BBC Research & Development
White Paper WHP xxx

Dirac Specification 0.9

The Dirac Team

Abstract

This document is the specification for the Dirac Video Compression system. Dirac is a hybrid video compression system using wavelets and overlapped block motion compensation. The specification is a normative description of the Dirac bitstream syntax and the operation of the compliant Dirac decoders. There is no normative text for encoding.

Additional key words:

White Papers are distributed freely on request.

Authorisation of the Chief Scientist is required for publication.

Table of Contents

Table of Figures

Table of Tables

BBC Research & Development
White Paper WHP xxx

Dirac Specification 0.9

The Dirac Team

1 Introduction

1.1 General

Dirac is a general-purpose video codec aimed at resolutions from QCIF (180x144) to HDTV (1920x1080) progressive or interlaced. Dirac is a hybrid motion compensated codec. Such coders incorporate both motion compensated prediction and a transform process. Frames are typically predicted from other, reference, frames. Such predicted frames are called Inter frames. The prediction process has to start at some point, so some frames are compressed without reference to other frames. These frames are called Intra frames. Dirac differs from some other codecs in common use (for example MPEG2, MPEG AVC, and VC-1) in that both Intra frames and the difference between prediction and Inter frames (the prediction residue) are transformed by a wavelet transform. The transform coefficients are then quantised and entropy coded using arithmetic coding. The motion vectors are also entropy coded and transmitted along with the compressed transform coefficients so that the predictions can be repeated at the decoder.

The header parameters in the Dirac bit stream are encoded using variable length codes. This allows a wide variation of the video and coding parameters, which permits compression of a wide range of video formats. By using variable length codes as indices to tables the tables may be extended in future without changes to the bit stream syntax.

1.2 Structure of this document

This document is organised according to the natural decoding hierarchy employed by a decoder, starting passing from sequence level to frame, component, and coefficient levels in order. The overall decoding process including access to the bit stream is summarised in Section 5. The elements of the Random Access Point (RAP) header are explained in Section 4.

1.3 Acknowledgements

This document is the work of a number of people, who are listed in Annex [Contributors]. In style and format it draws on existing standards, such as the ISO MPEG standards series, and also the Xiph Theora specification. We have in general followed similar conventions to the Theora Specification, and in some instances have taken text from that specification.

2 Conventions and definitions

2.1 Notation

2.1.1 Numbers

The prefix b indicates that the following value is to be interpreted as a binary number (base 2).

Example: The value b1110100 is equal to the decimal value 116.

The prefix 0x indicates the following value is to be interpreted as a hexadecimal number (base 16).

Example: The value 0x74 is equal to the decimal value 116.

2.1.2 Arithmetic operations

All arithmetic defined by this specification is exact. However, any real numbers that do arise will always be converted back to integers again in short order. The entire specification can be implemented using only normal integer operations. All operations are to be implemented with sufficiently large integers so that overflow cannot occur. Where the result of a computation is to be truncated to a fixed-sized binary representation, this will be explicitly noted. The size given for all variables is the maximum number of bits needed to store any value in that variable. Intermediate computations involving that variable may require more bits.

The following operators are defined:

$|a|$ The absolute value of a number *a*:

$$|a| = \begin{cases} -a, & a < 0 \\ a, & a \geq 0 \end{cases}$$

*a*b* the multiplication of a number *a* by a number *b*.

$\frac{a}{b}$ the exact division of a number *a* by a number *b* producing a potentially noninteger result.

$\lfloor a \rfloor$ the largest integer less than or equal to a real number *a*.

$\lceil a \rceil$ the smallest integer greater than or equal to a real number *a*.

a//b integer division of *a* by *b*.

$$a // b = \begin{cases} \left\lceil \frac{a}{b} \right\rceil, & a < 0 \\ \left\lfloor \frac{a}{b} \right\rfloor, & a \geq 0 \end{cases}$$

i.e. integer division rounds towards zero for positive *b*.

a%b the remainder from integer division of *a* by *b*.

$a\%b = |a| - |b| * |a//b|$

Note that with this definition, the result is always non-negative and less than $|b|$.

$a << b$ the value obtained by left-shifting the two's complement integer a by b bits. For purposes of this specification, overflow is ignored, and so this is equivalent to integer multiplication of a by 2^b.

$a >> b$ the value obtained by right-shifting the two's complement integer a by b bits, filling in the leftmost bits of the new value with 0 if a is non-negative and 1 if a is negative. This is not equivalent to integer division of a by 2^b. Instead,

$$a >> b = \left\lfloor \frac{a}{2^b} \right\rfloor$$

i.e. bitshift always rounds down.

round(a) Rounds a number a to the nearest integer, with ties rounded away from 0.

$$\text{round}(a) = \begin{cases} \left\lceil a - \frac{1}{2} \right\rceil, & a \le 0 \\ \left\lfloor a + \frac{1}{2} \right\rfloor, & a > 0 \end{cases}$$

sign(a) Returns the sign of a given number.

$$\text{sign}(a) = \begin{cases} -1, & a < 0 \\ 0, & a = 0 \\ 1, & a > 0 \end{cases}$$

ilog(a) The minimum number of bits required to store a positive integer a in two's complement notation, or 0 for a non-positive integer a.

$$\text{ilog}(a) = \begin{cases} 0, & a \le 0 \\ \lceil \log_2 a \rceil, & a > 0 \end{cases}$$

min(a, b) The minimum of two numbers a and b.

max(a, b) The maximum of two numbers a and b.

mean(a_1, …, a_n) For non-integral values this is the mathematical mean $\frac{a_1 + ... + a_n}{n}$ of the numbers a_i. For integral values it is the integer mean $(a_1 + ... + a_n) // n$.

median(a_1, …, a_n) The mathematical median of the numbers a_i. If n is odd, this is the value in position $\frac{n+1}{2}$ if the values are placed in

increasing order, starting at position 1. Where n is even, this is the mean (as defined above) of the values in position $\frac{n}{2}$ and $\frac{n}{2}+1$.

mode(a_1, …, a_n) The mathematical mode of the numbers a_i, which is the most common value. Where two or more values both occur the same maximum number of times, the smaller value is chosen.

The mean, median and mode operations apply also to sets of n-tuples of values by applying individually to each component, i.e.

median($\mathbf{a}_1$, …, $\mathbf{a}_n$)$_k$= median($\mathbf{a}_1$(k), …, $\mathbf{a}_n$(k))

In particular, the median of a set of integer ordered pairs, which are identified with motion vectors in this Specification, is the ordered pair formed of the median of the first components and the median of the second component.

Clip(x,a,b) The value c=min(max(x, a), b), ensuring that $a \le c \le b$.

2.1.3 Bitwise operations

Bitwise operations follow C-style conventions:

& Bitwise AND

| Bitwise OR

! Bitwise NOT

^ Bitwise XOR

All operations apply to all bits of operands, and where operands have unequal length the shorter operand is assumed to be padded to the left with zeroes. For example,

b1011 & b101 = b0001

2.1.4 Logical operations and boolean values

Logical operations follow C-style conventions:

&& Logical AND

|| Logical OR

! Logical NOT

Logical operations apply to Boolean values and produce Boolean values. However, numerical values can be interpreted as Boolean values by interpreting 0 as FALSE and any non-zero value as TRUE.

2.2 Key words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in IETF RFC 2119.

Where such assertions are placed on the contents of a Dirac bitstream itself, implementations should be prepared to encounter bitstreams that do not follow these requirements. An application's behavior in the presence of such nonconforming bitstreams is not defined by this specification, but any reasonable method of handling them MAY be used. By way of example, applications MAY discard the current frame, retain the current output thus far, or attempt to continue on by assuming some default values for the erroneous bits. When such an error occurs in the bitstream headers, an application MAY refuse to decode the entire stream. An application SHOULD NOT allow such non-conformant bitstreams to overflow buffers and potentially execute arbitrary code, as this represents a serious security risk.

2.3 Version Number

The version number of the Dirac syntax specification (this document) is used by the decoder to determine whether it can decode the bitstream. It falls into two integer parts, the major and minor version, written as M.m.

2.3.1 Major Version

The major version defines the version of the syntax with which the bit steam complies. A decoder complies with a major version number if it can parse all bit streams that comply with that version number. Such a compliant decoder must be able to parse all previous versions too. Decoders that comply with a major version of the specification may not be able to parse the bit stream corresponding to a later specification.

Depending on the profile and level defined a decoder compliant with a given major version number may still not be able to decode a bitstream.

A decoder shall maintain a value DECODER_MAJOR_VERSION_NUMBER equal to the major version number of the bitstreams

2.3.2 Minor Version

All minor versions of the specification should be functionally compatible with earlier minor versions with the same major version number. Later minor versions may contain corrections, clarifications, and disambiguations; they must not contain new features.

2.4 Levels and profiles

A profile is a set of decoding tools necessary to decode a bit stream. A level is a set of decoder resource requirements that must be satisfied in order to decode a bitstream, together with a set of constraints on the bitstream that ensures that these requirements are not exceeded. This version of the specification does not define distinct levels and profiles.

A decoder shall maintain values LEVEL_DECODER and PROFILE_DECODER. These values shall be set to zero (0). This denotes Base Level and Base Profile.

Later versions of the specification will provide details of levels and profiles and specify additional levels and profiles.

2.5 Abbreviations

GMC	Global Motion Compensation
IETF	Internet Engineering Task Force
MCPR	Motion Compensated Prediction Residue
MPEG	Motion Picture Expert Group
OBMC	Overlapped Block Motion Compensation
RAP	Random Access Point
RFC	Request For Comment
W3C	World Wide Web Consortium

2.6 Glossary of terms

All terms specific to the Dirac specification are defined when first used herein. This section contains an informative summary of such terms for ease of reference.

Access Unit	A contiguous block of data in the bitstream, from the beginning of which data can be decoded
Macroblock	A 4x4 group of overlapping blocks.
Prediction Unit	A macroblock , sub-macroblock or block, used for the purposes of motion compensation.
Random Access Point	A header at the beginning of an Access Unit containing parameters controlling decoding within the Access Unit
Sub-macroblock	A 2x2 group of overlapping blocks.

2.7 Specification conventions

2.7.1 Decoder variables

This specification uses a number of decoder variables, written in UPPER CASE type. These are used to store decoded values, and the results of calculations or decoder processes. After setting a decoder variable, subsequent decoder operations may depend on the value.

Decoder variables may be of any elementary mathematical type (such as Boolean or integer) as well as types defined in this specification. They also may be arrays of values. One dimensional arrays are represented as ARRAY[], and two-dimensional arrays as ARRAY[][] and so on. The value of a one-dimensional array at an index i is

ARRAY[i]

and the value of a two-dimensional array at an index (i,j) follows standard C/C++ ordering conventions and is

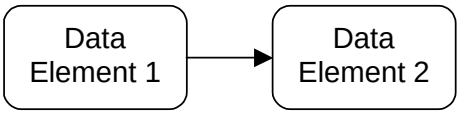
ARRAY[j][i]

The values ARRAY[j][i] for fixed j and i varying over its full range is called the jth row of ARRAY[][]; the values ARRAY[j][i] for fixed i and j varying over its full range is called the ith column of ARRAY[][].

2.7.2 Hierarchical bitstream elements

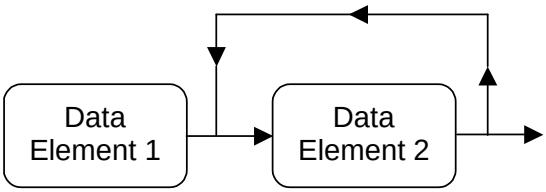
Hierarchical bitstream elements are defined in terms of their component data elements, which are presented as block diagrams.

A block diagram of the form



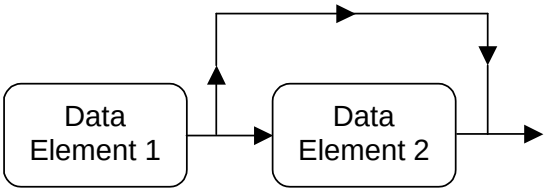
specifies that Data Element 1 is always followed in the bitstream by Data Element 2.

A block diagram of the form



specifies that one or more data elements of type Data Element 2 always follow Data Element 1. The Data Element 2 elements may be different, i.e. may be differently instantiated according to their own specification.

A block diagram of the form



specifies that Data Element 2 conditionally follows Data Element 1. Whether or not it does will depend on previously derived decoder variable values (not necessarily always contained in Data Element 1).

2.7.3 Elementary bitstream elements

Elementary bitstream elements comprise two sub-types. Firstly there are raw byte blocks which are provided as input to arithmetic decoding operations. This specification defines how the arithmetic decoding engine works (Section 6) and also how this engine is used for extracting data values in the context of other decoding operations (subband coefficient decoding and motion vector data decoding).

Secondly there are structured elementary bitstream elements, which are specified by means of figures such as Figure 1.

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
VALUE1	Value1Type	No	16	uegol(n)	≥0,≤31
if (VALUE1) {					

VALUE2	Integer	No	8	Literal	-
VALUE3	Bool	-	-	Literal	-
}					
for (i=0; i<VALUE1; ++i) {					
VALUE3[i]	Integer	Yes	8	se2gol(n)	-
VALUE4[i]	Bool	-	-	Literal	-
}					

Figure 1 Example structure of bitstream elements

The values in the bitstream element are encoded in the order in which they are presented in the table, using the encodings indicated. A full list of encodings is defined in Appendix A.

As in this example, these figures may contain pseudocode to indicate the conditional presence of certain values or of a multiplicity of values in an array. The syntax

if (VALUE) {					
…	…	…	…	…	…
}					

specifies that the values between curly braces are only included if VALUE is TRUE or (if not a Boolean value) non-zero.

The syntax

for (i=0; i<VALUE;++i) {					
…	…	…	…	…	…
}					

specifies that the values between curly braces are included VALUE times (where VALUE is an integral value). Variables specified therein are distinguished by being represented as arrays, with dependency on the loop variable.

An elementary bitstream element may also (but not necessarily) be required to be a whole number of bytes in length, in which case it which case it may be padded with extra bits. These pad bits are not included in the figure representations.

The process of decoding an elementary bitstream element specified by a table begins by

a) extracting all variables in the order in which they occur in the table using the elementary data format decoding rules specified in Appendix A.

b) discarding any padding bits

These stages are not specified in the decoding semantics of individual bitstream elements, but are deemed to have occurred prior to any other decoding stages, which may involve, amongst other processes, the mathematical manipulation of values, or

the initialisation of decoder variables. The semantic specification for decoding bitstream elements consists of specifications of these additional stages.

2.7.4 Decoder process hierarchy and inputs/outputs

Decoding processes, particularly of hierarchical bitstream elements, may be specified by invoking other decoding processes. Each decoder process is specified with a list of inputs and a list of outputs. The outputs are values produced by that decoding process, which may be used by subsequent decoder processes, especially the remainder of the invoking process.

The inputs are a list of decoder variables used within the specification of the decoder process itself, but NOT including those that may be used in other processes invoked by that process. Indeed, all decoder variables are available for all subsequent processes, and it is not strictly necessary to specify inputs at all: it is done so as to provide summary information to implementers.

In software terms decoder variables are global, and inputs do not specify an interface to a decoding function but merely indicates immediate dependencies in the specification that follows.

2.8 Bit packing convention and data input

Data is read from the bitstream in bits and bytes. A byte consists of 8 bits with an order defined from the the least significant bit (lsb or bit 0) to the most significant bit (msb or bit 7). Conventionally, a binary representation of a byte value represents the msb as the leftmost bit and the lsb as the rightmost bit. In Dirac, bits are read in this order also, from the msb to the lsb.

Since the Dirac bitstream includes arithmetically coded elements, any sequence of bits is in principle possible. For a decoder which is already decoding a bitstream correctly, this presents no problem. However, for a decoder which is seeking within a bitstream for certain header elements to begin decoding, such coincidences, although rare, could cause decoding failure. For this reason additional data may be present within the bistream to avoid collisions. This is specified in Section 3.5.2.

2.9 Decoded video

Decoded video is represented in this specification as a collection of two-dimensional non-negative integer-valued arrays, YDATA[][], C1DATA[][] and C2DATA[][]. YDATA[][] represents luma values and C1DATA[][] amd C2DATA[][] represent chroma values. For monochrome data C1DATA[][] and C2DATA[][] may not be present. The dimensions of these arrays are determined by the frame dimensions and the video colour format, as specified in Section 4.5.

The meaning of this data is outside the scope of this specification, although the bitstream contains display parameters(Section 4.5) that may be used to convey how this data is interpreted and displayed. The display parameters are informative and do not necessarily imply that decoded data lies within any particular colourspace, or that the display parameters accurately represent how the data is to be interpreted, although encoding incorrect or misleading display information is strongly deprecated.

In short, this specification determines a mathematical algorithm for populating integer arrays of data from the contents of a compliant bitstream.

3 High level bitstream structure

3.1 Overall structure

A Dirac bit stream commences with a Stream Identifier which shall be followed by one or more Access Units, and terminated by a Stop Sequence code (Figure 2).

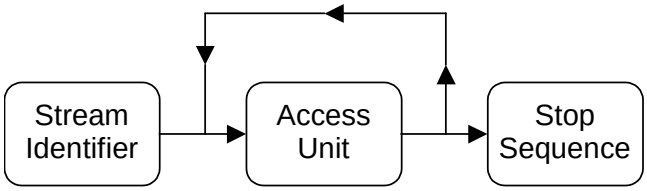


Figure 2 Sequence Structure

The Stream Identifier is a sequence of 8 bytes containing "KW-DIRAC"[1] in ASCII coding: i.e. the following sequence of hexadecimal bytes:

0x4B|0x57|0x2D|0x44|0x49|0x52|0x41|0x43

The stop sequence code is the following sequence of hexadecimal bytes:

0x42|0x42|0x43|0x44|0xD0

Correct decoding of a Dirac bitstream is only supported from the beginning of an Access Unit. If a bitstream contains multiple Access Units then decoding may begin from the start of any Access Unit. Therefore bitstreams that contain many Access Units allow random access to the video.

3.2 Access Unit structure

An Access Unit is a sequence comprising a Random Access Point (RAP) header followed by one or more Frame Data Units (Figure 3).

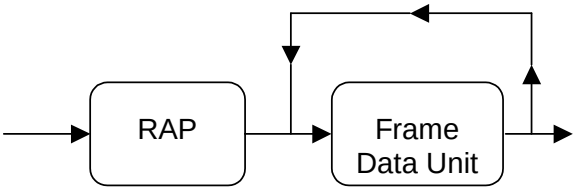


Figure 3 Structure of Access Units

The RAP contains parameters constraining the decoding of data up to the next RAP. These parameters and their decoding are defined in Section 4.

Frame Data Units contain data for decoding a single video frame. Decoding of Frame Data Units is defined in Section 5.3. They possess a frame type which indicates whether the frame is a reference frame or not, and whether the frame is Intra coded or not. This information is indicated in the Parse Code that begins each frame data unit (section 3.5.2). The meaning of these terms is defined in Section 3.3.

Frame Data Units also contain encoded frame numbers indicating display or output order. How these are encoded is specified in Section 5.3. Their order is not

[1] KW refers to Kingswood Warren, the current location (as of 2005) of BBC Research and Development.

necessarily the order in which decoded frame data is to be output from the decoder. A decoder may elect not to decode all frames (see section 3.5.2).

An Access Unit must contain at most 2^{30} Frame Data Units.

3.3 Frame types

To achieve efficient compression Dirac exploits the similarity between frames in a video sequence. This creates dependences between how frames are decoded, and in particular a frame may not be decodeable independently but only with respect to other frames.

A frame is defined to be one of four types, depending on two independent properties.

1. A frame must be either Intra or Inter. Intra frames are decoded without reference to any other frames. Inter frames are decoded using data derived from one or two Reference frames, using motion compensated prediction (Section 5.12).

2. A frame must be either Reference I or Non Reference (NR). Reference frames may be used for decoding other frames. Non Reference frames must not be used for decoding other frames.

Further restrictions are mandatory for determining which R frames may be used for decoding a given frame. These are given in Sections 3.4 and 5.3.

Four types of frame are defined:

- Intra Reference frame

- Intra Non Reference frame

- Inter Reference Frame

- Inter Non Reference frame

The decoder must maintain an array of decoded Reference frames, the REFERENCE_FRAME_BUFFER[]. For the purposes of this Specification, this buffer is considered to be ordered so that the order of frames in the buffer is the order in which they are added to the buffer. In particular, the oldest (first decoded) frame in the buffer is the first frame in the buffer, and the last frame in the buffer is the most recently decoded frame in the buffer. This ordering is independent of the frame numbers of the frames in the buffer.

3.4 Frame reordering

Each frame data unit contains data sufficient in conjunction with previous Parse Units (Section 3.5) to decode a video frame. The order in which frames are to be output from the decoder is defined to be output order or display order. The order in which corresponding frame data units are present in the bitstream may (and usually will) differ from output order. This order is termed coded order.

As a consequence of this restriction, Inter frames shall be decodeable with reference to Reference frames that have occurred earlier in the bitstream. As a result each frame data unit is decodeable immediately if all previous Parse Units (in bitstream order) have been decoded. In fact all frame data units are decodeable immediately if all Parse Units in the previous Access Unit and all Parse Units in the current Access Unit up to the current frame data unit have been decoded; most frame data units require

only data in the current Access Unit to have been decoded, those that don't being a consequence of prediction structures requiring data reordering. For full details, consult section 5.3.1.

Since coded order may differ from output order, a decoder may maintain an array of decoded video frames. The management of this array is outside the scope of this specification. However, the RAP does contain data which will ensure an adequate delay to support reordering whilst rendering pictures smoothly for continuous playback within an Access Unit.

INFORMATIVE

In practice the REFERENCE_FRAME_BUFFER[] and the array of decoded frames may well be implemented in a common picture store. They have been separated in this specification since not all implementations will consist of a straightforward playback of pictures. For example, video editing systems may well wish to decode only a subset of frames. It is intended that bitstream splicing should be possible to produce valid bitstreams, and in this case different prediction structures within Access Units may increase or decrease the level of buffering required for frame reordering. It is up to display applications to manage potential changes in buffering requirements. When specified in future versions of this specification, a level and profile will determine the maximum level of display frame buffering required, to ensure smooth playback across multiple Access Units.

3.5 Parse Units

A Parse Unit is either a RAP or a Frame Data Unit. A Parse Unit must be less than 2^{24} bytes in length.

3.5.1 Byte alignment

The start of each Parse Unit (Frame or RAP) shall be aligned with the start of a byte, counted from the start of the previous RAP. Each Parse Unit shall occupy a whole number of bytes. This is achieved by padding the end of the parse units with up to 7 (undefined) bits if necessary. Further padding with undefined bytes may also be applied.

INFORMATIVE

Additional padding may be required to ensure that Constant Bit Rate coding constraints are met, for example.

3.5.2 Parse Codes

Unique Parse Codes are used to identify the start of Parse Units in the bit stream. These are included to simplify parsing. They are included at the start of each Parse Unit and at the end of the sequence. All codes begin with the same four-byte identifier, which is (in hexadecimal)

0x42|0x42|0x43|0x44

This is the ASCII encoding of "BBCD".

A fifth byte indicates the meaning of the code.

Due to the nature of the arithmetic coded data in the bitstream, the first four bytes might, coincidentally, be output as part of compressed data. If these four bytes occur in the bitstream in any position other than at the beginning of a parse unit or as a Sequence Stop code, then they shall be followed by the Data code byte 0xFF.

This Data code byte shall be removed from the bitstream before any decoding process is appliedit is decoded. Therefore, when scanning a Dirac bitstream to find the start of a Parse Unit, the "Data" Parse Code must be ignored. The final Parse Code bytes are shown in Table 1. Parse codes not specified in Table 1 are undefined.

Value	Start code
0x42|0x42|0x43|0x44|0xD7	RAP
0x42|0x42|0x43|0x44|0xD1	Intra Reference Frame
0x42|0x42|0x43|0x44|0xD2	Intra Non Reference Frame
0x42|0x42|0x43|0x44|0xD3	Inter Reference Frame
0x42|0x42|0x43|0x44|0xD4	Inter Non Reference Frame
0x42|0x42|0x43|0x44|0xD0	Sequence Stop
0x42|0x42|0x43|0x44|0xFF	Data

Table 1 Values of the Parse Codes

The different types of frame are enumerated by different parse odes to provide information to the decoder when it is parsing the bit stream. This allows a decoder to estimate the computational resources required to decode the frame. If a decoder has limited resources it may choose to skip decoding a particular frame because it is unlikely to be decoded in the required time. Frame skipping is supported by the coding of offsets in the stream, Sections 4.3 and 5.3.1.

4 RAP header data

4.1 Purpose

Sometimes sequences are simply played from start to finish. Often, however, it is necessary to start playing a sequence part way through a stream (for example if a viewer has just connected to a broadcast/multicast transmission or following transmission errors). To achieve this the player must be able to start decoding at some point in the middle of a stream without requiring prior information.

Random Access Points (RAPs) are Parse Units in a Dirac bit stream at which a player can start decoding. A RAP provides the sequence and decoding parameters with which to configure the decoder. The RAP should be interpreted as giving an access point to the data *in display order*.

A RAP does NOT imply that all subsequent frames in *coded order* can be decoded. A RAP is followed immediately by an Intra frame. However, typically, some Inter frames that are earlier in display order, will be transmitted following an Intra frame. Such Inter frames might be predicted from frames prior to the RAP. Clearly such Inter frames cannot be decoded from only the information following the RAP. However

subsequent frames can be decoded. Given a limited reordering depth in a prediction structure, eventually all frames after a RAP will be decodeable.

All frames with higher frame numbers than the RAP may be decoded. More accurately (since frame numbers are modulo 2^{32}), all frames within an Access Unit with non-negative FRAME_NUMBER_OFFSET (Section 5.2) can be decoded using earlier data within the Access Unit. All frames within an Access Unit must be decodeable using data within the Access Unit and/or the preceding Access Unit.

The RAP contains Decode and Display Parameters. Generally these parameters should be the same in all RAPs in a compliant bit stream. An exception relates solely to the case where a video sequence contains sections of both interlaced and progressively scanned video. In this case it is permissible for the Scan Format to change to accommodate the changing format of the video. Most video sequences contain exclusively either interlaced or progressively scanned video. For such sequences neither the Decode nor Display Parameters should vary between RAPs.

4.2 RAP header structure

This section specifies the structure and decoding process for the RAP header.

Each RAP consists of three elements, Parse Parameters, Sequence Parameters and Display Parameters, which are presented in this order in the bitstream.

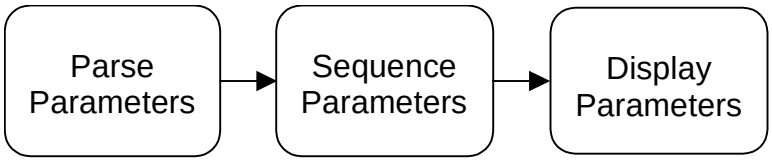


Figure 4 Random Access Point Structure

Each of the three elements of a RAP are byte aligned and occupy a whole number of bytes. Each component is padded with up to 7 (undefined) bits at the end of the component, if necessary, to achieve this. This is done to facilitate parsing.

Parse Parameters provide information that indicates to a decoder whether it will be able to decode the Access Unit, and information that allows a decoder to navigate within the Access Unit.

The Sequence Parameters contain the parameters of the video sequence (width, height chroma format and so on). These parameters are essential to decoding and displaying the bitstream. The Sequence Parameters are intended to change rarely if ever. If a change is necessary, it is recommended that the bitstream is terminated by a Stop Sequence Parse Code and a new bitstream is initiated.

Display Parameters, with the very important exception of signal range parameters, are not needed to decode the sequence. They indicate how the sequence should be displayed, but a display application interfacing with a decoder is not obliged to obey them (and may not be able to).

The Display Parameters should be the same in all RAPs. If changes other than those specified in Section 4.5 are necessary, it is recommended that the bitstream is terminated by a Stop Sequence Parse Code and a new bitstream is initiated.

The RAP is decoded by decoding the Parse Parameters, the Sequence Parameters and the Display Parameters.

4.3 Parse Parameter decoding

This section defines the process for decoding the RAP Parse Parameters.

Inputs to this process are the decoder variables DECODER_MAJOR_VERSION_NUMBER, DECODER_LEVEL, DECODER_PROFILE, indicating the capabilities of the decoder.

Outputs of this process are the decoder variables NEXT_PARSE_OFFSET, PREVIOUS_PARSE_OFFSET, RAP_FRAME_NUMBER, MAJOR_VERSION_NUMBER, MINOR_VERSION_NUMBER, PROFILE, LEVEL.

Parse Parameters are byte-aligned and occupy a whole number of bytes, being padded at the end with up to 7 (undefined) bits as necessary.

The structure of the Parse Parameters, excluding padding bits, is given in Table 2below:

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
PARSE_CODE	ParseCode	-	40	Literal	RAP header Parse Code
NEXT_PARSE_OFFSET	Integer	No	24	Literal	-
PREVIOUS_PARSE_OFFSET	Integer	No	24	Literal	-
RAP_FRAME_NUMBER	Integer	No	32	ue2gol(n)	-
MAJOR_VERSION_NUMBER	Integer	No	8	uegol(n)	>0
MINOR_VERSION_NUMBER	Integer	No	8	uegol(n)	≥0
PROFILE	Integer	No	8	uegol(n)	=0
LEVEL	Integer	No	8	uegol(n)	=0

Table 2 Parse Parameters

Once these variables are extracted, the decoder may terminate decoding if:

a) DECODER_MAJOR_VERSION_NUMBER<MAJOR_VERSION_NUMBER

or

b) PROFILE>DECODER_PROFILE

or

c) LEVEL>DECODER_LEVEL

A decoder that does not terminate at this point may decode correctly, but it cannot be guaranteed to do so.

The Parse Parameters allow a decoder to access the bitstream flexibly, the offset data supporting selective decoding, for example when scrubbing in a video editor. They also allow a decoder to immediately determine whether it can decode a given bitstream by checking the PROFILE and LEVEL values.

NOTE: In this draft of the specification only a single PROFILE and a single LEVEL value are supported.

PARSE_CODE shall be the RAP header parse code defined in Table 1 Values of the Parse Codes.

NEXT_PARSE_OFFSET shall be the offset in bytes to the start of the next Parse Unit. This must be 0 if there are no subsequent Parse Units in the stream.

PREVIOUS_PARSE_OFFSET shall be the offset in bytes to the start of the previous Parse Unit. This must be 0 if there are no prior Parse Units in the stream.

RAP_FRAME_NUMBER shall be the output frame number of the subsequent frame in bitstream order.

MAJOR_VERSION_NUMBER shall be the index of the major version of the bitstream specification to which the Access Unit complies.

MINOR_VERSION_NUMBER shall be the index of the minor version of the bitstream specification to which the Access Unit complies.

PROFILE shall be the index of the coding toolset profile required for decoding the Access Unit.

LEVEL shall be the index of the decoder level requirements necessary for decoding the Access Unit.

4.4 Sequence Parameter decoding

There are no inputs to this process.

Outputs to this process are decoder variables VIDEO_FORMAT, LUMA_WIDTH, LUMA_HEIGHT, CHROMA_FORMAT, CHROMA_H_SCALE, CHROMA_V_SCALE, CHROMA_WIDTH, CHROMA_HEIGHT

Sequence Parameters are byte aligned and occupy a whole number of bytes. The Sequence Parameters are padded with up to 7 (undefined) bits at the end if necessary.

The structure of the Sequence Parameters, excluding padding bits, is as shown in Figure 5.

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
VIDEO_FORMAT_INDEX	Integer	No	8	uegol(n)	-
CUSTOM_DIMENSIONS_FLAG	Bool	-	-	Literal	-
if (CUSTOM_DIMENSIONS_FLAG) {					
LUMA_WIDTH	Integer	No	32	uegol(n)	-
LUMA_HEIGHT	Integer	No	32	uegol(n)	-
}					
CHROMA_FORMAT_FLAG	Bool	-	-	Literal	-
if (CHROMA_FORMAT_FLAG) {					
CHROMA_FORMAT_INDEX	Integer	No	3	Literal	
}					

SIGNAL_RANGE_FLAG	Bool	-	-	Literal	-
if (SIGNAL_RANGE_FLAG) {					
SIGNAL_RANGE_INDEX	Integer	No	32	uegol(n)	≥0 ≤7
if (SIGNAL_RANGE_INDEX==0) {					
LUMA_OFFSET	Integer	No	-	uegol(n)	-
LUMA_EXCURSION	Integer	No	-	uegol(n)	-
CHROMA_OFFSET	Integer	No	-	uegol(n)	-
CHROMA_EXCURSION	Integer	No	-	uegol(n)	-
}					
}					

Figure 5 Sequence Parameter structure

4.4.1 Video format

The VIDEO_FORMAT_INDEX is an index into Table 3, which is the table of supported video formats. The VIDEO_FORMAT_INDEX determines the value of the VIDEO_FORMAT variable, which is in turn used to set default values for numerous frame decoding variables (Section 4.6).

VIDEO_FORMAT_INDEX	VIDEO_FORMAT
0	Custom Format
1	QSIF
2	QCIF
3	SIF
4	CIF
5	SD (NTSC)
6	SD (PAL)
7	SD (525 Digital)
8	SD (625 Digital)
9	HD 720
10	HD 1080
11	Advanced Video Format

Table 3 Supported video formats

4.4.2 Video dimensions

If CUSTOM_DIMENSIONS_FLAG is FALSE, then the luma image dimensions LUMA_WIDTH and LUMA_HEIGHT are set to the values specified in Section 4.6.

If CUSTOM_DIMENSIONS_FLAG is TRUE, then LUMA_WIDTH and LUMA_HEIGHT are set to the values contained in the Sequence Parameters.

The CHROMA_FORMAT_INDEX is an index into Table 4, the table of supported chroma (sampling) formats.

If CHROMA_FORMAT_FLAG is FALSE, then set CHROMA_FORMAT_INDEX=0 (i.e. 4:2:0 is the default chroma format). Otherwise, CHROMA_FORMAT_INDEX is extracted from the bitstream.

CHROMA_FORMAT_INDEX	CHROMA_FORMAT
0	4:2:0
1	4:2:2
2	4:1:1
3	4:4:4
4	Luma Only

Table 4 Supported chroma formats

If CHROMA_FORMAT is 4:2:0 then set

CHROMA_H_SCALE=2

CHROMA_V_SCALE=2

If CHROMA_FORMAT is 4:2:2 then

CHROMA_H_SCALE=2

CHROMA_V_SCALE=1

If CHROMA_FORMAT is 4:4:4 or Luma Only then

CHROMA_H_SCALE=1

CHROMA_V_SCALE=1

The dimensions of chroma components are set by:

CHROMA_WIDTH=LUMA_WIDTH//CHROMA_H_SCALE

CHROMA_HEIGHT=LUMA_WIDTH//CHROMA_V_SCALE

INFORMATIVE

D-Cinema formats are not included because they are insufficiently well defined at present. However the Advanced Video Format includes many features suitable for DCinema applications as well as lossless production applications

The table of video formats may be extended in future versions of this specification. This may be done without changing the bitstream syntax due to the use of variable length codes for the table index.

4.5 Display Parameter decoding

This section specifies the decoding of the Display Parameter element. The recommended interpretation of these values is specified in informative Section 4.7.

Display Parameters are byte-aligned and occupy a whole number of bytes. The Display Parameters are padded with up to 7 (undefined) bits at the end if necessary.

Inputs to this process are: VIDEO_FORMAT.

Outputs of this process are: INTERLACE, TOP_FIELD_FIRST, FRAME_RATE, PIXEL_ASPECT_RATIO, CLEAN_WIDTH, CLEAN_HEIGHT, CLEAN_TL_X, CLEAN_TL_Y, IMAGE_ASPECT_RATIO, LUMA_OFFSET, LUMA_EXCURSION, CHROMA_OFFSET, CHROMA_EXCURSION, ACC_BITS, RED_X, RED_Y, GREEN_X, GREEN_Y, BLUE_X, BLUE_Y,

WHITE_X, WHITE_Y, COLOUR_MATRIX_INDEX, K_R, K_G, K_B, TRANSFER_CHAR_INDEX.

Outputs must be accessible to any display mechanism interfacing to a compliant decoder.

The structure of the Display Parameters, excluding padding bits, is as shown in Figure 6

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
INTERLACE	Bool	-	-	Literal	-
if (INTERLACE) {					
TOP_FIELD_FIRST	Bool	-	-	Literal	-
}					
FRAME_RATE_FLAG	Bool	-	-	Literal	-
if (FRAME_RATE_FLAG) {					
FRAME_RATE_INDEX	Integer	No	32	uegol(n)	≥0 ≤11
if (FRAME_RATE_INDEX==0) {					
FR_NUMERATOR	Integer	No	32	ue2gol(n)	-
FR_DENOMINATOR	Integer	No	32	ue2gol(n)	-
}					
}					
PIXEL_ASPECT_RATIO_FLAG	Bool	-	-	Literal	-
if (PIXEL_ASPECT_RATIO_FLAG) {					
ASPECT_RATIO_INDEX	Integer	No	32	uegol(n)	≥0 ≤3
if (ASPECT_RATIO_INDEX==0) {					
AR_NUMERATOR	Integer	No	32	ue2gol(n)	-
AR_DENOMINATOR	Integer	No	32	ue2gol(n)	-
}					
}					
CLEAN_AREA_FLAG	Bool	-	-	Literal	-

if (CLEAN_AREA_FLAG) {					
CLEAN_TL_X	Integer	No	32	uegol(n)	≥0 ≤LUMA_WIDTH
CLEAN_TL_Y	Integer	No	32	uegol(n)	≥0 ≤LUMA_HEIGHT
CLEAN_WIDTH	Integer	No	32	uegol(n)	≥0 ≤LUMA_WIDTH
CLEAN_HEIGHT	Integer	No	32	uegol(n)	≥0 ≤LUMA_HEIGHT
}					
COLOUR_MATRIX_FLAG	Bool	-	-	Literal	-
if (COLOUR_MATRIX_FLAG) {					
COLOUR_MATRIX_INDEX	Integer	No	-	uegol(n)	≥0 ≤3
if (COLOUR_MATRIX_INDEX==0) {					
K_R	Float	Yes	-	Literal	-
K_B	Float	Yes	-	Literal	-
}					
}					
SIGNAL_RANGE_FLAG	Bool	-	-	Literal	-
if (SIGNAL_RANGE_FLAG) {					
SIGNAL_RANGE_INDEX	Integer	No	32	uegol(n)	≥0 ≤7
if (SIGNAL_RANGE_INDEX==0) {					
LUMA_OFFSET	Integer	No	-	uegol(n)	-
LUMA_EXCURSION	Integer	No	-	uegol(n)	-
CHROMA_OFFSET	Integer	No	-	uegol(n)	-
CHROMA_EXCURSION	Integer	No	-	uegol(n)	-
}					
}					
COLOUR_SPEC _FLAG	Bool	-	-	Literal	-

if (COLOUR_SPEC_FLAG) {					
COLOUR_PRIMARIES_FLAG	Bool	-	-	Literal	-
if (COLOUR_PRIMARIES_FLAG) {					
COLOUR_PRIMARIES_INDEX	Integer	No	-	uegol(n)	≥0 ≤3
if (COLOUR_PRIMARIES_INDEX==0) {					
RED_X	Float	Yes	-	Literal	-
RED_Y	Float	Yes	-	Literal	-
GREEN_X	Float	Yes	-	Literal	-
GREEN_Y	Float	Yes	-	Literal	-
BLUE_X	Float	Yes	-	Literal	-
BLUE_Y	Float	Yes	-	Literal	-
WHITE_X	Float	Yes	-	Literal	-
WHITE_Y	Float	Yes	-	Literal	-
}					
}					
TRANSFER_CHAR_FLAG	Bool	-	-	Literal	-
if (TRANSFER_CHAR_FLAG) {					
TRANSFER_CHAR_INDEX	Integer	No	-	uegol(n)	≥0 ≤3
}					
}					

Figure 6 Display Parameter structure

The decoding process is specified in the following sections.

4.5.1 Interlace

If INTERLACE is TRUE, all frames within the Access Unit are interlaced and should be displayed as interlaced (that is, even lines correspond to one field and odd lines to another). If INTERLACE is TRUE, the variable TOP_FIELD_FIRST is set and the field whose first line is the first line in the frame should be displayed before the other field.

4.5.2 Frame rate

If FRAME_RATE_FLAG is FALSE, then the variable FRAME_RATE is set to the video format default specified in Section 4.6. If FRAME_RATE_FLAG is TRUE then an index FRAME_RATE_INDEX is decoded, which is an index into Table 5.

FRAME_RATE_INDEX	FRAME_RATE
0	Custom Frame Rate
1	12
2	12.5
3	15
4	24000/1001
5	24
6	25
7	30000/1001
8	30
9	50
10	60000/1001
11	60

Table 5 Display frame rates and their encoded indices

If FRAME_RATE_INDEX is 0 then we have a Custom Frame Rate. In this case FRAME_RATE is set by

$$\text{FRAME\_RATE} = \frac{\text{FR\_NUMERATOR}}{\text{FR\_DENOMINATOR}}$$

4.5.3 Pixel aspect ratio

If PIXEL_ASPECT_RATIO_FLAG is FALSE, then PIXEL_ASPECT_RATIO is set to the video format default specified in Section 4.6. If PIXEL_ASPECT_RATIO_FLAG is TRUE then an index ASPECT_RATIO_INDEX is decoded, which is an index into Table 6..

ASPECT_RATIO_INDEX	PIXEL_ASPECT_RATIO
0	Custom Pixel Aspect Ratio
1	1
2	10/11 (525 line systems)
3	59/54 (625 line systems)

Table 6 Pixel aspect ratios and their indices

If ASPECT_RATIO_INDEX is 0, then we have a Custom Pixel Aspect Ratio. In this case PIXEL_ASPECT ratio is set by

$$\text{PIXEL\_ASPECT\_RATIO} = \frac{\text{AR\_NUMERATOR}}{\text{AR\_DENOMINATOR}}$$

4.5.4 Clean area

If CLEAN_AREA_FLAG is FALSE then CLEAN_WIDTH, CLEAN_HEIGHT, CLEAN_TL_X and CLEAN_TL_Y are set to the video format defaults specified in Section 4.6. If CLEAN_AREA_FLAG is TRUE then there is a non-default clean area and CLEAN_WIDTH, CLEAN_HEIGHT, CLEAN_TL_X, CLEAN_TL_Y are decoded.

From the clean area and the pixel aspect ratio, the true image aspect ratio is derived by

$$\text{IMAGE\_ASPECT\_RATIO} = \frac{\text{PIXEL\_ASPECT\_RATIO} * \text{CLEAN\_WIDTH}}{\text{CLEAN\_HEIGHT}}$$

4.5.5 Colour matrix

If COLOUR_SPEC_FLAG is TRUE and COLOUR_MATRIX_FLAG is FALSE, then COLOUR_MATRIX_INDEX is derived from the video format defaults and is non-zero. If COLOUR_MATRIX_FLAG is TRUE, then COLOUR_MATRIX_INDEX is decoded. COLOUR_MATRIX_INDEX is an index into Table 7.

COLOUR_MATRIX_INDEX	Colour Matrix	Colour Matrix Parameters
0	Custom Colour Matrix	-
1	SDTV(ITU-R BT 601, Digital 625 Line video; ITU-R BT 470, Analogue PAL System B&G; SMPTE 170M, NTSC; SMPTE 293M, 720x483 59.94Hz Progressive)	K_R=0.299 K_G=0.587 K_B=0.114
2	HDTV(ITU-R BT 709 (1125/60/2:1 only), Parameter Values for HDTV Standards; ITU-R BT 1361, Worldwide unified colorimetry of future TV systems; SMPTE 274M, 1920x1080 HDTV; SMPTE 296M, 1280x720 HDTV)	K_R=0.2126 K_G=0.7152 K_B=0.0722
3	YCgCo	K_R=0.25 K_G=0.5 K_B=0.25

Table 7 Colour matrix systems

If COLOUR_MATRIX_INDEX is 0, then K_R and K_B are decoded. K_G is derived by

K_G=1.0-(K_R+K_B)

In this case the chroma components should be interpreted as the usual chroma components C_b and C_r (see Section 4.7).

In case COLOUR_MATRIX_INDEX is 3, the chroma components should always be interpreted as the green and orange chroma differences C_g and C_o (see Section 4.7.6). This also affects clipping (Section 5.13) and default offset and excursion values Section 4.5.6).

4.5.6 Signal range

The signal range values determine how much of the data is scaled and clipped prior to matrixing and display. If SIGNAL_RANGE_FLAG is FALSE then SIGNAL_RANGE_INDEX is determined from the video format defaults as per Section 4.6, otherwise it is decoded.

SIGNAL_RANGE_INDEX is an index into Table 8. The offset and excursion values differ for YCgCo coding (COLOUR_MATRIX_INDEX=3), since the lossless integer matrixing operation gains a bit of resolution in the chroma components. In this case SIGNAL_RANGE_INDEX must not be 0, 2, 4 or 6 i.e. only full range formats are supported.

SIGNAL_RANGE_INDEX	Signal Range	Excursions and offsets (except YCgCo)	Excursions and offsets (YCgCo)
0	Custom Signal Range	-	-
1	8 bit Full Range	LUMA_OFFSET=0 LUMA_EXCURSION=255 CHROMA_OFFSET=128 CHROMA_EXCURSION=254	LUMA_OFFSET=0 LUMA_EXCURSION=255 CHROMA_OFFSET=255 CHROMA_EXCURSION=510
2	8 bit Video	LUMA_OFFSET=16 LUMA_EXCURSION=219 CHROMA_OFFSET=128 CHROMA_EXCURSION=224	-
3	10 bit Full Range	LUMA_OFFSET=0 LUMA_EXCURSION=1020 CHROMA_OFFSET=512 CHROMA_EXCURSION=1016	LUMA_OFFSET=0 LUMA_EXCURSION=1020 CHROMA_OFFSET=1020 CHROMA_EXCURSION=2040
4	10 bit Video	LUMA_OFFSET=64 LUMA_EXCURSION=876 CHROMA_OFFSET=512 CHROMA_EXCURSION=896	-
5	12 bit Full Range	LUMA_OFFSET=0 LUMA_EXCURSION=4080 CHROMA_OFFSET=2048 CHROMA_EXCURSION=4064	LUMA_OFFSET=0 LUMA_EXCURSION=4080 CHROMA_OFFSET=4080 CHROMA_EXCURSION=9060
6	12 bit Video	LUMA_OFFSET=256 LUMA_EXCURSION=3504 CHROMA_OFFSET=2048 CHROMA_EXCURSION=3584	-
7	16 bit Full Range	LUMA_OFFSET=0 LUMA_EXCURSION=65280 CHROMA_OFFSET=32768 CHROMA_EXCURSION=65024	LUMA_OFFSET=0 LUMA_EXCURSION=65280 CHROMA_OFFSET=65280 CHROMA_EXCURSION=1305(

Table 8 Signal range parameters

If SIGNAL_RANGE_INDEX is 0, then LUMA_OFFSET, LUMA_EXCURSION, CHROMA_OFFSET and CHROMA_EXCURSION are decoded from the Display Parameters.

Signal range bits over and above 8 are accuracy bits, and a value ACC_BITS is derived by:

$$\text{ACC\_BITS} = \begin{cases} 0 \text{ if SIGNAL\_RANGE\_INDEX=0, 1 or 2} \\ 2 \text{ if SIGNAL\_RANGE\_INDEX=3 or 4} \\ 4 \text{ if SIGNAL\_RANGE\_INDEX=5 or 6} \\ 8 \text{ if SIGNAL\_RANGE\_INDEX=7} \end{cases}$$

Section 4.7.4 has details of how these variables should be used to control matrixing into Red, Green and Blue display primaries.

4.5.7 Colour specification

If COLOUR_SPEC_FLAG is TRUE, then colour primaries, matrices or transfer characteristic may differ from those implied by the video format and are overridden.

If COLOUR_SPEC_FLAG is FALSE then default values for RED_X, RED_Y, GREEN_X, GREEN_Y, BLUE_X, BLUE_Y, WHITE_X, WHITE_Y, K_R, K_G, are as specified in Section 4.6.

If COLOUR_SPEC_FLAG is TRUE and COLOUR_PRIMARIES_FLAG is FALSE, then COLOUR_PRIMARIES_INDEX is derived from the video format defaults (and is non-zero). If COLOUR_PRIMARIES_FLAG is TRUE then COLOUR_PRIMARIES_INDEX is decoded. COLOUR_PRIMARIES_INDEX is an index into Table 9, the table of colour primary systems.

COLOUR_PRIMARIES_INDEX	Colour Primary Systems	Colour Primary Parameters
0	Custom Colour Primaries	-
1	NTSC (SMPTE 170M; also SMPTE RP 145, SMPTE C Primaries and SMPTE 293M, 720x483 59.94Hz Progressive)	RED_X=0.640 RED_Y=0.340 GREEN_X=0.310 GREEN_Y=0.595 BLUE_X=0.155 BLUE_Y=0.070 WHITE_X (CIE III D65) =0.3127 WHITE_Y (CIE III D65) =0.3290
2	PAL (EBU Tech 3213)	RED_X=0.640 RED_Y=0.330 GREEN_X=0.290 GREEN_Y=0.600 BLUE_X=0.150 BLUE_Y=0.060 WHITE_X (CIE III D65) =0.3127 WHITE_Y (CIE III D65) =0.3290
3	HDTV (ITU-R BT 709, Parameter Values for HDTV Standards; also ITU-R BT 1361, SMPTE 274M, 1920x1080 HDTV, SMPTE 296M, 1280x720 HDTV)	RED_X=0.640 RED_Y=0.330 GREEN_X=0.300 GREEN_Y=0.600 BLUE_X=0.150 BLUE_Y=0.060 WHITE_X (CIE III D65) =0.3127 WHITE_Y (CIE III D65) =0.3290

Table 9 Colour primary systems

4.5.8 Transfer characteristic

If TRANSFER_CHAR_FLAG is FALSE, then TRANSFER_CHAR_INDEX is derived from the video format defaults and is non-zero. If TRANSFER_CHAR_FLAG is TRUE then TRANSFER_CHAR_INDEX is decoded and is an index into Table 10.

TRANSFER_CHAR_INDEX	Transfer Characteristic
0	TV (ITU-R BT 709, Parameter Values for HDTV Standards; SMPTE 274M, 1920x1080 HDTV; SMPTE 296M, 1280x720 HDTV; SMPTE 170M, NTSC; SMPTE 293M, 720x483 59.94Hz Progressive)
1	Extended Colour Gamut (ITU-R BT 1361, Worldwide unified colorimetry of future TV systems)
2	Linear

Table 10 Transfer characteristics

4.6 Video format default values for RAP parameters

This section specifies default values for RAP Sequence and Display Parameters derived from the value of VIDEO_FORMAT. These values shall be used if not explicitly overridden by subsequent data in the RAP Header.

4.6.1 Custom Format

The default values in Table 11 shall be used for decoding processes if VIDEO_FORMAT is Custom Format.

Decoder parameter	Default value
LUMA_WIDTH	640
LUMA_HEIGHT	480
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	30
PIXEL_ASPECT_RATIO	1
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	640
CLEAN_HEIGHT	480
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	1 (8 bit Full Range)
COLOUR_PRIMARIES_INDEX	1 (NTSC)
TRANSFER_CHAR_INDEX	0 (TV)

Table 11 Custom Format default parameter values

4.6.2 QSIF

The default values in Table 12 shall be used for decoding processes if VIDEO_FORMAT is QSIF.

Decoder parameter	Default value
LUMA_WIDTH	176
LUMA_HEIGHT	120
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	15
PIXEL_ASPECT_RATIO	10/11
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	176
CLEAN_HEIGHT	120
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	1 (8 bit Full Range)
COLOUR_PRIMARIES_INDEX	1 (NTSC)
TRANSFER_CHAR_INDEX	0 (TV)

Table 12 QSIF default parameter values

4.6.3 QCIF

The default values in Table 13 shall be used for decoding processes if VIDEO_FORMAT is QCF.

Decoder parameter	Default value
LUMA_WIDTH	176
LUMA_HEIGHT	144
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	12.5
PIXEL_ASPECT_RATIO	59/54
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	176
CLEAN_HEIGHT	144
COLOUR_MATRIX_INDEX	1 (SDTV)

SIGNAL_RANGE_INDEX	1 (8 bit Full Range)
COLOUR_PRIMARIES_INDEX	2 (PAL)
TRANSFER_CHAR_INDEX	0 (TV)

Table 13 QCIF default parameter values

4.6.4 SIF

The default values in Table 14 shall be used for decoding processes if VIDEO_FORMAT is SIF.

Decoder parameter	Default value
LUMA_WIDTH	352
LUMA_HEIGHT	240
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	15
PIXEL_ASPECT_RATIO	10/11
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	352
CLEAN_HEIGHT	240
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	1 (8 bit Full Range)
COLOUR_PRIMARIES_INDEX	1 (NTSC)
TRANSFER_CHAR_INDEX	0 (TV)

Table 14 SIF default parameter values

4.6.5 CIF

The default values in Table 15 shall be used for decoding processes if VIDEO_FORMAT is CIF.

Decoder parameter	Default value
LUMA_WIDTH	352
LUMA_HEIGHT	288
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	12.5
PIXEL_ASPECT_RATIO	59/54

CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	352
CLEAN_HEIGHT	288
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	1 (8 bit Full Range)
COLOUR_PRIMARIES_INDEX	2 (PAL)
TRANSFER_CHAR_INDEX	0 (TV)

Table 15 CIF default parameter values

4.6.6 SD (NTSC)

The default values in Table 16 shall be used for decoding processes if VIDEO_FORMAT is SD (NTSC).

Decoder parameter	Default value
LUMA_WIDTH	704
LUMA_HEIGHT	480
INTERLACE	TRUE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	30000/1001
PIXEL_ASPECT_RATIO	10/11
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	704
CLEAN_HEIGHT	480
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	1 (NTSC)
TRANSFER_CHAR_INDEX	0 (TV)

Table 16 SD (NTSC) default parameter values

4.6.7 SD (PAL)

The default values in Table 17 shall be used for decoding processes if VIDEO_FORMAT is SD (PAL).

Decoder parameter	Default value
LUMA_WIDTH	704

LUMA_HEIGHT	576
INTERLACE	TRUE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	25
PIXEL_ASPECT_RATIO	59/54
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	704
CLEAN_HEIGHT	576
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	2 (PAL)
TRANSFER_CHAR_INDEX	0 (TV)

Table 17 SD (PAL) default parameter values

4.6.8 SD (525 Digital)

The default values in Table 18 shall be used for decoding processes if VIDEO_FORMAT is SD (525 Digital).

Decoder parameter	Default value
LUMA_WIDTH	720
LUMA_HEIGHT	480
INTERLACE	TRUE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	30000/1001
PIXEL_ASPECT_RATIO	10/11
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	720
CLEAN_HEIGHT	480
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	1 (NTSC)
TRANSFER_CHAR_INDEX	0 (TV)

Table 18 SD (525 digital) default parameter values

4.6.9 SD (625 Digital)

The default values in Table 19 shall be used for decoding processes if VIDEO_FORMAT is SD (625 Digital).

Decoder parameter	Default value
LUMA_WIDTH	720
LUMA_HEIGHT	576
INTERLACE	TRUE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	30
PIXEL_ASPECT_RATIO	59/54
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	720
CLEAN_HEIGHT	576
COLOUR_MATRIX_INDEX	1 (SDTV)
SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	2 (PAL)
TRANSFER_CHAR_INDEX	0 (TV)

Table 19 SD (625 digital) default parameter values

4.6.10 HD 720

The default values in Table 20 shall be used for decoding processes if VIDEO_FORMAT is HD 720.

Decoder parameter	Default value
LUMA_WIDTH	1280
LUMA_HEIGHT	720
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	24
PIXEL_ASPECT_RATIO	1
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	1280
CLEAN_HEIGHT	720
COLOUR_MATRIX_INDEX	2 (HDTV)

SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	3 (HDTV)
TRANSFER_CHAR_INDEX	0 (TV)

Table 20 HD 720 default parameter values

4.6.11 HD 1080

The default values in Table 21 shall be used for decoding processes if VIDEO_FORMAT is HD 1080.

Decoder parameter	Default value
LUMA_WIDTH	1920
LUMA_HEIGHT	1080
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	24
PIXEL_ASPECT_RATIO	1
CLEAN_TL_X	0
CLEAN_TL_Y	0
CLEAN_WIDTH	1920
CLEAN_HEIGHT	1080
COLOUR_MATRIX_INDEX	2 (HDTV)
SIGNAL_RANGE_INDEX	2 (8 bit Video)
COLOUR_PRIMARIES_INDEX	3 (HDTV)
TRANSFER_CHAR_INDEX	0 (TV)

Table 21 HD 1080 default parameter values

4.6.12 Advanced Video Format

The default values in Table 22 shall be used for decoding processes if VIDEO_FORMAT is the Advanced Video Format.

Decoder parameter	Default value
LUMA_WIDTH	1920
LUMA_HEIGHT	1080
INTERLACE	FALSE
TOP_FIELD_FIRST	TRUE
FRAME_RATE	50
PIXEL_ASPECT_RATIO	1
CLEAN_TL_X	0

CLEAN_TL_Y	0
CLEAN_WIDTH	1920
CLEAN_HEIGHT	1080
COLOUR_MATRIX_INDEX	3 (YCgCo)
SIGNAL_RANGE_INDEX	5 (12 bit Full Range)
COLOUR_PRIMARIES_INDEX	3 (HDTV)
TRANSFER_CHAR_INDEX	1 (Extended Colour Gamut)

Table 22 HD 1080 default parameter values

4.7 Interpretation of Display Parameters (INFORMATIVE)

The interpretation of Display Parameters by a display mechanism interfacing with a compliant decoder is non-normative. However, it should where possible follow the recommendations and interpretations specified in this section. Likewise, encoders should ensure that accurate display parameter information is encoded to maximise the quality of displayed video.

4.7.1 Video systems model

All current video systems use the following model for YUV coding of the RGB values (computer systems often omit coding to and from YUV).

DIAGRAM NEEDED HERE

The R, G and B are tristimulus values (e.g. candelas/meter^2). Their relationship to CIE XYZ tristimulus values can be derived from the set of primaries and white point defined in the colour primaries part of the colour specification below using the method described in SMPTE RP 177-1993. In this document the RGB values are normalised to the range [0,1], so that RGB=1,1,1 represents the peak white of the display device and RGB=0,0,0 represents black.

The E_R, E_G, E_B values, are related to the RGB values by non-linear transfer functions labelled "f()" and "g()" in the diagram. Normally, these values also fall in the range [0,1], but in the case of extended gamut, negative values may be allowed also. The transfer function "f()" is typically performed in the camera and is specified in the "Transfer Characteristic" part of the "Colour Specification". For aesthetic and psychovisual reasons the transfer function "g()" is not quite the inverse of "f()". In fact the combined effect of "f()" and "g()" is such that

$$g(f(x)) = x^{\gamma}$$

where γ is the "rendering intent" or end to end gamma of the system, which may vary between about 1.1 and 1.6 depending on viewing conditions. The rationale for this is given in [Digital Video and HDTV, Charles Poynton 2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7].

The non-linear E_R, E_G, E_B values are subject to a matrix operation (known as "non-constant luminance coding"), which transforms them into luma (E_Y) and chroma (normally E_{Cb} and E_{Cr}, but sometimes E_{Cg} and E_{Co}). E_Y is normally limited to the range [0,1] and the chroma values to the range [-0.5, 0.5]. This is "YUV" coding and sometimes the chroma components are subsampled, either horizontally or both

horizontally and vertically. UV sampling is specified by the CHROMA_FORMAT value.

The $E_{Y,}$ E_{Cb}, E_{Cr} (or $E_{Y,}$ E_{Cg}, E_{Co}) values are mapped to a range of integers Y, Cb, Cr (Y, Cg, Co). Typically they are mapped to an 8 bit range [0, 255]. The way this mapping occurs is defined by the signal range parameters. It is these integer values that are actually output from the decoder.In order to display video, the inverse to the above operations must be performed to convert this data to $E_{Y,}$ E_{Cb}, E_{Cr}, then to $E_{R,}$ E_G, E_B and thence to R, G and B.

The E values can be viewed as something of a mathematical abstraction. For example in digital display devices, R, G and B values are specified in terms of integer levels which are derived from the integral luma and chroma values by direct operations subsuming and approximating all the real-number operations described here. Generally, these approximations cause loss through quantisation of intermediate values, and the restriction of values to particular ranges also restricts the colour gamut. In the case of YCgCo coding, a lossless direct integer transform is used, so that in this mode (together with 4:4:4 sampling and lossless compression), Dirac supports lossless RGB coding.

4.7.2 Frame rate

The FRAME_RATE value encodes the intended rate at which frames should be displayed subsequent to decoding. If INTERLACE is TRUE, then fields are displayed at double the frame rate, in the order specified by the TOP_FIELD_FIRST flag.

4.7.3 Pixel aspect ratio and clean area

The PIXEL_ASPECT_RATIO value of an image is the ratio of the intended spacing of horizontal samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios are fundamental properties of sampled images because they determine the displayed shape of objects in the image. Failure to use the right value of PIXEL_ASPECT_RATIO will result in distorted images – for example, circles will be displayed as ellipses and so forth.

Some HDTV standards and computer image formats are defined to have pixel aspect ratios that are exactly 1:1.

The clean area is intended to define an area within which picture information is subjectively uncontaminated by all edge transient (and other) distortions. It may only be appropriate to display the clean area rather than the whole picture, which may be distorted at the edge.

The top-left corner of the clean area has coordinates (CLEAN_TL_X, CLEAN_TL_Y) and dimensions CLEAN_WIDTHxCLEAN_HEIGHT.

The clean area and the pixel aspect ratio determine the IMAGE_ASPECT_RATIO which is the ratio of the width of the intended display area to the height of the intended display area.

Given two separate sequences, with identical IMAGE_ASPECT_RATIO, if the top left corner and bottom left corners of their clean apertures are coincident when displayed, then the images as a whole should be exactly coincident. This is regardless

of the actual pixel dimensions of the images or their clean areas. This allows sequences to be combined together appropriately if they are appropriately scaled.

The defined pixel aspect ratios are designed to give standard image aspect ratios for typical TV broadcasts. For example, for a 525 line (American) 704 x 480 (clean area) picture the image aspect ratio is (704 x 10)/(480 x 11) which is exactly 4:3.

For 625 line systems the 59:54 pixel aspect ratio means (less conveniently) that a 702.9x576 image would have an exact 4:3 image aspect ratio. It might be argued that the pixel aspect ratio for 625 line systems should be such that a 702x576 image would have an exact 4:3 image aspect ratio. It could be said that this corresponds to the analogue 625 line TV specification. This requirement would lead to a pixel aspect ratio of 128:117. However, the tolerance of the analogue line length is 702± 3 pixels, which does not really seem to justify a ratio of exactly 128:117.

The values specified here are generally agreed to be the “correct” values. Then again not everyone agrees with this consensus. These arise from the “industry standard” sampling frequencies used for square pixels, which were originally designed for digitising composite analogue video signals. These “industry standard” sampling frequencies are 11+3/11 MHz for 525 line systems and 14.75MHz for 625 line systems. The ratio of these frequencies to the (standardised) 13.5MHz sampling frequency used for broadcasting yields the pixel aspect ratios given in Table 18 and Table 19.

You are strongly advised to use one of the default pixel aspect ratios. However, if you know what you are doing and don’t like the default values you can define your own ratio. You should be aware that many display devices may ignore your decision and may use different and unsuitable values.

4.7.4 Signal range

The offset and excursion values should be used to convert the integer-valued decoded luma and chroma data Y, Cb, Cr to intermediate values E_Y, E_{Cr}, and E_{Cb} by the recipe

$$E_Y = \frac{Y - \text{LUMA\_OFFSET}}{\text{LUMA\_EXCURSION}}$$

$$E_{Cb} = \frac{Cb - \text{CHROMA\_OFFSET}}{\text{CHROMA\_EXCURSION}}$$

$$E_{Cr} = \frac{Cr - \text{CHROMA\_OFFSET}}{\text{CHROMA\_EXCURSION}}$$

E_Y, is normally clipped to the range [0,1], and E_{Cr}, and E_{Cb} to the range [-0.5,0.5]. This effectively clips Y to

[LUMA_OFFSET, LUMA_OFFSET+LUMA_EXCURSION]

and Cb, Cr to

[CHROMA_OFFSET-LUMA_EXCURSION/2,
LUMA_OFFSET+LUMA_EXCURSION/2]

However, maintaining an extended RGB gamut may mean that either such clipping is not done, or non-standard offset and excursion values are used to extract the extended gamut from the non-negative decoded Y, Cr, and Cb values.

Non-default offset and excursion values cannot be coded if the chroma format is YCgCo: default parameters should be used. However, even in this case, E_Y, E_{Cg}, and E_{Co} should not be calculated. Instead, direct integer conversion to RGB should be done as described in Section 4.7.6. (In fact, excursion values will be ignored in this integer conversion.)

4.7.5 Colour primaries

The colour primaries allow device dependent linear RGB colour co-ordinates to be mapped to device independent linear CIE XYZ space. The primaries specified below are the CIE (1931) XYZ chromaticity co-ordinates of the primaries and the white point of the device. The maths required to convert between RGB and XYZ is reproduced below.

$$F = \begin{bmatrix} \frac{\text{RED\_X}}{\text{RED\_Y}} & \frac{\text{GREEN\_X}}{\text{GREEN\_Y}} & \frac{\text{BLUE\_X}}{\text{BLUE\_Y}} \\ 1 & 1 & 1 \\ \frac{\text{1-RED\_X-RED\_Y}}{\text{RED\_Y}} & \frac{\text{1-GREEN\_X-GREEN\_Y}}{\text{GREEN\_Y}} & \frac{\text{1-BLUE\_X-BLUE\_Y}}{\text{BLUE\_Y}} \end{bmatrix}$$

$$\begin{bmatrix} s_r \\ s_g \\ s_b \end{bmatrix} = F^{-1} \begin{bmatrix} \frac{\text{WHITE\_X}}{\text{WHITE\_Y}} \\ 1 \\ \frac{\text{1-WHITE\_X-WHITE\_Y}}{\text{WHITE\_Y}} \end{bmatrix}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} s_r R \\ s_g G \\ s_b B \end{bmatrix}$$

The colour primary specification therefore allows exact colour reproduction of decoded RGB values on different displays with different display primaries. It has to be said that often conversion between encoded primaries and display primaries is not done.

4.7.6 Colour matrix

Luma and chroma values $E_{Y,}$ E_{Cb}, E_{Cr} should be used to derive E_R, E_G, E_B values by the following equations.

$$E_R = E_Y + 2\left(1-\text{K\_R}\right)E_{Cr}$$

$$E_G = E_Y - 2\frac{\left(1-\text{K\_B}\right)\text{K\_B}}{\text{K\_R}}E_{Cb} - 2\frac{\left(1-\text{K\_R}\right)\text{K\_R}}{\text{K\_R}}E_{Cr}$$

$$E_B = E_Y + 2\left(1-\text{K\_R}\right)E_{Cb}$$

This follows by inverting the equations

$$E_Y = \text{K\_R} \cdot E_R + \text{K\_G} \cdot E_G + \text{K\_B} \cdot E_B \quad ; \quad \text{K\_R+K\_G+K\_B} = 1$$

$$E_{Cb} = \frac{E_B - E_Y}{2 \cdot (1 - \text{K\_B})}$$

$$E_{Cr} = \frac{E_R - E_Y}{2 \cdot (1 - \text{K\_R})}$$

In the case of YCgCo coding, E_R, E_G, E_B should be directly computed from the integer Y, Cg and Co values by the following recipe, whereby integer RGB I_R, I_G, I_B values are decoded by

Y-=LUMA_OFFSET

Cg-=CHROMA_OFFSET

Co-=CHROMA_OFFSET

TEMP=Y-(Cg>>1)

I_G=TEMP+Cg

I_B=TEMP-(Co>>1)

I_R=I_B+Co

These may be scaled down by dividing by (255<<ACC_BITS) and clipped to [0,1] to give E_R, E_G, E_B. If the inverse transform has been correctly applied prior to coding and lossless coding employed, then clipping will be unnecessary.

Note that this matrix implies that the chroma range is twice as large as the RGB range (and the luma range), since the chroma components involve subtraction. Although logically knowing the signal range and scaling signals is prior to performing matrixing, the matrix parameters are coded first in the Display Parameters in order to allow the signal ranges to be correctly determined in this case.

4.7.7 Transfer characteristic

TV transfer characteristic

Denoting R or G or B as “L” (light) and E_R, E_G, E_B as “E” then E=f(L) such is that;

$$E = \begin{cases} 4.5L & 0 \le L < 0.018 \\ 1.099L^{0.45} & 0.018 \le L \le 1 \end{cases}$$

All modern TV systems use this transfer characteristic at present. ITU-R BT 470 (Conventional Television systems PAL, NTSC and SECAM) specifies an “assumed gamma value of the receiver for which the primary signals are pre-corrected” as 2.2 for NTSC and 2.8 for PAL. This specification is incomplete, incorrect and obsolete and modern PAL and NTSC systems use the “TV” transfer characteristic above.

Extended Colour Gamut

ITU-R BT 1361, Worldwide unified colorimetry of future TV systems defines a transfer characteristic for systems with an extended colour gamut as follows.

Denoting R or G or B as “L” (light) and E_R, E_G, E_B as “E” then E=f(L) such that;

$$E = \begin{cases} 1.099L^{0.45} & 0.018 \le L \le 1.33 \\ 4.5L & -0.0045 \le L < 0.018 \\ -\dfrac{1.099(-4L)^{0.45} - 0.099}{4} & -0.25 \le L < -0.0045 \end{cases}$$

This transfer characteristic is intended to be used with systems using an extended colour gamut.

Linear

A linear transfer characteristic has f(x)=x.

5 Decoding process

5.1 Overview

The decoding process described in this section specifies the decoding process that a decoder shall undergo to reconstruct frames from the coded bitstream. This section does not specify how frames are encoded, nor how frames are presented for display, which are outside the scope of this specification.

A decoder may decode frames in any order provided that it has already decoded and stored the reference frames (if any) to which the corresponding frame data unit refers.

5.2 Decoder initialisation

Before decoding a frame a decoder must be initialised by setting the RAP parameters. This is done by decoding the RAP header as per Section 4. The parameters set in the RAP header apply to all frames decoded from an Access Unit. They imply defaults for a number of frame decoding operations, which may be overriden by parameters set within individual frame data units. Details are given in subsequent sections.

RAP parameters apply across the Access Unit, which may include frames which are temporally prior to the frame indicated by the RAP_FRAME_NUMBER. These are indicated by a negative FRAME_NUMBER_OFFSET. In particular Sequence and Display parameters including frame dimensions and picture format (interlace or progressive) shall be common across an Access Unit. Where Sequence or Display parameters change with respect to the previous RAP header, frames with negative FRAME_NUMBER_OFFSET must not occur within the Access Unit. Changing RAP parameters within a bitstream is deprecated. A bitstream should terminate with a stop code and a new bitstream should start with the new RAP header data.

INFORMATIVE

This prescription is roughly equivalent, in MPEG-2 terms, to demanding that a closed GOP be used at the boundary between interlace and non-interlace pictures, or between SD and HD pictures. The prescription ensures that the temporal scope of RAP parameters and their scope within the bitstream can be aligned. It makes sense since such a boundary almost certainly represents a cut, and prediction across the boundary is useless.

5.3 Frame decoding

This section specifies the process for decoding a frame data unit.

Inputs to this process are: None.

Outputs to this process are: the decoder variable FRAME_NUMBER, and the decoded video data arrays YDATA[][], C1DATA[][], C2DATA. If the chroma format is Luma Only then the chroma arrays C1DATA[][] and C2DATA[][] shall be empty.

Frame data units contain the parameters and data need to reconstruct a single video frame. They consist of several parts and their structure is specified by Figure 7.

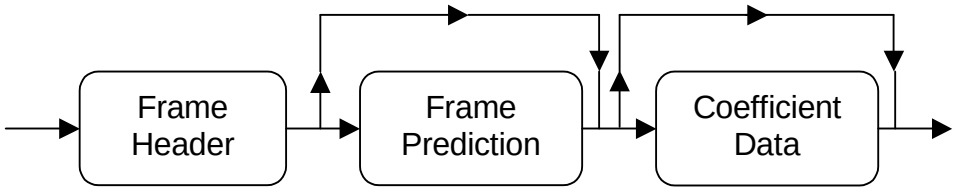


Figure 7 Frame Data Unit structure

Frame Prediction is included for Inter Frames and omitted for Intra frames. Each part is byte aligned and padded with up to 7 zero bits to a whole number of bytes. These parts are described in subsequent sections.

The frame decoding procedure is as follows.

1. Decode the Frame Header as per Section 5.3.1
2. If the frame is an Inter frame, decode the Frame Prediction as per Section 5.5
3. a) If ZERO_COEFFS_FLAG is FALSE, decode Coefficient Data producing YDATA[][], C1DATA[][] and C2DATA[][] as per Section 5.7

   b) If ZERO_COEFFS_FLAG is TRUE, set all elements of YDATA[][], C1DATA[][] and C2DATA[][] to be identically zero.
4. If the frame is an Inter frame, motion compensate the data arrays as per Section 5.12
5. Clip the frame data as per Section 5.13
6. If the frame is a Reference frame, then

   a) If REFERENCE_FRAME_BUFFER[] is full, then first remove the first (oldest) frame in the buffer;

   b) Add the frame to REFERENCE_FRAME_BUFFER[].

INFORMATIVE

Allowing the REFERENCE_FRAME_BUFFER to become full is deprecated. Bitstreams are expected to use the method of explicitly retiring frames to manage the buffer.

5.3.1 Parse Parameter decoding

This section specifies the process for decoding the Frame Header.

Inputs to this process RAP_FRAME_NUMBER.

Outputs of this process are FRAME_NUMBER, IS_REF, IS_INTRA, NUM_REFS, REFS[], NEXT_PARSE_OFFSET, PREVIOUS_PARSE_OFFSET

The Frame Header provides information that allows the Frames to be correctly interpreted and decoded. The Frame Header is byte aligned and padded to a whole number of bytes with up to 7 zero bits. The structure of the Frame Header is specified by Figure 8 below.

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
PARSE_CODE	Parse Code	-	40	Literal	=Intra Reference, Intra Non Reference, Inter Reference, or Inter Non Reference
NEXT_PARSE_OFFSET	Integer	No	24	Literal	-
PREVIOUS_PARSE_OFFSET	Integer	No	24	Literal	-
FRAME_NUMBER_OFFSET	Integer	Yes	32	se2gol(n)	$\geq -2^{30}+1$ $\leq 2^{30}-1$
if (!IS_INTRA) {					
TWO_REFS_FLAG	Bool	-	1	Literal	-
for (i=0; i<NUM_REFS; ++i) {					
REFS_OFFSET[i]	Integer	Yes	32	se2gol(n)	$\geq -2^{31}+1$ $\leq 2^{31}-1$
}					
NON_DEFAULT_WEIGHTS	Bool	-	1	Literal	-
if (NON_DEFAULT_WEIGHTS) {					
REF1_WEIGHT	Integer	Yes	6	segol(n)	≥ -16 ≤ 16
if (NUM_REFS==2) {					
REF2_WEIGHT	Integer	Yes	6	segol(n)	≥ -16 ≤ 16
}					
}					
LIST_SIZE	Integer	No	8	uegol(n)	-

for (i=0; I<LIST_SIZE; ++I) {					
RTD_LIST_OFFSET[i]	Integer	Yes	32	se2gol(n)	$\geq 2^{31}+1$ $\leq 2^{31}-1$
}					

Figure 8 Structure of Frame Header

The Reference Frames element is only present for Inter frames.

Parse Code

PARSE_CODE is decoded by reading the first four bytes and comparing the fifth byte with the Parse Code table values in Table 1 to derive the frame type. If the frame is of Intra type, the IS_INTRA flag is set to TRUE, else it is set to FALSE.

If the frame is a reference frame the IS_REF flag is set to TRUE, else it is set to FALSE.

Frame Numbers

FRAME_NUMBER is set as

FRAME_NUMBER=RAP_FRAME_NUMBER +FRAME_NUMBER_OFFSET (mod 2^{32})

Each value of FRAME_NUMBER shall occur at most once in any consecutive pair of Access Units.

If the frame data unit is the first frame data unit after a RAP, then FRAME_NUMBER=RAP_FRAME_NUMBER i.e. FRAME_NUMBER_OFFSET=0. In this case also the frame shall be an Intra frame.

INFORMATIVE

The restrictions on the value of FRAME_NUMBER_OFFSET and on the number of frame data units in an Access Unit ensures that an unambiguous frame number can be assigned in this manner for all frames in the current Access Unit and the previous Access Unit.

Reference Frames

If IS_INTRA is FALSE, then reference frame data is decoded.

If TWO_REFS_FLAG is TRUE, then NUM_REFS is set to 2; otherwise it is set to 1.

The reference frame numbers are coded differentially with respect to the RAP_FRAME_NUMBER, and are decoded by:

for (i=0 i<NUM_REFS i++)

REFS[i]=+RAP_FRAME_NUMBER+REFS_OFFSET[i]

If FRAME_NUMBER_OFFSET>0, each reference frame number REFS[i] shall be the frame number of a Reference frame in the same Access Unit which has occurred prior to the current frame data unit in bitstream order.

If FRAME_NUMBER_OFFSET<0, each reference frame number REFS[i] shall be the frame number of a Reference frame in the same Access Unit which has occurred prior to the current frame data unit in bitstream order, or it shall be the frame number of a frame in the previous Access Unit.

If the NON_DEFAULT_WEIGHTS flag is TRUE, then prediction values will be scaled by scaling factor values in the motion compensation process. The values REF1_WEIGHT and REF2_WEIGHT determine the scaling values to be used, with three precision bits i.e. in eighths. So a scaling value of 1 for Reference 1 is indicated by setting REF1_WEIGHT=8 and a scaling value of 1/2 for Reference 2 is indicated by setting REF2_WEIGHT=4. Negative scaling values and scaling values >1 are permitted.

INFORMATIVE

The frame number restrictions ensure that the scope of temporal prediction is limited to the range where frames can be uniquely identified by their frame numbers.

Retired Frame List

The retired list frame numbers are coded differentially with respect to the RAP_FRAME_NUMBER. The pseudocode recipe is:

for (i=0 i<LIST_SIZE i++)

RTD_LIST[i]=RAP_FRAME_NUMBER+RTD_LIST_OFFSET[i]

For each frame number in RTD_LIST[], discard any frame with that frame number from REFERENCE_FRAME_BUFFER[]. Retired frames shall be discarded immediately before further decoding of the current frame takes place.

INFORMATIVE

The Retired Frame List is the recommended method for removing reference frames from the reference frame buffer.

Zero coefficients

If ZERO_COEFFS_FLAG is TRUE, then all pixel data in all components shall be set to zero.

5.4 Frame number arithmetic (INFORMATIVE)

For display, it is necessary for a display process to be able to determine the correct order in which frames should appear. Since FRAME_NUMBER is an unsigned 32 bit integer, as some point the values of FRAME_NUMBER will repeat, and a frame with FRAME_NUMBER=2^{32}-1 and a subsequent frame with FRAME_NUMBER=0 may both be in the same display buffer. The usual numerical order will cause the second frame to be displayed first.

However, if we define a new relation

$x \prec y$

if and only if either

a) $x<y$ and $y-x<2^{31}$

or

b) $y<x$ and $x-y \geq 2^{31}$.

This relation is no longer transitive, but allows an order to be discriminated. This will be the correct order provided that the encoder maintains a reordering depth less than 2^{31} frames. This will certainly be true for any practical profile.

5.5 Frame Prediction decoding process

This section specifies the process for decoding motion vector data used for motion compensation. This process is invoked in decoding a frame if IS_INTRA is FALSE. The Frame Prediction element is only present for Inter frames.

Inputs to this process are: None.

Outputs of this process are: XBLEN_LUMA, XBLEN_CHROMA, YBLEN_LUMA, YBLEN_CHROMA, XBSEP_LUMA, XBSEP_CHROMA, YBSEP_LUMA, YBSEP_CHROMA, X_NUM_MB, Y_NUM_MB, MC_LUMA_WIDTH, MC_LUMA_HEIGHT, MC_CHROMA_WIDTH, MC_CHROMA_HEIGHT; a value of MB_SPLIT and MB_COMMON for each macroblock in the frame; a value of PRED_MODE for each Prediction Unit in each macroblock; a value DC_VAL for each prediction unit for which PRED_MODE is INTRA; a motion vector MV1 for each prediction unit for which PRED_MODE is REF1ONLY or REF1AND2; a motion vector MV2 for each prediction unit for which PRED_MODE is REF2ONLY or REF1AND2.

The structure of the Frame Prediction data unit is specified by Figure 9 below.

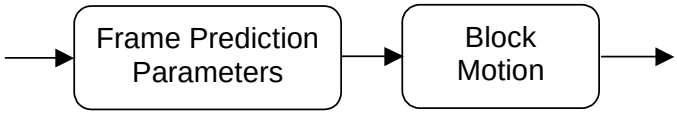


Figure 9 Frame Prediction structure

Each element of the Frame Prediction data unit is byte-aligned and occupies a whole number of bytes.

The Frame Prediction Parameters contain data determining motion vector precision, dimensions and overlaps of blocks used for motion compensation, global motion parameters (if any), and whether and how global motion is used.

The Block Motion data unit consists of raw arithmetically coded data bytes, which are decoded to produce motion vector data for prediction units within each MB.

The Frame Prediction decoding process is as follows:

1. Decode the Frame Prediction parameters as per Section 5.5.2
2. If GLOBAL_MOTION_FLAG is TRUE, generate the global motion field(s) as per Section 5.5.3
3. Decode the arithmetically coded Block Motion data as per Section 5.6

5.5.1 Motion model

Dirac predicts Inter frames from one or two reference frames using motion compensation. Dirac's motion model is overlapping block motion compensation (OBMC). The use of overlapping blocks mitigates blocking artefacts. The spatial

displacement of each block, from the corresponding position in a reference frame, is coded in the bitstream. These displacements are known as motion vectors. Motion vectors are the displacement that should be applied to the reference frame to predict the current frame.[2]

Two types of motion vector information are used: global and block motion vectors. Global motion is intended to describe the motion of the background, using a parametric model. The block motion vectors are intended to describe the more varied motion of the foreground. The two type of motion information are used together to define the overall motion vectors. The way in which they are combined is defined in subsequent sections.

In motion vector decoding, block data is organised into sub-macroblock s and macroblock s which are arrays of 2x2 and 4x4 blocks. Blocks, sub-macroblocks and macroblocks are all termed Prediction Units.

5.5.2 Frame Prediction Parameters decoding

This section specifies the process for decoding Frame Prediction Parameters.

Inputs to this process are: NUM_REFS

Outputs of this process are: XBLEN_LUMA, XBLEN_CHROMA, YBLEN_LUMA, YBLEN_CHROMA, XBSEP_LUMA, XBSEP_CHROMA, YBSEP_LUMA, YBSEP_CHROMA, MV_PRECISION, X_NUM_MB, Y_NUM_MB, MC_LUMA_WIDTH, MC_LUMA_HEIGHT, MC_CHROMA_WIDTH, MC_CHROMA_HEIGHT; global motion parameters **A[**0], **A**[1], **b**[0] ,**b**[1] , **c**[0], **c**[1], BLOCK_DATA_LENGTH

The structure of the Frame Prediction Parameters is shown in Figure 10.

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
BLOCK_PARAMS_FLAG	bool	-	1	Literal	-
if (BLOCK_PARAMS_FLAG) {					
BLOCK_PARAMS_INDEX	Integer	No	32	Uegol(n)	0,1,2,3
if (BLOCK_PARAMS_INDEX==0) {					
XBLEN_LUMA	Integer	No	32	Uegol(n)	-
YBLEN_LUMA	Integer	No	32	Uegol(n)	-
XBSEP_LUMA	Integer	No	32	Uegol(n)	-
XBSEP_LUMA	Integer	No	32	Uegol(n)	-
}					
}					

[2] The name “motion vector” is, therefore, a misnomer because they are actually prediction displacement vectors. Nevertheless the term motion vector is used for consistency with industry practice.

MV_PRECISION_FLAG	bool	-	1	Literal	-
if (MV_PRECISION_FLAG) {					
MV_PRECISION_INDEX	Integer	No	32	uegol(n)	0,1,2,3
}					
GLOBAL_MOTION_FLAG	bool	-	1	Literal	-
if (GLOBAL_MOTION_FLAG) {					
GLOBAL_ONLY_FLAG	bool	-	1	Literal	-
GLOBAL_PREC_BITS	Integer	No	8	uegol(n)	-
for (i=0; i<NUM_REFS; ++i) {					
PAN_ZERO[i]	bool	-	1	Literal	-
if (!PAN_ZERO[i]) {					
B_1[i]	Integer	Yes	32	segol(n)	-
B_2[i]	Integer	Yes	32	segol(n)	-
}					
MATRIX_ZERO[i]	bool	-	1	Literal	-
if (!MATRIX_ZERO[i]) {					
A_11[i]	Integer	Yes	32	segol(n)	-
A_12[i]	Integer	Yes	32	segol(n)	-
A_21[i]	Integer	Yes	32	segol(n)	-
A_22[i]	Integer	Yes	32	segol(n)	-
}					
PERSPECTIVE_ZERO	bool	-	1	Literal	-
if (!PERSPECTIVE_ZERO) {					
C_1[i]	Integer	Yes	32	segol(n)	-
C_2[i]	Integer	Yes	32	segol(n)	-
}					
}					
}					
BLOCK_DATA_LENGTH	Integer	No	32	uegol(n)	>0

Figure 10 Frame Prediction Parameters

Frame Prediction Parameters are byte-aligned and occupy a whole number of bytes. They are padded with up to 7 unspecified bits as necessary.

The process for decoding the Frame Prediction Parameters is as follows.

Block parameters

Block parameters determine how OBMC is performed, by setting the horizontal and vertical block separation and the horizontal and vertical length.

These parameters also affect wavelet transform operation and coefficient data decoding, since they require the picture to be padded if there is a non-integral number of macroblocks vertically or horizontally. See Section 5.7.2.

If BLOCK_PARAMS_FLAG is FALSE, then default values of BLOCK_PARAMS_INDEX are used. These are determined from the video format according to Table 23 below

VIDEO_FORMAT	BLOCK_PARAMS_INDEX
Custom Format	2
QSIF	1
QCIF	1
SIF	2
CIF	2
SD (NTSC)	2
SD (PAL)	2
SD (525 Digital)	2
SD (625 Digital)	2
HD 720	3
HD 1080	4

Table 23 Default values of BLOCK_PARAMS_INDEX

If BLOCK_PARAMS_FLAG is FALSE then the BLOCK_PARAMS_INDEX is decoded. This is an index into Table 24, giving luma block dimensions and separations.

BLOCK_PARAMS_INDEX	Luma Block Parameters:	
	Size (XBLEN_LUMA x YBLEN_LUMA)	Separation (XBSEP_LUMA x YBSEP_LUMA)
0	Custom	Custom
1	8x8	4x4
2	12x12	8x8
3	16x16	10x12
4	24x24	16x16

Table 24 Preset block dimensions and separations for OBMC

If BLOCK_PARAMS_INDEX is 0, then custom parameters are present. The values of XBLEN_LUMA, YBLEN_LUMA, XBSEP_LUMA and YBSEP_LUMA are decoded.

Derivation of chroma block parameters

If custom parameters are not present, chroma block dimensions are determined according to the value of CHROMA_FORMAT and BLOCK_PARAMS_INDEX as per Table 25 below.

BLOCK_PARAMS_INDEX	CHROMA_FORMAT	Chroma Block Parameters:	
		Size (XBLEN_CHROMA x YBLEN_CHROMA)	Separation (XBSEP_CHROMA x YBSEP_CHROMA)
1	4:2:0	4x4	2x2
	4:2:2	4x8	2x4
	4:1:1	Not supported	
	4:4:4	8x8	4x4
2	4:2:0	6x6	4x4
	4:2:2	6x12	4x8
	4:1:1	4x12	2x8
	4:4:4	12x12	8x8
3	4:2:0	9x8	5x6
	4:2:2	9x16	5x12
	4:1:1	Not supported	
	4:4:4	16x16	10x12
4	4:2:0	12x12	8x8
	4:2:2	12x24	8x16
	4:1:1	6x24	4x16
	4:4:4	24x24	16x16

Table 25 Non-custom chroma block parameters

If BLOCK_PARAMS_INDEX=0, then XBLEN_LUMA and YBLEN_LUMA shall be such that:

XBSEP_LUMA%CHROMA_H_SCALE=0
YBSEP_LUMA%CHROMA_V_SCALE=0

Then

XBSEP_CHROMA = XBSEP_LUMA//CHROMA_H_SCALE
YBSEP_CHROMA = YBSEP_LUMA//CHROMA_V_SCALE

XBLEN_CHROMA shall be set to be the smallest integral value such that

a) XBLEN_CHROMA>XBSEP_CHROMA

b) (XBLEN_CHROMA-XBSEP_CHROMA)%2=0

c) XBLEN_CHROMA≥(XBLEN_LUMA/CHROMA_H_SCALE)

d) XBLEN_CHROMA≤2*XBSEP_CHROMA

YBLEN_CHROMA is set to be the smallest integral value such that

e) YBLEN_CHROMA>YBSEP_CHROMA

f) (YBLEN_CHROMA-YBSEP_CHROMA)%2=0

g) YBLEN_CHROMA≥(YBLEN_LUMA/CHROMA_V_SCALE)

h) YBLEN_CHROMA≤2*YBSEP_CHROMA

XBLEN_LUMA, YBLEN_LUMA, XBSEP_LUMA and YBSEP_LUMA shall be set so that conditions a) to h) may be satisfied simultaneously.

Frame padding and block dimensions

Frame data for Inter frames must be padded in order to support an integral number of macroblocks vertically and horizontally. Further padding may be added for the purposes of applying the wavelet transform (Section 5.7.2), but this further padding is removed prior to motion compensation (Section 5.11.2). The motion compensation padding is set as follows.

The number of macroblocks horizontally is set as

$$X\_NUM\_MB=\left\lceil \frac{LUMA\_WIDTH}{4*XBSEP\_LUMA} \right\rceil$$

The number of macroblocks vertically is set as

$$Y\_NUM\_MB=\left\lceil \frac{LUMA\_HEIGHT}{4*YBSEP\_LUMA} \right\rceil$$

The number of blocks horizontally is set as

X_NUM_BLOCKS=4*X_NUM_MB

The number of blocks vertically is set as

Y_NUM_BLOCKS=4*Y_NUM_MB

The padded height and width are set as

MC_LUMA_WIDTH=4*XBSEP_LUMA*X_NUM_MB

MC_LUMA_HEIGHT=4*YBSEP_LUMA*Y_NUM_MB

MC_CHROMA_WIDTH=MC_LUMA_WIDTH//CHROMA_H_SCALE

MC_CHROMA_HEIGHT=MC_LUMA_HEIGHT//CHROMA_V_SCALE

These values are used to determine the amount of padding to be applied to the frame prior to performing motion compensation (Section 5.12).

Motion Vector Precision

If MV_PRECISION_FLAG is TRUE, then the MV_PRECISION_INDEX is decoded. Value 0 corresponds to integer pixel accuracy, 1 to half-pixel, 2 to quarter pixel and 3 to eighth pixel accuracy.

If MV_PRECISION_FLAG is FALSE, then MV_PRECISION_INDEX is set to be 1 i.e. default motion vector precision is half-pixel.

Global motion

The GLOBAL_MOTION_FLAG indicates the presence of global motion data. If TRUE, the GLOBAL_ONLY_FLAG is decoded. This indicates whether only global motion data is present.

If GLOBAL_MOTION_FLAG is FALSE then GLOBAL_ONLY_FLAG is set to FALSE by default.

If GLOBAL_MOTION_FLAG is TRUE then global motion model data is decoded.

For each i from 0 to NUM_REFS-1, if PAN_ZERO[i] is TRUE then B_1[i] and B_2[i] are set to 0.

For each i from 0 to NUM_REFS-1, if MATRIX_ZERO[i] is TRUE then A_11[i], A_12[i], A_21[i] and A_22[i] are set to 0.

For each i from 0 to NUM_REFS-1, if PERSPECTIVE_ZERO[i] is TRUE then C_1[i] and C_2[i] are set to 0.

GLOBAL_PREC_BITS is the number of precision bits with which the matrix and perspective elements of the global motion model have been encoded. Real-valued matrices **A**[i] are derived by the recipe:

$$\mathbf{A}[i]=2^{\text{-GLOBAL\_PREC\_BITS}} \cdot \begin{pmatrix} \text{A\_11}[i] & \text{A\_12}[i] \\ \text{A\_21}[i] & \text{A\_22}[i] \end{pmatrix}$$

Real-valued perspective vectors **c**[i] are derived by

$$\mathbf{c}[i]=2^{\text{-GLOBAL\_PREC\_BITS}} \cdot \begin{pmatrix} \text{C\_1}[i] \\ \text{C\_2}[i] \end{pmatrix}$$

The accuracy pan elements are given to integer accuracy:

$$\mathbf{b}[i]=\begin{pmatrix} \text{B\_1}[i] \\ \text{B\_2}[i] \end{pmatrix}$$

Block data length

BLOCK_DATA_LENGTH shall be the length in bytes of the subsequent Block Motion data unit.

5.5.3 Global motion field generation

This section specifies how global motion fields are generated from the global motion parameters **A**[i], **b**[i], and **c**[i].

A three-dimensional array of motion vectors GLOBAL[][][] is derived as follows, where the first index runs from 0 to NUM_REFS-1, the second index from 0 to Y_NUM_BLOCKS-1 and the third index runs from 0 to X_NUM_BLOCKS-1:

$$\text{GLOBAL}[i][y][x] = \text{round}\left(\frac{\mathbf{A}[i]\mathbf{x} + \mathbf{b}[i]}{\mathbf{c}[i]^{\mathbf{T}}\mathbf{x} + 1} \right), \text{ where } \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$$

The two-dimensional arrays GLOBAL[i] correspond to global motion vector fields for each reference, providing a motion vector for each block. The motion vectors are in units of $2^{\text{-MV\_PRECISION}}$.

INFORMATIVE

B_1[i] and B_2[i] are not scaled because a simple pan (**A**[i]=**0**, **c**[i]=**0**) must be computed to MV_PRECISION bits of accuracy.

5.6 Block Motion Data decoding

This section specifies the process for decoding the Block Motion data unit. This process is invoked for Inter frames when decoding the Frame Prediction data unit. The Block Motion data unit is byte aligned and occupies a whole number of bytes, padded with zero bits as necessary. Its size in bytes must be equal to BLOCK_DATA_LENGTH.

Inputs to this process are: global motion field GLOBAL[][][]; GLOBAL_MOTION_FLAG, X_NUM_MB, Y_NUM_MB, X_NUM_BLOCKS, Y_NUM_BLOCKS, NUM_REFS,

Outputs from this process are: a value of MB_SPLIT and MB_COMMON for each MB; a value of PRED_MODE for each prediction unit in each MB; a motion vector MV1 for each prediction unit with PRED_MODE equal to REF1ONLY or REF1AND2; a motion vector MV2 for each prediction unit with PRED_MODE equal to REF2ONLY or REF1AND2.

The Block Motion data unit is a single block of arithmetically coded binary data. This section specifies the decoding operations to be used in conjunction with the arithmetic decoding engine specified in Section 6 with the contexts and initialisation defined in Section5.6.10.

Block motion data is used in predicting Inter frames using either global or block motion or both. When block motion is used it encodes the motion vectors to be used. When two reference frames are used it encodes motion vectors for both references. With two references it also encodes which reference, or both references, are to be used and which blocks are to be coded Intra (i.e. without using motion compensated prediction). The Block Motion data also encodes any other information that is needed to perform motion compensated prediction, such as macroblock splitting.

Motion vector data is organised into macroblock s, which are 4x4 arrays of blocks. The motion data is decoded by decoding the data in each MB, scanning in raster order from the top-left corner.

Numbering

For the purposes of this specification, macroblocks are numbered by x- and y-coordinates in raster order, from the top-left MB. Blocks are also numbered from by x- and y-coordinates in raster order from the top-left block. The indices of blocks within a macroblock with coordinates (x,y) therefore run from (4x,4y) (top-left) through to (4x+3,4y+3) (bottom right).

Overall decoding process

The decoding process iterates across all macroblocks, first decoding the macroblock data and then decoding the prediction unit data pertaining to each macroblock. The macroblock data constrains how much block data there is and how it is interpreted. The decoder maintains a value MB_COUNT which is used to reset statistics periodically.

In pseudocode the decoding process is:

MB_COUNT=0

```
for (y=0;  y<Y_NUM_MB;  y++)
{
   for (x=0;  x<X_NUM_MB;  x++)
   {
        decode_MB_data (x,y)
        decode_MB_block_data(x,y)
        MB_COUNT++
        if (MB_COUNT>32)
           halve_all_counts()
   }
}
```

The arithmetic decoding engine function halve_all_counts() is specified in Section 6.4.3.

5.6.1 Motion vector data prediction apertures

This section specifies the aperture used for predicting motion vector data, namely the previously decoded motion vector data elements which may be used for prediction should a suitable value exist.

All motion vector data is differentially decoded. The differential element is performed by forming a prediction from previously decoded values. Various methods of prediction are used for predicting different elements of motion vector data for differential, however in each case the data elements used to form the prediction are defined in the same way. Data elements which may be available for prediction are termed the prediction aperture.

MB data prediction aperture

The nominal prediction aperture for macroblock data is defined to be the existing macroblocks to the top, left and above the current MB. For macroblocks on the left-hand side of the frame, but not the top-left MB, the aperture consists of only the macroblock above for macroblocks on the top of the frame, but not the top-left macroblock the aperture consists solely of the macroblock to the left for the top-left macroblock the aperture is empty. The nominal aperture for macroblock data is illustrated in Figure 11 below.

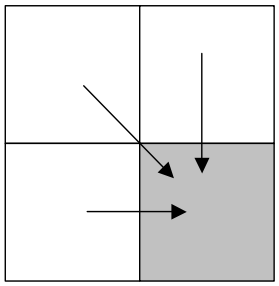


Figure 11 Macroblock prediction aperture

Block motion data prediction aperture

The nominal prediction aperture for block motion data is defined to be: the prediction units containing the blocks to the left, top and top-left of the top-left block in the current prediction unit. An example aperture for different macroblock splittings is shown in Figure 12. Prediction units at the top and left of the frame have prediction aperture restricted to those prediction units in the nominal aperture that lie within the frame.

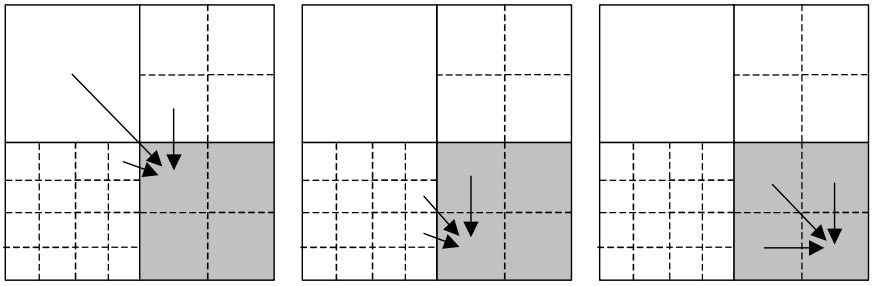


Figure 12 Prediction aperture for prediction units within a MB.

Where a prediction unit in the prediction aperture has a value of the same type as that being decoded, it forms part of the prediction. However, not all prediction units in the prediction aperture may have such a value. For example if Reference 2 motion vectors are being decoded but the top-left prediction unit only has Reference 1 motion vectors, then it will not provide values in the prediction.

5.6.2 MB data

This section specifies the decode_MB_data(x,y) process.

Inputs to this process are: none.

Outputs of this process are: a value of MB_USING_GLOBAL, MB_SPLIT and MB_COMMON for the macroblock at position (x,y).

MB data is only present if GLOBAL_ONLY_FLAG is FALSE.

The structure of the macroblock data is shown in Figure 13.

Name	Type	Signed	Size (bits)	Value restrictions
if (!GLOBAL_ONLY_FLAG) {				
if (GLOBAL_MOTION_FLAG) {				
MB_USING_GLOBAL	Bool	-	1	-
}				
if (!MB_USING_GLOBAL) {				
MB_SPLIT	Integer	No	2	0,1,2

}				
if (MB_SPLIT!=0) {				
MB_COMMON	Bool	-	1	-
}				
}				

Figure 13 Macroblock Structure

The decoding process for a macroblock at coordinates (x,y) is as follows, in pseudocode

```
if (!GLOBAL_ONLY_FLAG)
{
   MB_SPLIT=2
   MB_COMMON=TRUE
}
else
{
    if (GLOBAL_MOTION_FLAG)
       decode_mb_using_global(x,y)
    else
       MB_USING_GLOBAL=FALSE

    if (!MB_USING_GLOBAL)
       decode_mb_split(x,y)
    else
       MB_SPLIT=2

    if (MB_SPLIT!=0)
       decode_mb_common(x,y)
    else
       MB_COMMON=TRUE
}
```

The subsidiary decoding functions decode_mb_using_global(), decode_mb_split(), decode_mb_common() are specified in subsequent sections.

The MB_SPLIT parameter determines to what level the macroblock shall be split i.e. how many Prediction Units it contains and what size they are. If MB_SPLIT=2, then the macroblock is split into a 4x4 array of macroblock s. If MB_SPLIT=1, then the macroblock is split into a 4x4 array of sub-macroblocks. If MB_SPLIT=0, then the

macroblock is not split. For each prediction unit in the MB, motion vectors and possibly prediction modes will be extracted from the bitstream, so MB_SPLIT determines how many values must be extracted for the MB.

The MB_COMMON parameter determines whether a common prediction mode is to be decoded for the whole macroblock contained within the bitstream, or whether different prediction modes are to be decoded for each prediction unit. If MB_SPLIT=0, MB_COMMON is not necessary, since in this case there is only one prediction unit are the two cases are identical.

5.6.3 MB_USING_GLOBAL decoding

This section specifies the function decode_mb_using_global(x,y) for a macroblock at coordinates (x,y).

MB_USING_GLOBAL is decoded differentially with respect to a prediction. The reconstruction procedure is

MB_USING_GLOBAL(x,y)=binary_arith_decode()+mb_using_global_prediction(x, y) (mod 2)

A single context is used: MB_USING_GLOBAL_CTX.

The prediction function mb_using_global_prediction(x,y) is defined as the mean of available previously-decoded predictors in the prediction arrangement defined in Section5.6.1:

if (x>0 && y>0)

mb_using_global_prediction(x,y)=(MB_USING_GLOBAL (x-1,y)+
MB_USING_GLOBAL (x-1,y-1)+
MB_ USING_GLOBAL (x,y-1))/3

It returns a value 0 or 1.

5.6.4 MB_SPLIT decoding

This section specifies the function decode_mb_split(x,y) for a macroblock at coordinates (x,y).

MB_SPLIT is decoded differentially with respect to a prediction. The MB_SPLIT prediction residue is encoded in the bitstream using truncated unary binarisation, since the values are constrained to lie in the range 0-2. The reconstruction procedure is therefore

MB_SPLIT(x,y)=ut_arith_decode()+mb_split_prediction(x,y) (mod 3)

Two contexts are used for bin 1 and bin 2: MB_SPLIT_CTX_BIN1 and MB_SPLIT_CTX_BIN2.

The prediction function mb_split_prediction(x,y) is defined as the mean of available previously-decoded MB_SPLIT values in the neighbouring predictor macroblocks, as per Section 5.6.1:

if (x>0 && y>0)

mb_split_prediction(x,y)=(MB_SPLIT(x-1,y)+ MB_SPLIT(x-1,y-1)+ MB_SPLIT(x,y-1))/3

It returns a value 0,1 or 2.

5.6.5 MB_COMMON decoding

This section specifies the function decode_mb_split(x,y) for a macroblock at coordinates (x,y).

MB_COMMON is decoded differentially with respect to a prediction. The reconstruction procedure is

MB_COMMON(x,y)=binary_arith_decode()+mb_split_prediction(x,y) (mod 2)

The MB_COMMON prediction residue is encoded in the bitstream as a single binary bit, with context MB_COMMON_CTX.

The prediction function mb_common_prediction(x,y) is defined as the mean of available previously-decoded predictors in the prediction arrangement defined in Section 5.6.1:

if (x>0 && y>0)

mb_common_prediction(x,y)=(MB_COMMON (x-1,y)+ MB_COMMON(x-1,y-1)+
MB_COMMON(x,y-1))/3

It returns a value 0 or 1.

5.6.6 Prediction modes

Four prediction modes shall be supported by the decoder:

INTRA

REF1ONLY

REF2ONLY

REF1AND2.

These correspond to using DC prediction, using a motion compensated prediction from Reference 1 only, using a motion compensated prediction from Reference 2 and using a motion compensated prediction composed from data from both Reference 1 and Reference 2.

The prediction modes shall be identified with two-bit words as follows:

INTRA=b00

REF1ONLY=b01

REF2ONLY=b10

REF1AND2=b11

In this way, Reference 1 is used for prediction if and only if (PRED_MODE&b01) and Reference 2 is used for prediction if and only if (PRED_MODE&b10).

5.6.7 Block motion data decoding

This section specifies the operation of decode_MB_block_data(x,y). This process is invoked in the decoding of each MB.

Block motion data consists of decoding the prediction mode and motion vectors for all the prediction units in the MB. The process depends upon the values of MB_SPLIT and MB_COMMON. The decoding process is as follows: scan the prediction units within a macroblock in raster order and for each prediction unit:

1. If MB_COMMON is TRUE, decode a mode PRED_MODE for the top-left prediction unit in the macroblock as per Section 5.6.8. This mode shall also be used for all prediction units in the MB.

2. Loop over all the prediction units in the macroblock in raster order, and for each prediction unit:

    a) If MB_COMMON is FALSE decode a mode to be used for the prediction unit as per Section 5.6.8 and set as PRED_MODE

    b) If PRED_MODE is REF1ONLY or REF1AND2 decode a motion vector MV1

    c) If PRED_MODE is REF2ONLY or REF1AND2 decode a motion vector MV2

    d) If PRED_MODE is INTRA, decode a DC value PU_DC

INFORMATIVE

The identification of the PRED_MODE value for the macroblock with that for the top-left prediction unit is so that the prediction aperture defined in Section 5.6.1 can be consistently applied.

5.6.8 Block prediction mode decoding

PRED_MODE consists of two bits of information (for reference 1 and reference 2) and is decoded differentially with respect to predictions applying to each bit.

Two contexts are used: PRED_MODE_BIT1_CTX and PRED_MODE_BIT2_CTX.

The decoding process is:

PRED_MODE=binary_arith_decode(PRED_MODE_BIT1_CTX)

PRED_MODE|= (binary_arith_decode(PRED_MODE_BIT2_CTX)<<1)

PRED_MODE^= block_mode_pred()

PRED_MODE prediction

PRED_MODE is predicted by predicting the first and second bit independently as the mode of the corresponding bits of prediction modes for prediction units in the prediction aperture.

I.e. prediction units not on at the top or left side of the frame:

block_mode_pred()&1=mode($PRED\_MODE_L$&1, $PRED\_MODE_T$&1, $PRED\_MODE_T$&1)

block_mode_pred()&2=mode($PRED\_MODE_L$&2, $PRED\_MODE_T$&2, $PRED\_MODE_T$&2)

For prediction units on the left side of the frame other than the top-left prediction unit in the frame:

block_mode_pred()=PRED_MODE$_T$

For prediction units on the top side of the frame other than the top-left prediction unit in the frame:

block_mode_pred()=PRED_MODE$_L$

For the top-left prediction unit:

block_mode_pred()=INTRA[**NB: this is inconsistent with current software**]

5.6.9 Block motion vector decoding

This section specifies the decoding process for motion vectors in a prediction unit whose top-left block has coordinates (x,y).

If MB_USING_GLOBAL is TRUE then: if PRED_MODE&1 is TRUE, MV1 is equal to GLOBAL[0][y][x]; if PRED_MODE&2 is TRUE, MV2 is GLOBAL[1][y][x].

If MB_USING_GLOBAL is FALSE then block motion vectors are differentially decoded with respect to a prediction. The decoding process is as follows.

Motion vector prediction residues are decoded horizontal component first, followed by the vertical component. The values use signed unary binarisation. The decoding process for each motion vector is:

1. MV_x=mv_pred()+su_arith_decode()
2. MV_y=mv_pred()+su_arith_decode()

Different contexts are used for Reference 1 and Reference 2 vectors, for horizontal and vertical components, for different bins and for sign and magnitude data, as per Table 26.

MV contexts			
Reference 1			
Horizontal		**Vertical**	
Magnitude Contexts	**Sign Context**	**Magnitude**	**Sign Context**
Bin **Context** 1 REF1x_BIN1_CTX 2 REF1x_BIN2_CTX 3 REF1x_BIN3_CTX 4 REF1x_BIN4_CTX ≥5 REF1x_BIN5+_CTX	REF1x_SIGN_CTX	**Bin** **Context** 1 REF1y_BIN1_CTX 2 REF1y_BIN2_CTX 3 REF1y_BIN3_CTX 4 REF1y_BIN4_CTX ≥5 REF1y_BIN5+_CTX	REF1y_SIGN_CTX
Reference 2			
Horizontal		**Vertical**	
Magnitude	**Sign**	**Magnitude**	**Sign Context**

Bin	Context	REF2y_SIGN_CTX	Bin	Context	REF2y_SIGN_CTX
2	REF2x_BIN1_CTX		2	REF2y_BIN1_CTX	
2	REF2x_BIN2_CTX		2	REF2y_BIN2_CTX	
3	REF2x_BIN3_CTX		3	REF2y_BIN3_CTX	
4	REF2x_BIN4_CTX		4	REF2y_BIN4_CTX	
≥5	REF2x_BIN5+_CTX		≥5	REF2y_BIN5+_CTX	

Table 26 Contexts for Reference1 and Reference2 Vectors

Prediction

The prediction process mv_pred() predicts Reference 1 motion as the median of available Reference 1 motion vectors from prediction units in the prediction aperture, and Reference 2 motion vectors of available Reference 2 motion vectors from prediction units in the prediction aperture. The prediction aperture is as specified in Section 5.6.1.

Reference 1 motion vectors in a prediction unit are available if PRED_MODE&1 is non-zero.

Reference 2 motion vectors in a prediction unit are available if PRED_MODE&2 is non-zero.

5.6.10 Motion vector data context initialisation

Motion vector data contexts are initialised prior to decoding according to Table 27.

Context	COUNT0	COUNT1
REF1x_BIN1_CTX	1	1
REF1x_BIN2_CTX	1	1
REF1x_BIN3_CTX	1	1
REF1x_BIN4_CTX	1	1
REF1x_BIN5+_CTX	1	1
REF1y_BIN1_CTX	1	1
REF1y_BIN2_CTX	1	1
REF1y_BIN3_CTX	1	1
REF1y_BIN4_CTX	1	1
REF1y_BIN5+_CTX	1	1
REF2x_BIN1_CTX	1	1
REF2x_BIN2_CTX	1	1
REF2x_BIN3_CTX	1	1
REF2x_BIN4_CTX	1	1
REF2x_BIN5+_CTX	1	1
REF2y_BIN1_CTX	1	1

REF2y_BIN2_CTX	1	1
REF2y_BIN3_CTX	1	1
REF2y_BIN4_CTX	1	1
REF2y_BIN5+_CTX	1	1
REF1x_SIGN_CTX	1	1
REF1y_SIGN_CTX	1	1
REF2x_SIGN_CTX	1	1
REF2y_SIGN_CTX	1	1
MB_SPLIT_BIN1_CTX	1	1
MB_SPLIT_BIN2_CTX	1	1
MB_COMMON_CTX	1	1
PRED_MODE_BIT1_CTX	1	1
PRED_MODE_BIT2_CTX	1	1

Table 27 Motion vector data context initialisation [True values TBD]

5.7 Coefficient Data decoding

This section defines the decoding process for decoding the Coefficient Data. This is only present in the bitstream if ZERO_COEFFS_FLAG is FALSE.

Inputs to this process are: CHROMA_FORMAT

Outputs to this process are: data arrays YDATA[][], C1DATA[][], C2DATA[][].

Coefficient Data consists of data for each of the video components. The structure of the Coefficient Data unit is given in

Figure 14.

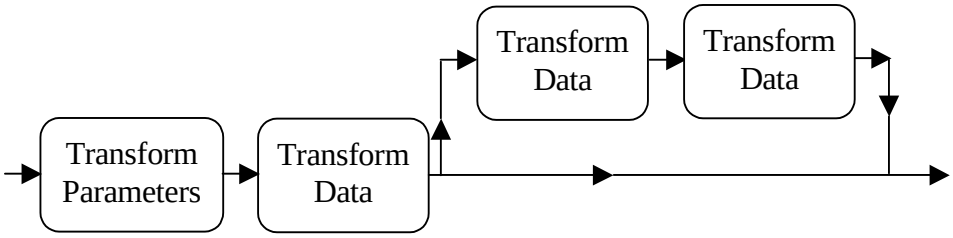


Figure 14 Coefficient Data unit structure

The overall decoding procedure for Coefficient Data is as follows.

1. Decode Transform Parameters
2. Decode the first Transform Data element to produce a decoded data array DATA[][] and set as the luma data YDATA[][]
3. If CHROMA_FORMAT is not YONLY then decode the second Transform Data Element to produce a decoded data array DATA[][] and set as

C1DATA[][]. Then decode the Third Transform Data element to produce a decoded data array DATA[][] and set as C2DATA[][].

5.7.1 Transform Parameters

This sections specifies the decoding process for the Transform Parameters.

Inputs to this process are: None.

Outputs of this process are: NON_DEFAULT_TRANSFORM, WAVELET_FILTER, TRANSFORM_DEPTH, IWT_CHROMA_WIDTH, IWT_CHROMA_HEIGHT, IWT_LUMA_WIDTH, IWT_LUMA_HEIGHT SPATIAL_PARTITION, MAX_XBLOCKS, MAX_YBLOCKS, MULTI_QUANT.

The Transform Parameters data unit consists of the following data elements shown in Figure 15.

Name	Type	Signed	Size (bits)	Encoding	Value restrictions
NON_DEFAULT_TRANSFORM	bool	-	-	Literal	-
if (NON_DEFAULT_TRANSFORM) {					
WAVELET_FILTER_INDEX	Integer	No	8	uegol(n)	≤4
}					
NON_DEFAULT_DEPTH	bool	-	-	Literal	-
if (NON_DEFAULT_DEPTH) {					
TRANSFORM_DEPTH	Integer	No	8	uegol(n)	≤6
}					
SPATIAL_PARTITION	bool	-	-	Literal	-
if (SPATIAL_PARTITION) {					
PARTITION_INDEX	Integer	No	8	uegol(n)	-
if (PARTITION_INDEX==0) {					
MAX_XBLOCKS	Integer	No	8	uegol(n)	-
MAX_YBLOCKS	Integer	No	8	uegol(n)	-
}					
MULTI_QUANT	bool	-	-	Literal	-
}					

Figure 15 Transform parameters data unit

WAVELET_FILTER_INDEX is an index into Table 28 and determines the filter WAVELT_FILTER to be used in the IWT (note that this Table may be extended in future Profiles). If NON_DEFAULT_TRANSFORM is TRUE, then

WAVELET_FILTER_INDEX is decoded if NON_DEFAULT_TRANSFORM is FALSE then WAVELET_FILTER shall be set to DAUBECHIES97 i.e. the default wavelet filters are the Daubechies (9,7) filters.

WAVELET_FILTER_INDEX	WAVELET_FILTER
0	DAUBECHIES97
1	APPROX_DAUBECHIES
2	FIVETHREE
3	THIRTEENFIVE

Table 28 Wavelet filters for IWT operation.

NON_DEFAULT_DEPTH is a flag indicating whether a non-default depth value follows. If FALSE, then TRANSFORM_DEPTH is set to 4. If TRUE then TRANSFORM_DEPTH is decoded.

TRANSFORM_DEPTH represents the number of vertical-horizontal synthesis operations performed in the IWT operation (Section 5.11). Once TRANSFORM_DEPTH is determined, the dimensions

IWT_CHROMA_WIDTH
IWT_CHROMA_HEIGHT
IWT_LUMA_WIDTH
IWT_LUMA_HEIGHT

of the padded data which will be output from the IWT are derived, as per Section 5.7.2.

SPATIAL_PARTITION is a flag indicating whether coefficient data is spatially partitioned into code blocks within a subband. If TRUE, an index PARTITION_INDEX into Table 29 below is decoded, which determines the partitioning scheme employed.

PARTITION_INDEX	Partition
0	Custom
1	Default

Table 29 Spatial partition methods for coefficient decoding

If PARTITION_INDEX is 0, a custom partition is encoded by setting the maximum number of code blocks horizontally and vertically, MAX_XBLOCKS and MAX_YBLOCKS. The operation of the default and custom partition modes is specified in Section 5.10.3.

If SPATIAL_PARTITION is TRUE, a flag MULTI_QUANT is decoded indicating whether separate quantisers are encoded for each code block in a subband or a single quantiser is used for the whole subband. If SPATIAL_PARTITION is FALSE then MULTI_QUANT is set to FALSE.

INFORMATIVE

These different spatial partition modes can be used to support region of interest coding.

5.7.2 Frame padding and subband dimensions

This section specifies the derivation of the dimensions of frame data arrays in the IWT and Subband Data decoding processes. These dimensions in turn determine the dimensions of wavelet subbands and code blocks. This process is invoked in the decoding of the Transform Parameters once TRANSFORM_DEPTH has been determined.

Inputs to this process are: IS_INTRA, MC_CHROMA_WIDTH, MC_CHROMA_HEIGHT, LUMA_WIDTH, LUMA_HEIGHT, TRANSFORM_DEPTH

Outputs of this process are: IWT_CHROMA_WIDTH, IWT_CHROMA_HEIGHT, IWT_LUMA_WIDTH, IWT_LUMA_HEIGHT.

For subband reconstruction, video data must be padded in order to accommodate the full depth of the wavelet transform. This is in addition to any padding performed to accommodate motion compensation block parameters, as specified in Section 5.5.2.

For Inter frames, the padded chroma dimensions are specified by

$$IWT\_CHROMA\_WIDTH = 2^{TRANSFORM\_DEPTH} * \left\lceil \frac{MC\_CHROMA\_WIDTH}{2^{TRANSFORM\_DEPTH}} \right\rceil$$

$$IWT\_CHROMA\_HEIGHT = 2^{TRANSFORM\_DEPTH} * \left\lceil \frac{MC\_CHROMA\_HEIGHT}{2^{TRANSFORM\_DEPTH}} \right\rceil$$

For Intra frames, the padded dimensions are specified by

$$IWT\_CHROMA\_WIDTH = 2^{TRANSFORM\_DEPTH} * \left\lceil \frac{CHROMA\_WIDTH}{2^{TRANSFORM\_DEPTH}} \right\rceil$$

$$IWT\_CHROMA\_HEIGHT = 2^{TRANSFORM\_DEPTH} * \left\lceil \frac{CHROMA\_HEIGHT}{2^{TRANSFORM\_DEPTH}} \right\rceil$$

For all frames the padded luma dimensions are defined by:

IWT_LUMA_WIDTH=IWT_CHROMA_WIDTH*CHROMA_H_SCALE

IWT_LUMA_HEIGHT= IWT_CHROMA_HEIGHT*CHROMA_V_SCALE

This definition ensures that component data has been padded sufficiently to accommodate all macroblocks. Note that this recipe implies that padded frame data may be of different dimensions for Inter and Intra frames.[**NB this differs from the current software]**

INFORMATIVE

These dimensions refer solely to the dimensions of reconstructed component data. Padded values will be discarded after performing the IWT, and so their value is immaterial and they may be generated by an encoder is any fashion. An encoder is likely to pad video data prior to motion compensation and transform coding, in which case a variety of different strategies are available to the encoder in order to mitigate any increase in bit rate. However, an encoder may use any suitable means.

5.7.3 Transform Data

This section specifies the process for decoding Transform Data elements.

Inputs to this process are: None.

Outputs of this process are: a two-dimensional array DATA[][] of decoded luma or chroma coefficients

If CHROMA_FORMAT is YONLY then only one Transform Data unit is present in the frame data unit, representing the Y component data. In all other cases, there are three Transform Data units in the frame data unit. The first Transform Data unit is the Y Transform Data unit, the second is the U Transform Data unit and the third is the V Transform Data unit.

Transform Data units contain no header data, but instead consist of a number of Subband Header Data and Subband Coefficient Data units as shown in Figure 16.

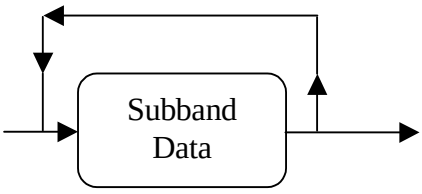


Figure 16 Transform Data unit structure

The number of Subband Data units is NUM_SUBBANDS, which is derived by:

NUM_SUBBANDS = 3*TRANSFORM_DEPTH+1

Each Subband Data unit encapsulates data from a single wavelet subband. The Subband Data units shall be presented in the bitstream in reverse numeric order, from NUM_SUBBANDS to 1 inclusive.

The decoding procedure for the Transform Data unit is:

1. Decode each subband:

   ```
   for (SUBBAND_NUM=NUM_SUBBANDS; SUBBAND_NUM>0;
        --SUBBAND_NUM)
           decode_subband()
   ```

2. Perform the Inverse Wavelet Transform on the resulting subband coefficient data to reconstruct the video component (Y, U or V):

   ```
   iwt( subbands() )
   ```

Subband data decoding is specified in Section 5.8. The Inverse Wavelet Transform is specified in Section 5.11.

5.8 Subband Data decoding

This sections specifies the process decode_subband(). This process is invoked for decoding each of the NUM_SUBBANDS Subband Data units in a Transform Data unit.

Inputs to this process are: None.

Outputs of this process are: A two-dimensional array SUBBAND[][] of decoded subband coefficients.

The structure of Subband Data elements is specified by Figure 17.

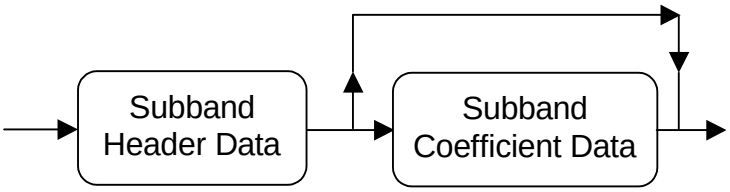


Figure 17 Subband Data structure

Both Subband Header Data and Subband Coefficient Data are byte-aligned and occupy a whole number of bytes. The decoding process is as follows.

1. Derive the subband dimensions as per Section 5.8.1.

2. Decode Subband Header Data as per Section 5.9.

2. If ZERO_SUBBAND_FLAG is FALSE, decode Subband Coefficient Data as per Section 5.10 to obtain SUBBAND[][].

3. If ZERO_SUBBAND_FLAG is TRUE, set all coefficients in SUBBAND[][] to zero.

5.8.1 Subband numbers and dimensions

This section specifies the derivation of the dimensions of the subband data arrays SUBBAND[][] from SUBBAND_NUM and the IWT padded frame dimensions.

Inputs to this process are: IWT_CHROMA_WIDTH, IWT_CHROMA_HEIGHT, IWT_LUMA_WIDTH, IWT_LUMA_HEIGHT

Outputs of this process are: SUBBAND_WIDTH, SUBBAND_HEIGHT, SCALE_FACTOR

Subbands are numbered from subband 1 to subband NUM_SUBBANDS, and are presented in reverse numerical order. The value of the subband number SUBBAND_NUM is derived from its position in the bitstream.

The correspondence between subbands and spatial frequency bands is given by Figure 18.

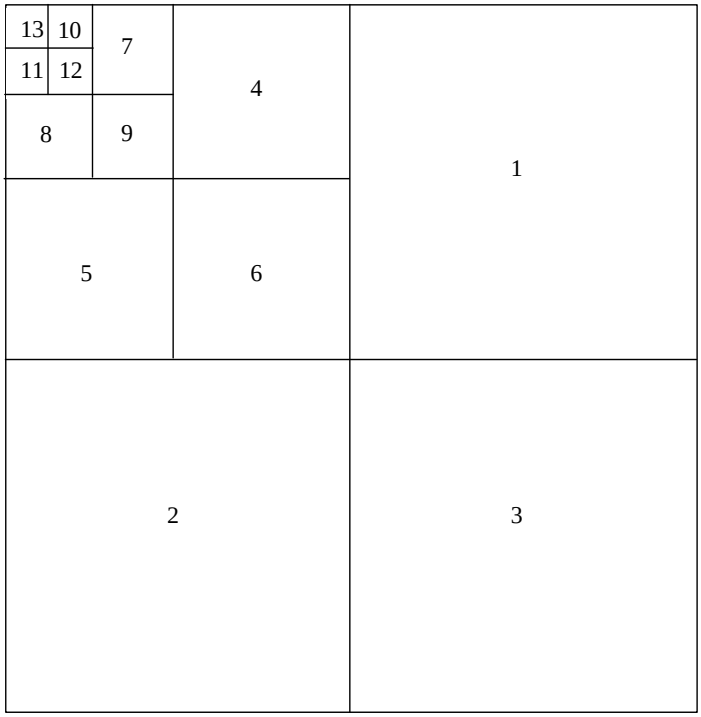


Figure 18 Subband decomposition of the spatial frequency domain showing subband numbering, for a 4-level wavelet decomposition.

For each subband a factor SCALE_FACTOR may be defined. For subbands 1 through NUM_SUBBANDS-1 this scale factor is defined by

$SCALE\_FACTOR=2^{(SUBBAND\_NUM//3)+1}$

If SUBBAND_NUM=NUM_SUBBANDS then the subband is the DC subband. In this case SCALE_FACTOR is defined by

$SCALE\_FACTOR=2^{(SUBBAND\_NUM//3)}$

The dimensions of luma subbands are defined by

SUBBAND_WIDTH=IWT_LUMA_WIDTH/SCALE_FACTOR

SUBBAND_HEIGHT=IWT_LUMA_HEIGHT/SCALE_FACTOR

The dimensions of chroma subbands are defined by

SUBBAND_WIDTH=IWT_CHROMA_WIDTH/SCALE_FACTOR

SUBBAND_HEIGHT=IWT_CHROMA_HEIGHT/SCALE_FACTOR

5.9 Subband Header Data decoding

This section specifies the process for decoding the Subband Header Data unit.

Inputs to this process are: None.

Outputs to this process are: ZERO_SUBBAND_FLAG, QUANT_INDEX, SUBBAND_LENGTH

The structure of the Subband Header Data unit is defined in Figure 19.

Name	Type	Signed	Size	Encoding	Value

			(bits)		restrictions
ZERO_SUBBAND_FLAG	bool	-	-	Literal	-
if (!ZERO_SUBBAND_FLAG) {					
QUANT_INDEX	Integer	No	8	uegol(n)	≤60
SUBBAND_LENGTH	Integer	No	32	uegol(n)	-
}					

Figure 19 Subband Header data

Subband Header Data is byte-aligned and occupies a whole number of bytes, padded with up to 7 (undefined) bits as necessary.

ZERO_SUBBAND is a flag indicating the presence of further data. If ZERO_SUBBAND is TRUE, then all data within the subband shall be set to zero.

If ZERO_SUBBAND is FALSE, QUANT_INDEX and SUBBAND_LENGTH are decoded.

5.10 Subband Coefficient Data decoding

This Section specifies the process for decoding Subband Coefficient Data elements of the bitstream. This process is invoked in decoding Subband Data elements.

Inputs to this process are: IS_INTRA, SUBBAND_NUM, NUM_SUBBANDS, SPATIAL_PARTITION, PARTITION_INDEX, MULTI_QUANT, SUBBAND_WIDTH, SUBBAND_HEIGHT; previously decoded subband data arrays within the same Transform Data element.

Outputs of this process are: A two-dimensional array SUBBAND[][] of decoded subband coefficients.

The Subband Coefficient Data unit is only present is ZERO_SUBBAND is FALSE. It is byte-aligned and occupies a whole number of bytes, padded with zero bits as necessary. It consists of a byte-aligned block of raw arithmetically-coded bytes of length SUBBAND_LENGTH.

Subband decoding conventions are specified in Section 5.10.1 and the overall subband decoding process is specified in Section 5.10.2.

5.10.1 Decoded subband data conventions

For the purposes of this specification a subband SUBBAND will be identified with a two dimensional data array of coefficients SUBBAND[][] where the first index ranges from 0 to SUBBAND_HEIGHT-1 and the second index from 0 to SUBBAND_WIDTH-1, so that

SUBBAND[y][x]

represents the x^{th} coefficient on row y.

Parents and children

If SUBBAND_NUM<NUM_SUBBANDS-3 then the subband has a parent subband. The index of the parent subband is given by SUBBAND_NUM+3.

If a subband has a parent subband then each coefficient has a parent coefficient. The coordinates of the parent coefficient to a coefficient at coordinates (x,y) are (x>>1,y>>1) in the parent subband. A parent coefficient therefore has four children (Figure 20). Parent subbands and their coefficients occur earlier in the bitstream than their children.

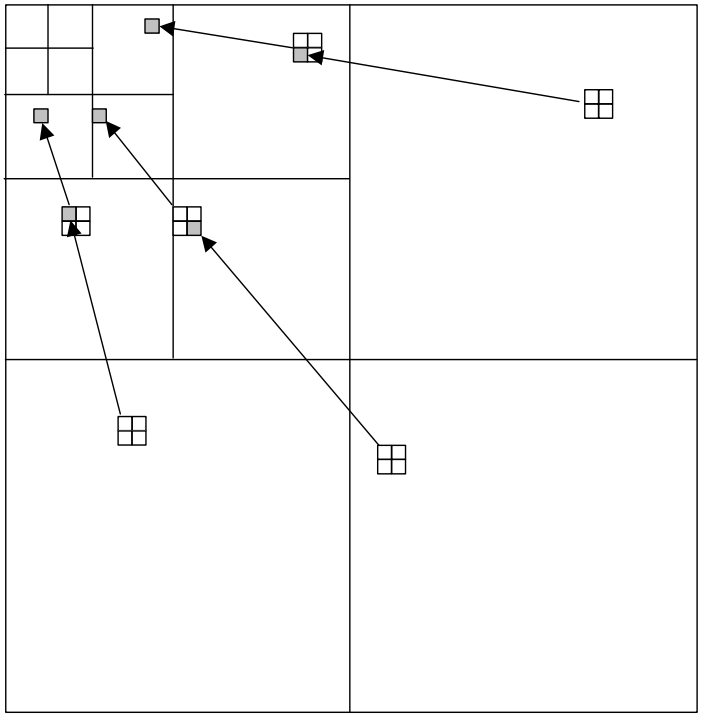


Figure 20 The relationship between child coefficients and their parents (shaded).

Orientation

All subbands other than the DC subband have an associated orientation. If SUBBAND_NUM%3=1 then the subband is vertically oriented. If SUBBAND_NUM%3=2 then the subband is horizontally oriented. If SUBBAND_NUM%3=0 then the subband is diagonally oriented.

5.10.2 Overall subband decoding process

This sections specifies the overall decoding process for coefficients within a subband. Data within a subband is divided into code blocks, representing rectangular blocks of coefficients. If SPATIAL_PARTITION is TRUE, then subbands may be divided into more than one code block, otherwise only a single block is used. The horizontal number of code blocks is XNUM_CODEBLOCKS. The vertical number of code blocks is YNUM_CODEBLOCKS.

The value of XNUM_CODEBLOCKS and YNUM_CODEBLOCKS is derived as per Section 5.10.3.

Before any coefficient data is decoded, a count of coefficients is initialised by

COEFF_COUNT=0

A reset interval COUNT_RESET is set by

COUNT_RESET=max(25, min((SUBBAND_WIDTH*SUBBAND_HEIGHT)//32, 800))

Code blocks are decoded in raster order. In pseudocode,

```
for (v=0 ; v<YNUM_CODEBLOCKS ; ++v)
{
   for (u=0 ; u<XNUM_CODEBLOCKS ; ++u)
      decode_code_block(u,v)
}
```

The state of the arithmetic decoder, including all contexts is maintained between codeblocks.

5.10.3 Number and dimension of code blocks

This section specifies the number and dimensions of code blocks used in decoding subband data.

Inputs to this process are: PARTITION_INDEX, IS_INTRA, SUBBAND_NUM, MAX_XBLOCKS, MAX_YBLOCKS

Outputs to this process are: XNUM_CODEBLOCKS, YNUM_CODEBLOCKS

If SPATIAL_PARTITION is FALSE, then

XNUM_CODEBLOCKS=1
YNUM_CODEBLOCKS=1

If SPATIAL_PARTITION_FLAG is TRUE then XNUM_CODEBLOCKS and YNUM_CODEBLOCKS are derived from the values of PARTITION_INDEX and SUBBAND_NUM as follows.

Default

If PARTITION_INDEX is 0 then the default spatial partition is used. The minimum dimension of a code block is set by

MIN_BLOCK_DIM=4

The partition is derived from the subband number and whether the frame is intra or not. Initial values of XNUM_CODEBLOCKS and YNUM_CODEBLOCKS are derived from Table 30.

		IS_INTRA	
		TRUE	FALSE
SUB BAN	<NUM_SUBBANDS-6	XNUM_CODEBLOCKS=4 YNUM_CODEBLOCKS=3	XNUM_CODEBLOCKS=12 YNUM_CODEBLOCKS=8

	≥NUM_SUBBANDS-6 <NUM_SUBBANDS-3	XNUM_CODEBLOCKS=1 YNUM_CODEBLOCKS=1	XNUM_CODEBLOCKS=8 YNUM_CODEBLOCKS=6
	≥NUM_SUBBANDS-3	XNUM_CODEBLOCKS=1 YNUM_CODEBLOCKS=1	XNUM_CODEBLOCKS=1 YNUM_CODEBLOCKS=1

Table 30 Matrix of code block numbers

XNUM_CODEBLOCKS and YNUM_CODEBLOCKS are then adjusted to ensure that minimum codeblock sizes are respected:

XNUM_CODEBLOCKS=min(XNUM_CODEBLOCKS , SUBBAND_WIDTH//MIN_BLOCK_DIM)

YNUM_CODEBLOCKS=min(YNUM_CODEBLOCKS , SUBBAND_HEIGHT//MIN_BLOCK_DIM)

Custom

If PREDICTION_INDEX is 0 then the number of codeblocks is configurable on a frame basis by setting MAX_XBLOCKS and MAX_YBLOCKS. XNUM_CODEBLOCKS. The minimum dimension of a code block is set by

MIN_BLOCK_DIM=4

and

XNUM_CODEBLOCKS=min(MAX_XBLOCKS , SUBBAND_WIDTH//MIN_BLOCK_DIM)

YNUM_CODEBLOCKS=min(MAX_YBLOCKS , SUBBAND_HEIGHT//MIN_BLOCK_DIM)

5.10.4 Code block decoding process

This section specifies the operation of the decode_code_block(u,v) process for a code block in position (u,v).

Code block parameters

The code block in position (u,v) consists of coefficients in positions (i,j) such that

XSTART≤u<XSTOP
YSTART≤v<YSTOP

where

XSTART=(u*SUBBAND_WIDTH)//XNUM_CODEBLOCKS
XSTOP=((u+1)*SUBBAND_WIDTH)//XNUM_CODEBLOCKS
YSTART=(v*SUBBAND_HEIGHT)//YNUM_CODEBLOCKS
YSTOP=((v+1)*SUBBAND_HEIGHT)//YNUM_CODEBLOCKS

Skip flag

If the number of code blocks in the subband is greater than 1 i.e. XNUM_CODEBLOCKS>1 or YNUM_CODEBLOCKS>1, a skip flag is decoded by the recipe

SKIP[j][i]=binary_arithdecode()+skip_prediction(i,j) (mod 2)

skip_prediction() is a prediction for the skip value. It is defined by

$$\text{skip\_prediction}(i, j) = \begin{cases} \text{FALSE if } i = 0 \text{ and } j = 0 \\ \text{SKIP}[j][i-1] \text{ if } i > 0 \\ \text{SKIP}[j-1][0] \text{ if } i = 0 \text{ and } j > 0 \end{cases}$$

If XNUM_CODEBLOCKS=1 and YNUM_CODEBLOCKS=1 then SKIP[j][i] is set to FALSE.

If SKIP[j][i] is TRUE, then all coefficients within the code block are set to zero and code block decoding terminates.

Quantisation parameters

If SKIP[j][i] is FALSE and if MULTI_QUANT is TRUE, then a quantisation index is decoded by the recipe

QF[j][i]=su_arith_decode()+QUANT_INDEX

If there is only one code block or MULTI_QUANT is FALSE then

QF[j][i]=QUANT_INDEX

The quantisation factor QUANT_FACTOR[j][i] and offset value OFFSET[j][i] are derived as follows:

QUANT_FACTOR[j][i]= round($2^{QF[j][i]/4}$) [**NB this differs from software]**

OFFSET[j][i]=round(QUANT_FACTOR[j][i] * 0.375)

A table of QUANT_FACTOR and OFFSET values is given in Table 44 in Appendix C.

Coefficient data

Coefficients within a code block are decoded in raster order according to the specification of Section 5.10.5.

If SKIP[j][i] is FALSE then the remaining code block data is decoded by the following process

```
for (y=YSTART ; y<YSTOP ; ++y)
{
   for (x=XSTART ; x<XSTOP ; ++x)
      decode_coeff(x,y)
}
```

5.10.5 Subband coefficient decoding process

This section describes the operation of the decode_coeff(x,y) process for a coefficient in position (x,y), within a code block in position (i,j).

A value magnitude is decoded by the recipe

SUBBAND[y][x]=uu_arith_decode()

The magnitude is then scaled by:

SUBBAND[y][x] *= QF[j][i]

If SUBBAND[y][x] is not zero, an offset is added:

SUBBAND[y][x] += OFFSET[j][i]

If SUBBAND[y][x]>0, then a sign value is decoded

SIGN=binary_arith_decode()

and if SIGN is FALSE

SUBBAND[y][x] = -SUBBAND[y][x]

If the frame is an INTRA frame and the subband is the DC band, then the coefficient is modified by a prediction:

SUBBAND[y][x] += intra_dc_prediction(x,y)

The operation of intra_dc_prediction() is defined in Section 5.10.11.

The derivation of contexts for use in magnitude and sign decoding is defined in Sections 5.10.6 and 5.10.7.

After the coefficient has been decoded, then the count of coefficients is updated:

```
COUNT++
if (COUNT>RESET_COUNT)
{
   COUNT=0
   halve_all_counts()
}
```

5.10.6 Magnitude context modelling

This section specifies the procedure for choosing contexts for decoding the magnitude of a coefficient in position (x,y) in the subband.

Neighbourhood sum

A value NHOOD_SUM is defined as the sum of the absolute values of previously decoded and reconstructed coefficients:

$$\text{NHOOD\_SUM}=\begin{cases}|\text{SUBBAND}[y][x-1]|+|\text{SUBBAND}[y-1][x]|+|\text{SUBBAND}[y-1][x-1]| & \text{if } x>0\\ |\text{SUBBAND}[0][x-1]| & \text{if } x>0 \text{ and } y=0\\ |\text{SUBBAND}[y-1][0]| & \text{if } x=0 \text{ and } y>0\\ 0 & \text{if } x=0 \text{ and } y>0\end{cases}$$

A value NTOP is defined by the recipe

```
NTOP=(SCALE_FACTOR>>1)*QF[j][i]
```

Parent flag

A Boolean flag PARENT_ZERO is set to TRUE if the subband has a parent subband and the parent coefficient at position (x>>1,y>>1) in that subband in zero.

PARENT_ZERO set to FALSE if the subband has a parent subband and the parent coefficient at position (x>>1,y>>1) in that subband is not zero.

If the subband does not have a parent subband then if the subband is the DC subband (SUBBAND_NUM=NUM_SUBBANDS) then PARENT_ZERO is FALSE for all coefficients in the subband. If the subband does not have a parent subband and is not the DC subband then PARENT_ZERO is TRUE for all coefficients in the subband.

[**NB: I think this is what the software does. Needs checking]**

Context selection

Magnitude contexts are selected by bin and, PARENT_ZERO value and NHOOD_SUM value, according to Table 31.

Subband coefficient magnitude contexts					
PARENT_ZERO=TRUE			**PARENT_ZERO=FALSE**		
Bin	**NHOOD_SUM**	**Context**	**Bin**	**NHOOD_SUM**	**Context**
1	0	Z_BIN1z_CTX	1	0	NZ_BIN1z_CTX
1	>0	Z_BIN1nz_CTX	1	>0,≤NTOP	NZ_BIN1a_CTX
2	-	Z_BIN2_CTX	1	>NTOP	NZ_BIN1b_CTX
3	-	Z_BIN3_CTX	2	-	NZ_BIN2_CTX
4	-	Z_BIN4_CTX	3	-	NZ_BIN3_CTX
≥5	-	Z_BIN5+_CTX	4	-	NZ_BIN4_CTX
			≥5	-	NZ_BIN5+_CTX

Table 31 Contexts for coefficient magnitude decoding

5.10.7 Sign context modelling

This section specifies the procedure for choosing contexts for decoding the sign of a coefficient in position (x,y) in the subband.

Sign contexts are selected based on the sign of previously decoded coefficients. The choice of the coefficients used for the decision depends on the orientation of the subband.

A value PREVIOUS_VALUE is defined to be

SUBBAND[y][x-1] if x>0 and the subband is horizontally oriented

SUBBAND[y-1][x] if y>0 and the subband is vertically oriented

0 otherwise

The sign context used is specified by Table 32.

Subband coefficient sign contexts	
PREVIOUS_VALUE	**Context**

>0	SIGN_POS_CTX
<0	SIGN_NEG_CTX
0	SIGN_ZERO_CTX

Table 32 Contexts for coefficient signs

5.10.8 Skip parameter context modelling

The skip parameter statistics are modelled by a single context BLOCK_SKIP_CTX.

5.10.9 Quantisation index context modelling

The quantisation index magnitude statistics are modelled by a single context QUANT_MAG_CTX for all bins. The quantisation index sign statistics are modelled by a single context QUANT_SIGN_CTX.

5.10.10 Context initialisation

Prior to decoding the subband, contexts are initialised according to Table 33.

Context	COUNT0	COUNT1
BLOCK_SKIP_CTX	1	1
QUANT_MAG_CTX	1	1
QUANT_SIGN_CTX	1	1
SIGN_POS_CTX	1	1
SIGN_NEG_CTX	1	1
SIGN_ZERO_CTX	1	1
Z_BIN1z_CTX	1	1
Z_BIN1nz_CTX	1	1
Z_BIN2_CTX	1	1
Z_BIN3_CTX	1	1
Z_BIN4_CTX	1	1
Z_BIN5+_CTX	1	1
NZ_BIN1z_CTX	1	1
NZ_BIN1a_CTX	1	1
NZ_BIN1b_CTX	1	1
NZ_BIN2_CTX	1	1
NZ_BIN3_CTX	1	1
NZ_BIN4_CTX	1	1
NZ_BIN5+_CTX	1	1

Table 33 Subband decoding context initialisation [NB: these context initialisations are subject to change]

5.10.11 Intra DC subband prediction

This section specifies the procedure intra_dc_prediction() used for predicting coefficients in the DC band of Intra frames.

The prediction value to be returned is denoted PRED_VALUE and is derived based on previously decoded and reconstructed coefficients, by the recipe:

$$\text{PRED\_VALUE} = \begin{cases} (\text{SUBBAND}[y][x-1] + \text{SUBBAND}[y-1][x] + \\ \qquad \text{SUBBAND}[y-1][x-1]) // 3 \text{ if } x > 0 \text{ and } y > 0 \\ \text{SUBBAND}[0][x-1] \text{ if } x > 0 \text{ and } y = 0 \\ \text{SUBBAND}[y-1][0] \text{ if } x = 0 \text{ and } y > 0 \\ 0 \text{ if } x = 0 \text{ and } y > 0 \end{cases}$$

5.11 Inverse wavelet transform

This section defines the process iwt() for reconstructing component data from decoded subbands using the Inverse Wavelet Transform (IWT). iwt() is invoked in decoding Transform Data units which encapsulate component data, if and only if ZERO_COEFFS is FALSE.

The iwt() process consists of two sub-processes:

1. iwt_synthesis()

2. iwt_pad_removal()

The output of this process is a two-dimensional array DATA[][] of pixel data representing Y, U or V component data.

5.11.1 IWT synthesis operation

This section defines the process iwt_synthesis() invoked by iwt().

This is an iterative procedure operating on four subbands at each iteration stage to produce a new subband. In pseudocode the procedure is:

```
LL=SUBBAND[NUM_SUBBANDS]

for (n=0 , k=NUM_SUBBANDS-1; n<TRANSFORM_DEPTH ; ++n , k-=3)

{

   LL=vh_synthesis( LL , SUBBAND[k-2] ,  SUBBAND[k-1] , SUBBAND[k] )

}
```

The decoded component data values will comprise the subband coefficients LL[][], which are identified with the two-dimensional array DATA[][].

5.11.2 Removal of IWT pad values

This section defines the decoding process iwt_pad_removal(). This process is invoked after iwt_synthesis(). In this process values not used subsequently are discarded.

Values WIDTH and HEIGHT are defined as follows. For Intra frame components, if the component is the luma component, the values are set as

WIDTH = LUMA_WIDTH

HEIGHT = LUMA_HEIGHT

If the component is a chroma component, then

WIDTH = CHROMA_WIDTH

HEIGHT = CHROMA_HEIGHT

For Inter frame component data, the values are set as follows. For luma components,

WIDTH = MC_LUMA_WIDTH

HEIGHT = MC_LUMA_HEIGHT

If the component is a chroma component, then

WIDTH = MC_CHROMA_WIDTH

HEIGHT = MC_CHROMA_HEIGHT

All component data with horizontal index greater than or equal to WIDTH or with vertical index greater than or equal to HEIGHT is discarded (Figure ??).

[Figure ?? Discarded data values]

5.11.3 Vertical and horizontal synthesis

This section specifies the operation of the vertical and horizontal synthesis process vh_synthesis().

vh_synthesis(LL , HL , LH , HH) is repeatedly invoked by iwt_synthesis(). It operates on four subbands labelled LL, HL, LH and HH to produce a new subband SYNTH_BAND, which is returned as the result of the process. The subbands LL, HL, LH and HH shall all have identical dimensions.

First, the data from the four subbands are interleaved to form a single two-dimensional array A[][] whose vertical and horizontal dimensions are twice that of each of the original subbands, using the interleave() process specified in Section 5.11.4:

A=interleave(LL , HL, LH , HH)

Vertical synthesis is performed second. For each column of coefficients in the array A[][], the 1d_synthesis() procedure is applied.

Horizontal synthesis is performed third. For each row of coefficients in the array A[][], the 1d_synthesis procedure is applied.

The new subband SYNTH_BAND comprises the data in the processed 2D array A[][].

5.11.4 Interleaving

This section specifies the interleaving process interleave(LL , HL , LH , HH) by which the data elements belonging to four subband data arrays are combined into a single data array A. The elements of A are defined by:

A[2*j][2*i] = LL[j][i]

A[2*j][2*i+1] = HL[j][i]

A[2*j+1][2*i] = LH[j][i]

A[2*j+1][2*i+1] = HH[j][i]

for all i such that 0≤i<width(LL) and all j such that 0≤j<height(LL).

INFORMATIVE

Note that the interleaving process always creates an array with even dimensions.

5.11.5 One-dimensional synthesis

This section specifies the one-dimensional synthesis process 1d_synthesis() applied either to rows or columns of the matrix A[][].

1d_synthesis() applies filtering to a one dimensional array C[] of coefficients.

Where 1d_synthesis() is applied to a row of data, then a value C[k] shall be identified with a value A[j][k] where j is the index of the row being processed.

Where 1d_synthesis() is applied to a column of data, then a value C[k] shall be identified with a value A[k][i] where i is the index of the column being processed.

The one-dimensional synthesis process comprises the application of a number of reversible integer lifting filter operations lift1() to liftN().

The number of lifting operations and their definition depends upon the choice of wavelet filter derived from the WAVELET_FILTER_INDEX value defined for the frame data unit. The definition of lifting and of lifting operations for particular wavelet filters are given in subsequent sections.

5.11.6 Integer lifting

This section defines the general operation of a single integer lifting operation.

Each lifting process applies either to odd or to even coefficients. If it applies to even coefficients, then the even coefficients are modified using the values of odd coefficients only. If it applies to odd coefficients, then the odd coefficients are modified using the values of even coefficients only. Specifically, a single lifting filtering operation is of the form:

$$C(2n) = C(2n) + \left(\sum_{i=-N}^{M} t_i C(2(n+i)-1) \right) >> s$$

if it operates on even coefficients and of the form

$$C(2n+1) = C(2n+1) + \left(\sum_{i=-N}^{M} t_i C(2(n+i)) \right) >> s$$

if it operates on odd coefficients. The values t_i are the lifting filter tap values; the value s is the lifting filter scale factor.

The lifting operation is applied for all the even, or all the odd, coefficients in the row or column array C. A decoder may perform the individual coefficient filtering operations in any order, as this does not affect the result.

The filtering process uses reflection extension. Where filtering requires values which fall out of the range of values of the array C, then the value selected is determined as follows:

- C[2k] is identified with C[2length(C)-2-2k] if 2k≥length(C)

- C[2k] is identified with C[-2k] if 2k<0

- C[2k+1] is identified with C[2length(C)-1-2k] if 2k+1>length(C)

- C[2k+1] is identified with C[-(2k+1)] if 2k+1<0

[**NB. Current software does not consistently use reflection – may be edge extension in some cases. Edge extension may be better and easier to specify!]**

INFORMATIVE

This specification defines the lifting process on the basis of lifting procedures applied to an entire row or column consecutively. It is possible to implement lifting filtering operations so that a filtering operation associated with one lifting filter is followed by a filtering operation associated with another lifting filter. I.e. the order of iteration is changed. In this case, the order in which filtering is applied to coefficients does affect the outcome of the process as even lifting operations may be followed by odd ones, and care must be taken that values are not modified in the wrong order. Nevertheless such an implementation may be more efficient, and complies with this specification if it produces identical results.

5.11.7 Daubechies (9,7) lifting filters

This section specifies the lifting filters that must be used if WAVELET_FILTER_INDEX=0.

There are four lifting filters lift1, lift2, lift3, and lift4. Their filtering operations are given in Table 34.

Filter	Parity	Filter equation
lift1	Even	C[2n] -= (1817 * (C[2n-1] + C[2n+1]))>>12
lift2	Odd	C[2n+1] -= (3616 * (C[2n] + C[2n+2]))>>12
lift3	Even	C[2n] += (217 * (C[2n-1] + C[2n+1]))>>12
lift4	Odd	C[2n+1] += (6497 * (C[2n] + C[2n+2]))>>12

Table 34 Lifting filters for WAVELET_FILTER_INDEX=0

5.11.8 Approximate Daubechies (9,7) lifting filters

This section specifies the lifting filters that must be used if WAVELET_FILTER_INDEX=1.

There are two lifting filters lift1 and lift2. Their filtering operations are given in Table 35.

Filter	Parity	Filter equation
lift1	Even	C[2n] -= (C[2n-1] + C[2n+1])>>2

lift2	Odd	C[2n+1] += (9*(C[2n] + C[2n+2]) –(C[2n-2]+C[2n+4]))>>4

Table 35 Lifting filters for WAVELET_FILTER_INDEX=1

5.11.9 (5,3) lifting filters

This section specifies the lifting filters that must be used if WAVELET_FILTER_INDEX=2.

There are two lifting filters lift1 and lift2. Their filtering operations are given in Table 36.

Filter	Parity	Filter equation
lift1	Even	C[2n] -= (C[2n-1] + C[2n+1])>>2
lift2	Odd	C[2n+1] += (C[2n] + C[2n+2])>>1

Table 36 Lifting filters for WAVELET_FILTER_INDEX=2

5.11.10 (13,5) lifting filters

This section specifies the lifting filters that must be used if WAVELET_FILTER_INDEX=3.

There are two lifting filters lift1 and lift2. Their filtering operations are given in Table 37.

Filter	Parity	Filter equation
lift1	Even	C[2n] -= (9*(C[2n-1] + C[2n+1]) –(C[2n-3]+C[2n+3]))>>4
lift2	Odd	C[2n+1] += (9*(C[2n] + C[2n+2]) –(C[2n-2]+C[2n+4]))>>5

Table 37 Lifting filters for WAVELET_FILTER_INDEX=3

5.12 Motion compensation

This section defines the operation of the motion_compensate() decoding process. This process is invoked for each Inter frame component and is applied subsequently to the IWT.

Motion compensation employs up to 2 reference frames resident in the REFERENCE_FRAME_BUFFER[]. Motion compensation is applied to luma and (if the video data is not monochrome) chroma components separately.

The inputs to this process are: a component data array DATA[][] of values extracted from the IWT, and the motion vector data decoded as per Section 5.5.

The outputs of this process are: a modified component data array DATA[][].

5.12.1 Overall process

For each component, motion compensation is effected by forming a prediction for each pixel within the component and adding it to the pixel value. Specifically, for luma components, set

WIDTH=MC_LUMA_WIDTH

HEIGHT=MC_LUMA_HEIGHT

and for chroma components

set

WIDTH=MC_CHROMA_WIDTH

HEIGHT=MC_CHROMA_HEIGHT

The pseudocode motion compensation process is

```
for (j=0; j<HEIGHT; ++j)
{
   for (i=0; i<WIDTH; ++i)
   {
      DATA[j][i] += predict( i , j )
    }
}
```

The use of OBMC implies that each pixel may fall into between 1 and 4 Prediction Units (Figure 21), and the prediction value is a weighted combination of predictions from each prediction unit of which the pixel is a member.

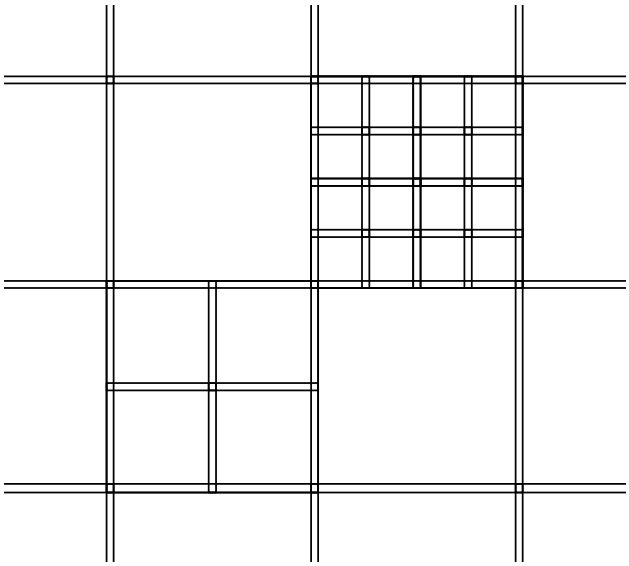


Figure 21 Overlapping blocks/prediction units]

5.12.2 Pixel prediction

This section specifies the operation of the predict(i,j) function for predicting a pixel at position (i,j) within a component data array.

predict() comprises a weighted sum

$$\text{predict}(i,\ j) = \left(\sum_k w_k(i,\ j)\text{pu\_predict}_k(i,\ j) + 1024 \right) >> 11$$

where the sum is over all prediction units defined by the motion vector data, and the $w_k(i, j)$ is a non-negative integral weighting function defined for each prediction unit. For any pixel coordinates (i,j) at most 4 of the values $w_k(i, j)$ is non-zero. The location and dimensions of prediction units are specified in Section 5.12.3.

Sections 5.12.8 and 5.12.9 define the operation of pu_predict() for different prediction modes. Section 5.12.10 derives the value of the weighting factors $w_k(i, j)$.

The motion compensation process, although defined in this specification in terms of the mathematical operation performed on pixels, is designed to be efficiently implemented by means of block operations. Further informative details are given in Section 5.12.11.

5.12.3 Prediction unit coordinates and dimensions

This section specifies the location and dimensions of a prediction unit.

A prediction unit corresponds to a rectangular area of pixels in the current frame. A prediction unit is determined by the coordinates (mx,my) of the macroblock to which it belongs, the macroblock splitting level MB_SPLIT and the coordinates (px,py) of the prediction unit within the MB, which must be such that

$0 \leq px < (2 << MB\_SPLIT)$

and

$0 \leq py < (2 << MB\_SPLIT)$

A prediction unit within a macroblock is a block if and only if MB_SPLIT=2; it is a sub-MB if and only if MB_SPLIT=1; and it is a macroblock if and only if MB_SPLIT=0. Prediction unit dimensions for each type of prediction unit differ. Prediction unit dimensions also differ for prediction units in luma components and prediction unit in chroma dimensions.

Define XBLEN=XBLEN_LUMA if the component is the luma component, XBLEN=XBLEN_CHROMA otherwise.

Define YBLEN=YBLEN_LUMA if the component is the luma component, YBLEN=YBLEN_CHROMA otherwise.

Define XBSEP=XBSEP_LUMA if the component is the luma component, XBLEN=XBSEP_CHROMA otherwise.

Define YBSEP=YBSEP_LUMA if the component is the luma component, YBSEP=YBSEP_CHROMA otherwise.

If the prediction unit is a block, the prediction unit dimensions are:

PU_WIDTH=XBLEN

PU_HEIGHT=YBLEN

If the prediction unit is a sub-MB the prediction unit dimensions are:

PU_WIDTH=XBLEN+XBSEP

PU_HEIGHT=YBLEN+YBSEP

If the prediction unit is a macroblock the prediction unit dimensions are:

PU_WIDTH=XBLEN+3*XBSEP

PU_HEIGHT=YBLEN+3*YBSEP

Values OFFSETX and OFFSETY are defined by:

OFFSETX=(XBLEN-XBSEP)>>1

OFFSETY=(YBLEN-YBSEP)>>1

The location of the top-left corner pixel of a prediction unit with macroblock coordinates (mx,my) and coordinates (px,py) within the macroblock are derived as follows. For a block (MB_SPLIT=2), they are

PU_TL_X=(4*mx+px)*XBSEP-OFFSETX

PU_TL_Y=(4*my+py)*YBSEP-OFFSETY

If the prediction unit is a sub-MB the coordinates are

PU_TL_X=(4*mx+2*px)*XBSEP-OFFSETX

PU_TL_Y=(4*my+2*py)*YBSEP-OFFSETY

If the prediction unit is a macroblock the coordinates are

PU_TL_X=4*mx)*XBSEP-OFFSETX

PU_TL_Y=4*my*YBSEP-OFFSETY

If the prediction unit falls at the edges of the picture , the prediction unit coefficients will extend OFFSETX pixels horizontally and OFFSETY pixels vertically beyond the picture boundaries. These pixels are not motion compensated and predictions for these values are discarded.

5.12.4 Motion vector scaling for chroma components

This section specifies how motion vectors shall be scaled for compensating chroma components.

For each prediction unit, if the prediction unit prediction mode is not INTRA, then one or more motion vectors have been decoded as per Section 5.6.9. These motion vectors represent displacements from the luma component of the current frame into the luma component of reference frames. When motion compensating chroma components, the motion vector must be scaled in order to take into account different chroma sampling formats.

MV_CHROMA_x = MV_x//CHROMA_H_SCALE

MV_CHROMA_y = MV_y//CHROMA_V_SCALE

5.12.5 Subpixel vectors

The decoded frame motion vectors represent displacements scaled by the motion vector precision MV_PRECISION. The integer part of a motion vector MV, integer_mv(MV), is defined to be the vector with horizontal component

$MV\_x//2^{MV\_PRECISION}$

and vertical component

$MV\_y//2^{MV\_PRECISION}$

The subpixel remainder, subpel(MV) is defined to be the vector with horizontal component

$MV\_x-(MV\_x//2^{MV\_PRECISION})* 2^{MV\_PRECISION}$

and vertical component

$MV\_y-(MV\_y//2^{MV\_PRECISION})* 2^{MV\_PRECISION}$

5.12.6 Subpixel interpolation method

This section defines the method by which values corresponding to sub-pixel offsets from reference picture pixels are derived.

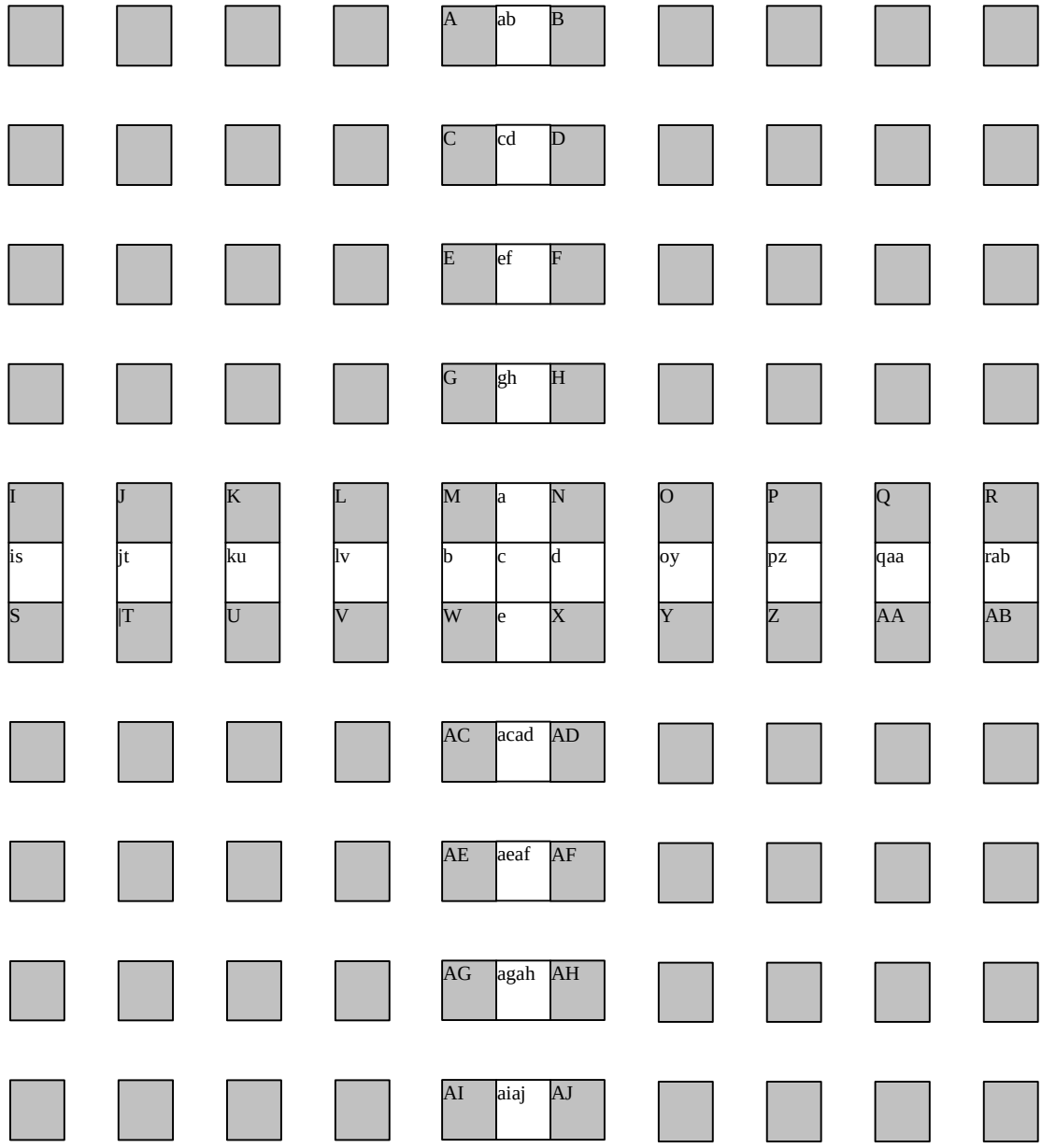


Figure 22 Interpolation of half-pixel values

Half-pixel interpolation

Half-pixel interpolation is performed if MV_PRECISION>0. Half-pixel values are interpolated by applying a 10-tap filter vertically then horizontally, as follows. The filter taps are defined by Table 38:

Tap	Value
t0	167
t1	-56
t2	25
t3	-11
t4	3

Table 38 Interpolation filter coefficients

Values a,b,d and e are calculated first. Values a and e are calculated by applying the filter to the nearest integer pixels in the horizontal direction:

a=(t0*(M+N)+t1*(L+O)+t2*(K+P)+t3*(J+Q)+t4*(I+R)+128)>>8

e=(t0*(W+X)+t1*(V+Y)+t2*(U+Z)+t3*(T+AA)+t4*(S+AB) +128)>>8

Values b and d are calculated by applying the filter to the nearest integer pixels in the vertical direction:

b=(t0*(M+W)+t1*(G+AC)+t2*(E+AE)+t3*(C+AG)+t4*(A+AI)+128)>>8

d= (t0*(N+X)+t1*(H+AD)+t2*(F+AF)+t3*(D+AH)+t4*(B+AJ) +128)>>8

Values jt, ku, is, lv, oy, pz, qaa, rab are derived by applying the same operation as applied to derive sample b to the corresponding columns of values in which these samples lie.

Value c is derived by applying the horizontal filter to the derived half-pixel values jt, is, ku, lv, oy,b,d,,pz, qaa, rab:

c= (t0*(b+d)+t1*(lv+oy)+t2*(ku+pz)+t3*(jt+qaa)+t4*(is+rab) +128)>>8

Quarter and eighth pixel interpolation

Subpixel resolutions higher than half-pixel are derived by linear interpolation from half-pixel values. Quarter pixel interpolation is performed if MV_PRECISION is 2. First, half-pixel interpolated values A, B, C, D are derived as described above. Half-pixel values are marked as A, B, C and D in Figure 23 Quarter- and one-eighth pixel interpolation. The shaded pixels are original pixel values or interpolated half-pixel values..

Secondly, quarter-pixel values in positions b, h, j are derived by:

b=(A+B+1)>>1

h=(A+C+1)>>1

j=(A+B+C+D+2)>>2

Eighth-pixel interpolation is performed if MV_precision is 3. First, half-pixel interpolated values A, B, C, D are derived as described above. Values b,h,j are defined as in the previous derivation. The remaining values a,c,d,e,f,g,i,k,l,m,o are derived by:

a=(12*A+4*B+8)>>4

c=(4*A+12*B+8)>>4

d=(12*A+4*C+8)>>4

e=(9*A+3*B+3*C+D+8)>>4

f=(6*A+6*B+2*C+2*D+8)>>4

g=(3*A+9*B+C+3*D+8)>>4

i=(6*A+2*B+6*C+2*D+8)>>4

k=(2*A+6*B+2*C+6*D+8)>>4

l=(4*A+12*C+8)>>4

m=(3*A+B+9*C+3*D+8)>>4

n=(2*A+2*B+6*C+6*D+8)>>4

o=(A+3*B+3*C+9*D+8)>>4

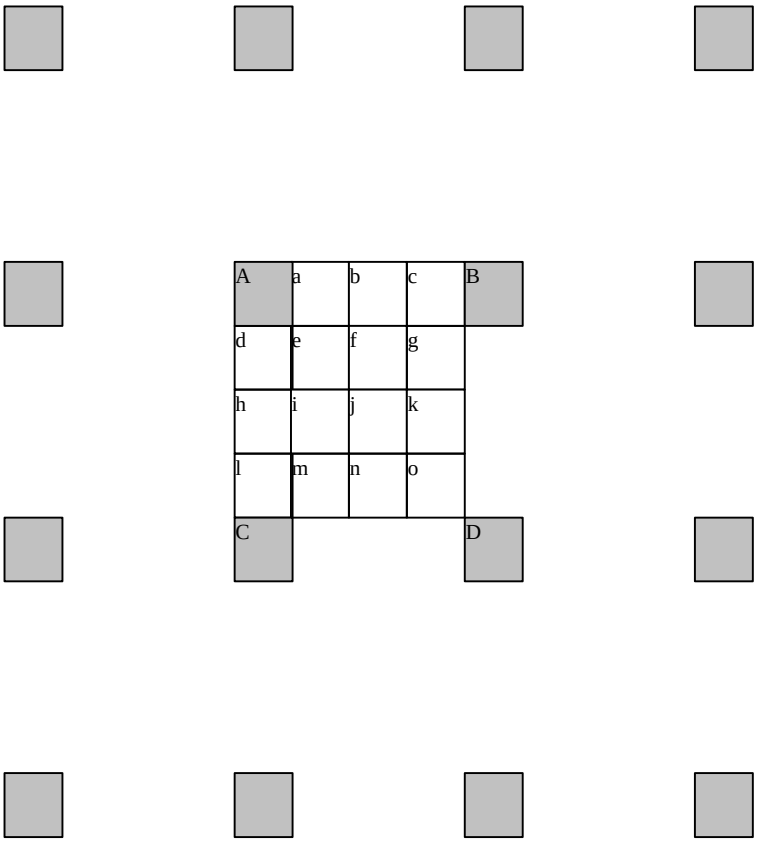


Figure 23 Quarter- and one-eighth pixel interpolation. The shaded pixels are original pixel values or interpolated half-pixel values.

5.12.7 Formation of a prediction from a motion vector

This section defines how a prediction value pred is derived from a pixel location (i,j) in the current frame component and a motion vector MV of resolution MV_PRECISION for a reference frame component.

Let IMV=integer_mv(MV) be the integer part of the motion vector, and SMV=subpel(MV) be the fractional part. The location of the reference pixel is determined in subpixel fractional offsets as

$(rx,ry)=(2^{MV\_PRECISION}*(IMV\_x+i)+SMV\_x, 2^{MV\_PRECISION}*(IMV\_y+j)+SMV\_y)$

That is, if MV_PRECISION=2 this gives the number of quarter-pixels from the top-left hand corner of the upconverted reference frame.

The prediction value pred is derived as the value of the reference frame at the location

$(Clip(rx,0, 2^{MV\_PRECISION}*(WIDTH-1)), Clip(ry,0, 2^{MV\_PRECISION}*(HEIGHT-1))$

where WIDTH and HEIGHT are as defined in Section 5.11.2.

5.12.8 Formation of unweighted predictions for non-intra prediction units

This section defines the procedure by which a prediction is derived for a pixel within a prediction unit for which the prediction mode PRED_MODE is not INTRA, resulting in a prediction value pu_predict(i,j) where (i,j) are the coordinates of the pixel such that

PU_TL_X≤i<PU_TL_X+PU_WIDTH

PU_TL_Y≤j<PU_TL_Y+PU_HEIGHT

If PRED_MODE&1 is non-zero, then there is a reference 1 motion vector MV1=(MV1_x,MV1_y). A prediction PRED1 is derived as per Section 5.12.7 from the corresponding reference frame component.

If PRED_MODE&1=0, then PRED1 shall be 0.

If PRED_MODE&2 is non_zero, then there is a reference 2 motion vector MV2=(MV2_x,MV2_y). A prediction PRED2 is derived as per Section 5.12.7 from the corresponding reference frame component.

If PRED_MODE&2=0 then PRED2 shall be 0.

If NON_DEFAULT_WEIGHTS is TRUE the prediction value returned from pu_predict(i,j) is

(REF1_WEIGHT*PRED1+REF2_WEIGHT*PRED2 +4)>>3

If NON_DEFAULT_WEIGHTS is FALSE the prediction value returned from pu_predict(i,j) is

PRED1+PRED2

5.12.9 Formation of unweighted predictions for intra prediction units

This section defines the procedure by which a prediction is derived for each pixel within a prediction unit for which the prediction mode is INTRA, resulting in a prediction value pu_predict(i,j) where (i,j) are the coordinates of the pixel.

The decoded motion vector data has decoded a value PU_DC. This is the 8-bit-accurate DC value of the prediction unit. The reconstructed DC is shifted by the number of accuracy bits ACC_BITS.

DC=PU_DC<<ACC_BITS

If ACC_BITS>0 an offset is also added to improve reconstruction,

DC+=1<<(ACC_BITS-1)

DC is then returned as the output of pu_predict(i,j)

5.12.10 Calculation of weighting values

This section defines how the values of the weighting function $w_k(i,j)$ are derived.

The inputs to this function are:

- the values WIDTH and HEIGHT of the video component after the IWT;

- the coordinates (i,j) of a pixel in the video component

- a prediction unit P_k to which (i,j) belongs

The outputs of this process is a weighting value which is returned.

Given a prediction unit P_k with coordinates $PU\_TL\_X_k$, $PU\_TL\_X_k$ and dimensions PU_WIDTH_k, PU_HEIGHT_k then the normalised coordinates of the pixel are defined to be:

ni=i-$PU\_TL\_X_k$

nj=j-$PU\_TL\_Y_k$

Set L=MB_SPLIT_k denotes the macroblock splitting level of the macroblock to which P_k belongs. The weighting matrices $OBMC\_WT_n[][]$ are as specified in Appendix B. From these, the weighting value $w_k(i,j)$ is derived by the pseudocode:

$w_k(i,j) = OBMC\_WT_L[nj][ni]$

if (i<XBLEN/2 || i≥WIDTH-(XBLEN/2))

  $w_k(i,j) = 1024$

if (j<YBLEN/2 || j≥HEIGHT-(YBLEN/2))

  $w_k(i,j) = 1024$

if (!NON_DEFAULT_WEIGHTS && PRED_MODE!=REF1AND2)

  $w_k(i,j) *= 2$

These conditions ensure that weighting occurs correctly at the edges of the picture, as no overlapping block predictions are available outside the edges of the picture.

5.12.11 Implementation (INFORMATIVE)

The motion compensation process is defined in this specification in terms of the prediction determined for each pixel. Typically, an implementation will motion compensate all pixels within a prediction unit or a block together, as they share motion parameters and hence will have a contiguous prediction data set in the reference frames. Contiguity of the reference data is important for efficient use of the

cache in software implementations and in reducing overall memory bandwidth in both hardware and software implementations.

In this case, the prediction process consists of locating an array (PU_WIDTH by PU_HEIGHT) of prediction data as a sub-array of the reference data by using the motion vector as an offset into the reference frame component. Where MV_PRECISION>0 this array will have a stride (1<<MV_PRECISION). The weighting matrix is then applied directly to the prediction block as a whole.

In order to calculate the overlaps correctly, an intermediate prediction buffer can be built up by the recipe:

for ($j=PU\_TL\_Y_k$; $i< PU\_TL\_Y_k+PU\_HEIGHT_k$;++j)

{

  for ($i=PU\_TL\_X_k$; $i< PU\_TL\_X_k+PU\_WIDTH_k$;++i)

    $predict\_buffer(i, j) += w_k(i, j)pu\_predict_k(i, j)$

}

and then iterating over all the prediction units P_k.

The prediction for a pixel (i,j) is then

(predict_buffer(i,j)+1024)>>11

This is mathematically equivalent to the formulation in this specification because the overlapping block weights are constrained to sum to 2048.

It is not necessary for the prediction buffer to consist of the whole video component – it could be a single row of macroblocks or even blocks, with values being overwritten successively.

The use of a half-weight matrix for REF1AND2 prediction units when NON_DEFAULT_WEIGHTS is FALSE implies that Reference 1 data and Reference 2 data may be added to a prediction buffer in any order. That is, all Reference 1 data may be added to the prediction buffer first, followed by all Reference 2 data. This allows for more flexible memory management as there is no need to access Reference 1 and Reference 2 simultaneously.

5.13 Clipping of frame data

This section specifies the operation of the clip_frame() decoding process. This process is invoked in decoding each frame data unit subsequent to motion compensation (in the case of Inter frames) or the IWT (in the case of Intra frames.

Inputs to this process are: reconstructed frame data arrays YDATA[][], C1DATA[][], C2DATA[][]; LUMA_OFFSET, LUMA_EXCURSION, CHROMA_OFFSET, CHROMA_EXCURSION

Outputs to this process are: clipped frame data arrays YDATA[][], C1DATA[][], C2DATA[][]

5.13.1 Maximum and minimum data values

Values YMAX, YMIN, CMAX, and CMIN are defined by

YMIN=LUMA_OFFSET

YMAX=LUMA_OFFSET+LUMA_EXCURSION

CMIN=CHROMA_OFFSET-CHROMA_EXCURSION//2

CMAX=CHROMA_OFFSET+CHROMA_EXCURSION//2

5.13.2 Clipping operation

Each luma sample YDATA[j][i] is clipped by the operation:

YDATA[j][i]=clip(YDATA[j][i] , YMIN, YMAX)

Each of the chroma samples C1DATA[j][i], C2DATA[j][i] is clipped by the operation:

C1DATA[j][i]=clip(C2DATA[j][i] , CMIN, CMAX)

C2DATA[j][i]=clip(C2DATA[j][i] , CMIN, CMAX)

[**NB current software assumes 8 bit full range clipping only]**

6 Arithmetic decoding

This section describes the semantics of arithmetic decoding, which is used to extract coefficient and motion vector data from the bitstream.

6.1 Arithmetic decoding engine

The arithmetic decoding engine consists of an internal state, and functions which extract data values according to the state and also change the state.

The internal state consists of

a) three unsigned 16 bit unsigned integer values CODE, LOW and HIGH

b) an integer value UNDERFLOW

c) a set of contexts

Bits will be shifted into CODE during the decoding process.

6.2 Data input

The arithmetic decoding process accesses data in a contiguous block of length LENGTH bytes in the bitstream (LENGTH=MV_DATA_LENGTH or SUBBAND_DATA_LENGTH depending on whether motion data or subband coefficient data is being decoded).

A function input_bit() is defined such that: if less than LENGTH full bytes of data have been input into the arithmetic decoding process, input_bit() shall return the next bit in bitstream order according to the conventions of Section 2.8; otherwise 0 shall be returned.

INFORMATIVE

When decoding some implementations may require more than LENGTH bytes of data input in order to stabilise and decode the last values.

6.3 Initialisation

At the beginning of the decoding of any data unit, the arithmetic decoding state is initialised as follows:

LOW=0x0

HIGH=0xffff

UNDERFLOW=0

CODE=0

for (b=15 ; b≥0 ; --b)

  CODE |= (input_bit()<<b)

init_contexts()

6.4 Contexts

The decoder shall maintain the statistics relating to the set of contexts that may be used in decoding a data element. How these shall be initialised is set out in Sections 5.6.10 and 5.10.10. A context is modelled as consisting of the following unsigned integer values, which constitute its state:

COUNT0

COUNT1

COUNT0 represents a count of the number of zeroes that have occurred in the context, COUNT1 a count of the number of ones that have occurred in the context.

From these values WEIGHT, SCALED_COUNT0 and SCALED_COUNT1 are derived. WEIGHT shall always be equal to COUNT0+COUNT1. WEIGHT shall always be less than 1024

SCALED_COUNT0 is a count of zeroes scaled to a total count of 1024. SCALED_COUNT1 is a count of ones, likewise scaled to a total count of 1024. The derivation of the scaled counts is specified in Section 6.4.1.

6.4.1 Scaling counts

This section specifies the operation of the scale_counts(CONTEXT) process for a context CONTEXT.

Values SCALED_COUNT0 and SCALED_COUNT1 are derived from the values COUNT0, COUNT1 and WEIGHT associated with CONTEXT as follows:

SCALED_COUNT0 = (COUNT0 * DIVISION_FACTOR[WEIGHT-1])>>21

SCALED_COUNT1 = 1024-SCALED_COUNT0

DIVISION_FACTOR[] is an array (Look-Up Table) ranging over indices 0 to 1023 such that

DIVISION_FACTOR[k]=2^{31}//WEIGHT.

INFORMATIVE

DIVISION_FACTOR[] is defined so that SCALED_COUNT0 approximates

COUNT0*1024/WEIGHT

whilst avoiding using a division.

6.4.2 Initialisation

This section specifies the operation of the init_contexts() arithmetic decoding process.

Context initialisation consists of

a) setting COUNT0 and COUNT1 to the appropriate initial values for each context being used

b) setting WEIGHT=COUNT0+COUNT1

b) invoking scale_counts() for each context in order to determine SCALED_COUNT0 and SCALED_COUNT1

Sections 5.6.10 and 5.10.10 specify the contexts used and initial values for COUNT0 and COUNT1 for decoder operations.

6.4.3 Halving counts

This section specifies the operation of the process halve_ counts(CONTEXT) for a context CONTEXT, and the process halve_all_counts().

The values COUNT0, COUNT1 and WEIGHT associated with are modified by:

COUNT0 >>= 1

COUNT0++

COUNT1 >>= 1

COUNT1++

WEIGHT=COUNT0+COUNT1

scale_counts(CONTEXT)

The process halve_all_counts() invokes halve_counts() for each context in use.

6.4.4 Updating counts

This section specifies the operation of update(CONTEXT , VALUE) for a context CONTEXT and a Boolean or binary value VALUE. This process is invoked in the binary_arith_decode() process.

The values COUNT0, COUNT1 and WEIGHT associated to the context CONTEXT are modified by the recipe:

```
if (VALUE )
   COUNT1++
else
   COUNT0++
```

```
WEIGHT=COUNT0+COUNT1
if (WEIGHT≥1024)
   halve_counts( CONTEXT )
```

6.5 Decoder functions

The arithmetic decoding engine is a multi-context, adaptive binary arithmetic decoder, performing binary renormalisation and producing binary outputs. For each bit decoded, the semantics of the relevant decoder function determine a statistical context (probability model) to be used for the internal rescaling functions. Assuming this context, the raw arithmetic decoding function is written

binary_arith_decode()

Its operation is defined in Section 6.5.3.

Signed and unsigned integer outputs can be derived by the use of binarisation schemes which turn a symbol into a string of bits. There are two basic binarisations employed for arithmetic coding: unary and truncated unary binarisation. These are defined in Appendix A, giving rise to four decoder functions

Unsigned unary arithmetic decoding: uu_arith_decode()

Signed unary arithmetic decoding: su_arith_decode()

Unsigned truncated unary arithmetic decoding: ut_arith_decode()

In addition a decoding process may invoke halve_all_counts() to reset context statistics.

INFORMATIVE

There is no signed truncated unary arithmetic decoding function in Dirac, since truncated unary values have been derived from modulo arithmetic and have no sign.

6.5.1 Unary arithmetic decoding

This section specifies the operation of arithmetic decoding functions uu_arith_decode() and su_arith_decode() in terms of binary arithmetic decoding operations.

uu_arith_decode()

Pseudo-code for unsigned unary arithmetic decoding uu_arith_decode() is as follows:

```
VALUE=0
while ( !binary_arith_decode( choose_context() ) )
   VALUE++
```

choose_context() is a function that produces a context with which the binary bit shall be decoded. The value it returns can depend on any values known to the decoder at the time it is called, especially including the binarisation bin (the bin number is equal to VALUE+1 according to the conventions of AppendixA).

su_arith_decode

Signed unary arithmetic decoding su_arith_decode() has the following pseudocode representation:

```
VALUE= uu_arith_decode()
if ( VALUE!=0 )
{
   if ( !binary_arith_decode( choose_context() ) )
   VALUE=-VALUE
}
```

6.5.2 Truncated unary arithmetic decoding

This section specifies the operation of the unsigned truncated unary arithmetic decoding function ut_arith_decode() in terms of binary arithmetic coding operations.

Pseudo-code for ut_arith_decode() is as follows, for values known to be in the range :

```
VALUE=0
while ( !binary_arith_decode( choose_context() ) && VALUE<N )
   VALUE++
```

choose_context() is a function that produces a context with which the binary bit shall be decoded. The value it returns can depend on any values known to the decoder at the time it is called, especially including the binarisation bin (the bin number is equal to VALUE+1 according to the conventions of Appendix A).

6.5.3 Binary arithmetic decoding

This section specifies the operation of the binary arithmetic decoding function binary_arith_decode(CONTEXT).

This function is fundamental to all the arithmetic decoding functions. It consists of three stages:

1. Determine the binary output value VALUE as per Section 6.5.4
2. Modify the decoder state as per Section 6.5.5
3. Update the context statistics by invoking update(CONTEXT , VALUE) as per Section 6.4.4

6.5.4 Binary arithmetic decoding output

This section specifies the derivation of the output value which shall be returned by the binary_arith_decode(CONTEXT) process.

Inputs to this process are: LOW, HIGH, CODE

Outputs to this process are: VALUE, RANGE

Define a value COUNT by

```
COUNT=( (CODE-LOW+1)<<10 )-1
```

Set

```
RANGE=HIGH-LOW+1
if (COUNT<RANGE*COUNT0)
   VALUE=0
else
   VALUE=1
```

INFORMATIVE

This process approximately maps the interval [LOW, HIGH] to the interval [0, 1024] and determines if the value represented by CODE is less than COUNT0. If so, zero has been coded.

6.5.5 Binary arithmetic decoding state update

This section specifies how the arithmetic decoder state shall be updated. This process shall be invoked by the binary_arith_decode(CONTEXT) process after the return value VALUE has been derived as per Section 6.5.4.

Inputs to this process are: VALUE, RANGE, LOW, HIGH, COUNT

Outputs of this process are: LOW, HIGH, COUNT

First, the values of HIGH and LOW are modified by:

```
if (VALUE==0)
   HIGH=LOW+ ( RANGE*COUNT0 )>>10 )-1
else
   LOW=LOW+ ( ( RANGE*COUNT0 )>>10 )
```

Then, LOW, HIGH and CODE are iteratively modified and data input as follows:

```
do
{
   if ( HIGH & 1<<15 ) == ( LOW & 1<<15 ) )
   {}
  else if ( (LOW & 1<<14 ) && !( HIGH & 1<<14 ) )
  {
     CODE      ^= 1<<14
     LOW  ^= 1<<14
     HIGH ^= 1<<14;
   }
   else
        break

   LOW  <<= 1
   HIGH <<= 1
```

```
    ++HIGH

    CODE <<= 1
        CODE += input_bit();

} while ( TRUE )
```

7 References

A Data encoding

A.1 Data encoding formats

Data is encoded in the Dirac bitstream in a number of ways. These split into two sorts: fixed-length and variable length. The syntax and decoding semantics for these formats are described in the succeeding sections.

These sections describe how bit sequences shall be interpreted as values of various types. For the purposes of this section, a bit sequences are numbered arrays of bits which are represented graphically with the first bit in the leftmost position:

b[0] b[1] b[2] …

This order is the order that bits are extracted from the bitstream in accordance with the bit packing convention (Section 2.8).

A.2 Fixed-length code formats

bool

Encoded as a single bit. A value of 1 shall be decoded as TRUE and 0 shall be decoded as false.

n-bit literal

An n-bit number in literal format shall be decoded by extracting n bits in order and placing the first bit in the leftmost position, the second bit in the next position and so on. The resulting value is to be interpreted according to the conventions of the type to which it is deemed to belong. For example, signed integers are represented in 2's complement format [ISO ref].

n-byte literal

An n-byte number in literal format shall be decoded by extracting n bytes in order and placing the first byte in the leftmost position, the second byte in the next position and so on. The resulting value is to be interpreted according to the conventions of the type to which it is deemed to belong.

A.3 Variable-length code formats

Seven different variable-length code formats are used. Of these three, the unary binarisation formats are not used directly for data encoding but for binarising values so that integer values may be produced from strings of bits and vice-versa. Binarisation is used in the context of arithmetic decoding.

A.3.1 Unsigned unary uu(n)

Unsigned unary data represents unsigned integer values. The value N≥0 is represented as N zeroes followed by a 1:

Value	Binarisation
0	1
1	01
…	…
N	$\underbrace{0...0}_{N}1$

Table 39 Conversion from unsigned unary to binary

The bit values in the unary representation of a value are referred to by bin numbers: the first bit lies in bin 1, the second in bin 2 and so on. A 0 in bin N therefore indicates that the value is greater than or equal to N, where a 1 indicates the value equals N-1.

A.3.2 Signed unary su(n)

Signed unary data is represents signed integer values. It consists of a uu-coded unsigned value followed by a sign symbol, if the unsigned value is non-zero. The sign symbol is 1 for positive and 0 for negative:

Value	Binarisation
0	1
1	011
-1	010
2	0011
-2	0010
…	…
N	$\underbrace{0...0}_{N}11$
N	$\underbrace{0...0}_{N}10$

Table 40 Conversion from signed unary to binary

A.3.3 Unsigned truncated unary ut(n)

If a value lies in a range 0≤x≤N, then truncated unary binarisation can be used: that is, for the last value the final 1 can be omitted, since its presence is inevitable.

Value	Binarisation
0	1
1	01
…	…

N-1	$\underbrace{0...0}_{N-1}1$
N	$\underbrace{0...0}_{N}$

Table 41 Conversion from unsigned truncated unary to binary

A.3.4 Unsigned exp-Golomb uegol(n)

Unsigned exp-Golomb data is decoded to produce unsigned integer values. The format consists of two parts. A prefix part, consisting of n zeroes followed by a one, indicates how many further bits to read. A suffix part, consisting of n bits, is then used to determine the value. The decoding procedure for extracting a value VALUE is mathematically equivalent to:

```
COUNT= 0
while( !read_bits(1) )
{
   COUNT++
}
VALUE = (1<<COUNT) -1 + read_bits( COUNT )
```

where the value returned by read_bits(COUNT) is interpeted as a binary representation of an unsigned integer with most-significant bit first. The bit sequences corresponding to some values are shown in Table 42.

Bit sequence	Decoded Value
1	0
0 1 0	1
0 1 1	2
0 0 1 0 0	3
0 0 1 0 1	4
0 0 1 1 0	5
0 0 1 1 1	6
0 0 0 1 0 0 0	7
0 0 0 1 0 0 1	8
0 0 0 1 0 1 0	9
0 0 0 1 0 1 1	10

Table 42 Example conversions from uegol-coded values to binary

A.3.5 Signed exp-Golomb segol(n)

Signed exp-Golomb data is decoded to produce signed integer values. It consists of a uegol-coded unsigned value followed by a sign symbol, if the unsigned value is non-zero. The decoding procedure is:

1. Decode an unsigned value value using the uegol(n) decoding procedure. Set as VALUE.

2. Then

```
if ( VALUE )
{
   if ( !read_bits( 1 ) )
      VALUE = -VALUE
}
```

A.3.6 Unsigned exp-exp-Golomb ue2gol(n)

As the name suggests, exp-exp-Golomb is an iterated form of exp-Golomb coding, whereby the prefix is itself coded with exp-Golomb codes. The decoding procedure for producing a value VALUE is:

```
COUNT = uegol_decode()
VALUE = (1<<COUNT) -1 + read_bits( COUNT )
```

Table 43 shows the bit sequences corresponding to some values.

Bit sequence	Decoded Value
1	0
0 1 0 0	1
0 1 0 1	2
0 1 1 0 0	3
0 1 1 0 1	4
0 1 1 1 0	5
0 1 1 1 1	6
0 0 1 0 0 0 0 0	7
0 0 1 0 0 0 0 1	8
0 0 1 0 0 0 1 0	9
0 0 1 0 0 0 1 1	10

Table 43 Bit sequence for ue2gol-encoded values

For small values, uegol is more efficient than ue2gol, but for large values such as frame numbers, ue2gol uses fewer bits.

A.3.7 Signed exp-exp-Golomb se2gol(n)

Signed exp-exp-Golomb data is decoded to produce signed integer values. It consists of a ue2gol-coded unsigned value followed by a sign symbol, if the unsigned value is non-zero. The decoding procedure is:

1. Decode an unsigned value VALUE using the ue2gol(n) decoding procedure. Set as VALUE.

2. Then

```
if ( VALUE )
{
   if ( !read_bits( 1 ) )
      VALUE = -VALUE
}
```

B Overlapped Block Motion Compensation block parameters

This section specifies the weighting matrices that shall be used for motion compensation.

If LH is the horizontal block length (XBLEN_LUMA or XBLEN_CHROMA), SH is the horizontal block separation (XBSEP_LUMA or XBSEP_CHROMA), LV is the vertical block length (XBLEN_LUMA or XBLEN_CHROMA) and SV is the vertical block separation (YBSEP_LUMA or YBSEP_CHROMA) then the weighting matrices $OBMC\_WT_k[][]$ for k=0, 1, 2 are product matrices such that:

$OBMC\_WT_k[j][i]=OBMC_{k,LH,SH}[i]*OBMC_{k,LV,SV}[j]$

The one-dimensional arrays $OBMCX_{k,L,S}[]$ are determined as follows.

Define a rolloff factor R by

$$R=\frac{L-S+1}{S}$$

Define also the raised cosine function with rolloff R, by

$$rc_R(t)=\begin{cases} 0 \text{ if } |t|>\frac{1+R}{2} \\ 1 \text{ if } |t|<\frac{1-R}{2} \\ \frac{1}{2}+\frac{1}{2}\cos\left(\frac{\pi}{R}\left(|t|-\frac{(1-R)}{2}\right)\right) \text{ otherwise} \end{cases}$$

For n between 0 and L-1, define VAL by

$$VAL[n]=\frac{n-\frac{(L-1)}{2}}{S}$$

$OBMC_{2,L,S}[n]=round(32.rc_R(VAL[n]))$

The matrices $OBMC_{1,L,S}[]$ and $OBMC_{0,L,S}[]$ are derived by glueing copies of $OBMC_{2,L,S}[]$ together.

$OBMC_{1,L,S}[]$ has length L+S, and

$$OBMC_{1,L,S}[n]=\begin{cases} OBMC_{2,L,S}[n] \text{ if } n<\frac{L}{2} \\ OBMC_{2,L,S}[n-S] \text{ if } n>L+S-1-\frac{L}{2} \\ 32 \text{ otherwise} \end{cases}$$

$OBMC_{0,L,S}[]$ has length L+3S, and

$$OBMC_{0,L,S}[n]=\begin{cases} OBMC_{2,L,S}[n] \text{ if } n<\frac{L}{2} \\ OBMC_{2,L,S}[n-S] \text{ if } n>L+S-1-\frac{L}{2} \\ 32 \text{ otherwise} \end{cases}$$

C Quantisation parameter arrays

The inverse quantisation process for reconstructing subbands defined in Section 5.10.4 requires that quantisation factors and offsets are derived from the encoded index QUANT_INDEX. Values of QUANT_FACTOR and OFFSET for QUANT_INDEX between 0 and 60 are listed in Table 44.

QUANT_INDEX	QUANT_FACTOR	OFFSET
0	1	0
1	1	0
2	1	0
3	2	1
4	2	1
5	2	1
6	3	1
7	3	1
8	4	2
9	5	2
10	6	2
11	7	3
12	8	3
13	10	4
14	11	4
15	13	5
16	16	6
17	19	7
18	23	9
19	27	10
20	32	12
21	38	14
22	45	17
23	54	20
24	64	24
25	76	29
26	91	34
27	108	41
28	128	48
29	152	57
30	181	68

31	215	81
32	256	96
33	304	114
34	362	136
35	431	162
36	512	192
37	609	228
38	724	272
39	861	323
40	1024	384
41	1218	457
42	1448	543
43	1722	646
44	2048	768
45	2435	913
46	2896	1086
47	3444	1292
48	4096	1536
49	4871	1827
50	5793	2172
51	6889	2583
52	8192	3072
53	9742	3653
54	11585	4344
55	13777	5166
56	16384	6144
57	19484	7307
58	23170	8689
59	27554	10333
60	32768	12288

Table 44 Derivation of quantisation factors and offsets from the encoded index QUANT_INDEX

D List of Decoder Variables and section first described.

A	5.11.3
A_11	5.5.2
A_12	5.5.2
A_21	5.5.2
A_22	5.5.2
ACC_BITS	4.5
ACC_BITS	5.12.9
APPROX_DAUBECHIES	5.7.1
AR_DENOMINATOR	4.5
AR_NUMERATOR	4.5
ASPECT_RATIO_INDEX	4.5
B_1	5.5.2
B_2	5.5.2
BLOCK_DATA_LENGTH	5.5.2
BLOCK_DATA_LENGTH	5.6
BLOCK_PARAMS_FLAG	5.5.2
BLOCK_PARAMS_INDEX	5.5.2
BLOCK_SKIP_CTX	5.10.8
BLUE_X	4.5
BLUE_Y	4.5
C	5.11.5
C_1	5.5.2
C_2	5.5.2
C1DATA	2.9
C2DATA	2.9
CHROMA_EXCURSION	4.5
CHROMA_FORMAT	4.4
CHROMA_FORMAT_INDEX	4.4.2
CHROMA_H_SCALE	4.4
CHROMA_HEIGHT	4.4
CHROMA_OFFSET	4.5
CHROMA_OFFSET	4.5.6
CHROMA_V_SCALE	4.4
CHROMA_WIDTH	4.4
CLEAN_AREA_FLAG	4.5
CLEAN_HEIGHT	4.5
CLEAN_TL_X	4.5
CLEAN_TL_Y	4.5
CLEAN_WIDTH	4.5
CMAX	5.13.1
CMIN	5.13.1
CODE	6.1
COEFF_COUNT	5.10.2
COLOUR_MATRIX_FLAG	4.5
COLOUR_MATRIX_INDEX	4.5
COLOUR_PRIMARIES_FLAG	4.5
COLOUR_PRIMARIES_INDEX	4.5

REF2x_BIN2_CTX	5.6.9
REF2x_BIN3_CTX	5.6.9
REF2x_BIN4_CTX	5.6.9
REF2x_BIN5+_CTX	5.6.9
REF2x_SIGN_CTX	5.6.9
REF2y_BIN1_CTX	5.6.9
REF2y_BIN2_CTX	5.6.9
REF2y_BIN3_CTX	5.6.9
REF2y_BIN4_CTX	5.6.9
REF2y_BIN5+_CTX	5.6.9
REF2y_SIGN_CTX	5.6.9
REFERENCE_FRAME_BUFFER	3.3
REFS	5.3.1
REFS_OFFSET	5.3.1
RTD_LIST_OFFSET	5.3.1
S	B
SCALE_FACTOR	5.8.1
SCALED_COUNT0	6.4
SCALED_COUNT1	6.4
SIGN	5.10.5
SIGN_NEG_CTX	5.10.7
SIGN_POS_CTX	5.10.7
SIGN_ZERO_CTX	5.10.7
SIGNAL_RANGE_FLAG	4.5
SIGNAL_RANGE_INDEX	4.5
SKIP	5.10.4
SMV	5.12.7
SPATIAL_PARTITION	5.7.1
SUBBAND	5.8
SUBBAND_HEIGHT	5.8.1
SUBBAND_LENGTH	5.9
SUBBAND_NUM	5.7.3
SUBBAND_WIDTH	5.8.1
SYNTH_BAND	5.11.3
THIRTEENFIVE	5.7.1
TOP_FIELD_FIRST	4.5
TRANSFER_CHAR_FLAG	4.5
TRANSFER_CHAR_INDEX	4.5
TRANSFORM_DEPTH	5.7.1
UNDERFLOW	6.1
VALUE	6.5.1
VIDEO_FORMAT	4.4
VIDEO_FORMAT_INDEX	4.4.1
WAVELET_FILTER	5.7.1
WAVELET_FILTER_INDEX	5.7.1
WEIGHT	6.4
WHITE_X	4.5
WHITE_Y	4.5
WIDTH	5.11.2
X_NUM_BLOCKS	5.5.2
X_NUM_MB	5.5