Open Media Stack

Video Specification

V0.9

Open Media Stack ("OMS") Specification is an open community project with the goal of developing an open source royalty-free video compression solution. OMS Participants would like to receive diverse, wide, open, public, free and unrestricted input, suggestions, comments, contributions and any other feedback ("Feedback") on any materials in or with which these OMS Terms appear and on any other specifications, binary or source code software ("Software"), and plans and approaches proposed by OMS Participants ("OMS Materials").

Specifications, Software and other OMS Materials may be accompanied by or include or require agreement to additional terms that are imposed by Host (e.g., the OSI-approved CDDL license,GPLv.2.0, etc.) ("Other Terms"). Other Terms will govern the specifications, Software and other OMS Materials, and prevail to the extent of any conflicts with these OMS Terms. Subject to these OMS Terms, access to or use of the OpenMediaCommons.org website and any other website, controlled owned or operated by Host is governed by the web site's Terms of Use ("Terms of Use"). Nothing is intended to create a partnership, franchise, joint venture, agency or a fiduciary or employment relationship. All entities will comply with U.S. export, trade, security and privacy laws and applicable laws of any jurisdiction. You may not assign any rights or obligations under these OMS Terms. These OMS Terms will be governed by and construed in the English language and in accordance with the laws of the United States of America and the State of California, without giving effect to conflict of law principles of any jurisdiction or to the United Nations Convention on Contracts for the International Sale of Goods (1980), which is specifically excluded in its entirety. Any proceeding arising out of these OMS Terms must be exclusively brought in the appropriate courts in Santa Clara County, California, and You submit to the exclusive jurisdiction of those courts. The failure of Host to exercise or enforce any right or provision will not constitute a waiver of the right or provision. To the extent any provision of this Agreement is found by a court of competent jurisdiction to be invalid or unenforceable, it will be amended to achieve as nearly as possible the effect of its original form, and the remaining provisions will remain in full force. These OMS Terms constitute the entire agreement and supersede any prior oral or written communications, proposals, regarding the subject matter of these OMS Terms, except to the extent otherwise subsequently agreed in writing by Host.

"Host" means Sun Microsystems, Inc. and any of its assignees and any of the parties who control, own or operate the OpenMediaCommons.org website and any other website owned or operated by the foregoing parties, and the affiliates of the foregoing, in connection with OMS. "OMS Participants" means Host and parties who agree to and comply with these OMS Terms. "You" means you and any entity(ies) on whose behalf you provide Feedback.

EXCEPT AS MAY BE EXPRESSLY STATED HEREIN, HOST MAKES NO EXPRESS OR IMPLIED TRANSFERS, LICENSES, REPRESENTATIONS OR WARRANTIES, INCLUDING WITHOUT LIMITATION ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL ANY HOST BE LIABLE FOR ANY INDIRECT, INCIDENTAL, PUNITIVE, SPECIAL, EXEMPLARY OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT LIMITATION, ANY LOST PROFITS, LOST SALES, LOST REVENUE, LOSS OF GOODWILL, BUSINESS INTERRUPTION, OR LOSS OF PROGRAMS OR DATA, ARISING FROM OR RELATING TO THIS AGREEMENT. IN NO EVENT WILL HOST'S AGGREGATE LIABILITY ARISING FROM OR RELATING TO THIS AGREEMENT EXCEED U.S. $5 (FIVE U.S. DOLLARS). THE FOREGOING LIMITATIONS APPLY (A) EXCEPT TO THE EXTENT PROHIBITED BY LAW, (B) WHETHER THE LIABILITY, LOSS, OR DAMAGE ARISES IN CONTRACT (INCLUDING, WITHOUT LIMITATION, BREACH OF WARRANTY), TORT (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE, AND (C) EVEN IF A PARTY KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH LIABILITY, LOSSES OR DAMAGES AND EVEN IF ANY REMEDY FAILS OF ITS ESSENTIAL PURPOSE.

BY REVIEWING OR PROVIDING FEEDBACK ON OMS MATERIALS, YOU ACKNOWLEDGE YOU HAVE READ, UNDERSTOOD AND AGREE TO THESE OMS TERMS.

Please note that as participation and involvement progresses, the parties involved will likely be required to agree to additional terms and conditions.

Please forward any Feedback via Sun's OpenMediaCommons.org web site at http://www.openmediacommons.org/forums/ . Sun very much looks forward to your involvement. Thank you very much.

List of Authors

Mike DeMoney
Peter Farkas, PhD
Gerard Fernando, PhD
Rob Glidden
Kelly Kishore
Bo Liu
Cliff Reader, PhD
George Simion
Ryan Zhang

The authors wish to thank Nancy Snyder, Nissa Strottman and Jeanie Treichel for their help with this specification.

Table of Contents

1. Introduction

1.1. Open Media Stack Video Specifications

The Open Media Stack (OMS) Video Specification defines a video decoder and the associated bitstream syntax. It is intended for delivery, storage and playback of video streams.

The OMS Video Specification has been developed based on the principles of the Open Media Commons (OMC) (http://www.openmediacommons.org) initiative that Sun announced in August 2005.

1.2. Open Media Commons

OMC is an Open Source community project to develop royalty-free technologies and open solutions for digital content creation, duplication, distribution, and consumption.

OMC seeks to drive cross-industry growth and prosperity as well as promote both intellectual property protection and user privacy.

OMC's goals include:

- Develop open, royalty-free digital media solutions
- Promote the creation, distribution, and consumption of digital content
- Collaborate with like-minded Open Source communities
- Influence organizations to standardize royalty-free specifications

OMC's principles include:

- Innovation flourishes through openness - open standards, reference architectures and implementations
- Creators are users, and users are creators
- Respect for users' privacy is essential
- Code (both laws and technology) should encourage innovation

1.3. Profiles and Configurations

Profiles shall be defined as subsets of the tools in the OMS specification and must be identified by a profile number. The configuration shall be defined as a constrained set of values for parameters for a given profile which at a configuration point is a conformance point for the decoder, including the compliant bitstream.

The current version of the OMS video specification defines the Simple profile; however, it is not required that subsequent profiles defined for OMS video be backward compatible with Simple profile.

1.4. Notation

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119[RFC2119].

2. References, Definitions, Abbreviations

2.1. References

2.1.1. Normative References

1. **ITU-R BT.470-2**: “Television systems”
2. **ITU-R BT.601**: “Encoding parameters for digital television for studios”
3. **ITU-R BT.624**: “Characteristics of television systems”
4. **ITU-R BT.709**: “Parameter values for the HDTV standards for production and international programme exchange”
5. **ITU-T T.81**: “JPEG - Digital compression and coding of continuous-tone still images – requirements and guidelines”
6. **RFC2119**: Bradner, S ,"Key words for use in RFCs to Indicate Requirement Levels", 1997, http://www.ietf.org/rfc/rfc2119.txt
7. **Chen1977**: Wen-Hsiung Chen, C. Harrison Smith, and S. C. Fralick.”A fast computational algorithm for the DCT”, IEEE Trans. COM-25 No.9 P 1004-1009, Sept. 1977.
8. **MLIB**: MediaLib libraries from Sun Microsystems Inc. http://www.sun.com/processors/vis/mlibform.html

2.2. Definitions

Arithmetic coding: Method of lossless data compression.

Binarization: The process of converting the value of a variable to a sequence of binary values.

Block: A unit of data consisting of either 8x8 pixels or 4x4 pixels of luminance samples, and the corresponding 4x4 pixels and 2x2 pixels of chrominance samples.

Chrominance/Chroma: The parts of the video signal that contains the “color” information. Chrominance is often referred to with the symbols Cr and Cb (alternatively U and V). Chrominance data is obtained by performing a transform from the RGB (Red/Green/Blue) image planes.

Codec: Combination of the words **co**der and **dec**oder.

Coding tool: A discrete, separable, step, algorithm or functional element in a coding scheme.

Coding/Compression: A technique or scheme that transforms data from one representation to another, comprising a collection of rules that map elements of a first set onto the elements of a second set; intended to increase delivery or processing efficiency, typically by reducing the amount of data required to represent the data.

Color space: Model for defining colors in terms of a set of numbers. For video compression the color space consists of typically mapping RGB colors into the luminance and two color difference signals. These are referred to as YC_bC_r or as YUV.

Configuration: A constrained set of values for parameters that define a configuration of the decoder.

Decoder: The system that decodes input data stream and outputs decompressed video. The decoder is specified in the OMS video specification.

Encoder: A system that encodes input video and outputs a compressed data stream that is conformant with the specific profile and configuration of OMS video. The encoder itself is not part of the specification, except for the inverse DCT process that is used in the encoder (See Section 5.4.2).

Entropy coding: Method of lossless data compression.

Field: A field is the set of either the even or odd rows of a frame. There is no explicit support for field data in OMS.

Frame: A complete unit of media data which is nominally at a specific time instance. For video data, it consists of a triplet of arrays made up of luminance (often referred to as luma) and two chrominance (often referred to as chroma) data sets. For audio data it consists of a unit of sampled audio.

Frame rate. The rate at which frames are made available to the encoder or presented at the output of the decoder.

Image. A two-dimensional spatial array of pixel values at a given time instance.

Inter-frame coding: Method of temporal coding/compression that is performed on a sequence of frames, intended to take advantage of redundancy between adjacent or non-adjacent frame sequences.

Interlace scanning: Representation of video data in the form of a sequence of fields.

Intra-frame coding: Method of spatial coding/compression that is performed on a region of a frame using adjacent data in order to take advantage of spatial redundancy within a frame.

Luminance/Luma: The part of the video signal that contains the “brightness” information. Luminance is often referred to with the symbol Y. Luminance data is obtained by performing a transform from the RGB image planes.

Macroblock: A unit of data consisting of 8x8 pixels of luminance samples, and the corresponding 4x4 pixels of chrominance samples.

Motion compensated prediction: Method of applying inter-frame coding to predict data in the decoder using motion information.

Profile: A predefined set of coding tools associated with a profile number.

Progressive scanning: Representation of video data in the form of a sequence of frames. Progressive scanning is also referred to as non-interlace scanning.

Skipped: A block or macroblock for which no data needs to be coded except for a “flag” to indicate that this unit of data is skipped.

Spatial prediction: Method of applying intra-frame coding to predict data in the decoder using spatial information.

Subframe: A unit of data, that is a rectangular region of a frame, whose horizontal and vertical dimensions are integer multiples of the macroblock dimensions.

Subpixel interpolation: Method of interpolating the pixel values at fractional positions of an image.

Temporal prediction: Method of applying inter-frame coding to predict data within a frame.

Variable length coding: Method of lossless data compression (i.e., entropy coding) that assigns a variable number of bits to source data.

2.3. Abbreviations

DCT Discrete Cosine Transform
FSM Finite State Machine
IDCT Inverse Discrete Cosine Transform
IPR Intellectual Property Rights
LPS Least Probable Symbol
MPS Most Probable Symbol
OMC Open Media Commons
OMS Open Media Stack

3. Overview of the OMS Video Compression Specification

The goal of the OMS video compression specification is to provide a balance between compression performance and implementation complexity, while ensuring that implementations will be royalty-free. Digital video compression is over 40 years old and most of the fundamental inventions were either not patented or their patents have expired. With the notable exception of a few techniques, the innovations in video compression in recent years have been incremental and most of the large number of patents associated with contemporary standards have concerned marginal improvements in performance, specific implementation techniques, or specific coded bitstream syntax. The OMS video compression specification is based on legacy royalty-free technology and avoids the recently patented technology. While this approach sacrifices the performance gains of recently patented techniques, the OMS video compression specification includes innovations of its own that at least partially compensate for these losses. In addition, the OMS video compression specification takes advantage of contemporary semiconductor technology both to avoid the patented technology that was directed toward specific implementations, and to allow more sophisticated processing. Where appropriate, patenting the innovations associated with the OMS video compression specification will provide a defensive portfolio of IPR to help maintain OMS as a mainstream royalty-free compression solution.

The OMS video compression specification is targeted for implementation in software on generic 32-bit integer processors. It is assumed such processors include 32-bit integer multipliers, and are supported with sufficient memory to buffer multiple video frames. The compression specification does not specify unique techniques such as those designed for multiplier-free 16-bit implementation. The number of coding modes has been minimized. As a result, the OMS video compression specification is much simpler than contemporary compression specifications.

The OMS video architecture is based on hybrid motion compensated transform coding with entropy coding. This architecture was established in the late 1970s, and forms the foundation for all major compression solutions today. The encoder is illustrated in Figure 3.1. The video compression specification shall be described as a set of video coding tools. There are three major coding tools:

- Hybrid motion compensated coding
- Spatial transform and linear quantization coding
- Entropy coding

Each of these includes many small coding tools that collectively provide the functionality and performance of the compression solution. A brief introduction to the OMS video tool set follows after an overview of the data structures in the OMS video compression specification.

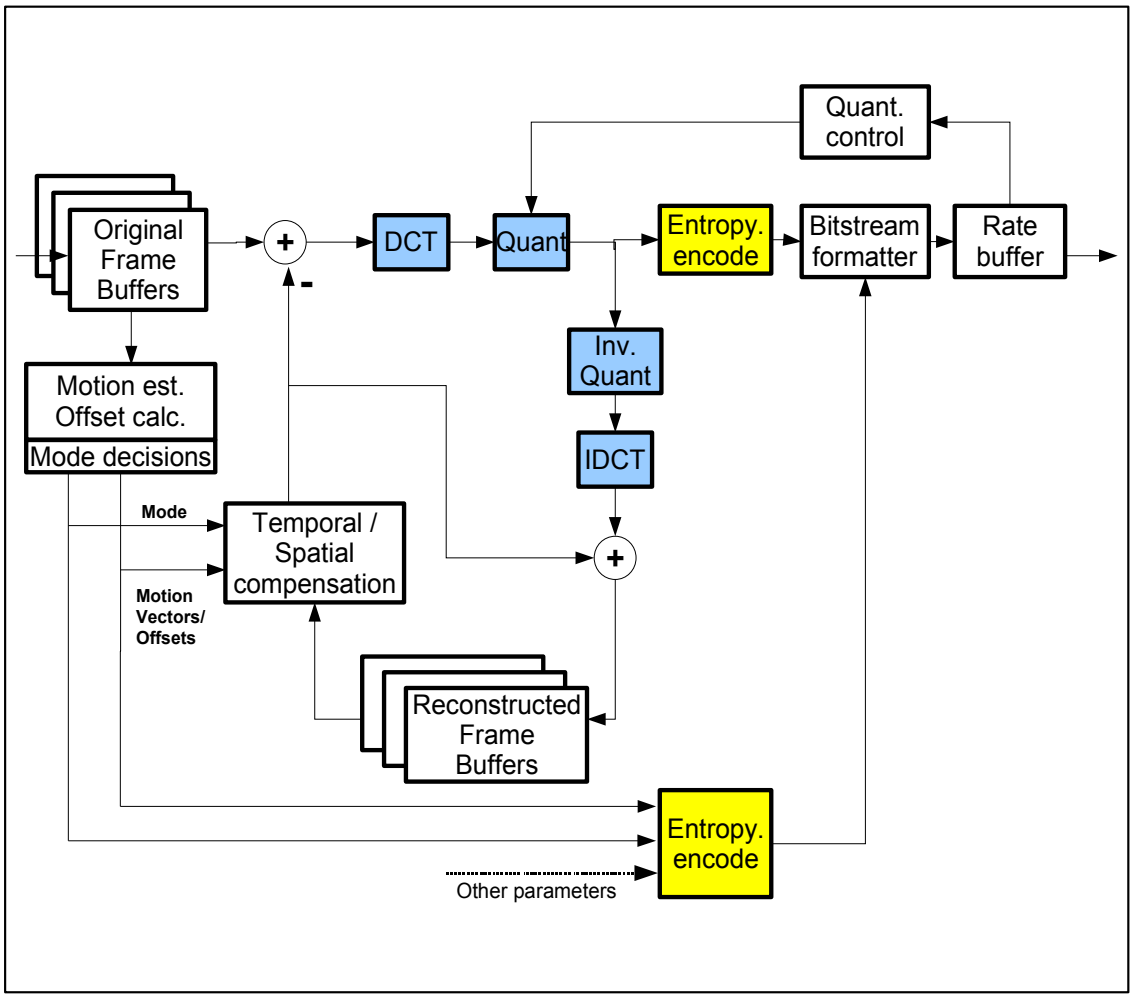

Figure 3.1 Architecture of the encoder

The decoder architecture is illustrated in Figure 3.2.

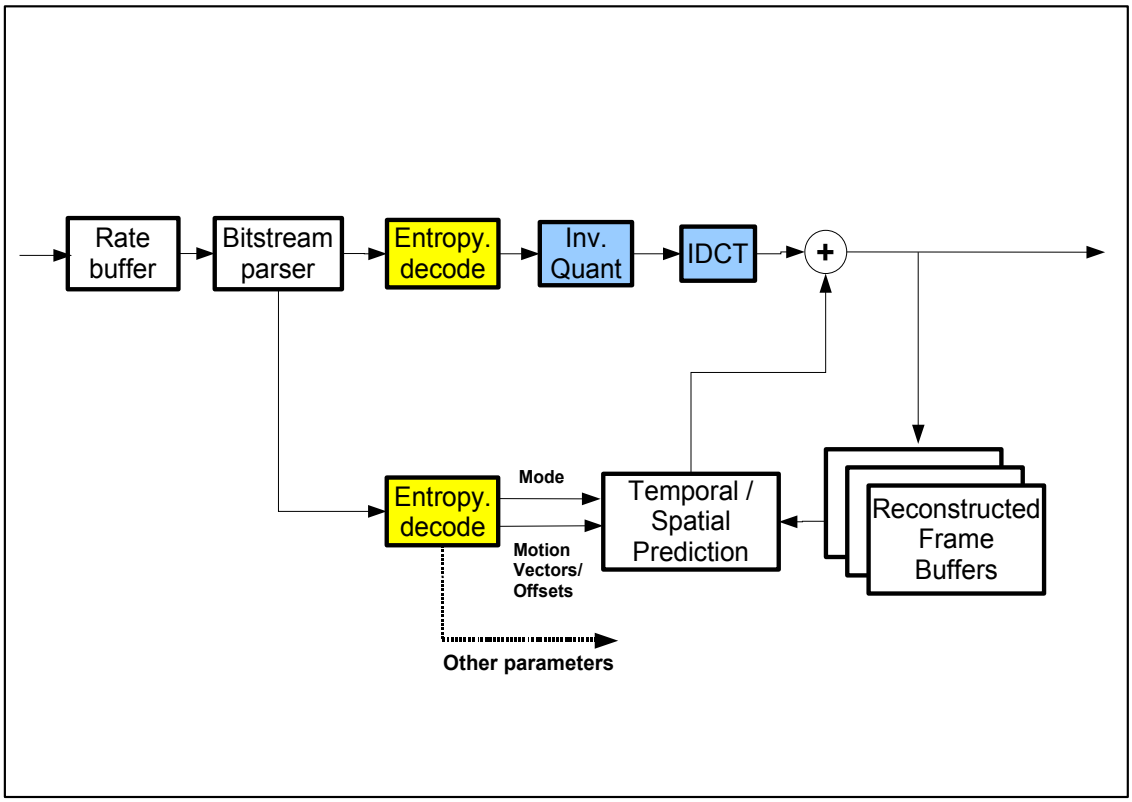

Figure 3.2 Architecture of the decoder

3.1. Data Structures

A hierarchy of data structures provides correspondence between the structure of raw video sequences and coded video sequences. The hierarchy comprises the following structures with associated functionality:

- Sequence header
  - A complete set of information required to decode and playback a video sequence
- Frame header
  - A set of information with associated time-stamp describing a spatial frame of displayable data at a specific time instant
- Subframe object
  - A rectangular subset of a frame containing prediction data needed to complete a displayable frame at the corresponding time instant
- Macroblock object
  - The smallest coded structure for a region of a subframe
- Block object
  - A component of a macroblock

These structures may be encapsulated in various ways to provide complete video presentations in different transmission and storage environments. Examples of transmission networks include synchronous time-division-multiplexed communication networks, dedicated broadcast networks, and asynchronous packet-switched networks. Encapsulation may also be made in suitable ways for storage applications.

Figure 3.3 illustrates a video sequence beginning with a sequence header. A succession of frames follows, each with a time stamp, and it should be noted these frames do not have to form a periodic time sequence. Repeat sequence headers may be inserted periodically that contain the same information as the sequence header to facilitate entering the sequence e.g., in broadcast applications.

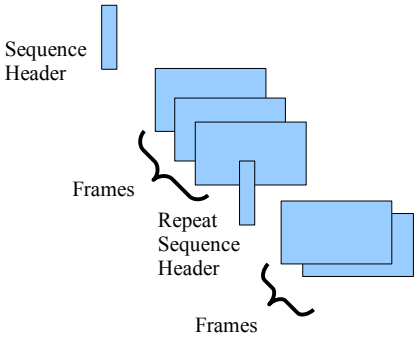

Figure 3.3: Video Sequence

Figure 3.4 shows a video frame, with a number of predicted subframes. Predicted subframes contain macroblocks with coded motion vector and/or prediction error data. The remainder of the frame, shown as white space, contains skipped predicted macroblocks. These regions can be reconstructed for display entirely from reference data.

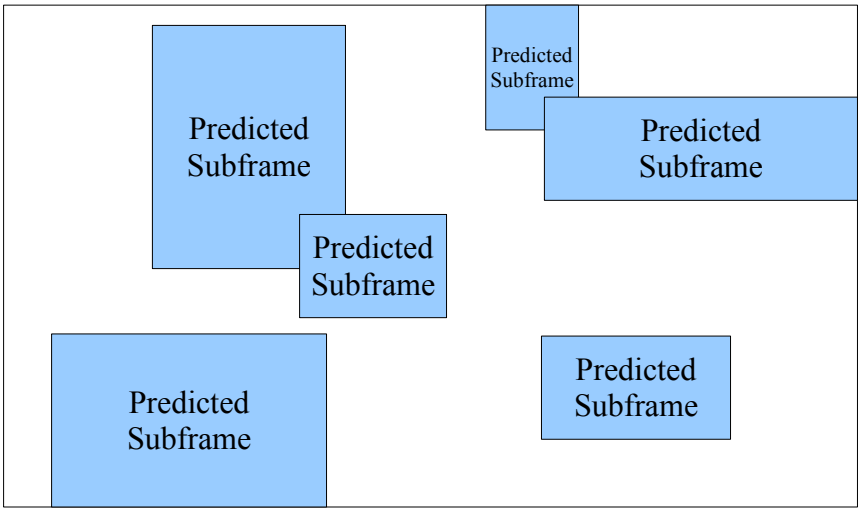

Figure 3.4. Video Frame and Subframes

Frames are addressed with a coordinate system, F(i,j) in which the origin is in the upper left corner. Positive address directions are right and down.

Predicted subframes must be rectangular regions whose dimensions shall be integer multiples of the macroblock dimensions. Predicted subframes overlap, and thus need not be contiguous. The location of the subframes shall be defined by the coordinates of their upper-left corner. The width of the subframe shall be specified, and its height shall be determined by the number of coded macroblocks

Macroblocks are addressed in raster sequence within the subframe as illustrated in Figure 3.5. Skipped macroblocks are illustrated in a different color (yellow), with the skipped address shown in a dashed line.

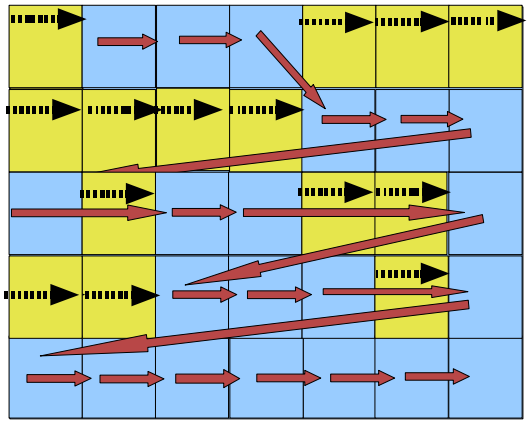

Figure 3.5. Subframe Macroblock Addressing

Figure 3.6 illustrates the pre- and post- processing transformation of the macroblock and the filtering operations that convert red-green-blue pixels to luminance-chrominance format. The chrominance must be filtered and subsampled by 2:1 in each dimension.

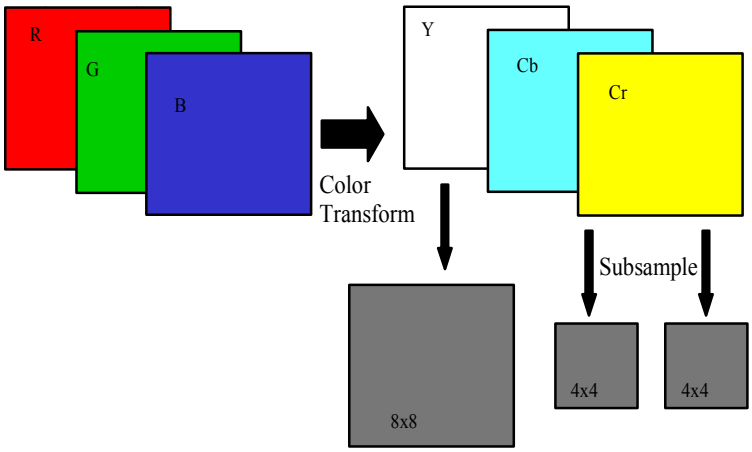

Figure 3.6. Macroblock

OMS Video supports a single, square macroblock size of 8x8 pixels. Luminance blocks may be 8x8 or 4x4 pixels. Chrominance blocks may be 4x4 or 2x2 pixels. The macroblock shall consist of either (a) one 8x8 luminance block and a pair of two 4x4 chrominance blocks or (b) four 4x4 luminance blocks and four 2x2 chrominance blocks each for Cr and Cb. Figure 3.7 illustrates the arrangement and ordering of luminance and chrominance blocks of 4x4 pixels and 2x2 pixels respectively within a macroblock.

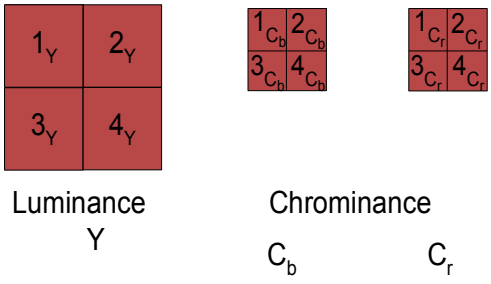

Figure 3.7. Block Order

Similarly in Figure 3.8, when the luminance block size is 8x8, the size of the luminance block is equivalent to the size of the luminance macroblock, and the size of the chrominance blocks is 4x4.

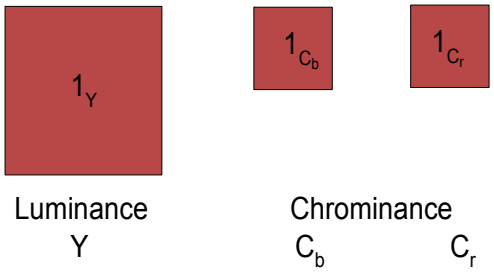

Figure 3.8. Block Order

3.2. Hybrid Motion Compensated Coding

The OMS video compression solution provides hybrid motion compensated coding with the following coding tools:

- Motion compensated prediction
  - Region block-matching
    - ¼-pixel precision
      - Bicubic spline luminance subpixel interpolation
      - Bilinear chrominance subpixel interpolation
- Spatial compensated prediction
  - Neighbor block-matching
  - DC

Motion estimation and motion prediction are performed with NxN blocks, where N=4 and N=8 (subject to verification testing).

Sequences contain two types of coded frame data:

- Spatial-predicted frames – frames that are spatial-predicted only
- Temporal predicted frames – frames that may be temporal-predicted or spatial-predicted

Spatial-predicted frames contain subframes with one type of coded macroblock data:

- Spatial predicted data
  - Data coded by spatial prediction at a time instant without reference to data at other time instants

Temporal-predicted frames contain subframes with two types of coded macroblock data:

- Spatial predicted data
  - Data coded by spatial prediction at a time instant without reference to data at other time instants
- Temporal predicted data
  - Data predicted from the previous frame for the current display frame
  - Prediction may be “perfect” and the data will be “skipped”

Each luminance block in a macroblock is predicted with a separate motion vector. Chrominance

blocks inherit the motion vectors of the luminance blocks and each quadrant of a chrominance block is predicted separately. Motion vectors are coded differentially within macroblocks in the order shown in Figures 3.7 and 3.8, and are coded differentially between macroblocks by predicting the first motion vector from the last motion vector in the previous macroblock.

Coded temporal predicted blocks may contain coded differential motion vector data, and may contain coded prediction error data, also called residual data. If a temporal predicted block contains no coded differential motion vector data and coded prediction error data, the block is skipped. If all blocks in a macroblock are skipped, the macroblock is skipped. Due to motion vector prediction, no coded motion vector data does not necessarily mean the motion vector is zero.

3.3. Spatial Transformation and Linear Quantization Coding

OMS video coding removes spatial redundancy with an integer transform that approximates the traditional Discrete Cosine Transform (the term DCT shall be used to refer to this integer approximation) and scalar quantization. To avoid the problem of mismatch between inverse transforms at the encoder and decoder, the OMS video codec normatively specifies the forward and inverse transforms in this standard. The spatial transformation, linear quantization, and their respective inverse operations shall be computed using 32-bit integer precision arithmetic.

As illustrated in Figures 3.7 and 3.8, OMS video specifies various block sizes. The transformation of the block shall be performed using the DCT with the same corresponding size. For example, if the luminance block size is 8x8, then the 8x8 DCT would be selected for the transformation of the luminance blocks. With respect to the 8x8 luminance block, the chrominance block size would be 4x4, so the 4x4 DCT would be selected for the transformation of the chrominance blocks.

To provide both sufficient precision and dynamic range when quantizing the transform coefficients, OMS video provides a method for setting the transmitted quantization step size in terms of a user defined quantization parameter, QP. The quantization step size shall be determined via a table lookup indexed by QP.

3.4. Entropy Coding

OMS video uses the context adaptive binary arithmetic coding specified by JPEG Annex-D as an option. Three types of data are arithmetic coded within the subframe:

- Subframe and Macroblock header data
- Prediction modes and prediction data: Motion Vectors and Spatial Offsets
- Residual coefficient data

Basic Algorithm

Arithmetic coding is based on dividing the probability interval {0,1} into sub-intervals determined by probability estimates of a variable to be coded. In binary arithmetic coding there are only two values, which are called the MPS (most probable symbol) and LPS (least probable symbol). In context-adaptive arithmetic coding, the probability estimates are conditional, and are dynamically updated based on the symbols already coded. The current probability estimate is maintained in an entity known as a context and a separate context (or set of contexts) is maintained for each variable being coded.

In the basic algorithm, when the next symbol to be coded has the MPS value, no data is coded into the bitstream and the probability interval is redefined to span the sub-interval of the MPS. The next symbol is then coded. When the symbol to be coded has the LPS value, the size of the MPS sub-interval is coded into the bitstream. The basic algorithm has to be augmented by procedures that

provide a practical implementation. As successive MPS symbols are coded, the probability interval becomes smaller and smaller, requiring more and more bits to represent it. After a certain number of MPS events, the representation will equal the precision of the implementation (e.g., 16 bits), necessitating renormalization. Coding an LPS value also triggers renormalization and may trigger conditional exchange. Conditional exchange occurs because practical methods for subdividing the probability interval are imprecise and the resultant LPS sub-interval may be larger than the MPS sub-interval. The context also maintains which symbol is assigned to the MPS. Renormalization and conditional exchange will be described in Section 5.5.2.

Context adaptive binary arithmetic coding tracks the probability of the symbol values being coded with multiple mechanisms. The first mechanism is the most basic - it tracks the probability of an LPS for one particular binary variable. This probability is tracked by a finite state machine (FSM) that alters its prediction based on the history of previous values of that variable. The state of the probability finite state machine and its corresponding probability estimate for the next value of this particular binary variable is maintained in the context. The encoder and decoder must agree on a common definition of the probability estimator finite state machine.

Because there are multiple variables, each possibly having different statistical behavior, the arithmetic coder allows for multiple contexts (e.g., independent transitions of probability FSM's). As each different variable is coded, the arithmetic coder utilizes and updates the context specific to that variable. Thus the arithmetic coder consists of a core coding engine that for each binary decision being coded uses one of a multiplicity of contexts, each context tracking the probability estimate for a specific variable. The encoder and decoder must agree on the assignment of contexts to variables to be coded. For example, there is a **macroblockSkipFlag** for every macroblock in a subframe. A separate context is maintained for this variable that is initialized at the beginning of each subframe and then adapts from macroblock to macroblock throughout the subframe.

Multivalued variables, e.g., enumerations, modes and integer values, are binarized by a binarization algorithm appropriate for the statistical properties of that variable. Each bit of the binarized variable is then arithmetically coded. Each binarization algorithm can be thought of as a binary tree where each leaf of the tree represents a particular symbol in the alphabet of the variable. The two edges leaving a node are labeled 0 and 1. The path of edges from the root of the tree to a particular leaf is the binarization of the symbol associated with that leaf. The binarization tree is generally constructed so that the most frequent symbols have the shortest paths, so the binarization trees represent a variable length coding of the variable.

The statistics of the binary decision represented by the outgoing two edges of the different nodes vary, hence differing nodes of the binarization tree for a single variable may have a different context associated with it to track the probability of that particular node decision. Thus non-binary values may require multiple contexts to track the probability of the individual decisions in the binarization tree. The encoder and decoder must agree on common binarization algorithms for all variables coded and the assignment of contexts to the nodes of the binarization tree. For example, there is a **differentialSimpleHorizontalMotionVector** for every coded macroblock in a subframe. The binarization algorithm for this syntax element utilizes different contexts for the various decisions that make up the binarization of this value. All of these contexts are initialized at the beginning of each subframe and then adapts from macroblock to macroblock.

The binarization algorithms and how they associate contexts to binary decisions coded within the algorithm are described in sections 5.5.3 (encoding) and 6.2.4 (decoding). Sections 5.5.4 through 5.5.9 (encoding) and sections 6.2.5 through 6.2.10 (decoding) describe for each variable specified in one of the syntax diagrams in Section 4.4 the binarization algorithm and parameters to pass to that binarization algorithm when coding that variable. The parameters include the context vector, the length of the context vector, and any parameters specfic to the particular binarization algorithm.

3.5. Rate Control

The initial version of the OMS video compression specification operates in variable bitrate environments. Later versions may operate in fixed bitrate environments, or environments in which a maximum bitrate is specified, below which the bitrate may vary.

4. Specification of the Coded Video Bitstream

4.1. Specification of Syntax, Functions, and Descriptors

The following descriptors define data types used in the bitstream:

bslbf – Bit String left bit first. Bit strings are written as 1's and 0's e.g. '1011 0111' or for convenience in hexadecimal.

uimsbf – Unsigned integer, most significant bit first

guimsbf – Unsigned integer, most significant bit first that is scaled according to the granularity parameter.

flag() – Context adaptive arithmetic coded single bit syntax element

unary() – Context adaptive arithmetic coded syntax element with unary binarization

fixed(nBit) – Context adaptive arithmetic coded syntax element with fixed bit width binariation that produces a nBit binarization

signedFixed(nBit) – Context adaptive arithmetic coded syntax element with signed fixed bit width binarization that produces a nBit binarization

signedUnary() – Context adaptive arithmetic coded syntax element with signed unary binarization

trUnary(maxVal) – Context adaptive arithmetic coded syntax element with truncated unary binarization

expGolomb(k) – Context adaptive arithmetic coded syntax element with exponential-Golomb binarization

signedExpGolomb(k) – Context adaptive arithmetic coded syntax element with an exponential-Golomb binarization that has been extended to handle signed values

Where:

nbit is the number of binary bits allocated in the binarization to represent bits of the value to be coded

maxVal is the maximum integer that is valid to be encoded by the binarization.

k is the order of the code

4.2. Method of Specifying Syntax in Tabular Form

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

Syntax	*Mnemonic*
/* An "if(..) … else if (...)" structure specifies a test of whether a condition is true, and if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of a sequence of else if statements. */	
if (condition-1)	
statements-1	
else if (condition-2)	
statements-2	
....	
else	
statements-N	
/* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. */	
for (initial statements; condition; subsequent statements)	
primary statements	

4.3. Functions, Data Structures and Temporary Variables

This section declares all functions, and defines all data structures and variables used in this specification.

4.3.1. Syntax Functions

4.3.1.1. Sequence Layer

sequenceHeader()

sequenceHeaderExtension()

userData()

4.3.1.2. Frame Layer

frameHeader()

frameHeaderExtension()

userData()

4.3.1.3. Subframe Layer

subframe()

subframeHeader()

initializeArithmeticCoder()

terminateArithmeticCoder()

downloadContexts()

deltaSubframeHorizontalAddressContext()

deltaSubframeVerticalAddressContext()

subframeWidthContext()

macroblockSkipFlagContext()

lastMacroblockFlagContext()

quantizerParameterUpdateContext()

blockSize4x4Context()

blockSkipFlagContext()

spatialPredictionModeContext()

spatialPredictiOffsetContext()

differentialSimpleHorizontalMotionVectorContext()

differentialSimpleVerticalMotionVectorContext()

eobFlagContext()

predictedCoefficientErrorContext()

subframeHeaderExtension()

userData()

4.3.1.4. Macroblock Layer

macroblock()

spatialPrediction(blockSize)

simpleMotionPrediction(blockSize)

4.3.1.5. Block Layer

predictedBlock(blockSize)

4.3.2. Data Structures

horizontalSize Sec. 4.4.1 The width of a frame

verticalSize Sec. 4.4.1 The height of a frame

deltaSubframeHorizontalAddress Sec. 4.6.1 The offset of the current subframe horizontal address from the previous subframe horizontal address. Subframes are ordered in a raster sequence.

deltaSubframeVerticalAddress Sec. 4.6.1 The offset of the current subframe vertical address from the previous subframe vertical address.

subframeWidth Sec. 4.6.1 The width of a subframe

See Figure 4.1 for illustration of subframe and frame data structures

numberSubBlocks Sec. 4.7.1 The number of luminance sub blocks in a macroblock

blockSize Sec. 4.7.1 (See Fig 3.7 and Figure 3.8 for details on block ordering)

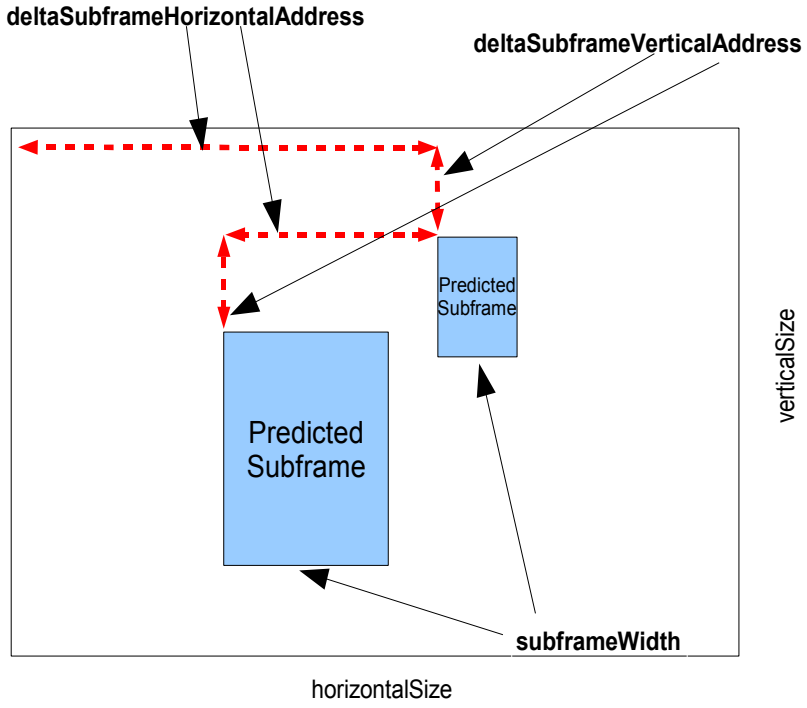

Figure 4.1: Subframe addressing

4.3.3. Temporary Variables

Syntax & Semantics

sizeGranularity Sec. 4.4.2 The granularity with which the horizontal and vertical size of the frame is specified

bitRateGranularity Sec. 4.4.2 The granularity with which the bitrate is specified

rateControlBufferGranularity Sec. 4.4.2 The granularity with which the rate control buffer size is specified

subframeGranularity Sec. 4.6.2 The granularity with which the delta horizontal size, delta vertical size, and width of the subframe is specified

spatialPredicted Sec. 4.6.2 A subframe mode

temporalPredicted Sec. 4.6.2 A subframe mode

spatialPredictionVertical Sec. 4.7.1 A spatial prediction mode

spatialPredictionHorizontal Sec. 4.7.1 A spatial prediction mode

4.4. Sequence Layer

4.4.1. Syntax

Syntax	***Mnemonic***
sequenceHeader(){	
sequenceSerialNumber	uimsbf(12)
granularity	uimsbf(4)
profileNumber	uimsbf(6)
versionNumber	uimsbf(10)
horizontalSize	guimsbf(6)
verticalSize	guimsbf(6)
pixelAspectRatio	uimsbf(4)
colorSpace	uimsbf(4)
sourceFrameRate	uimsbf(4)
bitRate	guimsbf(18)
rateControlBufferSize	guimsbf(18)
sequenceHeaderExtensionFlag	bslbf(1)
userDataFlag	bslbf(1)
zeroFill	bslbf(2)
If (sequenceHeaderExtensionFlag == 1)	

Syntax	*Mnemonic*
sequenceHeaderExtension()	
If (userDataFlag == 1)	
userData()	
}	

Syntax	*Mnemonic*
sequenceHeaderExtension(){	
}	

Syntax	*Mnemonic*
userData(){	
userDataLength	uimsbf(16)
For (i=0; i<userDataLength; i++)	
userData[i]	uimsbf(8)
}	

4.4.2. Semantics

sequenceSerialNumber – Number that may be used to uniquely identify each media stream.

granularity – A parameter that scales the value of the following syntax elements:

- **horizontalSize**
- **verticalSize**
- **bitrate**
- **rateControlBufferSize**
- **deltaSubframeHorizontalAddress**; See Section 4.6.1
- **deltaSubframeVerticalAddress**; See Section 4.6.1
- **subframeWidth**; See Section 4.6.1

profileNumber – 6-bit integer that defines the profile version number. In Version 1 of this specification shall be set to “000001”

versionNumber – 10-bit integer that defines the version number.

horizontalSize – Width of the decoded video frame in units of sizeGranularity*8 pixels, where sizeGranularity is defined in the following table.

verticalSize – Height of the decoded video frame in units of sizeGranularity*8 pixels, where sizeGranularity is defined in the following table.

granularity	sizeGranularity
0000	forbidden
0001	1
0010	2
0011	4
0100	8
0101	16
0110-1111	reserved

pixelAspectRatio – Four bit integer defined in the following table:

pixelAspectRatio	height/width	Comments
0000	forbidden	
0001	1.00000	Square pixel aspect ratio
0010	0.9157	ITU-R BT.601, 625line
0011	1.0950	ITU-R BT.601, 525line
0100-1111	reserved	

colorSpace – Four bit integer defined in the following table:

value	colorSpase
0000	forbidden
0001	Recommendation ITU-R BT.709
0010	Recommendation ITU-R BT.470-2 System M
0011	Recommendation ITU-R BT.470-2 System B, G
0100	Unspecified Video
0101-1111	reserved

sourceFrameRate – Four bit integer defined in the following table:

frameRate	per second
0000	forbidden
0001	23.976*
0010	24

frameRate	per second
0011	25
0100	29.97
0101	30
0110	50
0111	59.94*
1000	60
1001	72
1010	75
1011	Variable**
1100-1111	reserved

* : Refer to [ITU-R BT.624] on method of deriving exact values for these entries
** : Determined by frame time stamps

bitRate – This is an integer specifying the bitrate of the stream in increments of bitRateGranularity bits/sec up to 100Mbits/sec. Value zero is forbidden. $3FFFF_H$ identifies variable bit rate.

granularity	bitRateGranularity
0000	forbidden
0001	100
0010	500
0011	1000
0100	5000
0101	10000
0110-1111	reserved

rateControlBufferSize – This is an integer specifying the buffer size required for rate control. The buffer size is defined in increments of rateControlBufferGranularity bytes up to 25Mbytes. rateControlBufferGranularity is defined in the table below.

granularity	rateControlBufferGranularity
0000	forbidden
0001	100
0010	500
0011	1000
0100	5000
0101	10000
0110-1111	reserved

sequenceHeaderExtensionFlag – This flag must be set to zero in the current version of the specification.

userDataFlag – If this flag is set to one, user data follows in the bitstream. Any such user data must comprise an integral number of bytes. If this flag is set to zero, no user data follows.

zeroFill – This field is set to zero to enable byte alignment of the sequence header.

4.5. Frame Layer

4.5.1. Syntax

Syntax	*Mnemonic*
frameHeader(){	
timeStamp	uimsbf(37)
frameCodingType	uimsbf(3)
quantizerParameter	bslbf(6)
frameHeaderExtensionFlag	bslbf(1)
userDataFlag	bslbf(1)
If (frameHeaderExtensionFlag == 1)	
frameHeaderExtension()	
If (userDataFlag == 1)	
userData()	
}	

4.5.2. Semantics

timeStamp – The timeStamp is defined at a precision of 90KHz

frameCodingType – Identifies the frame coding type as defined in the table below:

Code	frameCodingType
000	forbidden
001	spatialPredicted
010	temporalPredicted
011-111	reserved

quantizerParameter – An unsigned 6 bit integer value used to scale the reconstruction level of the DCT coefficient levels. This is a non-linear function, and it is defined in Section 5.4.2.

frameHeaderExtensionFlag – This flag must be set to zero in the current version of the specification.

userDataFlag – If this flag is set to one, user data follows in the bitstream. Any such user data must comprise an integral number of bytes. If this flag is set to zero, no user data follows.

4.6. Subframe Layer

4.6.1. Syntax

Syntax	*Mnemonic*
subframe(){	
subframeHeader()	
initializeArithmeticCoder()	
deltaSubframeHorizontalAddress	signedUnary()
deltaSubframeVerticalAddress	unary()
subframeWidth	unary()
do{	
do{	
macroblockSkipFlag	flag()
} while (macroblockSkipFlag)	
macroblock()	
lastMacroblockFlag	flag()
} while (!lastMacroblockFlag)	
terminateArithmeticCoder()	
}	

Syntax	*Mnemonic*
subframeHeader(){	
subframeType	uismbf(3)
downloadContextsFlag	bslbf(1)
subframeHeaderExtensionFlag	bslbf(1)
userDataFlag	bslbf(1)
zeroFill	bslbf(2)
if (downloadContextsFlag == 1)	
downloadContexts()	
If (subframeHeaderExtensionFlag == 1)	

Syntax	*Mnemonic*
subframeHeaderExtension()	
If (userDataFlag == 1)	
userData()	
}	

Syntax	*Mnemonic*
downloadContexts(){	
subframeContextsFlag	bslbf(1)
if (subframeContextsFlag){	
deltaSubframeHorizontalAddressContext()	
deltaSubframeVerticalAddressContext()	
subframeWidthContext()	
macroblockSkipFlagContext()	
lastMacroblockFlagContext()	
}	
macroblockContextsFlag	bslbf(1)
if (macroblockContextsFlag){	
quantizerParameterUpdateContext()	
blockSize4x4Context()	
blockSkipFlagContext()	
}	
spatialPredictionContextsFlag	bslbf(1)
if (intraPrediction ContextsFlag){	
spatialPredictionModeContext()	
spatialPredictionOffsetContext()	
}	
simpleMotionPredictionContextsFlag	bslbf(1)
if (simpleMotionPredictionContextsFlag){	
differentialSimpleHorizontalMotionVectorContext()	
differentialSimpleVerticalMotionVectorContext()	
}	
blockContextsFlag	bslbf(1)
if (blockContextsFlag){	
eobFlagContext()	
predictedCoefficientErrorContext()	

Syntax	*Mnemonic*
}	
}	

Syntax	*Mnemonic*
deltaSubframeHorizontalAddressContext(){	
for (i = 0; i < 4; i++) {	
deltaSubframeHorizontalAddressContext[i].mps	bslbf(1)
deltaSubframeHorizontalAddressContext[i].state	uimsbf(5)
}	
}	
deltaSubframeVerticalAddressContext(){	
for (i = 0; i < 4; i++) {	
deltaSubframeVerticalAddressContext[i].mps	bslbf(1)
deltaSubframeVerticalAddressContext[i].state	uimsbf(5)
}	
}	
subframeWidthContext(){	
for (i = 0; i < 4; i++) {	
subframeWidthContext[i].mps	bslbf(1)
subframeWidthContext[i].state	uimsbf(5)
}	
}	
macroblockSkipFlagContext(){	
macroblockSkipFlagContext[0].mps	bslbf(1)
macroblockSkipFlagContext[0].state	uimsbf(5)
}	
lastMacroblockFlagContext(){	
lastMacroblockFlagContext[0].mps	bslbf(1)
lastMacroblockFlagContext[0].state	uimsbf(5)
}	
quantizerParameterUpdateContext(){	
for (i = 0; i < 4; i++) {	
quantizerParameterUpdateContext[i].mps	bslbf(1)
quantizerParameterUpdateContext[i].state	uimsbf(5)
}	
}	
blockSize4x4Context(){	
blockSize4x4Context[0].mps	bslbf(1)
blockSize4x4Context[0].state	uimsbf(5)
}	
blockSkipFlagContext(){	
blockSkipFlagContext[0].mps	bslbf(1)
blockSkipFlagContext[0].state	uimsbf(5)

Syntax	*Mnemonic*
}	
simpleMotionPredictionFlag(){	
simpleMotionPredictionFlag[0].mps	bslbf(1)
simpleMotionPredictionFlag[0].state	uimsbf(5)
}	
spatialPredictionModeContext(){	
for (i = 0; i < 2; i++) {	
spatialPredictionModeContext[i].mps	bslbf(1)
spatialPredictionModeContext[i].state	uimsbf(5)
}	
}	
spatialPredictionSpatialOffsetContext(){	
for (i = 0; i < 7; i++) {	
spatialPredictionSpatialOffsetContext[i].mps	bslbf(1)
spatialPredictionSpatialOffsetContext[i].state	uimsbf(5)
}	
}	
differentialSimpleHorizontalMotionVectorContext(){	
for (i = 0; i < 6; i++) {	
differentialSimpleHorizontalMotionVectorContext[i].mps	bslbf(1)
differentialSimpleHorizontalMotionVectorContext[i].state	uimsbf(5)
}	
}	
differentialSimpleVerticalMotionVectorContext(){	
for (i = 0; i < 6; i++) {	
differentialSimpleVerticalMotionVectorContext[i].mps	bslbf(1)
differentialSimpleVerticalMotionVectorContext[i].state	uimsbf(5)
}	
}	
eobFlagContext(){	
eobFlagContext[0].mps	bslbf(1)
eobFlagContext[0].state	uimsbf(5)
}	
predictedCoefficientErrorContext(){	
for (i = 0; i < 33*6; i++) {	
luma8x8Context[i].mps	bslbf(1)
luma8x8Context[i].state	uimsbf(5)
}	
for (i = 0; i < 9*6; i++) {	
luma4x4Context[i].mps	bslbf(1)
luma4x4Context[i].state	uimsbf(5)
}	
for (i = 0; i < 9*6; i++) {	

Syntax	*Mnemonic*
chroma4x4Context[i].mps	bslbf(1)
chroma4x4Context[i].state	uimsbf(5)
}	
for (i = 0; i < 4; i++) {	
chroma2x2Context[i].mps	bslbf(1)
chroma2x2Context[i].state	uimsbf(5)
}	
}	

Syntax	*Mnemonic*
subframeHeaderExtension(){	
}	

4.6.2. Semantics

deltaSubframeHorizontalAddress – Horizontal position of the upper-left corner of the subframe in multiples of subframeGranularity*8 pixels from the previous subframe. The position of the first subframe in a frame is coded relative to the upper-left corner of the frame. See section 4.3 for definition of positive direction within a frame.

deltaSubframeVerticalAddress - Vertical position of the upper-left corner of the subframe in multiples of subframeGranularity*8 pixels from the previous subframe. The position of the first subframe in a frame is coded relative to the upper-left corner of the frame. See section 4.3 for definition of positive direction within a frame.

subframeWidth – width of subframe in multiples of subframeGranularity*8 pixels.

granularity	subframeGranularity
0000	forbidden
0001	1
0010	2
0011	4
0100	8
0101	16
0110-1111	reserved

subframeHeight shall be determined by the **subframeWidth** and the number of macroblocks in the subframe.

macroblockSkipFlag – if macroblockSkipFlag = 1, the current macroblock is skipped (not coded); if macroblockSkipFlag = 0, the current macroblock is coded.

lastMacroblockFlag – if lastMacroblockFlag = 1, the current subframe is terminated; if lastMacroblockFlag = 0, the current subframe contains more coded macroblocks.

subframeType – Identifies the subframe coding type as defined in Table below:

Code	subframeCodingType
000	forbidden
001	spatialPredicted macroblocks
010	temporalPredicted macroblocks
011-111	reserved

subframes in spatial predicted frames shall be spatial prediction only, hence, subFrameType = 001.

downloadContextsFlag – if downloadContextsFlag = 1, contexts for arithmetic coding shall be downloaded and overwrite the existing contexts; if downloadContextsFlag = 0, the existing contexts shall be used. Default contexts are defined by the standard and shall be used unless overwritten.

subframeHeaderExtensionFlag – This flag must be set to zero in the current version of the specification.

userDataFlag – if this flag is set to one, user data follows in the bitstream. Any such user data must comprise an integral number of bytes. If this flag is set to zero, no user data follows.

subframeContextsFlag – if subframeContextsFlag = 0, subframeContexts are not downloaded; – if subframeContextsFlag = 1, subframeContexts are downloaded

macroblockContextsFlag – if macroblockContextsFlag = 0, macroblockContexts are not downloaded; – if macroblockContextsFlag = 1, macroblockContexts are downloaded

spatialPredictionContextsFlag – if spatialPredictionContextsFlag = 0, spatialPredictionContexts are not downloaded; – if spatialPredictionContextsFlag = 1, spatialPredictionContexts are downloaded

simpleMotionPredictionContextsFlag – if simpleMotionPredictionContextsFlag = 0, simpleMotionPredictionContexts are not downloaded; – if simpleMotionPredictionContextsFlag = 1, simpleMotionPredictionContext are downloaded

blockContextsFlag – if blockContextsFlag = 0, blockContexts are not downloaded; – if blockContextsFlag = 1, blockContexts are downloaded

deltaSubframeHorizontalAddressContext[i].mps – assigns the initial value of the MPS symbol

deltaSubframeHorizontalAddressContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

deltaSubframeVerticalAddressContext[i].mps – assigns the initial value of the MPS symbol

deltaSubframeVerticalAddressContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

subframeWidthContext[i].mps – assigns the initial value of the MPS symbol

subframeWidthContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

macroblockSkipFlagContext[0].mps – assigns the initial value of the MPS symbol

macroblockSkipFlagContext[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

lastMacroblockFlagContext[0].mps – assigns the initial value of the MPS symbol

lastMacroblockFlagContext[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

quantizerParameterUpdateContext[i].mps – assigns the initial value of the MPS symbol

quantizerParameterUpdateContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

blockSize4x4Context[0].mps – assigns the initial value of the MPS symbol

blockSize4x4Context[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

blockSkipFlagContext[0].mps – assigns the initial value of the MPS symbol

blockSkipFlagContext[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

simpleMotionPredictionFlag[0].mps – assigns the initial value of the MPS symbol

simpleMotionPredictionFlag[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

spatialPredictionModeContext[i].mps – assigns the initial value of the MPS symbol

spatialPredictionModeContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

spatialPredictionSpatialOffsetContext[i].mps – assigns the initial value of the MPS symbol

spatialPredictionSpatialOffsetContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

differentialSimpleHorizontalMotionVectorContext[i].mps – assigns the initial value of the MPS symbol

differentialSimpleHorizontalMotionVectorContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

differentialSimpleVerticalMotionVectorContext[i].mps – assigns the initial value of the MPS symbol

differentialSimpleVerticalMotionVectorContext[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

eobFlagContext[0].mps – assigns the initial value of the MPS symbol

eobFlagContext[0].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

luma8x8Context[i].mps – assigns the initial value of the MPS symbol

luma8x8Context[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

luma4x4Context[i].mps – assigns the initial value of the MPS symbol

luma4x4Context[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

chroma4x4Context[i].mps – assigns the initial value of the MPS symbol

chroma4x4Context[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

chroma2x2Context[i].mps – assigns the initial value of the MPS symbol

chroma2x2Context[i].state – a pointer into the fsm table (Annex-D) which defines the initial state for a variable

4.7. Macroblock Layer

4.7.1. Syntax

Syntax	***Mnemonic***
macroblock(){	
quantizerParameterUpdate	signedUnary()
blockSize4x4	flag()
if(blockSize4x4) {	
numberSubBlocks=4	
blockSize=4	
} else {	
numberSubBlocks=1	
blockSize=8	
}	
if(subframeType=spatialPredicted) {	
for (i=0; i<numberSubBlocks; i++){	
spatialPrediction(blockSize)	
}	

Syntax	*Mnemonic*
} else {	
for (i=0; i<numberSubBlocks; i++){	
blockSkipFlag	flag()
if (!blockSkipFlag){	
if (**simpleMotionPredictionFlag**{	flag()
simpleMotionPrediction(blockSize)	
} else {	
spatialPrediction(blockSize)	
}	
}	
}	
}	

Syntax	*Mnemonic*
spatialPrediction(blockSize){	
spatialPredictionMode	fixed(2)
if(spatialPredictionVertical || spatialPredictionHorizontal){	
spatialPredictionSpatialOffset	signedFixed(7)
predictedBlock(blockSize) // Luma	
predictedBlock(blockSize / 2) // Cb	
predictedBlock(blockSize / 2) // Cr	
}	
}	

Syntax	*Mnemonic*
simpleMotionPrediction(blockSize) {	
differentialSimpleHorizontalMotionVector	signedExpGolomb(0)
differentialSimpleVerticalMotionVector	signedExpGolomb(0)
predictedBlock(blockSize) // Luma	
predictedBlock(blockSize / 2) // Cb	
predictedBlock(blockSize / 2) // Cr	
}	

4.7.2. Semantics

quantizerParameterUpdate – A signed integer to be added to quantizerParameter The resultant value shall be used by the decoder until another quantizerParameterupdate is encountered at the macroblock layer.

blockSize4x4 – if blockSize4x4 = 1, then luminance block size of 4x4 pixels is used for the current macroblock, and the corresponding chrominance blocks will be 2x2 pixels.

blockSkipFlag – if blockSkipFlag = 1, the block is not coded and no data follows in the bitstream. if blockSkipFlag = 0, the block is coded and data follows in the bitstream.

simpleMotionPredictionFlag – if simpleMotionPredictionFlag = 1, a differential forward motion vector is coded and follows in the bitstream. if simpleMotionPredictionFlag = 0, the differential forward motion vector is zero and is not coded.

spatialPredictionMode – If spatialPredictionMode = 0, DC prediction is used; if spatialPredictionMode = 1, vertical spatial prediction is used with offset spatialPredictionSpatialOffset; if spatialPredictionMode = 2, horizontal spatial prediction is used with offset spatialPredictionSpatialOffset.

spatialPredictionSpatialOffset – spatial offset in subpixel units for prediction block. If blockSize4x4 = 1 , the range for horizontal offset is {-4,+4}, the range for vertical offset is {-4,0}. If blockSize4x4 = 0, the range for horizontal offset is {-8,+8}, the range for vertical offset is {-8,0}. Encoders shall ensure offsets do not point outside the frame. See section 5.3 for definition of positive direction within a frame.

differentialSimpleHorizontalMotionVector[i] – an increment to be added to the predicted motion vector.

differentialSimpleVerticalMotionVector[i] – an increment to be added to the predicted motion vector.

4.8. Block Layer

4.8.1. Syntax

Syntax	***Mnemonic***
predictedBlock(blockSize){	
i=0	
do {	
eobFlag	flag()
if (!eobFlag){	
do {	
predictedCoefficientError[i++]	signedExpGolomb(0)
} while (predictedCoefficientError[i - 1] == 0)	
}	

Syntax	*Mnemonic*
} while (i < blockSize * blockSize && ! eobFlag)	
while (i < blockSize * blockSize) {	
predictedCoefficientError[i++] = 0	
}	
}	

4.8.2. Semantics

predictedCoefficientError[i]– the sign and magnitude of the i^{th} coefficient error of a predicted block

eobFlag – if eobFlag = 1, there are no more coded coefficients; if eobFlag = 0, there are more coded coefficients

5. Encoding Process

5.1. Introduction

In this section the normative and informative sections for the encoding process will be defined.

5.1.1. Functions and Temporary Variables

5.1.1.1. Temporal Prediction

currentPixel(i,j) Sec. 5.2.2

referencePixel(i,j) Sec. 5.2.2

residualError(i,j) Sec. 5.2.2

5.1.1.2. Spatial Prediction

W_b Sec. 5.3.1 A notation for blockSize Sec. 4.7.1

H_b Sec. 5.3.1 A notation for blockSize Sec. 4.7.1

5.1.1.3. Arithmetic Encoding

Functions

encode()

renormalize()

carryPropagate()

putByte(byte)

initialize()

outputByte()

flush()

Temporary Variables

code Sec. 5.5.2 The variable in which arithmetically coded bits are accumulated

codeBits Sec. 5.5.2 The count of bits remaining until it is necessary to remove a byte from the code variable

interval Sec. 5.5.2 The current probability interval

nStackedFFs Sec. 5.5.2 The count of the number of 0xFF bytes through which a carry could

propagate

pendingByte Sec. 5.5.2 A variable containing the byte before the buffered 0xFF bytes

isPendingByte Sec. 5.5.2 A 1-bit variable that indicates whether a value is pending in the pendingByte variable

nStackedZeros Sec. 5.5.2 A count of trailing zero bytes

contextp Sec. 5.5.1 A pointer to the probability estimation context for the current binary decision being encoded. The probability estimation context contains the following fields:

state Sec. 5.5.1 A field that indicates the current index within the probability estimation finite state machine (represented by the table probFsm, below).

mps Sec. 5.5.1 A field that indicates the most probable symbol as determined by the probability estimate finite state machine.

Constant Tables

probFsm[state] Sec. 5.5.1 The definition of the probabiility tracking finite state machine. This table is indexed by the state field of the current context and each entry has the following fields:

lpsInterval Sec. 5.5.1 The value of the probability interval for the least probable symbol

nextLpsState Sec. 5.5.1 The next state in the probability interval if the current symbol is the LPS

nextMpsState Sec. 5.5.1 The next state in the probability interval if the current symbol is the MPS

doSwitchMps Sec. 5.5.1 A flag indicating whether conditional exchange should occur for the current state

5.2. Temporal Prediction (*Informative*)

5.2.1. Motion Estimation

Motion estimation is performed with 4x4 or 8x8 blocks over a window size of +/-granularity*32 pixels (subject to verification testing). The accuracy of motion estimation is ¼-pixel, with subpixel interpolation according to the cubic-spline method described in section 5.2.2.1 Subpixel Interpolation.

5.2.2. Motion Compensation

Motion compensation is performed with 4x4 or 8x8 blocks. The accuracy of motion compensation is ¼-pixel, with subpixel interpolation according to the cubic-spline method described in section 5.2.2.1 Subpixel Interpolation. The reference block and decoded residual error block are combined according to equation 5.1:

currentPixel(i,j) = referencePixel(i,j) + residualError(i,j) 5.1

5.2.2.1. Subpixel Interpolation for Luminance

The cubic spline interpolator shall be used for interpolation of luminance values up to 1/4 pixel accuracy. A detailed description is provided in Annex-B.

5.2.2.2. Subpixel Interpolation for Chrominance

The bilinear interpolator shall be used for interpolation of chrominance values up to 1/8 pixel accuracy. A detailed description is provided in Annex-B.

5.3. Spatial Prediction (*Informative*)

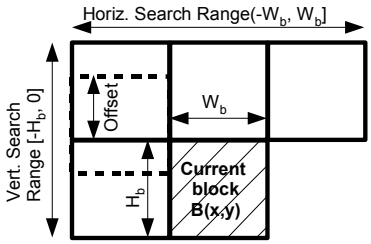

Figure 5.1: Spatial prediction

5.3.1. Spatial Prediction for luminance Blocks

Spatial prediction shall determine the best reference block and its relative offset to the current block for a given luminance block (indicated as "current block" in Figure 5.1) by using the three luminance blocks above, and the one luminance block to the left - all of which are considered to be neighboring blocks. Only neighboring blocks contained in the same subframe as the current block shall be used for spatial prediction. Spatial prediction is performed with block sizes of 4x4 and 8x8. To determine the best block to predict the current block, the following two blocks are computed:

(a) A *DC reference block* is formed by constructing a block where all of the elements are equal to the DC mean value and the dimension of the reference block is the same as the current block. The DC mean value shall be computed using all pixels in the three blocks above and the block to the left of the "current block". If one or more neighboring blocks are unavailable, the mean value shall be computed with the remaining neighboring blocks. If no neighboring blocks are available, the value of 128 must be used as the mean value to construct the DC reference block.

(b) A *candidate neighboring block* is defined as the neighboring block which is the best match and has the same dimensions as the current block. To determine the candidate neighboring block with the smallest prediction error, a search is carried out in the horizontal direction across the blocks above the current block with horizontal offsets in the range $(-W_b, W_b]$ where W_b is the width of the block. In the vertical direction, a search is carried out over the two blocks to the left over the range, $[-H_b, 0]$, where H_b is the height of the block. Figure 5.1 illustrates a vertical direction candidate with its offset from the current block. The best reference block is then selected between the DC reference block and the candidate neighboring block.

The selection of DC, horizontal direction, or vertical direction shall be encoded as follows. If the DC mode is selected then **spatialPredictionMode** is set to 0. If vertical direction is selected then **spatialPredictionMode** is set to 1 and **spatialPredictionSpatialOffset** is set to the vertical offset. If horizontal direction is selected, then **spatialPredictionMode** is set to 2 and **spatialPredictionSpatialOffset** is set to the horizontal offset.These parameters are then encoded.

The spatially predicted residual is computed as follows:

residual(i,j) = currentBlock(i,j) – bestReferenceBlock(i,j) 5.2

The residual shall be forwarded to the transformation process.

Where the luminance block size is 4x4 pixels, an extra restriction is applied for the luminance block 4 (based on the numbering as defined in Fig 3.7) in a given macroblock. This is illustrated in Fig 5.2. The prediction region for this block is limited, thus spatial prediction of luminance block 4 is based only on the three hatched blocks - left, upper left and upper blocks.

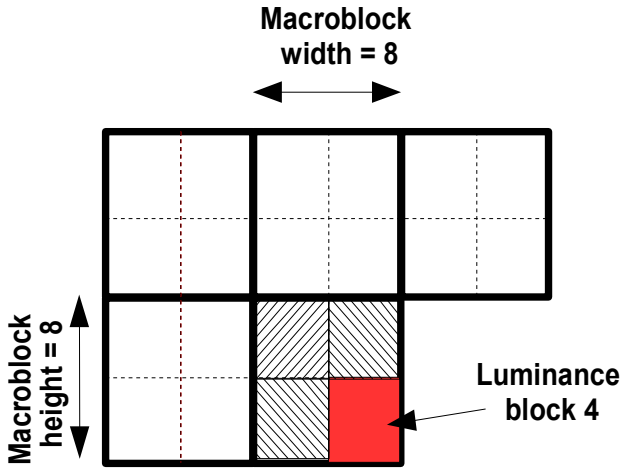

Figure 5.2: Limited region for spatial prediction of luminance block 4

5.3.2. Spatial Prediction for Chrominance Blocks

The encoder does not perform block matching of chrominance blocks for spatial prediction. Instead spatial prediction of the chrominance data shall be carried out using the **spatialPredictionMode** and **spatialPredictionSpatialOffset** derived from the spatial prediction for luminance blocks that is describe in Section 5.3.1.

Where there is one luminance block of 8x8 pixels in a macroblock, if **spatialPredictionMode** is set to 0 then DC prediction shall be applied to the corresponding 4x4 chrominance pixel block of C_b and C_r. If **spatialPredictionMode** is set to 1 then **spatialPredictionSpatialOffset** corresponding to the vertical offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 4x4 chrominance pixel block of C_b and C_r. If **spatialPredictionMode** is set to 2 then

spatialPredictionSpatialOffset corresponding to the horizontal offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 4x4 chrominance pixel block of C_b and C_r.

Where there are 4 luminance blocks in a macroblock, where each block consists of 4x4 pixels, then each chrominance block of 4x4 pixels shall be treated as 4 "sub-blocks" of 2x2 chrominance pixels for purposes of spatial prediction or motion compensated prediction. A chrominance sub-block in this context means the chrominance pixels that correspond spatially to each of the luminance blocks. If **spatialPredictionMode** is set to 0 then DC prediction shall be applied to the corresponding 2x2 chrominance pixel sub-block of C_b and C_r. If **spatialPredictionMode** is set to 1 then **spatialPredictionSpatialOffset** corresponding to the vertical offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 2x2 chrominance pixel sub-block of C_b and C_r. If **spatialPredictionMode** is set to 2 then **spatialPredictionSpatialOffset** corresponding to the horizontal offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 2x2 chrominance pixel sub-block of C_r and C_b.

5.4. Transform and Quantization

5.4.1. Forward Transform – DCT (*Informative*)

The transforms used are 2x2, 4x4 and 8x8 DCT transforms, which are approximations to the traditional DCT and IDCT. To simplify the language, we will refer to them as DCT and IDCT.

The input to the DCT is a 2x2, 4x4 or 8x8 block of 9-bit values. The output consist of a 2x2, 4x4 or 8x8 block of transform coefficients respectively. The transform coefficients are 9-bit, 11-bit and 15-bit values respectively. The input values and the output transform values are stored in 32-bit integer format, and occupy the least significant bits in the 32-bit integer.

The Annex-A contains the precise algorithms an implementation shall follow. Variations in the order in which operations are performed are allowed as long as the results are identical to the results of the algorithms described in the Annex.

5.4.2. Inverse Transform – DCT (N*ormative*)

The inverse transform (IDCT) is required in the encoder for purposes of prediction and reconstruction. This transform is the same as that used in the decoder. See Annex-A for details of the IDCT for 2x2, 4x4 and 8x8 blocks.

5.4.3. Quantization (*Informative)*

5.4.4.

The quantization used in OMS video is an adaptive linear quantization controlled by a variable stepsize. The stepsize is specified in the bitstream at the frame level by a 6-bit non-linear, table driven quantization parameter, **quantizerParameter**. This parameter can be updated with a signed differential value, **quantizerParameterUpdate** at the subframe and macroblock levels.

The linear quantizer forward and inverse operators are

quantization : RQ = round(R, quantizerScale)
dequantization : RDEQ = (RQ * quantizerScale) >> 10

where round() is a rounding function (not necessarily rounding to the nearest integer) and quantizerScale is the quantizer stepsize derived from **quantizerParameter** as specified in Annex-B.

There are two possible mechanisms which control the choice of **quantizerParameter**. One mechanism is to set the value for **quantizerParameter** before the encoding process starts by user input (for example, by specifying a parameter in a configuration file) or a parameter set by the program or device starting the encoding process. The other mechanism allows changing **quantizerParameter** at the frame, subframe or macroblock level by rate control in the encoder itself, for example when trying to achieve a constant bit rate. This might be done by a module in the encoder which would estimate the bit rate for coding a frame or subframe a priori, and adjust **quantizerParameter** for increasing or decreasing the bit rate, or by an a posteriori technique in which the distortion and rate for a range of **quantizerParameter** values is measured in a set of trial encodings, and the optimum values are selected to perform the final encoding.

5.4.5. Zig-Zag Scanning of Quantized Transform Coefficients (Informative)

The quantized transform coefficients are scanned in a zig-zag pattern so that the 2-D matrix of coefficients are converted into a 1-D list. This is carried out for the 2x2, 4x4 and 8x8 matrices of transform coefficients. Figures 5.3, Figure 5.4 and Figure 5.5 illustrate the process of generating 1-D lists from the 2x2, 4x4 and 8x8 blocks respectively.

0	1
2	3

Figure 5.3: Zig-zag scanning for generating 4 values from 2x2 block

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

Figure 5.4: Zig-zag scanning for generating 16 values from 4x4 block

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure 5.5: Zig-zag scanning for generating 64 values from 8x8 block

5.5. Entropy Coding (*Informative*)

Arithmetic coding is used to encode all data within a subframe other than the subframeHeader. The algorithm uses the context-adaptive binary arithmetic coding specified in optional Annex-D in the JPEG standard [ITU-T T.81].

5.5.1. Context Adaptation

Context adaptive arithmetic coding tracks the probability of the symbol values being arithmetic coded with two basic mechanisms. The first mechanism is the most basic; it tracks the probability of the value for one particular binary variable (or multiple binary variables if they share common statistics), An example might be a flag that appears for every macroblock. This probability is tracked by a finite state machine that alters its prediction based on the history of previous values of that variable. The state of the probability finite state machine (fsm) and its probability estimate for the next value of this particular binary variable is maintained in a "context". The encoder and decoder must agree on a common definition of the probability estimator finite state machine.

Because there are multiple variables, each possibly having different statistical behavior, the arithmetic coder allows for multiple contexts (e.g., independent transitions of probability fsm's). As each different variable is coded, the arithmetic coder utilizes and updates the context specific to that variable. Thus the arithmetic coder consists of a core coding engine that for each binary decision being coded uses one of a multiplicity of contexts, each context tracking the probability estimate for a specific variable. The encoder and decoder must agree on the assignment of contexts to variables to be coded.

The initial states of an fsm definition are shown below. The complete fsm definition is specified in Annex-D.

```
struct ProbState {
      int   lpsInterval;
      int   nextStateLps;
      int   nextStateMps;
      bool  doSwitchMps;
};

struct ProbState standardProbFsm = {
    /* state    lpsInterval   nextLpsState nextMpsState doSwitchMps */
    /*   0 */ { 0x5a1d,             1,         1,           1 },
    /*   1 */ { 0x2586,            14,         2,           0 },
    /*   2 */ { 0x1114,            16,         3,           0 },
```

```
        /*   3 */ { 0x080b,          18,         4,          0 },
        /*   4 */ { 0x03d8,          20,         5,          0 },
        /*   5 */ { 0x01da,          23,         6,          0 },
        /*   6 */ { 0x00e5,          25,         7,          0 },
        /*   7 */ { 0x006f,          28,         8,          0 },
        /*   8 */ { 0x0036,          30,         9,          0 },
        /*   9 */ { 0x001a,          33,        10,          0 },
        /*  10 */ { 0x000d,          35,        11,          0 },
        /*  11 */ { 0x0006,           9,        12,          0 },
        /*  12 */ { 0x0003,          10,        13,          0 },
        /*  13 */ { 0x0001,          12,        13,          0 },
        /*  14 */ { 0x5a7f,          15,        15,          1 },
        /*  15 */ { 0x3f25,          36,        16,          0 },
      ….
};
```

A context, which maintains a probability prediction state, is defined by the following structure:

```
struct ProbContext {
        bool                mps;
        int                 state;
        struct ProbState*   probFsm;
};
```

A reference to a probability context, ProbContext, is passed to the arithmetic coding engine in order to implement the probability tracking described.

Context Initialization
All contexts are initialized during subframe processing at the point indicated by initializeArithmeticCoder() in the subframe syntax diagram. Initialization consists of assigning 0 to both the mps and state fields. For all contexts except the eqProbContext, the probFsm field is initialized to reference the standardProbFsm array defined in Annex-D. The eqProbContext has the probFsm field initialized to reference the eqProbFsm array defined in Annex-D.

5.5.2. Arithmetic Encoding Engine

5.5.2.1. Externally Defined Functions Used by Arithmetic Encoding Engine

The following variables are used by the arithmetic encoding algorithm:

Function	Purpose
outputByte(byte)	Emit byte of arithmetically coded data into coded bitstream

5.5.2.2. Variables Used by Arithmetic Encoding Engine

The following variables are used by the arithmetic encoding algorithm:

Variable	Purpose
interval	Current probability interval size
code	Least significant bits of code value
codeBits	Remaining bits until code byte can be emitted

pendingByte	Most recently emitted non-FF code byte
isPendingByte	If true, pendingByte holds a valid code byte
nStackedFFs	Number of stack FF bytes after pendingByte
nStackedZeros	Number of trailing zero code bytes emitted

Probability Interval Subdivision

The encoder uses fixed precision integer arithmetic and avoids multiplication and division by approximating the subdivision of the probability interval and renormalizing by factors of 2 that can be implemented with shifts. The probability interval is stored in a variable *interval* and is kept in the integer range X'8000 to X'10000. This is equivalent to keeping the probability interval in the decimal range $0.75 \le interval < 1.5$ which allows a simple arithmetic approximation to be used in the probability interval subdivision. In the basic algorithm, if the current estimate of the LPS probability for context-index S is *lpsInterval(S)*, precise calculation of the sub-intervals would require:

lpsInterval(S) × *interval* Probability sub-interval for the LPS;
interval – (*lpsInterval(S)* × *interval*) Probability sub-interval for the MPS.

Because the decimal value of *interval* is of order unity, these can be approximated by

lpsInterval(S) Probability sub-interval for the LPS;
interval – lpsInterval(S) Probability sub-interval for the MPS.

The data being coded into the bitstream is initially stored in a variable *code*. Periodically – to keep the *code* variable from overflowing – a byte of data is removed from the high order bits of the *code* variable and placed in the bitstream.

The *interval* and *code* variables are illustrated in Figure 5.6. The “a” bits are the bits in the *interval* variable representing the current probability interval value and the “x” bits are the sub-interval bits in the *code* variable. The “s” bits are optional guard bits which provide temporary storage for carry bits, and the “b” bits indicate the bit positions from which the completed bytes of data are removed from the *code* variable. The “c” bit is a carry bit. Except at the time of initialization, bit 15 of the *interval* variable is always set and bit 16 is always clear (the LSB is bit 0).

	MSB			***LSB***
code	0000cbbb	bbbbbsss	xxxxxxxx	xxxxxxxx
interval	00000000	00000000	aaaaaaaa	aaaaaaaa

Figure 5.6: Encoder registers.

These variable conventions illustrate one possible implementation. However, any variable conventions which allow resolution of carry-over in the encoder and which produce the same entropy-coded segment may be used. The handling of carry-over and the byte stuffing following X’FF’ will be described in a later part of this annex.

Whenever the LPS is coded, the value of *interval – lpsInterval(S)* is added to the *code* variable and the probability interval is reduced to *lpsInterval*(S). Whenever the MPS is coded, the *code* variable is left unchanged and the interval is reduced to *interval – lpsInterval(S)*. The precision range required for *interval* is then restored, if necessary, by renormalization of both the *interval* and *code* variables.

5.5.2.3. Initialization of Arithmetic Encoding Engine

The variables of the arithmetic coding engine are initialized during subframe processing at the point indicated by initializeArithmeticCoder() in the subframe syntax diagram. The coder is initialized by

setting all variables to their initial values. The interval variable is set to 0x10000 representing a decimal value of 1.5. The codeBits variable is set to 11 indicating that the renormalization process can left shift the code variable by 11 bits before the first code byte is to be removed. The isPendingByte variable is set to false indicating that there is no code byte value currently in the pendingByte variable. All other variables are set to zero. At this time, all contexts are also initialized as described in section 5.5.1.

```
initializeArithEncodeEngine()
{
    interval = 0x10000;
    code = 0;
    codeBits = 11;
    pendingByte = 0;
    isPendingByte = false;
    nStackedFFs = 0;
    nStackedZeros = 0;
}
```

5.5.2.4. Arithmetic Encoding of a Binary Decision

A binary decision is coded by calling the routine arithEncode() passing a reference to a probability estimation context, contextp, and the binary decision to be coded. The assignment of contexts to a particular binary decision is specified in the defintions of the binarization algorithms in section 5.5.3.

```
arithEncode(struct ProbContext* contextp, bool decision)
{
    struct ProbState *probFsm = contextp->probFsm;
    interval -= probFsm[contextp->state].lpsInterval;
    if (decision == contextp->mps) {
      // code MPS
      if (interval < 0x8000) {
          if (interval < probFsm[contextp->state].lpsInterval) {
            code += interval;
            interval = probFsm[contextp->state].lpsInterval;
          }
          contextp->state = probFsm[contextp->state].nextStateMps;
          renormalize();
      }
    } else {
      // code LPS
      if (interval >= probFsm[contextp->state].lpsInterval) {
          code += interval;
          interval = probFsm[contextp->state].lpsInterval;
      }
      if (probFsm[contextp->state].doSwitchMps) {
          contextp->mps = ! contextp->mps;
      }
      contextp->state = probFsm[contextp->state].nextStateLps;
      renormalize();
    }
}
```

Renormalization

Renormalization is the process by which the interval is kept near a value of 1. Renormalization occurs after every coding of a LPS and after coding an MPS when the interval has been reduced to less than 0x8000.

Renormalizing entails multiplying both the interval variable and the code variable by 2 and decrementing the *codeBits* variable until the *interval* variable exceeds 0x8000. During this process, if the *codeBits* variable becomes zero, then carryPropagate() is called to remove a byte from the *code* variable and handle any carry propagation.

```
renormalize()
{
    do {
      interval <<= 1;
      code <<= 1;
      if (--codeBits == 0) {
          carryPropagate();
          codeBits = 8;
      }
    } while (interval < 0x8000);
}
```

With the procedure described above, the approximations in the probability interval subdivision process can sometimes make the LPS sub-interval larger than the MPS sub-interval. If, for example, the value of l*psInterval(S)* is 0.5 and *interval* is at the minimum allowed value of 0.75, the approximate scaling gives one-third of the probability interval to the MPS and two thirds to the LPS. To avoid this size inversion, conditional exchange is used. The probability interval is subdivided using the simple approximation, but the MPS and LPS sub-interval assignments are exchanged whenever the LPS sub-interval is larger than the MPS sub-interval. This MPS/LPS conditional exchange can only occur when a renormalization will be needed.

Carry Propagation

It is possible for this procedure to generate successive carry bits without limitation. In order to limit the length of the *code* variable, carry bits may be accumulated in bytes, and a count may be kept of the number of such bytes until the propagation ends. At that point, the number of such bytes of all-ones may be stuffed into the bitstream.

The function carryPropagate() removes a byte from the *code* variable and resolves any carry propagation out of that byte.

To allow for carries to propagate out of the *code* variable, carryPropagate buffers earlier bytes of code that could have their value changed by a carry. This entails buffering all previous contiguous trailing 0xFF valued bytes of the code stream along with the immediately previous byte before the string of 0xFFvalues.

The 0xFF valued code stream bytes are not actually buffered, rather a count is maintained in the variable *nStackedFFs*. The byte before the buffered 0xFF values is maintained in the *pendingByte* variable.

If the byte removed from the *code* variable is less than 0xFF, then it will absorb any future carry without generating a further carry; in that case, the currently buffered pending byte and 0xFF values are output because they are no longer subject to a carry.

The value 0xFF is used as a flag byte in the code stream to indicate various marker conditions, thus a 0xFF byte is always immediately followed by a second byte that indicates the particular type of marker. Markers and the associated type byte are:

Literal 0xFF	0x01
End of Code Stream	0x02

Stacked 0xFF Break	0x03
Stacked 0xFF Break Carry	0x04

An actual code byte of 0xFF is represented in the code stream as a 0xFF flag byte followed by a 0x01 byte. The arithmetic decoder should replace this byte pair with a single value of 0xFF. The end of the code stream is indicated by a 0xFF byte followed by a 0x02 byte. The arithmetic decoder should remove these bytes from the code stream, stop reading the code stream, and source 0 bytes as needed to the arithmetic decoder. The Stacked 0xFF Break byte pair indicates that an implementation limit was reached in the encoder that precluded further stacking of 0xFF values; this value should simply be removed from the code stream by a post-processing step of the code stream. If the Stacked 0xFF Break Carry flag indicates that a carry occurred into a Stacked 0xFF Break flag, a post-processor should generate a carry into the preceding byte (handling carry propagation as necessary) and eliminate this byte pair. Most software implementations will not need to utilize the Stacked 0xFF Break codes.

```
carryPropagate()
{
    unsigned int byte = code >> 19;
    code &= 0x7ffff;

    if (byte < 0xFF) {
      while (nStackedFFs != 0) {
          putByte(0xFF);
          putByte(0);
          nStackedFFs -= 1;
      }
      putByte(byte);
    } else if (byte > 0xFF) {
      // FF introduces escape sequences, real FF's are coded
      // as FF 00, so if this is a real FF follow it with a 0
      if (++pendingByte == 0xFF) {
          putByte(0);
      }
      // All intervening FF's have carry propagated to 00's
      while (nStackedFFs != 0) {
          putByte(0);
          nStackedFFs -= 1;
      }
      putByte(byte);
    } else {
      nStackedFFs += 1;
    }
}
```

Trailing Zero Elimination

The function putByte maintains a count of trailing zero bytes so that these trailing zero bytes may be easily eliminated from the end of the code stream when terminating the coding process. Trailing zeros are counted at the time the pendingByte is output to ensure that no further modification by carry propagation can occur. The function outputByte represents an external function that is passed the coded byte stream.

```
putByte(char byte)
{
    if (isPendingByte) {
      if (pendingByte == 0) {
          nStackedZeros += 1;
```

```
            } else {
                while (nStackedZeros > 0) {
                  outputByte(0);
                  nStackedZeros -= 1;
                }
                outputByte(pendingByte);
                if (pendingByte == 0xFF && byte == 0) {
                  outputByte(byte);
                  isPendingByte = false;
                  return;
                }
            }
        }
        pendingByte = byte;
        isPendingByte = true;
    }
```

5.5.2.5. Termination of the Arithmetic Encoding Engine

Termination of the arithmetic encoding engine occurs in the subframe processing at the point indicated by terminateArithmeticCoder() in the subframe syntax diagram. Termination of the arithmetic encoding process consists of zeroing the most low order bits of the code variable while still remaining inside the final interval. Then this final value is aligned with the code stream by shifting left by the value in the codeBits variable and a byte is removed from the code variable by the carryPropagate function. The code variable is then shifted 8 more bits to the left and another byte is removed by the carryPropagate function. Finally, the pendingByte variable is output by calling putByte(0). At this point, only trailing zero bytes have not been output, since the intent is to eliminate trailing zeros from the code stream. The process is completed by clearing the nStackedZeros variable.

```
terminateArithEncodingEngine()
{
    t = (code + interval - 1) & 0xFFFF0000;
    if (t < code) t += 0x8000;

    code = t << codeBits;
    carryPropagate();

    code <<= 8;
    carryPropagate();

    putByte();
    nStackedZeros = 0;

    // write end of stream flag
    outputByte(0xFF);
    outputByte(0x2);
}
```

5.5.3. Binarization of Multi-State Variables

Many variables to be coded are not simple binary values, but have many large symbol alphabets (e.g., enumerations, modes, integer values). These variables must be converted to a representation that is a sequence of binary values before they can be coded; this process is called "binarization." The binarization algorithms used in the OMS video codec are described later, but all can be thought of as a binary tree where each leaf of the tree represents a particular symbol in the alphabet of the variable. The two edges leaving a node are labeled 0 and 1. The path of edges from the root of the

tree to a particular leaf is the binarization of the symbol associated with that leaf. The binarization tree is generally constructed so that the most frequent symbols have the shortest paths, so the binarization trees represent a variable length coding of the variable.

The statistics of the binary decision represented by the outgoing two edges of the different nodes vary, hence differing nodes of the binarization tree for a single variable may have a different context associated with it to track the likelihood of that particular node decision. Thus non-binary values may require multiple contexts to track the probability of the individual decisions in the binarization tree. The encoder and decoder must agree on common binarization algorithms for all variables coded and the assignment of contexts to the nodes of the binarization tree.

OMS video specifies six different binarization algorithms. Each algorithm specifies how an alphabet of symbols is mapped to a binary tree where each leaf of the tree represents a symbol in the alphabet and the path of edges from the root of the tree to a particular leaf represents the binarization of that particular symbol in the alphabet. The OMS binarization algorithms also define how contexts used during arithmetic coding are assigned to the nodes of the binarization tree.

Each variable to be coded is associated with one of the six binarization algorithms. The selected binarization algorithm is specialized for that particular variable by a set of parameters.
The specialized binarization algorithm for a variable is specified by a string of the form "NAME(PARAMLIST). NAME specifies the general algorithm for the binarization process; each algorithm used by OMS video is described below. PARAMLIST is a comma-separated list of values that specializes the algorithm for a particular variable, The PARAMLIST also defines how many arithmetic coder contexts are to be associated with the binarization tree. The assignment of those contexts to specific nodes is specified in the binarization algorithm description.

Binarization Algorithms

Binarization is the process of converting a value to a sequence of binary decisions that can be coded by the binary arithmetic coding process. The binarization algorithms to be used to binarize a particular syntax item is summarized in the mnemonic field of the specific syntax item as shown in the Syntax diagrams in Section 4. For each syntax item, the binarization algorithm and the specific values of the parameters to be passed to that algorithm are specified in sections 5.5.2 through 5.5.6.

flag() Encoding
The simplest binarization: a single bit is arithmetically coded using a single context.

Pseudo-code implementation:

```
flagEncode(contexts[], value)
{
    arithEncode(contexts[0], value != 0);
}
```

fixed(nBits) Encoding
An unsigned, fixed width integer value of nBits binary bits is coded directly, most significant bit first. There are nContext contexts; each context is assigned to a single bit from the most significant bit towards the least significant bit. If there are fewer contexts than bits (nContext < nBits), then the final context is reused for the remaining least significant bits.

Pseudo-code implementation:

```
fixedEncode(contexts[], nContexts, nBits, value)
```

```
{
    // PRECONDITION: value must be representable in
    // nBits bits
    n = 0;

    for (i = nBits - 1; i >= 0; i--) {
      arithEncode(contexts[n], value & (1 << i));
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
}
```

signedFixed(nBits) Encoding

A signed, integer value is first mapped onto the unsigned integers by the function: f(x) = 2 * abs(x) – (x > 0 ? 0 : 1) and then coded as described in fixed binarization. There are nContext contexts; each context is assigned to a single bit from the most significant bit of the mapped value towards the least significant bit. If there are fewer contexts than bits (nContext < nBits), then the final context is reused for the remaining least significant bits.

Pseudo-code implementation:

```
signedFixedEncode(contexts[], nContexts, nBits, value)
{
    // PRECONDITION: sign folded value must be representable in
    // nBits bits
    n = 0;

    // map signed value to unsigned value
    value = (abs(value) << 1) - (value > 0 ? 1 : 0);

    for (i = nBits - 1; i >= 0; i--) {
      arithEncode(contexts[n], value & (1 << i));
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
}
```

unary() Encoding

An unsigned, integer value is coded as a string of 1 decisions the length of which represents the value to be coded. The string of 1's is then followed by a 0 decision. nContext contexts are allocated, The contexts are assigned individually to the first nContext decisions. If the value requires more than nContext decisions, the last context is reused for all remaining decisions.

Pseudo-code implementation:

```
unaryEncode(contexts[], nContexts, value)
{
    n = 0;  // context index

    while (value-- > 0) {
      arithEncode(contexts[n], 1);
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
    arithEncode(contexts[n], 0);
}
```

signedUnary() Encoding
A signed, integer value is first mapped onto the unsigned integers by the function: f(x) = 2 * abs(x) – (x > 0 ? 0 : 1) and then coded as described in unary binarization. nContext contexts are allocated, The contexts are assigned individually to the first nContext decisions. If the value requires more than nContext decisions, the last context is reused for all remaining decisions.

Pseudo-code implementation:

```
signedUnaryEncode(contexts[], nContexts, value)
{
    n = 0;  // context index

    // map signed value to unsigned value
    value = (abs(value) << 1) - (value > 0) ? 1 : 0;

    while (value-- > 0) {
      arithEncode(contexts[n], 1);
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
    arithEncode(contexts[n], 0);
}
```

trUnary(maxVal) Encoding
An unsigned, integer value up to and including maxValue is coded as a string of 1 decisions. If the value is less than maxValue, a final 0 decision is coded. It is invalid to code a value greater than maxValue. nContext contexts are allocated. The contexts are assigned individually to the first nContext decisions. If the value requires more than cContext decisions, the last context is reused for all remaining decisions.

Pseudo-code implementation:

```
trUnaryEncode(contexts[], nContexts, maxValue, value)
{
    // PRECONDITION: value MUST be <= maxValue
    n = 0;

    while (value-- > 0) {
      arithEncode(contexts[n], 1);
      maxValue -= 1;
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
    // Code terminating 0 if value < maxValue
    if (maxValue > 0) {
      arithEncode(contexts[n], 0);
    }
}
```

expGolomb(k) Encoding
An unsigned integer value is coded. The value is coded in two stages:

Part A:
While the value is greater than or equal to 2^k, code a 1 decision, reduce the value by 2^k, and increment k by 1. Then code a 0 decision.

Part B:

For i from k down to zero, code the 2^k bit of the modified value directly as a decision.

nContext contexts are allocated. Contexts are assigned individually to each successive decision coded in Part A; if there are more decisions than nContext, the last context is reused for the remaining decisions.

Decisions in Part B all use the equal probability context. The equal probability context is a special, single state probability finite state machine that estimates all bits as approximately 50% likely to be a 1.

```
expGolombEncode(contexts[], nContexts, k, value)
{
    n = 0;  // context index

    while (value >= (pow2 = (1 << k))) {
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
      // code that value is > 2**k
      arithEncode(contexts[n], 1);
      value -= pow2;
      k += 1;
    }
    // advance to next context if sufficient contexts
    if (n < nContexts - 1) n++;
    // code zero to mark end of exponent
    arithEncode(contexts[n], 0);
    // code low order bits
    while (k-- > 0) {
      arithEncode(eqProbContext, value & (1 << k));
    }
}
```

signedExpGolomb(k) Encoding

A signed integer value is coded. The value is coded in three stages.

Part A:

If the value is zero, code a 1 decision and conversion is complete (Parts B and C not done). Else code a 0.
If the value is negative, code a 1 decision and the value is converted to its absolute value; otherwise code a 0 decision and the value is unchanged.
Decrement the value by 1 and proceed to Part B.

Part B:

While the value is greater than or equal to 2^k, code a 1 decision, reduce the value by 2^k, and increment k by 1.
When value is no longer greater than or equal to 2k, code a 0 decision.
Proceed to Part C.

Part C:

For i from k down to zero, code the 2^k bit of the modified value directly as a decision.

nContext contexts are allocated. Contexts are assigned individual to each successive decision coded in Part A and B; if there are more decisions than nContext, the last context is reused for the remaining decisions.

Decisions in Part C all use the equal probability context. The equal probability context is a special, single state probability finite state machine that estimates all bits as approximately 50% likely to be a 1.

Pseudo-code implementation:

```
signedExpGolombEncode(contexts[], nContexts, k, value)
{
    n = 0;  // context index

    // Code zero test of value
    isZero = (value == 0);
    arithEncode(contexts[n], isZero);
    if (isZero) return;

    // advance to next context if sufficient contexts
    // and code sign of value
    if (n < nContexts - 1) n++;
    isNegative = (value < 0);
    arithEncode(contexts[n], isNegative);

    // value = abs(value) - 1
    if (value < 0) value = -value;
    value -= 1;

    while (value >= (pow2 = (1 << k))) {
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
      // code that value is > 2**k
      arithEncode(contexts[n], 1);
      value -= pow2;
      k += 1;
    }
    // advance to next context if sufficient contexts
    if (n < nContexts - 1) n++;
    // code zero to mark end of exponent
    arithEncode(contexts[n], 0);

    // code low order bits
    while (k-- > 0) {
      arithEncode(eqProbContext, value & (1 << k));
    }
}
```

5.5.4. Arithmetic Encoding of Flags

This section describes for each arithmetically coded syntax element the binarization algorithm to be used and the parameters to pass to that binarization algorithm. The parameters include the context array, its length, and any binarization-specific parameters. In all cases, the final value passed to the binarization algorithm is the value to be coded.

5.5.4.1. MacroblockSkipFlag

macroblockSkipFlag is context modeled with the context array macroblockSkipFlagContext of length 1, declared as:

```
struct ProbContext macroblockSkipFlagContext[1];
```

macroblockSkipFlag is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(macroblockSkipFlagContext, macroblockSkipFlagValue);
```

5.5.4.2. lastMacroblockFlag

lastMacroblockFlag is context modeled with the context array lastMacroblockFlagContext of length 1, declared as:

```
struct ProbContext lastMacroblockFlagContext[1];
```

lastMacroblockFlag is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(lastMacroblockFlagContext, lastMacroblockFlagValue);
```

5.5.4.3. blockSize4x4

blockSize4x4 is context modeled with the context array blockSize4x4Context of length 1, declared as:

```
struct ProbContext blockSize4x4Context[1];
```

blockSize4x4 is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(blockSize4x4Context, blockSize4x4Value);
```

5.5.4.4. BlockSkipFlag

blockSkipFlag is context modeled with the context array blockSkipFlagContext of length 1, declared as:

```
struct ProbContext blockSkipFlagContext[1];
```

blockSkipFlag is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(blockSkipFlagContext, blockSkipFlagValue);
```

5.5.4.5. SimpleMotionPredictionFlag

simpleMotionPredictionFlag is context modeled with the context array simpleMotionPredictionFlagContext of length 1, declared as:

```
struct ProbContext simpleMotionPredictionFlagContext[1];
```

simpleMotionPredictionFlag is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(simpleMotionPredictionFlagContext,
simpleMotionPredictionFlagValue);
```

5.5.4.6. eobFlag

eobFlag is context modeled with the context array eobFlagContext of length 1, declared as:

```
struct ProbContext eobFlagContext[1];
```

eobFlag is binarized and arithmetically encoded with the **flag** binarization invoked as:

```
flagEncode(eobFlagContext, eobFlagValue);
```

5.5.5. Subframe Overhead Data Arithmetic Coding

5.5.5.1. deltaSubframeHorizontalAddress

deltaSubframeHorizontalAddress is context modeled with the context array deltaSubframeHorizontalAddressContext of length 4, declared as:

```
struct ProbContext deltaSubframeHorizontalAddressContext[4];
```

deltaSubframeHorizontalAddress is binarized and arithmetically encoded with the **signedUnary** binarization invoked as:

```
signedUnaryEncode(deltaSubframeHorizontalAddressContext, 4,
deltaSubframeHorizontalValue);
```

5.5.5.2. deltaSubframeVerticalAddress

deltaSubframeVerticalAddress is context modeled with the context array deltaSubframeVerticalAddressContext of length 4, declared as:

```
struct ProbContext deltaSubframeVerticalAddressContext[4];
```

deltaSubframeVerticalAddress is binarized and arithmetically encoded with the u**nary** binarization invoked as:

```
unaryEncode(deltaSubframeVerticalAddressContext, 4,
deltaSubframeVerticalValue);
```

5.5.6. Macroblock Overhead Data Arithmetic Coding

5.5.6.1. Quantizer Parameter Update

quantizerParameterUpdate is context modeled with the context array quantizerParameterUpdateContext of length 4, declared as:

```
struct ProbContext quantizerParameterUpdateContext[4];
```

quantizerParameterUpdate is binarized and arithmetically encoded with the **signedUnary** binarization invoked as:

```
signedUnaryEncode(quantizerParameterUpdateContext, 4,
macroblockQuantizerScaleValue);
```

5.5.7. Spatial Prediction Arithmetic Coding

5.5.7.1. Spatial Prediction Mode

spatialPredictionMode is context modeled with the context array spatialPredictionModeContext of length 2, declared as:

```
struct ProbContext spatialPredictionModeContext[2];
```

spatialPredictionMode is binarized and arithmetically encoded with the **fixed** binarization invoked as:

```
fixedEncode(spatialPredictionModeContext, 2, spatialPredictionModeValue);
```

5.5.7.2. Spatial Prediction Spatial Offset

spatialPredictionOffset is context modeled with the context array spatialPredictionOffsetContext of length 7, declared as:

```
struct ProbContext spatialPredictionModeContext[7];
```

spatialPredictionOffset is binarized and arithmetically encoded with the **signedFixed** binarization invoked as:

```
signedFixedEncode(spatialPredictionOffsetContext, 7, 7,
spatialPredictionOffsetValue);
```

5.5.8. Motion Vector Arithmetic Coding

5.5.8.1. Differential Simple Horizontal Motion Vector

differentialSimpleHorizontalMotionVector is context modeled with the context array differentialSimpleHorizontalMotionVectorContext of length 6, declared as:

```
struct ProbContext differentialSimpleHorizontalMotionVectorContext[6];
```

differentialSimpleHorizontalMotionVector is binarized and arithmetically encoded with the **signedExpGolomb** binarization invoked as:

```
signedExpGolombEncode(differentialSimpleHorizontalMotionVectorContext, 6,
0, differentialSimpleHorizontalMotionVectorValue);
```

5.5.8.2. Differential Simple Vertical Motion Vector

differentialSimpleVerticalMotionVector is context modeled with the context array differentialSimpleVerticalMotionVectorContext of length 6, declared as:

```
struct ProbContext differentialSimpleVerticalMotionVectorContext[6];
```

differentialSimpleVerticalMotionVector is binarized and arithmetically encoded with the **signedExpGolomb** binarization invoked as:

```
signedExpGolombEncode(differentialSimpleVerticalMotionVectorContext, 6, 0,
differentialSimpleVerticalMotionVectorValue);
```

5.5.9. Arithmetic Coding of DCT Coefficient

5.5.9.1. Coding of Differential Coefficients

PredictedCoefficientError is context modeled with the following context arrays:

For predictedCoefficientError when contained in 8x8 luma blocks:

```
struct ProbContext luma8x8Context[33][6];
```

For predictedCoefficientError when contained in 4x4 luma blocks:

```
struct ProbContext luma4x4Context[9][6];
```

For predictedCoefficientError when contained in 4x4 Cb or Cr chroma blocks:

```
struct ProbContext chroma4x4Context[9][6];
```

For predictedCoefficientError when contained in 2x2 Cb or Cr chroma blocks:

```
struct ProbContext chroma4x4Context[4][6];
```

The i'th **predictedCoefficientError** as ordered by the zig-zag coefficient scan pattern defined in section 5.4.3 is binarized and arithmetically encoded with the **signedExpGolomb** binarization invoked as follows:

For predictedCoefficientError when contained in 8x8 luma blocks:

```
signedExpGolombEncode(luma8x8Context[i < 32 ? i : 32], 6, 0,
predictedCoefficientErrorValue);
```

For predictedCoefficientError when contained in 4x4 luma blocks:

```
signedExpGolombEncode(luma4x4Context[i < 8 ? i : 8], 6, 0,
predictedCoefficientErrorValue);
```

For predictedCoefficientError when contained in 4x4 Cb or Cr chroma blocks:

```
signedExpGolombEncode(chroma4x4Context[i < 8 ? i : 8], 6, 0,
predictedCoefficientErrorValue);
```

For predictedCoefficientError when contained in 2x2 Cb or Cr chroma blocks:

```
signedExpGolombEncode(chroma2x2Context[i], 6, 0,
predictedCoefficientErrorValue);
```

6. Decoding Process

6.1. Introduction

In this section the sections for the decoding process shall be defined normatively.

6.1.1. Functions and Temporary Variables

6.1.1.1. Temporal Prediction

previousHorizontalMotionVector Sec. 6.4.2: The value of the horizontal component of the previous block motion vector, used as a predictor for the horizontal component of the current block motion vector

previousVerticalMotionVector Sec. 6.4.2: The value of the vertical component of the previous block motion vector, used as a predictor for the vertical component of the current block motion vector.

reconstructedHorizontalMotionVector Sec. 6.4.2: The value of the horizontal component of the current block motion vector.

reconstructedVerticalMotionVector Sec. 6.4.2: The value of the vertical component of the current block motion vector.

6.2. Entropy Decoding

6.2.1. High Level Syntax

Syntax elements for the Sequence, Frame, and Subframe header are fixed-length coded. They are parsed and interpreted according to the syntax and semantics in Section 4. Syntax elements related to the size of the video frame are interpreted according to the granularity syntax element. These syntax elements are coded efficiently with a limited number of bits that are interpreted to represent a dynamic range and resolution for the encoded value.

6.2.2. Contexts Adaptation

Context adaptive arithmetic coding tracks the probability of the symbol values being arithmetically coded with two basic mechanisms. The first mechanism is the most basic, it tracks the probability of a 1 value for one particular binary variable (or multiple binary variables if they share common statistics). An example might be a flag that appears for every macroblock. This probability is tracked by a finite state machine that alters its prediction based on the history of previous values of that variable. The state of the probability finite state machine (fsm) and its probability estimate for the next value of this particular binary variable is maintained in a "context". The encoder and decoder must agree on a common definition of the probability estimator finite state machine.

Because there are multiple variables, each possibly having different statistical behavior, the arithmetic coder allows for multiple contexts (e.g. independent transitions of probability fsm's). As

each different variable is coded, the arithmetic coder utilizes and updates the context specific to that variable. Thus the arithmetic coder consists of a core coding engine that for each binary decision being coded uses one of a multiplicity of contexts, each context tracking the probability estimate for a specific variable. The encoder and decoder must agree on the assignment of contexts to variables to be coded.

The initial states of an fsm definition are shown below. The complete fsm definition is specified in Annex-D.

```
struct ProbState {
      int   lpsInterval;
      int   nextStateLps;
      int   nextStateMps;
      bool  doSwitchMps;
};

struct ProbState standardProbFsm = {
    /* state    lpsInterval   nextLpsState nextMpsState doSwitchMps */
    /*   0 */ { 0x5a1d,             1,         1,           1 },
    /*   1 */ { 0x2586,            14,         2,           0 },
    /*   2 */ { 0x1114,            16,         3,           0 },
    /*   3 */ { 0x080b,            18,         4,           0 },
    /*   4 */ { 0x03d8,            20,         5,           0 },
    /*   5 */ { 0x01da,            23,         6,           0 },
    /*   6 */ { 0x00e5,            25,         7,           0 },
    /*   7 */ { 0x006f,            28,         8,           0 },
    /*   8 */ { 0x0036,            30,         9,           0 },
    /*   9 */ { 0x001a,            33,        10,           0 },
    /*  10 */ { 0x000d,            35,        11,           0 },
    /*  11 */ { 0x0006,             9,        12,           0 },
    /*  12 */ { 0x0003,            10,        13,           0 },
    /*  13 */ { 0x0001,            12,        13,           0 },
    /*  14 */ { 0x5a7f,            15,        15,           1 },
    /*  15 */ { 0x3f25,            36,        16,           0 },
   ….
};
```

A context, which maintains a probability prediction state, is defined by the following structure:

```
struct ProbContext {
      bool             mps;
      int              state;
      struct ProbState* probFsm;
};
```

A reference to a probability context, ProbContext, is passed to the arithmetic decoding engine in order to implement the probability tracking described.

6.2.2.1. Context Initialization

All contexts are initialized during subframe parsing at the point indicated by initializeArithmeticCoder(). For all contexts, the mps and probState fields are initialized to zero. The probFsm field is initialized to standardProbFsm for all contexts except the eqProbFsm, where it is initialized to eqProbFsm. Both standProbFsm and eqProbFsm are defined in Annex-D.

```
initContext(struct ProbContext *contextp, struct ProbState probFsm[])
{
    contextp->mps = 0;
```

```
    contextp->probState = 0;
    contextp->probFsm = probFsm;
}
```

6.2.3. Arithmetic Decoding Engine

6.2.3.1. Externally Defined Functions Used by Arithmetic Decoding Engine

The following variables are used by the arithmetic decoding algorithm:

Function	Purpose
inputtByte()	Returns next byte of arithmetically coded data from coded bitstream

6.2.3.2. Variables Used by Arithmetic Decoding Engine

The following variables are used by the arithmetic decoding algorithm:

Variable	Purpose
interval	Current probability interval size
code	Least significant bits of code value
codeBits	Remaining bits until code byte can be emitted
bin	Temporary variable holding binary decision value
state	Temporary variable holding current probability FSM state
b	Temporary variable holding next code byte from bitstream

Probability Interval Subdivision

The decoder utilizes equivalent principles of interval subdivision as used in the encoding process, but in the decoding process the task is to determine which sub-interval the code value lies within. As each sub-interval and the coding of the corresponding decision is determined, the decoder subtracts from the code register the equivalent value added by the encoder. This process continues recursively in order to process the entire arithmetically coded bitstream. The decoder uses fixed precision integer arithmetic and avoids multiplication and division by approximating the subdivision of the probability interval and renormalizing by factors of 2 that can be implemented with shifts.

The data being decoded from the bitstream is loaded in a variable *code*. Periodically – to keep the *code* variable from underflowing – a byte of data is read from the coded bitstream and added to *code* variable.

The *interval* and *code* variables are illustrated in Table 6.1. The “a” bits are the bits in the *interval* variable representing the current probability interval value and the “x” bits are the sub-interval bits in the *code* variable. The “b” bits indicate the bit positions from which the input bytes from the coded bitstream are introduced into the *code* variable.

	MSB			***LSB***
code	xxxxxxxx	xxxxxxxx	bbbbbbbb	00000000
interval	00000000	00000000	aaaaaaaa	aaaaaaaa

Table 6.1: Encoder registers.

Whenever the LPS is coded, the value of *interval – lpsInterval(S)* is subtracted from the *code* variable and the probability interval is reduced to *lpsInterval*(S). Whenever the MPS is coded, the *code* variable is left unchanged and the interval is reduced to *interval – lpsInterval(S)*. The precision range required for *interval* is then restored, if necessary, by renormalization of both the *interval* and *code* variables.

6.2.3.3. Initialization of Arithmetic Decoding Engine

The arithmetic decoding engine is initialized during subframe parsing at the point indicated by initializeArithmeticCoder(). isEof is set to false to indicate that end of the arithmetically coded input stream has not been seen. interval is set to 0x10000 which represents an initial value of 1.5. The code register is initialized to zero and then byteIn() is called sucessively to shift in the first three bytes of the arithmetically coded input stream. At this time, all contexts are also initialized as described in section 6.2.2.1.

```
initDecoder()
{
    // advance input stream to byte alignment
    isEof = false;
    interval = 0x10000;
    code = 0;

    byteIn();
    code <<= 8;
    byteIn();
    code <<= 8;
    byteIn();
}
```

6.2.3.4. Arithmetic Decoding Input Stream to Recover Coded Decisions

The decoding procedure, arithDecode(), decodes a single binary decision. It is invoked by the debinarization algorithms described later in section 6.2.4. The binary decision is determined by the comparison of the high order bits of the code variable to interval value; that decision is subject to conditional exchange if the interval value is less than the current estimation of the lps interval.

```
// decodes one bin, returns either 0 or 1
bool
arithDecode(struct ProbContext *contextp)
{
    bin = contextp->mps;
    state = contextp->state;

    interval -= contextp->probFsm[state].lpsInterval;
    if ((code >> 16) < interval) {
      if (interval < 0x8000) {
          if (interval < contextp->probFsm[state].lpsInterval) {
            bin = ! bin;
            if (contextp->contextp->probFsm[state].doSwitchMps) {
                contextp->mps = bin;
            }
            contextp->state = contextp->probFsm[state].nextLps;
```

```
            } else {
              contextp->state = contextp->probFsm[state].nextMps;
            }
            renormDecoder();
          }
        } else {
            code -= interval << 16;
          if (interval < contextp->probFsm[state].lpsInterval) {
            contextp->state = contextp->probFsm[state].nextMps;
          } else {
            bin = ! bin;
            if (contextp->probFsm[state].doSwitchMps) {
              contextp->mps = bin;
            }
            contextp->state = contextp->probFsm[state].nextLps;
          }
          interval = contextp->probFsm[state].lpsInterval;
          renormDecoder();
        }
        return bin;
    }
```

Renormalization of Decoder

Renormalization is performed in order to maintain the value of the interval variable greater than or equal to 0x8000. The renormDecoder() procedure left shifts both the interval and code variables by one until the interval variable value is greater than or equal to 0x8000. The codeBits variable indicates how many bits remain for the current byte from the coded bitstream. When codeBits is decremented to zero, byteIn() is invoked to load a new byte from the coded bitstream.

```
renormDecoder()
{
    do {
      interval <<= 1;
      code <<= 1;
      if (--codeBits == 0) byteIn();
    } while (interval < 0x8000);
}
```

Input of Byte from Coded Bitstream

The byteIn() procedure reads the next byte from the coded bitstream, and handles the processing of the special processing of 0xFF bytes. A 0xFF byte indicates that the next byte in the coded bitstream will indicate whether the pair of input bytes (0xFF and its successor indicate the end of the arithmetically coded bitstream or a literal 0xFF byte. A successor value of 0x01 indicates that the pair of bytes (0xFF, 0x01) represents a 0xFF byte in the coded bitstream. 0XFF followed by any other value indicates the end of the arithmetically coded bitstream.

```
byteIn()
{
    codeBits = 8;
    if (isEof) return;
    b = inputByte();
    if (b == 0xFF) {
      isEof = (inputByte() != 0x01);
      if (isEof) return;
    }
    code += b << 8;
}
```

6.2.3.5. Termination of Arithmetic Decoder Engine

The arithmetic decoding engine is terminated during subframe parsing at the point indicated by terminateArithmeticCoder(). At this time, flushDecoder() is invoked to consume any trailing bytes of arithmetically coded input stream.

```
flushDecoder()
{
    while (! isEof) {
      b = inputByte();
      if (b == 0xFF) {
          isEof = (inputByte() != 0x01);
      }
    }
}
```

6.2.4. Debinarization Algorithms

Debinarization is the process of reversing the binarization process used in the binary arithmetic coding of values to arrive back at the original coded value. The debinarization algorithms to be used to debinarize a particular syntax item is summarized in the mnemonic field of the specific syntax item as shown in the Syntax diagrams in Section 4. For each syntax item, the debinarization algorithm and the specific values of the parameters to be passed to that algorithm are specified in sections 6.2.4 through 6.2.7. For all algorithms, the arithmetically decoded and debinarized value is returned.

flag() Debinarization
The simplest binarization: a single bit is arithmetically coded using a single context.

Pseudo-code implementation:

```
flagDecode(contexts[])
{
    return arithDecode(contexts[0]);
}
```

fixed(nBits) Debinarization
An unsigned, fixed width integer value of nBits binary bits is coded directly, most significant bit first. There are nContext contexts; each context is assigned to a single bit from the most significant bit towards the least significant bit. If there are fewer contexts than bits (nContext < nBits), then the final context is reused for the remaining least significant bits.

Pseudo-code implementation:

```
fixedDecode(contexts[], nContexts, nBits)
{
    n = 0;
    value = 0;

    for (i = nBits - 1; i >= 0; i--) {
      if (arithDecode(contexts[n]) value |= (1 << i));
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }

    return value;
```

```
}
```

signedFixed(nBits) Debinarization

First, an unsigned nBit “mapped” representation of the original coded value is decoded using the fixed debinarization procedure described above. Then, the mapped value is converted to the original signed integer value by the following process: The sign of the original value will be positive if the least significant bit of the mapped value is 1, the sign of the original value will be negative otherwise. Next, the mapped value is incremented by 1 and then divided by 2. Finally, the determined sign is applied to this value to form the original value.

Pseudo-code implementation:

```
signedFixedDecode(contexts[], nContexts, nBits)
{
    n = 0;
    value = 0;

    for (i = nBits - 1; i >= 0; i--) {
      if (arithDecode(contexts[n]) value |= (1 << i));
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }

    // perform inverse of sign folding
    isPositive = value & 1;
    value += 1;
    value >>= 1;
    if (! isPositive) value = -value;

    return value;
}
```

unary() Debinarization

An unsigned, integer value is coded as a string of 1 decisions the length of which represents the value to be coded. The string of 1's is then followed by a 0 decision. nContext contexts are allocated, The contexts are assigned individually to the first nContext decisions. If the value requires more than nContext decisions, the last context is reused for all remaining decisions.

Pseudo-code implementation:

```
unaryDecode(contexts, nContexts)
{
    n = 0;
    value = 0;

    while (arithDecode(contexts[n])) {
      value += 1;
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }

    return value;
}
```

signedUnary() Debinarization

First, an unsigned nBit "mapped" representation of the original coded value is decoded following the process as described above for unary debinarization. Then, the mapped value is converted to the original signed integer value by the following process: The sign of the original value will be positive if the least significant bit of the mapped value is 1, the sign of the original value will be negative otherwise. Next, the mapped value is incremented by 1 and then divided by 2. Finally, the determined sign is applied to this value to form the original value.

Pseudo-code implementation:

```
signedUnaryDecode(contexts[], nContexts)
{
    n = 0;
    value = 0;

    while (arithDecode(contexts[n])) {
      value += 1;
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }

    // perform inverse of sign folding
    isPositive = value & 1;
    value += 1;
    value >>= 1;
    if (! isPositive) value = -value;

    return value;
}
```

trUnary(maxValue) Debinarization
An unsigned, integer value up to and including maxValue is coded as a string of 1 decisions. If the value is less than maxValue, a final 0 decision is coded. It is invalid to code a value greater than maxValue. nContext contexts are allocated. The contexts are assigned individually to the first nContext decisions. If the value requires more than cContext decisions, the last context is reused for all remaining decisions.

Pseudo-code implementation:

```
trUnaryDecode(contexts[], nContexts, maxValue)
{
    n = 0;
    value = 0;

    while (value < maxValue && arithDecode(contexts[n])) {
      value += 1;
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
    }
    return value;
}
```

expGolomb(k) Debinarization
An unsigned integer value is decoded. The value is decoded in two stages:

Part A:

Initilize value to zero.
While decoded binary decisions are 1, increase value by 2^k, and then increment k by 1. Exit this loop when a decoded binary decision is 0.

Part B:

For i from k down to zero, if the decoded decision is one, add 2^k to the modified value.

nContext contexts are allocated. Contexts are assigned individual to each successive decision coded in Part A; if there are more decisions than nContext, the last context is reused for the remaining decisions.

Decisions in Part B all use the equal probability context. The equal probability context is a special, single state probability finite state machine that estimates all bits as approximately 50% likely to be a 1.

```
expGolombDecode(contexts[], nContexts, k)
{
    n = 0;  // context index
    value = 0;

    while (arithDecode(contexts[n])) {
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
      value |= 1 << k;
      k += 1;
    }

    // decode low order bits
    while (k-- > 0) {
      if (arithDecode(eqProbContext)) value += (1 << k);
    }

    return value;
}
```

signedExpGolomb(k) Debinarization

A signed integer value is decoded. The value is decoded in two stages:

Part A:

Initilize value to zero.
Decode binary decision, if 1, then value is zero and conversion is complete.
Otherwise, continue by decoding a binary decision, if 1, record that the value is negative; else the value is positive.

Part B:

While decoded binary decisions are 1, increase value by 2^k, and then increment k by 1. Exit this loop when a decoded binary decision is 0.

Part C:

For i from k down to zero, if the decoded decision is one, add 2^k to the modified value.

nContext contexts are allocated. Contexts are assigned individual to each successive decision coded in Part A and Part B; if there are more decisions than nContext, the last context is reused for the remaining decisions.

Decisions in Part C all use the equal probability context. The equal probability context is a special, single state probability finite state machine that estimates all bits as approximately 50% likely to be a 1.

Pseudo-code implementation:

```
signedExpGolombDecode(contexts[], nContexts, k)
```

```
{
    n = 0;  // context index
    value = 0;
    isNegative = false;

    // zero test of value
    if (arithDecode(contexts[n])) return value;
    // advance to next context if sufficient contexts
    if (n < nContexts - 1) n++;

    // sign of value
    isNegative = arithDecode(contexts[n]);
    if (n < nContexts - 1) n++;

    while (arithDecode(contexts[n])) {
      // advance to next context if sufficient contexts
      if (n < nContexts - 1) n++;
      value |= 1 << k;
      k += 1;
    }

    // code low order bits
    while (k-- > 0) {
      if (arithDecode(eqProbContext)) value += (1 << k);
    }

    value += 1;
    if (isNegative) value = -value;

    return value;
}
```

6.2.5. Arithmetic Decoding of Flags

6.2.5.1. MacroblockSkipFlag

macroblockSkipFlag is context modeled with the context array macroblockSkipFlagContext of length 1, declared as:

```
struct ProbContext macroblockSkipFlagContext[1];
```

macroblockSkipFlag is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(macroblockSkipFlagContext);
```

6.2.5.2. lastMacroblockFlag

lastMacroblockFlag is context modeled with the context array lastMacroblockFlagContext of length 1, declared as:

```
struct ProbContext lastMacroblockFlagContext[1];
```

lastMacroblockFlag is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(lastMacroblockFlagContext);
```

6.2.5.3. blockSize4x4

blockSize4x4 is context modeled with the context array blockSize4x4Context of length 1, declared as:

```
struct ProbContext blockSize4x4Context[1];
```

blockSize4x4 is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(blockSize4x4Context);
```

6.2.5.4. BlockSkipFlag

blockSkipFlag is context modeled with the context array blockSkipFlagContext of length 1, declared as:

```
struct ProbContext blockSkipFlagContext[1];
```

blockSkipFlag is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(blockSkipFlagContext);
```

6.2.5.5. SimpleMotionPredictionFlag

simpleMotionPredictionFlag is context modeled with the context array simpleMotionPredictionFlagContext of length 1, declared as:

```
struct ProbContext simpleMotionPredictionFlagContext[1];
```

simpleMotionPredictionFlag is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(simpleMotionPredictionFlagContext);
```

6.2.5.6. eobFlag

eobFlag is context modeled with the context array eobFlagContext of length 1, declared as:

```
struct ProbContext eobFlagContext[1];
```

eobFlag is debinarized and arithmetically decoded with the **flag** debinarization invoked as:

```
flagDecode(eobFlagContext);
```

6.2.6. Subframe Overhead Data Arithmetic Coding

6.2.6.1. deltaSubframeHorizontalAddress

deltaSubframeHorizontalAddress is context modeled with the context array deltaSubframeHorizontalAddressContext of length 4, declared as:

```
struct ProbContext deltaSubframeHorizontalAddressContext[4];
```

deltaSubframeHorizontalAddress is debinarized and arithmetically decoded with the **signedUnary** debinarization invoked as:

```
signedUnaryDecode(deltaSubframeHorizontalAddressContext, 4);
```

6.2.6.2. deltaSubframeVerticalAddress

deltaSubframeVerticalAddress is context modeled with the context array deltaSubframeVerticalAddressContext of length 4, declared as:

```
struct ProbContext deltaSubframeVerticalAddressContext[4];
```

deltaSubframeVerticalAddress is debinarized and arithmetically decoded with the **unary** debinarization invoked as:

```
unaryDecode(deltaSubframeVerticalAddressContext, 4);
```

6.2.7. Macroblock Overhead Data Arithmetic Coding

6.2.7.1. Quantizer Parameter Update

quantizerParameterUpdate is context modeled with the context array quantizerParameterUpdateContext of length 4, declared as:

```
struct ProbContext quantizerParameterUpdateContext[4];
```

quantizerParameterUpdate is debinarized and arithmetically decoded with the **signedUnary** debinarization invoked as:

```
signedUnaryDecode(macroblockQuantizerScaleUpdateContext, 4);
```

6.2.8. Spatial Prediction Arithmetic Coding

6.2.8.1. Spatial Prediction Mode

spatialPredictionMode is context modeled with the context array spatialPredictionModeContext of length 2, declared as:

```
struct ProbContext spatialPredictionModeContext[2];
```

spatialPredictionMode is debinarized and arithmetically decoded with the **fixed** debinarization invoked as:

```
fixedDecode(spatialPredictionModeContext, 2);
```

6.2.8.2. Spatial Prediction Spatial Offset

spatialPredictionOffset is context modeled with the context array spatialPredictionOffsetContext of length 7, declared as:

```
struct ProbContext spatialPredictionModeContext[7];
```

spatialPredictionOffset is debinarized and arithmetically decoded with the **signedFixed** debinarization invoked as:

```
signedFixedDecode(spatialPredictionOffsetContext, 7, 7);
```

6.2.9. Motion Vector Arithmetic Coding

6.2.9.1. Differential Simple Horizontal Motion Vector

differentialSimpleHorizontalMotionVector is context modeled with the context array differentialSimpleHorizontalMotionVectorContext of length 6, declared as:

```
struct ProbContext differentialSimpleHorizontalMotionVectorContext[6];
```

differentialSimpleHorizontalMotionVector is debinarized and arithmetically decoded with the **signedExpGolomb de**binarization invoked as:

```
signedExpGolombEncode(differentialSimpleHorizontalMotionVectorContext, 6,
0);
```

6.2.9.2. Differential Simple Vertical Motion Vector

differentialSimpleVerticalMotionVector is context modeled with the context array differentialSimpleVerticalMotionVectorContext of length 6, declared as:

```
struct ProbContext differentialSimpleVerticalMotionVectorContext[6];
```

differentialSimpleVerticalMotionVector is debinarized and arithmetically decoded with the **signedExpGolomb** binarization invoked as:

```
signedExpGolombDecode(differentialSimpleVerticalMotionVectorContext, 6,
0);
```

6.2.10. DCT Coefficient Arithmetic Coding

6.2.10.1. Coding of Differential Coefficients

PredictedCoefficientError is context modeled with the following context arrays:

For predictedCoefficientError when contained in 8x8 luma blocks:

```
struct ProbContext luma8x8Context[33][6];
```

For predictedCoefficientError when contained in 4x4 luma blocks:

```
struct ProbContext luma4x4Context[9][6];
```

For predictedCoefficientError when contained in 4x4 Cb or Cr chroma blocks:

```
struct ProbContext chroma4x4Context[9][6];
```

For predictedCoefficientError when contained in 2x2 Cb or Cr chroma blocks:

```
struct ProbContext chroma4x4Context[4][6];
```

The i'th **predictedCoefficientError** ordered as they appear in the coded bitstream is debinarized and arithmetically decoded with the **signedExpGolomb** debinarization invoked as follows:

For predictedCoefficientError when contained in 8x8 luma blocks:

```
signedExpGolombDecode(luma8x8Context[i < 32 ? i : 32], 6, 0);
```

For predictedCoefficientError when contained in 4x4 luma blocks:

```
signedExpGolombDecode(luma4x4Context[i < 8 ? i : 8], 6, 0);
```

For predictedCoefficientError when contained in 4x4 Cb or Cr chroma blocks:

```
signedExpGolombDecode(chroma4x4Context[i < 8 ? i : 8], 6, 0);
```

For predictedCoefficientError when contained in 2x2 Cb or Cr chroma blocks:

```
signedExpGolombDecode(chroma2x2Context[i], 6, 0);
```

6.3. Inverse Quantization and Transformation

6.3.1. Inverse Zig-Zag Scanning

Inverse zig-zag scanning is carried out prior to inverse quantization. This process generates a 2-D matrix of data values from a 1-D list for each of the 2x2, 4x4 and 8x8 blocks of data. Figure 6.1 shows the 4 values in the 1-D list that is re-ordered into the 2x2 block. Figure 6.2 shows the 16 values in the 1-D list that is re-ordered into the 4x4 block. Figure 6.3 shows the 64 values that are re-ordered into the 8x8 block.

0	1
2	3

Figure 6.1: Re-ordering of 4 values into 2x2 block

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

Figure 6.2: Re-ordering of 16 values into 4x4 block

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure 6.3: Re-ordering of 64 values into 8x8 block

There are two possible mechanisms which control the choice of **quantizerParameter**. One mechanism is to set the value for **quantizerParameter** before the encoding process starts by user input (for example by specifying a parameter in a configuration file) or a parameter set by the program or device starting the encoding process. The other mechanism allows changing **quantizerParameter** at the frame, subframe or macroblock level by rate control in the encoder itself, for example when trying to achieve a constant

6.3.2. Inverse Quantization

The quantization used in OMS video is an adaptive linear quantization controlled by a variable stepsize. The stepsize is specified in the bitstream at the frame level by a 6-bit non-linear, table driven quantization parameter, **quantizerParameter**. This parameter can be updated with a signed differential value, **quantizerParameterUpdate** at the subframe and macroblock levels.

Let prevQuantizerParameter be the quantizerParameter from the previous macroblock or subframe. Then quantizerParameter is updated according to the following equation:

quantizerParameter = prevQuantizerParameter + **quantizerParameterUpdate**

The linear quantizer inverse operator is

dequantization : RDEQ = (RQ * quantizerScale) >> 10

where quantizerScale is the quantizer stepsize derived from **quantizerParameter** as specified in Annex-B.

Pseudo-code is available in Annex-B.

6.3.3. Inverse Transformation

The transforms used are 2x2, 4x4 and 8x8 integer IDCT transforms, which are approximations to the traditional IDCT. For simplicity, they are called IDCT in this specification.

The input to the IDCT is a 2x2, 4x4 or 8x8 block of 9-bit, 11-bit or 15-bit transform coefficients respectively. The output consists of a 2x2, 4x4 or 8x8 block of 9-bit values. The input values and the output values are stored in 32-bit integer format, and occupy the least significant bits in the 32-bit integer.

Annex-A contains the precise algorithms an implementation shall follow. Variations in the order in which operations are performed are allowed as long as the results are identical to the results of the algorithms described in Annex-A

6.4. Temporal Prediction

6.4.1. Macroblock Modes

The macroblock type shown in Table 6.1 determines the macroblock motion mode. In motion compensated subframes macroblocks may be spatial predicted or temporal predicted. Spatial predicted macroblocks are always coded and are decoded using the procedures of Section 6.5.

Temporal predicted macroblocks may be coded or skipped, depending on whether there is coded motion vector data and coded predicted error (residual) data. These modes and conditions are summarized in Table 6.2.

Macroblock Mode	Conditions	Signaling
Skipped	Zero differential motion vector Zero residual data	**macroblockSkipFlag**=1
Coded	See Block modes At least one block has non-zero differential motion vector or residual data	**macroblockSkipFlag**=0

Table 6.2: Macroblock Coding Modes

If a macroblock is skipped, all motion vectors for the macroblock shall be set according to the following equations:

```
reconstructedHorizontalMotionVector = previousHorizontalMotionVector

reconstructedVerticalMotionVector = previousVerticalMotionVector
```

At the left edge of a subframe or following a spatial predicted macroblock the previous motion vector shall be zero:

```
previousHorizontalMotionVector = 0
```

```
previousVerticalMotionVector = 0
```

6.4.2. Block Modes

Spatial predicted blocks are always coded and are decoded using the procedures of Section 6.5. Temporal predicted blocks may be coded according to several modes, depending on whether there is coded motion vector data and coded predicted error data. These modes and conditions are summarized in Table 6.3.

Block Mode	Conditions	Signaling
Skipped	Zero differential motion vector && Zero residual data	**blockSkipFlag**=1
Coded differential motion vector only	Coded differential motion vector && Zero residual data	**simpleMotionPredictionFlag** = 1 1st **eobFlag** = 1
Coded	Coded differential motion vector && Coded residual data	**simpleMotionPredictionFlag** = 1 1st **eobFlag** = 0

Table 6.3: Block Coding Modes

If a block is skipped, motion vectors for the block shall be set according to the following equations:

```
reconstructedHorizontalMotionVector = previousHorizontalMotionVector

reconstructedVerticalMotionVector = previousVerticalMotionVector
```

At the left edge of a subframe or following a spatial predicted block the previous motion vector shall be zero:

```
previousHorizontalMotionVector = 0

previousVerticalMotionVector = 0
```

Chrominance motion vector is half the luminance motion vector.

6.4.3. Motion Vector Reconstruction

Reconstructed motion vectors for 4x4 luminance blocks are predicted from the motion vectors in the previous block according to the block order defined in Figure 3.7. The motion vector for the first block (1) in a macroblock is predicted from the motion vector of the last block (4) in the previous macroblock. Reconstructed motion vectors for 8x8 luminance blocks are predicted from the motion vectors in the previous macroblock. Prediction is performed according to the following equations:

```
reconstructedHorizontalMotionVector = previousHorizontalMotionVector +
                                      differentialSimpleHorizontalMotionVector

reconstructedVerticalMotionVector = previousVerticalMotionVector +
```

```
                                        differentialSimpleVerticalMotionVector

previousHorizontalMotionVector = reconstructedHorizontalMotionVector

previousVerticalMotionVector = reconstructedVerticalMotionVector
```

Motion vector predictors are set to zero for the following conditions:

- First block in a subframe
- Macroblocks on the left edge of a subframe
- Macroblocks following a spatial-predicted macroblock

According to the following equations:

previousHorizontalMotionVector = 0
previousVerticalMotionVector = 0:

pseudo-code for temporal prediction:

```
motionPrediction(simpleMotionPredictionFlag, blockSkipFlag,
differentialSimpleHorizontalMotionVector,
differentialSimpleVerticalMotionVector)
{
     if (simpleMotionPredictionFlag  == 1) { // Current block motion
                                             //  compensated.
           if (blockSkipFlag == 1) { // Current block is skipped

                  reconstructedHorizontalMotionVector =
                                        previousHorizontalMotionVector;

                 reconstructedVerticalMotionVector =
                             previousVerticalMotionVector;
           }
           else if ( simpleMotionPredictionFlag == 0) { // Current block
                                                        // is not skipped
                 reconstructedHorizontalMotionVector =
                              previousHorizontalMotionVector +
                              differentialSimpleHorizontalMotionVector;

                      reconstructedVerticalMotionVector =
                               previousVerticalMotionVector +
                               differentialSimpleVerticalMotionVector;
           }

           referenceLuminanceBlock = getReferenceLuminanceBlock(
                             reconstructedHorizontalMotionVector,
                             reconstructedVerticalMotionVector);

           referenceChrominanceBlock = getReferenceChrominanceBlock(
                             reconstructedHorizontalMotionVector/2,
                             reconstructedVerticalMotionVector/2);
     }
}
```

6.5. Spatial Compensation

6.5.1. Spatial Compensation of Luminance Data

For any 4x4 or 8x8 block flagged for spatial prediction compensation of luminance blocks shall be performed. Only neighboring blocks located within the current subframe shall be used in the spatial compensation process. Reference blocks shall be the same size as the residual.

If **spatialPredictionMode** equals 0, then DC prediction shall be performed. The DC mean value shall be computed using all luminance pixel values found in the neighboring reconstructed blocks. If one or more neighboring blocks are unavailable, the mean value shall be computed with the remaining neighboring blocks. If no neighboring blocks are available, the value of 128 must be used as the mean value to construct the DC reference block. All elements in the DC reference block shall be equal to the DC mean value.

If **spatialPredictionMode** equals 1, then spatial prediction shall be performed in the vertical direction and the **spatialPredictionSpatialOffset** is used as the vertical offset to locate the reference block. If **spatialPredictionMode** equals 2, then spatial prediction shall performed in the horizontal direction and the **spatialPredictionSpatialOffset** is used as the horizontal offset to locate the reference block.

Pseudo-code implementation:

```
spatialPredictionLuminance(spatialPredictionMode,
      spatialPredictionSpatialOffset)
{
      if(spatialPredictionMode == 0)  {  // DC mode
            referenceBlock = computeLuminanceDC(currentBlock);
      }
      else if(spatialPredictionMode == 1) { //  Vertical
            referenceBlock = getNeighborVerticalLuminance(currentBlock,
                              spatialPredictionSpatialOffset);
      }
      else if(spatialPredictionMode == 2) { //  Horizontal
            referenceBlock = getNeighborHorizontalLuminance(currentBlock,
                              spatialPredictionSpatialOffset);
      }
      return referenceLuminanceBlock;
}
```

6.5.2. Spatial Prediction of Chrominance Data

For each block if spatial prediction is indicated, then the **spatialPredictionMode** shall determine whether spatial prediction will be carried out with DC, vertical offset or horizontal offset.
Where there is one luminance block of 8x8 pixels in a macroblock, if **spatialPredictionMode** is set to 0 then DC prediction shall be applied to the corresponding 4x4 chrominance pixel block of C_b and C_r. If **spatialPredictionMode** is set to 1 then **spatialPredictionSpatialOffset** corresponding to the vertical offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 4x4 chrominance pixel block of C_b and C_r. If **spatialPredictionMode** is set to 2 then **spatialPredictionSpatialOffset** corresponding to the horizontal offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to each 4x4 chrominance pixel block of C_b and C_r.

Where there are 4 luminance blocks in a macroblock, where each block consists of 4x4 pixels, then each chrominance block of 4x4 pixels shall be treated as 4 "sub-blocks" of 2x2 chrominance pixels for purposes of spatial prediction or motion compensated prediction. A chrominance sub-block in this

context means the chrominance pixels that correspond spatially to each of the luminance blocks. If **spatialPredictionMode** is set to 0 then DC prediction shall be applied to the corresponding 2x2 chrominance pixel sub-block of C_b and C_r. If **spatialPredictionMode** is set to 1 then **spatialPredictionSpatialOffset** corresponding to the vertical offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to the corresponding 2x2 chrominance pixel sub-block of C_b and C_r. If **spatialPredictionMode** is set to 2 then **spatialPredictionSpatialOffset** corresponding to the horizontal offset shall be scaled by dividing by 2 and then spatial prediction shall be applied to the corresponding 2x2 chrominance pixel sub-block of C_b and C_r.

The DC predictor block for each chrominance shall be equivalent in size as the chrominance block, and its value must uniformly consist of its respective chrominance DC predictor value. The chrominance DC predictor value shall be obtained by computing the mean of the corresponding chrominance pixel values located in the adjacent reconstructed blocks. Any chrominance pixels which are not available must not be included in the computation of the mean.

Pseudo-code implementation:

```
spatialPredictionChrominance(spatialPredictionMode,
      spatialPredictionSpatialOffset)
{
      if(spatialPredictionMode == 0)  {  // DC mode
            referenceBlock = computeChrominanceDC(currentBlock);
      }
      else if(spatialPredictionMode == 1) { //  Vertical
             referenceBlock = getNeighborVerticalChrominance(currentBlock,
                                    spatialPredictionSpatialOffset/2);
      }
      else if(spatialPredictionMode == 2) { //  Horizontal
            referenceBlock = getNeighborHorizontalLuminance(currentBlock,
                                    spatialPredictionSpatialOffset/2);
      }
      return referenceChrominanceBlock;
}
```

6.6. Frame Reconstruction

The bitstream shall be decoded and frame reconstruction shall be carried out. These frames shall be then available for presentation.

Each block in either a macroblock that is not skipped (i.e. macroblockSkipFlag = 0) or for each block that is not skipped (i.e. blockSkipFlag = 0) blocks are reconstructed according to equation 6.1 and 6.2.

currentLuminanceBlock = referenceLuminanceBlock + residual 6.1

currentChrominanceBlock = referenceChrominanceBlock + residual 6.2

Each block in either a macroblock that is skipped (i.e. macroblockSkipFlag = 1) or for each block that is skipped (i.e. blockSkipFlag = 1) blocks are reconstructed according to equation 6.3 and 6.4.

currentLuminanceBlock = referenceLuminanceBlock 6.3

currentChrominanceBlock = referenceChrominanceBlock 6.4

6.7. Bitrate Control

The OMS Video Version 1 specification shall support the generation of variable bit rate bitstreams. These will enable specifying the maximum bitrate.

7. Normative Annexes

Annex-A

Pseudo C Code for DCT/IDCT Functions

The algorithms for the 2x2, 4x4 and 8x8 direct and inverse DCT transforms are expressed as pseudo C code. The forward transforms perform the operation:

$$X \rightarrow A * Y * A^t$$

while the inverse transforms perform the operation:

$$Y \rightarrow A^t * X * A$$

for a transform matrix A, and where A^t represents the transpose of A. The coefficients of the matrix A are chosen in such a way as to make these transforms approximations of the "classical" 2-D DCT and IDCT operators. Note that all operations are performed with 32-bit integer arithmetic.

Below is the pseudo-code for the six functions:

```
/*
 ************************************************************************
 * Copyright © 2008 Sun Microsystems, Inc.
 * 4150 Network Circle
 * Santa Clara, California 95054, U.S.A.
 * All rights reserved.
 ************************************************************************
 * /

/*
 ************************************************************************
 *
 * 1. matrixDct1Impl(8x8|4x4) is based on the method by Chen et al.
 *    [Chen1977] that has been implemented as part of the mediaLib
 *    libraries by Sun Microsystems Inc. [MLIB]
 *
 * 2. The matrixDct1Impl(8x8|4x4) is the "scale" integer version of
 *    Chen et al. [Chen1977].
 *
 *    <1> matrixDct1Impl DCT(8x8|4x4) were obtained from mediaLib
 *    functions:
 *    mlib_i_VideoDCT8x8.c, mlib_i_DCT4x4_S16_S16.c [MLIB]
 *
 *    <2> matrixDct1Impl IDCT (8x8|4x4) were written based on
 *    the mediaLib functions and on Chen et al. [Chen1977]
 *
 *    <3> These DCT/IDCT transforms use 32-bit variables.
 *
 *    <4> There are 16 MULs and 26 ADD operations in 8-way matrixDct1Impl
 *
 * 3. Derivation of the values CA0-CA6 and FCA3
 *    The 8-way DCT/IDCT has the following coefficients:
 *    FA0 = 0.49039264020162
 *    FA1 = 0.46193976625564
 *    FA2 = 0.41573480615127
```

```
 *    FA3 = 0.35355339059327
 *    FA4 = 0.27778511650980
 *    FA5 = 0.19134171618254
 *    FA6 = 0.09754516100806
 *    Multiplying scalar value 32768 to each of the above gives CA0-CA6.
 *    FCA3 is the result of multiplying 32768 * cos(pi/4)
 *
 * 4. Derivation of values b & c in 4x4 DCT/IDCT
 *    The 4-way DCT/IDCT has the following coefficients:
 *    FA0 = 0.50000000000000
 *    FA1 = 0.65328148243819
 *    FA2 = 0.27059805007310
 *    Multiplying scalar values 8192 to FA1 and FA2 gives b and c.
 *
 * 5. FA0 * 8192 = 4096 in 4. above
 *    In matrixDct1Impl(4x4) scaling is carried out as follows:
 *    ((c0 + c1) * 4096)) >> 12 = c0 + c1;
 *    By this scaling method multiplication operations are reduced
 ************************************************************************
 */

/*
 *
 ************************************************************************
 *  2x2 Integer DCT transform
 *
 * Input: x[2][2]
 *
 * Output: y[2][2]
 ************************************************************************
 */
void oms_dct_2x2(short *x, int xstride, short *y, int ystride) {
      y[0]=(x[0] + x[1] + x[xstride] + x[1 + xstride]) >> 1;
      y[1]=(x[0] - x[1] + x[xstride] - x[1 + xstride]) >> 1;
      y[ystride]=(x[0] + x[1] - x[xstride] - x[1 + xstride]) >> 1;
      y[1 + ystride]=(x[0] - x[1] - x[xstride] + x[1 + xstride]) >> 1;
}

/*
 *
************************************************************************
 *  2x2 Integer IDCT transform
 *
 * Input: x[2][2]
 *
 * Output: y[2][2]
 ************************************************************************
 */
void oms_idct_2x2(short *x, int xstride, short *y, int ystride) {
      y[0]=(x[0] + x[1] + x[xstride] + x[1 + xstride]) >> 1;
      y[1]=(x[0] - x[1] + x[xstride] - x[1 + xstride]) >> 1;
      y[ystride]=(x[0] + x[1] - x[xstride] - x[1 + xstride]) >> 1;
      y[1 + ystride]=(x[0] - x[1] - x[xstride] + x[1 + xstride]) >> 1;
}

/
 ************************************************************************
 *    4x4 integer DCT transform
 *
```

```
 *    Input: x[4][4]
 *
 *    Output: y[4][4]
 ************************************************************************
 */

/* ************************************************************ */
#define  ROUND(a)  (((a) + 2048) >> 12)
/* ************************************************************ */

void oms_dct_4x4(int x[4][4], int y[4][4]) {
  int a00, a10, a20, a30;
  int a01, a11, a21, a31;
  int a02, a12, a22, a32;
  int a03, a13, a23, a33;

  int b00, b10, b20, b30;
  int b01, b11, b21, b31;
  int b02, b12, b22, b32;
  int b03, b13, b23, b33;

  int b = 5351;
  int c = 2217;

  a00 = (int)x[0][0] + (int)x[0][3];
  a03 = (int)x[0][0] - (int)x[0][3];
  a01 = (int)x[0][1] + (int)x[0][2];
  a02 = (int)x[0][1] - (int)x[0][2];

  a10 = (int)x[1][0] + (int)x[1][3];
  a13 = (int)x[1][0] - (int)x[1][3];
  a11 = (int)x[1][1] + (int)x[1][2];
  a12 = (int)x[1][1] - (int)x[1][2];

  a20 = (int)x[2][0] + (int)x[2][3];
  a23 = (int)x[2][0] - (int)x[2][3];
  a21 = (int)x[2][1] + (int)x[2][2];
  a22 = (int)x[2][1] - (int)x[2][2];

  a30 = (int)x[3][0] + (int)x[3][3];
  a33 = (int)x[3][0] - (int)x[3][3];
  a31 = (int)x[3][1] + (int)x[3][2];
  a32 = (int)x[3][1] - (int)x[3][2];

  b00 = a00 + a01;
  b02 = a00 - a01;
  b01 = ROUND(a03 * b + a02 * c);
  b03 = ROUND(a03 * c - a02 * b);

  b10 = a10 + a11;
  b12 = a10 - a11;
  b11 = ROUND(a13 * b + a12 * c);
  b13 = ROUND(a13 * c - a12 * b);

  b20 = a20 + a21;
  b22 = a20 - a21;
  b21 = ROUND(a23 * b + a22 * c);
  b23 = ROUND(a23 * c - a22 * b);
```

```
    b30 = a30 + a31;
    b32 = a30 - a31;
    b31 = ROUND(a33 * b + a32 * c);
    b33 = ROUND(a33 * c - a32 * b);

    a00 = b00 + b30;
    a03 = b00 - b30;
    a01 = b10 + b20;
    a02 = b10 - b20;

    a20 = b02 + b32;
    a23 = b02 - b32;
    a21 = b12 + b22;
    a22 = b12 - b22;

    a10 = b01 + b31;
    a13 = b01 - b31;
    a11 = b11 + b21;
    a12 = b11 - b21;

    a30 = b03 + b33;
    a33 = b03 - b33;
    a31 = b13 + b23;
    a32 = b13 - b23;

    y[0][0] = (int)(a00 + a01) >> 2;
    y[2][0] = (int)(a00 - a01) >> 2;
    y[1][0] = (int)(ROUND(a03 * b + a02 * c)) >> 2;
    y[3][0] = (int)(ROUND(a03 * c - a02 * b)) >> 2;

    y[0][1] = (int)(a10 + a11) >> 2;
    y[2][1] = (int)(a10 - a11) >> 2;
    y[1][1] = (int)(ROUND(a13 * b + a12 * c)) >> 2;
    y[3][1] = (int)(ROUND(a13 * c - a12 * b)) >> 2;

    y[0][2] = (int)(a20 + a21) >> 2;
    y[2][2] = (int)(a20 - a21) >> 2;
    y[1][2] = (int)(ROUND(a23 * b + a22 * c)) >> 2;
    y[3][2] = (int)(ROUND(a23 * c - a22 * b)) >> 2;

    y[0][3] = (int)(a30 + a31) >> 2;
    y[2][3] = (int)(a30 - a31) >> 2;
    y[1][3] = (int)(ROUND(a33 * b + a32 * c)) >> 2;
    y[3][3] = (int)(ROUND(a33 * c - a32 * b)) >> 2;

}

/*
 ****************************************************************************
 *    4x4 IDCT transform
 *
 *    Input: x[4][4]
 *
 *    Output: y[4][4]
 ****************************************************************************
 */

void oms_idct_4x4(int x[4][4], int y[4][4]) {
  int a00, a10, a20, a30;
```

```
        int a01, a11, a21, a31;
        int a02, a12, a22, a32;
        int a03, a13, a23, a33;

        int b00, b10, b20, b30;
        int b01, b11, b21, b31;
        int b02, b12, b22, b32;
        int b03, b13, b23, b33;

        int b = 5351;
        int c = 2217;

        a00 = x[0][0] + x[0][2];
        a01 = x[0][0] - x[0][2];
        a02 = ROUND(x[0][1] * c - x[0][3] * b);
        a03 = ROUND(x[0][1] * b + x[0][3] * c);

        a10 = x[1][0] + x[1][2];
        a11 = x[1][0] - x[1][2];
        a12 = ROUND(x[1][1] * c - x[1][3] * b);
        a13 = ROUND(x[1][1] * b + x[1][3] * c);

        a20 = x[2][0] + x[2][2];
        a21 = x[2][0] - x[2][2];
        a22 = ROUND(x[2][1] * c - x[2][3] * b);
        a23 = ROUND(x[2][1] * b + x[2][3] * c);

        a30 = x[3][0] + x[3][2];
        a31 = x[3][0] - x[3][2];
        a32 = ROUND(x[3][1] * c - x[3][3] * b);
        a33 = ROUND(x[3][1] * b + x[3][3] * c);

        b00 = a00 + a03;
        b01 = a01 + a02;
        b02 = a01 - a02;
        b03 = a00 - a03;

        b10 = a10 + a13;
        b11 = a11 + a12;
        b12 = a11 - a12;
        b13 = a10 - a13;

        b20 = a20 + a23;
        b21 = a21 + a22;
        b22 = a21 - a22;
        b23 = a20 - a23;

        b30 = a30 + a33;
        b31 = a31 + a32;
        b32 = a31 - a32;
        b33 = a30 - a33;

        a00 = b00 + b20;
        a01 = b00 - b20;
        a02 = ROUND(b10 * c - b30 * b);
        a03 = ROUND(b10 * b + b30 * c);

        a10 = b01 + b21;
        a11 = b01 - b21;
```

```
    a12 = ROUND(b11 * c - b31 * b);
    a13 = ROUND(b11 * b + b31 * c);
    a20 = b02 + b22;
    a21 = b02 - b22;
    a22 = ROUND(b12 * c - b32 * b);
    a23 = ROUND(b12 * b + b32 * c);

    a30 = b03 + b23;
    a31 = b03 - b23;
    a32 = ROUND(b13 * c - b33 * b);
    a33 = ROUND(b13 * b + b33 * c);

    y[0][0] = (a00 + a03) >> 2;
    y[1][0] = (a01 + a02) >> 2;
    y[2][0] = (a01 - a02) >> 2;
    y[3][0] = (a00 - a03) >> 2;

    y[0][1] = (a10 + a13) >> 2;
    y[1][1] = (a11 + a12) >> 2;
    y[2][1] = (a11 - a12) >> 2;
    y[3][1] = (a10 - a13) >> 2;

    y[0][2] = (a20 + a23) >> 2;
    y[1][2] = (a21 + a22) >> 2;
    y[2][2] = (a21 - a22) >> 2;
    y[3][2] = (a20 - a23) >> 2;

    y[0][3] = (a30 + a33) >> 2;
    y[1][3] = (a31 + a32) >> 2;
    y[2][3] = (a31 - a32) >> 2;
    y[3][3] = (a30 - a33) >> 2;

}

/*
 ************************************************************************
 *    8x8 integer DCT transform
 *
 *    Input: x[8][8]
 *
 *    Output: y[8][8]
 ************************************************************************
 */


/* ************************************************************* */
#ifdef ROUND
#undef ROUND
#endif
#define  ROUND(a)   (((a) + (1<<14))>> 15)
/* ************************************************************* */
void oms_dct_8x8(int x[8][8], int y[8][8]) {
  int a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;
  int b0, b1, b2, b3, b4, b5, b6, b7;
  int c[8][8];
  int i;
  int CA0 = 16069, CA1 = 15137, CA2 = 13623,
    CA3 = 11585, CA4 = 9102, CA5 = 6270, CA6 = 3196, FCA3 = 23170;
```

```
        for( i=0; i<8; i++)
        {

          a0 = x[i][0] + x[i][7];
          a1 = x[i][1] + x[i][6];
          a2 = x[i][2] + x[i][5];
          a3 = x[i][3] + x[i][4];

          a4 = x[i][0] - x[i][7];
          a5 = x[i][1] - x[i][6];
          a6 = x[i][2] - x[i][5];
          a7 = x[i][3] - x[i][4];

          b0 = a0 + a3;
          b1 = a1 + a2;
          b2 = a0 - a3;
          b3 = a1 - a2;

          c[0][i] = ROUND((b0 + b1) * CA3);
          c[2][i] = ROUND(b2 * CA1 + b3 * CA5);
          c[4][i] = ROUND((b0 - b1) * CA3);
          c[6][i] = ROUND(b2 * CA5 - b3 * CA1);

          a8 = ROUND((a5 + a6) * FCA3);
          a9 = ROUND((a5 - a6) * FCA3);

          b4 = a4 + a8;
          b5 = a7 + a9;
          b6 = a4 - a8;
          b7 = a7 - a9;

          c[1][i] = ROUND(b4 * CA0 + b5 * CA6);
          c[3][i] = ROUND(b6 * CA2 - b7 * CA4);
          c[5][i] = ROUND(b6 * CA4 + b7 * CA2);
          c[7][i] = ROUND(b4 * CA6 - b5 * CA0);
        }

        for( i=0; i<8; i++)
        {
          a0 = c[i][0] + c[i][7];
          a1 = c[i][1] + c[i][6];
          a2 = c[i][2] + c[i][5];
          a3 = c[i][3] + c[i][4];

          a4 = c[i][0] - c[i][7];
          a5 = c[i][1] - c[i][6];
          a6 = c[i][2] - c[i][5];
          a7 = c[i][3] - c[i][4];

          b0 = a0 + a3;
          b1 = a1 + a2;
          b2 = a0 - a3;
          b3 = a1 - a2;

          y[0][i] = ROUND((b0 + b1) * CA3);
          y[2][i] = ROUND(b2 * CA1 + b3 * CA5);
          y[4][i] = ROUND((b0 - b1) * CA3);
          y[6][i] = ROUND(b2 * CA5 - b3 * CA1);
```

```
      a8 = ROUND((a5 + a6) * FCA3);
      a9 = ROUND((a5 - a6) * FCA3);

      b4 = a4 + a8;
      b5 = a7 + a9;
      b6 = a4 - a8;
      b7 = a7 - a9;

      y[1][i] = ROUND(b4 * CA0 + b5 * CA6);
      y[3][i] = ROUND(b6 * CA2 - b7 * CA4);
      y[5][i] = ROUND(b6 * CA4 + b7 * CA2);
      y[7][i] = ROUND(b4 * CA6 - b5 * CA0);
    }

  }

  /*
   ************************************************************************
   *    8x8 integer IDCT transform
   *
   *    Input: x[8][8]
   *
   *    Output: y[8][8]
   ************************************************************************
   */

  void oms_idct_8x8(int x[8][8], int y[8][8]) {
    int a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;
    int b0, b1, b2, b3, b4, b5, b6, b7;
    int c[8][8];
    int i;
    int CA0 = 16069, CA1 = 15137, CA2 = 13623,
      CA3 = 11585, CA4 = 9102, CA5 = 6270, CA6 = 3196, FCA3 = 23170;

    for( i=0; i<8; i++)
    {
      a0 = ROUND((x[i][0] + x[i][4]) * CA3);
      a4 = ROUND((x[i][0] - x[i][4]) * CA3);
      a2 = ROUND(x[i][2] * CA5 - x[i][6] * CA1);
      a6 = ROUND(x[i][2] * CA1 + x[i][6] * CA5);

      a8 = ROUND((x[i][3] + x[i][5]) * FCA3);
      a9 = ROUND((x[i][3] - x[i][5]) * FCA3);

      a1 = x[i][1] + a8;
      a3 = x[i][7] + a9;
      a5 = x[i][1] - a8;
      a7 = x[i][7] - a9;

      b0 = a0 + a6;
      b2 = a4 + a2;
      b4 = a4 - a2;
      b6 = a0 - a6;

      b1 = ROUND(a1 * CA6 - a3 * CA0);
      b7 = ROUND(a1 * CA0 + a3 * CA6);
      b3 = ROUND(a5 * CA4 + a7 * CA2);
      b5 = ROUND(a5 * CA2 - a7 * CA4);
```

```
            c[0][i] = b0 + b7;
            c[1][i] = b2 + b5;
            c[2][i] = b4 + b3;
            c[3][i] = b6 + b1;
            c[4][i] = b6 - b1;
            c[5][i] = b4 - b3;
            c[6][i] = b2 - b5;
            c[7][i] = b0 - b7;
          }

          for( i=0; i<8; i++)
          {
            a0 =  ROUND((c[i][0] + c[i][4]) * CA3);
            a4 =  ROUND((c[i][0] - c[i][4]) * CA3);
            a2 =  ROUND(c[i][2] * CA5 - c[i][6] * CA1);
            a6 =  ROUND(c[i][2] * CA1 + c[i][6] * CA5);

            a8 = ROUND((c[i][3] + c[i][5]) * FCA3);
            a9 = ROUND((c[i][3] - c[i][5]) * FCA3);

            a1 = c[i][1] + a8;
            a3 = c[i][7] + a9;
            a5 = c[i][1] - a8;
            a7 = c[i][7] - a9;

            b0 = a0 + a6;
            b2 = a4 + a2;
            b4 = a4 - a2;
            b6 = a0 - a6;

            b1 = ROUND(a1 * CA6 - a3 * CA0);
            b7 = ROUND(a1 * CA0 + a3 * CA6);
            b3 = ROUND(a5 * CA4 + a7 * CA2);
            b5 = ROUND(a5 * CA2 - a7 * CA4);

            y[0][i] = (b0 + b7);
            y[1][i] = (b2 + b5);
            y[2][i] = (b4 + b3);
            y[3][i] = (b6 + b1);
            y[4][i] = (b6 - b1);
            y[5][i] = (b4 - b3);
            y[6][i] = (b2 - b5);
            y[7][i] = (b0 - b7);

          }
        }
```

Annex-B

Pseudo C code for Quantization and dequantization functions

The quantization and dequantization functions are described explicitly below. This is a linear quantizer with a 6-bit non-linear quantization control. The **quantizerParameter** is in the range 0 to 56, Define

quantScale[7] = {1024, 1152, 1280, 1408, 1664, 1792, 1920}
quantOffset[7] = {250, 281, 312, 343, 406, 437, 468}

and

quantizerScale = quantScale[**quantizerParameter** % 7] << (**quantizerParameter** / 7)
quantizerOffset = 1 + (quantOffset[**quantizerParameter** % 7] >> (10 - **quantizerParameter** / 7))

The function round() is defined as follows

round(R, quantizerScale) = ((R + quantizerOffset) << 10) / quantizerScale

At the encoder, quantization is performed according to the following equation:

RQ = round(R, quantizerScale)

In the decoder section of the prediction loop at the encoder and at the decoder, inverse quantization is performed according to the following equation:

RDEQ = (RQ * quantizerScale) >> 10

Annex-C

Subpixel Interpolation of Luminance

For a given set of full pixel position

$$Z \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\}$$

interpolation shall be carried out at subpixel position

$$S \in \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o\}$$

as depicted in Figure C.1.

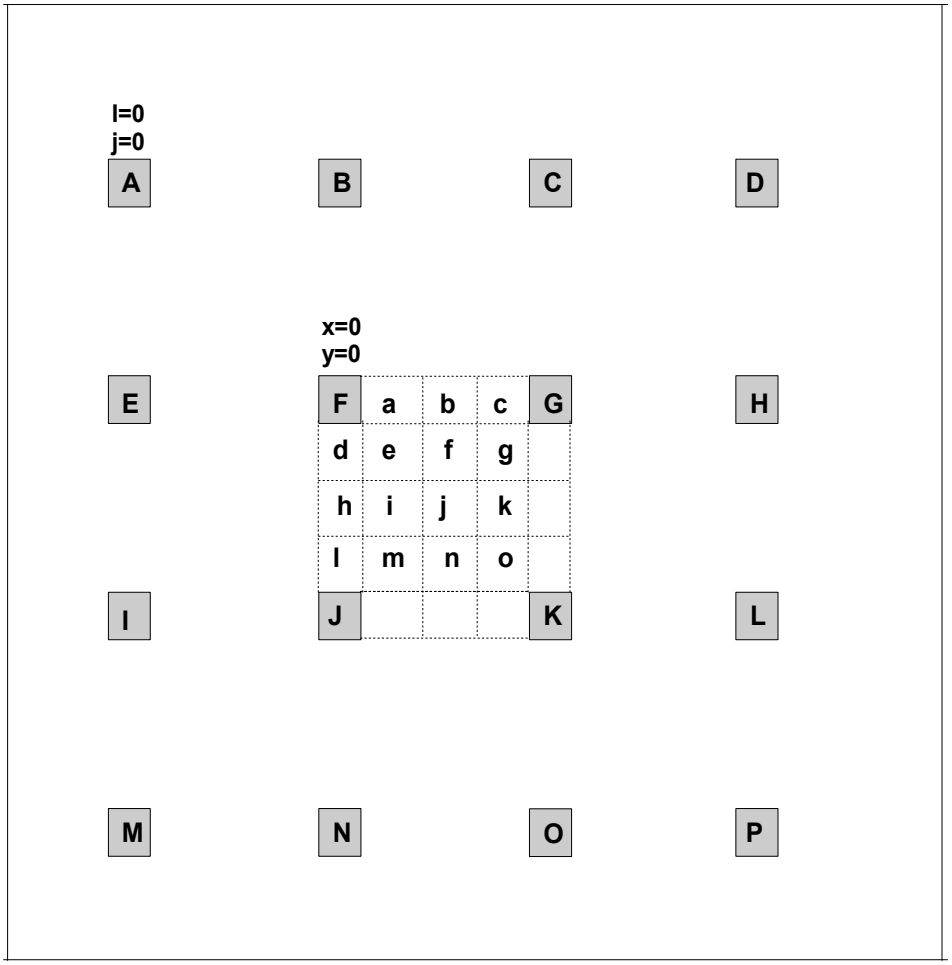

Figure C.1: Location of full and subpixel positions

The following is the generalized form of the cubic spline function:

$$F(x) = (a+2)\left|x^3\right| - (a+3)\left|x^2\right| + 1 \qquad \text{for } 0 \le |x| < 1$$

$$F(x) = a\left|x^3\right| - 5a\left|x^2\right| + 8a\left|x\right| - 4a \qquad \text{for } 1 \le |x| < 2$$

$$F(x) = 0 \qquad \text{for } |x| \ge 2$$

where ***a*** = -0.5.

The bicubic spline function derived from above is shown in Figure C.2

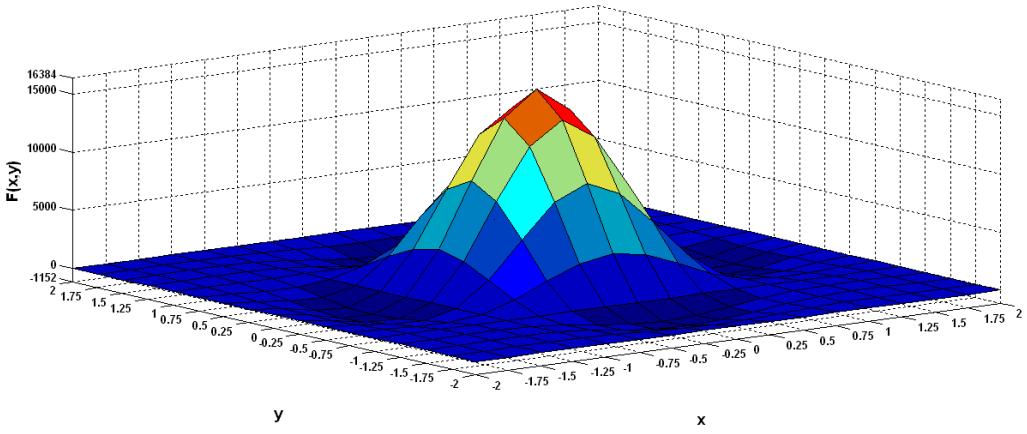

Figure C.2: Bicubic spline kernel function

For any member of subpixel **S**, the interpolation shall use a weighted sum of the neighboring full pixels **Z**. There is 4-fold symmetry with the 2-dimensional spline function, hence one quadrant of the filter coefficients:

16384	14208	9216	3712	0	-1152	-1024	-384	0
14208	12321	7992	3219	0	-999	-888	-333	0
9216	7992	5184	2088	0	-648	-576	-216	0
3712	3219	2088	841	0	-261	-232	-87	0
0	0	0	0	0	0	0	0	0
-1152	-999	-648	-261	0	81	72	27	0
-1024	-888	-576	-232	0	72	64	24	0
-384	-333	-216	-87	0	27	24	9	0
0	0	0	0	0	0	0	0	0

Figure C.3: The filter coefficients are scaled by a factor of 16384.

In Figure C.3 a full pixel position, fullPixel[j][i], may take any integer pixel position {A,....P}. For example, pixel B will have pixel value defined as fullPixel[0][1].

A subpixel S, where S may take values {**a, b, c, d, e, f, g, h, i, j, k, l, m, n, o**} are defined in the horizontal and vertical directions with coordinates x,y. In the general case, subPixel[y][x]. For example, subPixel, **e** will have a value defined as subPixel[1][1].

The value of the subpixel S, are computed from the summation of the products of full pixel values and their corresponding filter coefficients derived from the Figure C.3.

index and indexY define the indices for accessing the filter coefficients.

```
indexX = {x+4, x, 4-x, 8-x};
indexY = (y+4, y, 4-y, 8-y};
```

For example, when s = e (y=1, x=1),

```
indexX = {5, 1, 3, 7};
indexY = {5, 1, 3, 7};
```

For example, when s = h (y=0, x=2),

```
indexX = {4, 0, 4, 8};
indexY = {6, 2, 2, 6};
```

```
S0 = table[indexY[0]][indexX[0]]*fullPixel[0][0] +
                          table[indexY[0]][indexX[1]]*fullPixel[0][1];
S1 = table[indexY[0]][indexX[2]]*fullPixel[0][2] +
                          table[indexY[0]][indexX[3]]*fullPixel[0][3];
S2 = table[indexY[1]][indexX[0]]*fullPixel[1][0] +
                          table[indexY[1]][indexX[1]]*fullPixel[1][1];
S3 = table[indexY[1]][indexX[2]]*fullPixel[1][2] +
                          table[indexY[1]][indexX[3]]*fullPixel[1][3];
S4 = table[indexY[2]][indexX[0]]*fullPixel[2][0] +
                          table[indexY[2]][indexX[1]]*fullPixel[2][1];
S5 = table[indexY[2]][indexX[2]]*fullPixel[2][2] +
                          table[indexY[2]][indexX[3]]*fullPixel[2][3];
S6 = table[indexY[3]][indexX[0]]*fullPixel[3][0] +
                          table[indexY[3]][indexX[1]]*fullPixel[3][1];
S7 = table[indexY[3]][indexX[2]]*fullPixel[3][2] +
                          table[indexY[3]][indexX[3]]*fullPixel[3][3];

S8 = S0 + S1;
S9 = S2 + S3;
S10 = S4 + S5;
S11 = S6 + S7;

S12 = S8 + S9;
S13 = S10 + S11;

S = (S12 + S13) >> 14;
```

Because the value of S is bounded by [0, MAXVALUE], where

```
MAXVALUE = (1 << bitdepth) - 1;
```

the final value F(s) of the subpixel s is then bounded by:

$$F(s) = 0 \qquad for\ s < 0$$

$$F(s) = s \qquad for\ 0 \le s \le MAXVALUE$$

$$F(s) = MAXVALUE \qquad for\ s > MAXVALUE$$

Subpixel Interpolation of Chrominance

Interpolation of chrominance shall be done using bilinear interpolation.

$$F(x, y) = \frac{(A(8-x)(8-y) + Bx(8-y) + Cy(8-x) + Dxy)}{64}$$

where F(x,y) is the subpixel chrominance interpolated from surrounding A, B, C, and D full-sample chrominance values

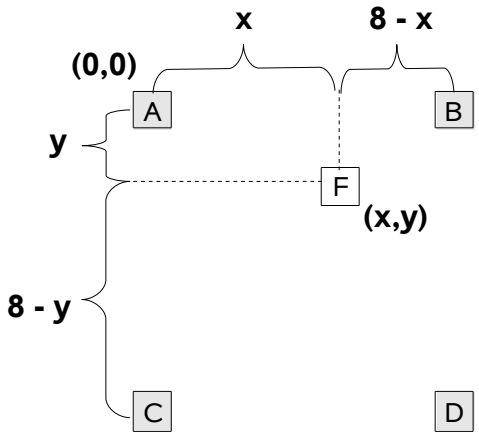

Figure C.4: Subpixel chrominance coordinates for bilinear interpolation

In the case of bilinear interpolation, F(x,y) remains within the boundaries of [0, MAXVALUE].

Annex-D

Finite State Machine (FSM) Definition

```
struct ProbState {
     int lpsInterval;
     int nextStateLps;
     int nextStateMps;
     bool doSwitchMps;
 };

 struct ProbState eaProbFsm[] = {
 /*   0 */ { 0x5555,         0,           0,         0 }
 };

 struct ProbState standardProbFsm[] = {
 /* state  lpsInterval nextStateLps nextStateMps doSwitchMps */
 /*   0 */ { 0x5a1d,         1,           1,         1 },
 /*   1 */ { 0x2586,        14,           2,         0 },
 /*   2 */ { 0x1114,        16,           3,         0 },
 /*   3 */ { 0x080b,        18,           4,         0 },
 /*   4 */ { 0x03d8,        20,           5,         0 },
 /*   5 */ { 0x01da,        23,           6,         0 },
 /*   6 */ { 0x00e5,        25,           7,         0 },
 /*   7 */ { 0x006f,        28,           8,         0 },
 /*   8 */ { 0x0036,        30,           9,         0 },
 /*   9 */ { 0x001a,        33,          10,         0 },
 /*  10 */ { 0x000d,        35,          11,         0 },
 /*  11 */ { 0x0006,         9,          12,         0 },
 /*  12 */ { 0x0003,        10,          13,         0 },
 /*  13 */ { 0x0001,        12,          13,         0 },
 /*  14 */ { 0x5a7f,        15,          15,         1 },
 /*  15 */ { 0x3f25,        36,          16,         0 },
 /*  16 */ { 0x2cf2,        38,          17,         0 },
 /*  17 */ { 0x207c,        39,          18,         0 },
 /*  18 */ { 0x17b9,        40,          19,         0 },
 /*  19 */ { 0x1182,        42,          20,         0 },
 /*  20 */ { 0x0cef,        43,          21,         0 },
 /*  21 */ { 0x09a1,        45,          22,         0 },
 /*  22 */ { 0x072f,        46,          23,         0 },
 /*  23 */ { 0x055c,        48,          24,         0 },
 /*  24 */ { 0x0406,        49,          25,         0 },
 /*  25 */ { 0x0303,        51,          26,         0 },
 /*  26 */ { 0x0240,        52,          27,         0 },
 /*  27 */ { 0x01b1,        54,          28,         0 },
 /*  28 */ { 0x0144,        56,          29,         0 },
 /*  29 */ { 0x00f5,        57,          30,         0 },
 /*  30 */ { 0x00b7,        59,          31,         0 },
 /*  31 */ { 0x008a,        60,          32,         0 },
 /*  32 */ { 0x0068,        62,          33,         0 },
 /*  33 */ { 0x004e,        63,          34,         0 },
 /*  34 */ { 0x003b,        32,          35,         0 },
 /*  35 */ { 0x002c,        33,           9,         0 },
 /*  36 */ { 0x5ae1,        37,          37,         1 },
 /*  37 */ { 0x484c,        64,          38,         0 },
 /*  38 */ { 0x3a0d,        65,          39,         0 },
 /*  39 */ { 0x2ef1,        67,          40,         0 },
 /*  40 */ { 0x261f,        68,          41,         0 },
```

```
    /*  41 */ { 0x1f33,        69,          42,          0 },
    /*  42 */ { 0x19a8,        70,          43,          0 },
    /*  43 */ { 0x1518,        72,          44,          0 },
    /*  44 */ { 0x1177,        73,          45,          0 },
    /*  45 */ { 0x0e74,        74,          46,          0 },
    /*  46 */ { 0x0bfb,        75,          47,          0 },
    /*  47 */ { 0x09f8,        77,          48,          0 },
    /*  48 */ { 0x0861,        78,          49,          0 },
    /*  49 */ { 0x0706,        79,          50,          0 },
    /*  50 */ { 0x05cd,        48,          51,          0 },
    /*  51 */ { 0x04de,        50,          52,          0 },
    /*  52 */ { 0x040f,        50,          53,          0 },
    /*  53 */ { 0x0363,        51,          54,          0 },
    /*  54 */ { 0x02d4,        52,          55,          0 },
    /*  55 */ { 0x025c,        53,          56,          0 },
    /*  56 */ { 0x01f8,        54,          57,          0 },
    /*  57 */ { 0x01a4,        55,          58,          0 },
    /*  58 */ { 0x0160,        56,          59,          0 },
    /*  59 */ { 0x0125,        57,          60,          0 },
    /*  60 */ { 0x00f6,        58,          61,          0 },
    /*  61 */ { 0x00cb,        59,          62,          0 },
    /*  62 */ { 0x00ab,        61,          63,          0 },
    /*  63 */ { 0x008f,        61,          32,          0 },
    /*  64 */ { 0x5b12,        65,          65,          1 },
    /*  65 */ { 0x4d04,        80,          66,          0 },
    /*  66 */ { 0x412c,        81,          67,          0 },
    /*  67 */ { 0x37d8,        82,          68,          0 },
    /*  68 */ { 0x2fe8,        83,          69,          0 },
    /*  69 */ { 0x293c,        84,          70,          0 },
    /*  70 */ { 0x2379,        86,          71,          0 },
    /*  71 */ { 0x1edf,        87,          72,          0 },
    /*  72 */ { 0x1aa9,        87,          73,          0 },
    /*  73 */ { 0x174e,        72,          74,          0 },
    /*  74 */ { 0x1424,        72,          75,          0 },
    /*  75 */ { 0x119c,        74,          76,          0 },
    /*  76 */ { 0x0f6b,        74,          77,          0 },
    /*  77 */ { 0x0d51,        75,          78,          0 },
    /*  78 */ { 0x0bb6,        77,          79,          0 },
    /*  79 */ { 0x0a40,        77,          48,          0 },
    /*  80 */ { 0x5832,        80,          81,          1 },
    /*  81 */ { 0x4d1c,        88,          82,          0 },
    /*  82 */ { 0x438e,        89,          83,          0 },
    /*  83 */ { 0x3bdd,        90,          84,          0 },
    /*  84 */ { 0x34ee,        91,          85,          0 },
    /*  85 */ { 0x2eae,        92,          86,          0 },
    /*  86 */ { 0x299a,        93,          87,          0 },
    /*  87 */ { 0x2516,        86,          71,          0 },
    /*  88 */ { 0x5570,        88,          89,          1 },
    /*  89 */ { 0x4ca9,        95,          90,          0 },
    /*  90 */ { 0x44d9,        96,          91,          0 },
    /*  91 */ { 0x3e22,        97,          92,          0 },
    /*  92 */ { 0x3824,        99,          93,          0 },
    /*  93 */ { 0x32b4,        99,          94,          0 },
    /*  94 */ { 0x2e17,        93,          86,          0 },
    /*  95 */ { 0x56a8,        95,          96,          1 },
    /*  96 */ { 0x4f46,       101,          97,          0 },
    /*  97 */ { 0x47e5,       102,          98,          0 },
    /*  98 */ { 0x41cf,       103,          99,          0 },
    /*  99 */ { 0x3c3d,       104,         100,          0 },
```

```
    /* 100 */ { 0x375e,        99,          93,         0 },
    /* 101 */ { 0x5231,       105,         102,         0 },
    /* 102 */ { 0x4c0f,       106,         103,         0 },
    /* 103 */ { 0x4639,       107,         104,         0 },
    /* 104 */ { 0x415e,       103,          99,         0 },
    /* 105 */ { 0x5627,       105,         106,         1 },
    /* 106 */ { 0x50e7,       108,         107,         0 },
    /* 107 */ { 0x4b85,       109,         103,         0 },
    /* 108 */ { 0x5597,       110,         109,         0 },
    /* 109 */ { 0x504f,       111,         107,         0 },
    /* 110 */ { 0x5a10,       110,         111,         1 },
    /* 111 */ { 0x5522,       112,         109,         0 },
    /* 112 */ { 0x59eb,       112,         111,         1 }
    };
```