5.5.1.7
iRun += RUN_REF			
}			
}			
Return iRun			
}			

Table 60. DecodeRun

DECODE_INDEX(iLocation, VLC) {	*Number of bits*	*Descriptor*	*Reference*
if (iLocation < 15) {			
INDEX1 // variable length decode with VLC	variable	uimsbf	5.5.1.2
iIndex = INDEX1			
}			
else if (iLocation == 15) {			
INDEX2	variable	uimsbf	Table 64
iIndex = INDEX2			
}			
else {			
INDEX3	1	uimsbf	5.5.1.4
iIndex = INDEX3			
}			
Return iIndex			
}			

Table 61. DecodeIndex

5.5.1.1 First index symbol (FIRST_INDEX)(Adaptive Variable size)

FIRST_INDEX is a variable length symbol that takes on values between 0 and 11. The coding of this symbol shall use one of five tables, adaptively determined as specified by the code. The codes are defined in Table 62.

FIRST_INDEX	Code 0	Code 1	Code 2	Code 3	Code 4
0	0000 1	0010	11	001	010
1	0000 01	0001 0	001	11	1
2	0000 000	0000 00	0000 000	0000 000	0000 001
3	0000 001	0000 01	0000 001	0000 1	0001
4	0010 0	0011	0000 1	0001 0	0000 010
5	010	010	010	010	011
6	0010 1	0001 1	0000 010	0000 001	0000 0000
7	1	11	011	011	0010
8	0011 0	011	100	0001 1	0000 011
9	0001	100	101	100	0011
10	0011 1	0000 1	0000 011	0000 01	0000 0001
11	011	101	0001	101	0000 1

Table 62. Code tables for FIRST_INDEX

5.5.1.2 Index symbol1 (INDEX1)(Adaptive Variable size)

INDEX1 is a variable length symbol that is present if iLocation <15 and takes on values between 0 and 5. The coding of this symbol shall use one of xxx tables, adaptively determined as specified by the code. The codes are defined in Table 63.

INDEX1	Code 0	Code 1	Code 2	Code 3
0	1	01	0000	0 0000
1	0 0000	0000	0001	0 0001
2	001	10	01	01
3	0 0001	0001	10	1
4	01	11	11	0001
5	0001	001	001	001

Table 63. Code tables for INDEX1

5.5.1.3 Index symbol2 (INDEX2)(Variable size)

INDEX2 shall be coded as defined in Table 64.

Bit pattern	Value
0	0
10	2
110	1
111	3

Table 64. Decoding of INDEX2

5.5.1.4 Index symbol3 (INDEX3)(1)

INDEX2 is a 1 bit syntax element that is present if iLocation == 16, and represents the presence of subsequent run/level symbols.

5.5.1.5 Run (RUN)(Variable size)

RUN is a variable length syntax element that is present if (iMaxRun > 1 && iMaxRun < 5), and represents the value of run. If iMaxRun == 2, the decoding of RUN shall be specified by Table 65. If iMaxRun == 3, the decoding of RUN shall be specified by Table 66. If iMaxRun == 2, the decoding of RUN shall be specified byTable 67.

Bit pattern	Value
1	1
0	2

Table 65. Decoding of RUN if iMaxRun == 2

Bit pattern	Value
1	1
01	2
00	3

Table 66. Decoding of RUN if iMaxRun == 3

Bit pattern	Value
1	1
01	2
001	3
000	4

Table 67. Decoding of RUN if iMaxRun == 4

5.5.1.6 Run Index (RUN_INDEX)(Variable size)

RUN_INDEX is a variable length syntax element that is present if iMaxRun >= 5, and represents the value of run. The decoding of RUN_INDEX shall be defined by Table 68.

Bit pattern	Value
1	1
01	2
001	3
0000	4
0001	5

Table 68. Decoding of RUN_INDEX

5.5.1.7 Run Refine (RUN_REF)(iFixed)

RUN_REF is a fixed length syntax element that refines the value of run. The presence and size of the LEVEL_REF syntax element shall be specified by iFixed, as defined in Table 60.

5.6 Flexbits

5.6.1 MB Flexbits (MB_FLEXBITS)

MB_FLEXBITS () {	Num bits	Descriptor	Reference
for (i = 0; i < NChannels; i++) {			
iIndx = i > 0			
iModelBits = ModelAC[iIndx]			
iNBlocks = 4			
if (iIndx && CLR_FMT == YUV_420)			
iNBlocks = 1			
else if (iIndx && CLR_FMT == YUV_422)			
iNBlocks = 2			
for (iBlock = 0; iBlock < iNBlocks; iBlock++)			
iNumNonZero = BLOCK_FLEXBITS (pCoeffs, iModelBits, TRIM_FLEXBITS, iQP, gbSkipFlexbits)	struct	uimsbf	Table 70
}			
}			

Table 69. Syntax of MB Flex bits

BLOCK_FLEXBITS (pCoefsf[], iModelBits, iTrimFlexBits, iQP, bSkipFlexbits) {	Num bits	Descriptor	Reference
pOrder[ ] = {0,5,1,6, 10,12,8,14, 2,4,3,7, 9,13,11,15}			
iFlexBitsLeft = iModelBits – iTrimFlexBits			
if (iFlex BitsLeft < 0 || bSkipFlexBits)			
iFlexBitsLeft = 0			
if (iFlexBitsLeft) {			
iQP1 = iQP << iTrim			
for (n =1; n < 16; n++) {			
j = pOrder[n]			
pCoeff[j] = DECODE_FLEX(pCoeff[j], iFlexBitsLeft, iQP1)	struct	uimsbf	Table 71
}			
}			

Table 70. Syntax of Block flexbits

DECODE_FLEX(Coeff, iFlexBitsLeft, iQP) {	Number of bits	Descriptor	Reference
FLEX_REF	iFlexBitsLeft	uimsbf	5.6.1.1
Coeff <<= iModelBits;			
if (Coeff > 0) {			
Coeff += FLEX_REF *iQP;			
}			
else if (Coeff < 0)			
Coeff –= FLEX_REF * iQP;			
}			
else {			
Coeff = FLEX_REF * iQP;			
if (Coeff) {			
SIGN	1	bool	5.4.1.9
if(SIGN)			
Coeff = –Coeff;			
}			
}			
return Coeff;			
}			

Table 71. Decoding Flex bits

5.6.1.1 Flex Refinement (FLEX_REF)(iFlexBitsLeft)

FLEX_REF is a syntax element that represents the flexbits part of the coefficient. The size of this syntax element is given by iFlexBits.

5.7 Functions

Functions refer to processing steps that do not involve the decoding of any syntax element. These steps shall be performed as specified in the syntax tables defining the decoding process, and are defined in this section.

5.7.1.1 ResetContext

The function ResetContext is defined in Table 72.

```
ResetContext () {
    // MB_x : macroblock index along width of image (in x direction)
    // StartOfTile_x : macroblock index of start of current tile along width of image
    // EndOfTile_x : macroblock index of end of current tile along width of image
    if ((MB_x == EndOfTile_x) || (((MB_x – StartOfTile_x) % 16) == 0))
        Return true
    Else
        Return false
}
```

Table 72. Pseudocode for function ResetContext

5.7.1.2 ResetTotals

The function ResetContext is defined in Table 73.

```
ResetTotals () {
    // MB_x : macroblock index along width of image (in x direction)
    // StartOfTile_x : macroblock index of start of current tile along width of image
    // EndOfTile_x : macroblock index of end of current tile along width of image
    if  (((MB_x – StartOfTile_x) % 16) == 0)
        Return true
    else
        Return false
}
```

Table 73. Pseudocode for function ResetTotals