# Dirac Specification

Version 2.2.3-unofficial
Issued: April 29, 2012

## Abstract

This document is the specification of the Dirac video decoder and stream syntax.

Dirac is a royalty-free video compression system developed by the BBC utilising wavelet transforms and motion compensation. It is designed to be simple, flexible, yet highly effective. It can operate across a wide range of resolutions and application domains, including: internet and mobile streaming, delivery of standard-definition and high-definition television, digital television and cinema production and distribution, and low-power devices and embedded applications.

The system offers several key features:

- lossy and lossless coding using a common tool set
- intra-coded modes for professional production applications
- a special low delay mode for link adaption applications, such as the carriage of HDTV over SDTV infrastructure
- motion-compensated ('long-GOP') modes for distribution applications
- gradual quality reduction with increasing compression

## Contents

## 1 Introduction

Dirac is an open video codec developed by the BBC. It has been developed to address the growing complexity and cost of current video compression technologies, which provide greater compression efficiency at the expense of implementing a very large number of tools. Dirac is a powerful and flexible compression system, yet uses only a small number of core tools. A key element of its flexibility is its use of the wavelet multi-resolution transform for compressing pictures and motion-compensated residuals, which allows Dirac to be used across a very wide range of resolutions without enlarging the toolset.

Dirac is an Open Source software project, and reference implementations of the decoder and encoder are available at http://sourceforge.net/projects/dirac. A high-performance implementation, called Schrodinger, is also available open source at http://schrodinger.sourceforge.net.

Dirac offers the following features:

**Multi-resolution transforms** Data is encoded using the wavelet transform, and packed into the bitstream subband by subband. High compression ratios result in a gradual loss of resolution. Lower resolution output can be obtained for low complexity decoding by extracting only the lower resolution data.

**Inter and intra frame coding** Pictures can be encoded using motion compensation for low bit rate, or without reference to other pictures for editing, archive and other professional applications.

**Frame and field coding** Both frames and fields can be coded.

**Dual syntax** A low delay syntax is available for applications requiring very low, fixed, latency. This can be of the order of a few lines of input or output video. The low delay syntax is suitable for light compression for the re-use of low bandwidth infrastructure, for example carrying HDTV over SD-SDI links. The low delay syntax uses intra coding and simple Variable Length Codes for entropy codes. The core syntax provides much greater compression efficiency at the expense of a whole picture delay. The core syntax can use a highly efficient form of binary adaptive arithmetic coding, as well as motion compensation, for maximum performance.

**CBR and VBR operation** Dirac supports both constant bit rate and variable bit rate operation.When the low delay syntax is used, the bit rate will be constant for each area (Dirac slice) in a picture to ensure constant latency.

**Variable bit depths** 8, 10, 12 and 16 bit formats are supported.

**Multiple chroma sampling formats** 444, 422 and 420 video are all supported.

**Lossless and RGB coding** A common toolset is used for both lossy and lossless coding. RGB coding is supported via the YCoCg integer color transform.

**Choice of wavelet filters** A wide range of wavelet filters can be used to trade off performance against complexity. The Daubechies (9,7) filter is supported for compatibility with JPEG2000. A Fidelity filter is provided for improved resolution scalability.

**Simple stream navigation** The encoded stream contains picture numbers and forms a doubly-linked list with each picture header indicating an offset to the previous and next picture, to support field-accurate high-speed navigation with no parsing or decoding required.

## 2 Scope

This specification defines the Dirac video compression system through the stream syntax, entropy coding, coefficient unpacking and picture decoding processes. The decoder operations are defined by means of a mixture of pseudocode and mathematical operations.

This is version 2.2.3-unofficial of the Dirac specification. The document includes a full description of the Dirac stream syntax and decoder operations.

Dirac is a long-GOP video codec that uses wavelet transforms and motion compensation together with entropy coding, that can be readily implemented in hardware or software. Dirac is a superset of the proposed SMPTE VC-2 video codec standard which comprises the intra coding parts of this specification.

This version is compatible with and extends Version 1 by the addition of motion compensated coding. Version 1 corresponds exactly to the proposed SMPTE VC-2 video codec standard.

Subsequent versions of this specification may contain additional tools.

## 3 Conformance notation

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: ‘shall’, ‘should’, or ‘may’. Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any section explicitly labelled as ‘Informative’ or individual paragraphs that are also indicated in this way.

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

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

The keywords ‘may’ and ‘need not’ indicate courses of action permissible within the limits of the document.

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

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

## 4 Normative References

Normative references are external documents referenced in normative text that are indispensable to the user. Bibliographic references are references made in informative text or are those otherwise not indispensable to the user.

The following standards contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent edition of the standards indicated below.

1. ITU-R BT.601-6: Studio Encoding Parameters of Digital Television for standard 4:3 and Wide-screen 16:9 Aspect Ratios.

2. ITU-R BT.709-5: Parameter values for the HDTV standards for production and international programme exchange, 2002.

3. SMPTE 428.1: Digital Cinema Distribution Master (DCDM) Image Characteristics.

4. ITU-T H.264 (03/2005): Advanced video coding for generic audiovisual services. (Note: ISO/IEC 14496-10:2005, Information Technology  Coding of Audio-Visual Objects  Part 10, is a direct equivalent to ITU-T H.264.)

5. ITU-BT.1361: Worldwide unified colorimetry and related characteristics of future television and imaging systems.

6. SMPTE 2036-1: Ultra High Definition Television  Image Parameter Values For Program Production.

## 5 Definition of abbreviations and terms

This section defines the abbreviations and terms used in the Dirac specification.

### 5.1 Abbreviations

**4CIF:** Four times CIF

**4SIF:** Four times SIF

**CIE:** Commission internationale de l'éclairage (International Commission on Illumination)

**CIF:** Common Image Format

**DC:** Direct Current

**DCI:** Digital Cinema Initiatives

**DWT:** Discrete Wavelet Transform

**HD(TV):** High Definition (Television)

**HH:** High-High (subband)

**HL:** High-Low (subband)

**IDWT:** Inverse Discrete Wavelet Transform

**ITU:** International Telecommunications Union

**LH:** Low-High (subband)

**LL:** Low-Low (subband)

**NTSC:** National Television Systems Committee

**QCIF:** Quarter CIF

**QSIF:** Quarter SIF

**SD(TV):** Standard Definition (Television)

**SIF:** Source Input Format

**VC:** Video Codec

**VLC:** Variable Length Code

### 5.2 Terms

**AC (sub)Band:** any signal band that is not the DC sub-band.

**Arithmetic coding:** a form of entropy coding used by Dirac, which is used in addition to exp-Golomb coding.

**Chroma:** a pair of colour difference components. The term chroma is the direct equivalent to luma (see luma definition below). In this specification, the term chroma is not the same as that used in composite colour television. It is used to cover both gamma-corrected and non-gamma-corrected signals.

**Codeblock:** a rectangular array of wavelet coefficients within a component subband.

**Codec:** a truncation of the terms "coder" and "decoder".

**DC subband:** the signal band that represents data composed from the lowest frequency band of a wavelet transform (0-LL).

**Discrete Wavelet Transform (DWT):** a means of transforming an array of values into space-frequency components through the use of a filter bank.

**Entropy Coding:** a term for describing any mathematical process used to encode data in a lossless manner, intended to reduce the required bit rate.

**Exp-Golomb:** a form of variable-length code. This specification uses an interleaved variant.

**Intra DC Prediction:** the prediction of coefficients within the dc subband of intra pictures from neighbouring coefficients..

**Inverse Discrete Wavelet Transform (IDWT):** the inverse of the DWT that converts an array of space-frequency components back into an array of values.

**Inverse Quantisation:** a process whereby each sample of a sub-band has its signal range expanded by a defined value.

**Lifting:** the name given to reducing a DWT filtering operations into a number of elementary filters, each operating on half the samples. (Note: see Bibliography item ”Ripples in Mathematics”, chapter 3, for more information. )

**Low Delay:** a term used to define a Dirac mode which can be used to compress video with a delay of less than one frame duration.

**Luma:** the weighted sum of RGB components of colour video, which may or may not be gamma-corrected. (This term is used to prevent confusion with the term luminance that is created only from linear light levels as used in colour science.)

**Parse Info header:** identifies the beginning of major Dirac syntax elements (sequence start, picture, sequence end, padding and auxiliary data) with defined parse code values.

**Parsing:** a process by which numerical and text strings within binary data are recognised and used to provide syntactic meaning.

**Picture:** a single frame or field of video.

**Quantisation:** a process whereby each sample of a sub-band has its signal range compressed by a defined value.

**Quantiser:** The defined value used for the purposes of quantisation or inverse quantisation.

**Raster scan:** any 2-dimensional array of samples, whether as video samples or as wavelet transformed values, that is scanned in accordance with television systems; namely left to right, then top to bottom.

**Sequence:** the data contained in a Dirac sequence corresponds to a single video sequence with constant video parameters as defined in Section 10. A Dirac sequence is preceded by a ‘Parse Info’ header that indicates the beginning of the sequence with a unique parse code. A Dirac sequence can be extracted from a Dirac bit-stream and decoded entirely independently.

**Slice:** a component part of the low-delay syntax that provides for compression of small parts (slices) of a picture in order to reduce delay.

**State:** the set of current decoder variable values.

**Stream:** a concatenation of Dirac sequences.

**Subband:** the signal band that represents data composed from a single space-frequency band of a wavelet transform.

# 6 Conventions

## 6.1 State representation

This standard uses a state model to express parsing and decoding operations. The state of the decoder/parser shall be stored in the variable state. Individual elements of the decoder **state** (state variables) may be accessed by means of named labels, e.g. **state**[VAR_NAME] (i.e. state is a map, as defined in Section 6.3).

The decoder state shall be globally accessible within the decoder. Other variables, not declared as inputs to a process, shall be local to that process. The parsing and decoding operations are defined in terms of modifying the decoder state. The state variables need not directly correspond to elements of the stream, but may be calculated from them taking into account the decoder state as a whole. For example, a state variable value may be differentially encoded with respect to another value, with the difference, not the variable itself, encoded in the stream.

The parsing process is defined by means of pseudocode and/or mathematical formulae. The conventions for these elements are described in the following sections.

## 6.2 Number formats

Numbers without a prefix shall be interpreted as decimal numbers.

The prefix b indicates that the following value shall be interpreted as a binary natural number (non-negative integer).

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

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

**Example** The value 0x7A is equal to the decimal value 122.

## 6.3 Data types

### 6.3.1 Elementary data types

Three basic types are used in the pseudo code:

**Boolean** - A Boolean variable that has only two possible values: **True** and **False**.

**Integer** - A positive or negative whole number or zero.

**Label** - a unique immutable value used in control structures and to access maps (see below).

### 6.3.2 Compound data types

Elementary and compound data types may be aggregated into a single compound data type. There are three compound data type:

**Set** A collection of data types. A set is indicated by enclosing the elements within curly braces, for example $\{a, b, c\}$ represents a set containing the values $a, b$ and $c$. An empty set may be indicated by $\{\}$. The usual set-theoretic operations such as: $\cup$ (union), $\cap$ (intersection), $\in$ (membership) apply to sets and the other compound data types.

**Map** A set of data types whose elements are accessed by their corresponding label. For example, $p[Y], p[C1], p[C2]$ might be the values of the different video components of a pixel. The set of labels corresponding to the elements of a map $m$ can be accessed by args($m$), so that, for example, $args(p)$ returns$\{Y, C1, C2\}$.

**Array** A collection of data types accessed by a non-negative integer index. This compound data type is typically used to represent an array of variables. Elements of a 1-dimensional array $a$ are accessed by $a[n]$ for $n$ in the range 0 to length$(a) - 1$.

A compound data type may contain other compound data types. For example, a two dimensional array is an array of one dimensional arrays. Elements of a 2-dimensional array are accessed by $a[n][m]$ for $0 \leq m \leq$ (width$(a) - 1$), and $0 \leq n \leq$ (height$(a) - 1$). Compound data types may be more complex. For example, picture data, pic, may be considered to be a map of arrays, where $pic[Y]$ is a 2-dimensional array storing luma data, and $pic[C1]$ and $pic[C2]$ are two-dimensional arrays storing chroma data.

Elements may be added to a map or array by assignment using the appropriate index (label or integer). For example, $a[7] = 2$, adds element 7 to the array $a$, if a does not already contain element 7, then this element is assigned the value 2.

## 6.4 Functions and operators

This section defines the functions and operators used in the pseudocode in this specification. Functions and operators are similar but functions use the syntax, $(arg1, arg2, \ldots)$ whereas operators are simply placed before or between operands, e.g. $a + b$. The difference is purely syntactic and is to correspond with conventional mathematical notation.

### 6.4.1 Assignment

The assignment operation = applies to all variable types. After performing

$$a = b$$

the value of $a$ shall become equal to that of $b$, and the value of $b$ shall remain unchanged. For a set (or map or array) this constitutes an element-wise copy i.e.

$$a[x] = b[x]$$

for all valid values of $x$.

### 6.4.2 Boolean functions and operators

The following functions and operators are defined for one or more Boolean arguments:

**not** (not a) or returns **True** for a boolean value $a$ if and only if $a$ is **False**

**and** (a and b) returns **True** if and only if a and b are both **True**. Operator "and" may be used in pseudo-code conditions to denote the logical AND between Boolean values, for example: if (condition1 and condition2): etc.

**or** (a or b) returns True if either a or b are True, else it returns False. Operator "or" may be used in pseudo-code conditions to denote the logical OR between Boolean values, for example: if (condition1 or condition2): etc.

**majority** Given a set, $S = \{s_0, , s_{n-1}\}$ of Boolean values, majority$(S)$ returns the majority condition. That is, if the number of **True** values is greater than or equal to the number of **False** values, $majority(S)$ returns **True**, otherwise it returns **False**.

Boolean operations are to be distinguished from bitwise operations which operate on non-negative integer values, and are defined in Section 6.4.3.

### 6.4.3 Integer functions and operators

The following functions and operators are defined on integer values:

**Absolute value** $|a| = \begin{cases} a \text{ if } a \geq 0 \\ -a \text{ otherwise} \end{cases}$.

**Sign** sign($a$) is defined by

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

**Addition** The sum of $a$ and $b$ is represented by $a + b$.

**Subtraction** $a$ minus $b$ is represented by $a - b$.

**Multiplication** $a$ times $b$ is represented, for clarity, by $a * b$.

**Integer division** Integer division is defined for integer values $a, b$ with $b > 0$ where: $n = a//b$ is defined to be the largest integer $n$ such that

$$n * b \leq a$$

i.e. numbers are rounded towards -infinity. N.B. this differs from C/C++ conventions of round towards 0.

**Remainder** For integers $a, b$, with $b > 0$, the remainder $a\%b$ is equal to

$$a\%b = a - (a//b) * b$$

$a\%b$ always lies between 0 and $b - 1$.

**Exponentiation** For integers $a, b$, $b > 0$ $a^b$ is defined as $a * a * \ldots * a$ ($b$ times). $a^0$ is 1.

**Maximum** max($a, b$) returns the largest of $a$ and $b$.

**Minimum** min($a, b$) returns the smallest of values $a$ and $b$.

**Clip** clip($a, b, t$) clips the value $a$ to the range defined by $b$ (bottom) and $t$ (top):

$$\text{clip}(a, b, t) = \min(\max(a, b), t)$$

**Shift down** For integers $a, b$, with $b \geq 0$, $a \gg b$ is defined as $a//2^b$.

**Shift up** For integers $a, b$, with $b \geq 0$, $a \ll b$ is defined as $a * 2^b$.

**Integer logarithm** $m = \text{intlog}_2(n)$, for $n > 0$, $m$ is the integer such that $2^{m-1} < n \leq 2^m$.

**Mean** Given a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of integer values, the integer unbiased mean, mean($S$), is defined to be

$$(s_0 + s_1 + \ldots + s_{n-1} + (n//2))//n$$

**Median** Given a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of integer values the median, median($S$), returns the middle value. If $t_0 \leq t_1 \leq \ldots \leq t_{n-1}$ are the values $s_i$ placed in ascending order, this is

$t_{(n-1)/2}$

if $n$ is odd and

mean($\{t_{(n-2)/2}, t_{n/2}\}$) if $n$ is even. If $S = \emptyset$, median($S$) returns 0.

The following bitwise operations are defined on non-negative integer values:

**&** Logical AND is applied between the corresponding bits in the binary representation of two numbers, e.g. 13&6 is b1101&b110, which equals b100, or 4.

**|** Logical OR is applied between the corresponding bits in the binary representation of two numbers, e.g. 13|6 is b1101—b110, which equals b1111, or 15.

**$\wedge$** Logical XOR is applied between the corresponding bits in the binary representation of two numbers, e.g. $13 \wedge 6$ is b1101 $\wedge$ b110, which equals b1011, or 11.

**&=** $a\& = b$ is equivalent to $a = (a\&b)$.

**|=** $a| = b$ is equivalent to $a = (a|b)$.

**$\wedge$=** $a\wedge = b$ is equivalent to $a = (a \wedge b)$.

Bitwise-not is not defined for integers to avoid ambiguity concerning leading zeroes

The following logical operators are defined for integer and boolean arguments:

**==** Test of equality of two variables. $a == b$ is **True** if and only if the value of a equals the value of b.

**!=** Not equal to. $a! = b$ is equivalent to not $(a == b)$

The following logical operators are defined for integer arguments only:

**<** Less than

**<=** Less than or equal to

**>** Greater than

**>=** Greater than or equal to

The following combined assignment operators are defined for integer arguments: Operators $+, -, *, //, \%, \gg, \ll, \&, |, \wedge$, may be combined with the assignment operator (as for the Boolean operators &, |, and $\wedge$ above). For example $a+ = b$ is equivalent to $a = (a + b)$.

### 6.4.4 Array and map functions and operators

The following functions and operators are defined for sets, maps and arrays.

**Indexing** For an array $a$, $a[index]$ returns an element of $a$. If $a$ is a map the index shall be a label, else if $a$ is an array the index shall be an integer.

**Scalar Assignment** Where the notation $a = 0$ is used for an array of integer values, it means "set all elements of the array to zero".

**Insertion** $a[index] = b$ inserts a copy of $b$ into set $a$ if the element does not already exist.

**Tokens** for a map $a$, args($a$) returns the set of the indexing tokens.

**Length** for a one dimensional array $a$, length($a$) returns the number of elements in the array.

**Width** for a two dimensional array $a$, width($a$) returns the width the array. The width is the number of scalar elements corresponding to the right most array index.

**Height** for a two dimensional array $a$, height($a$) returns the height the array. The height is the number of one dimensional arrays in the two dimensional array and the "height" dimension corresponds to the left most array index.

### 6.4.5 Precedence and associativity of operators

To avoid any confusion over the order of operator precedence, every equation makes extensive use of the expression operators ”(” and ”)”. All operations recursively execute the innermost expression(s) first until the calculation has been completed. In cases where the expression operators do not make clear the order of precedence, the following table defines the descending order of operator precedence and the associativity of each operator. [Table tbc]

## 6.5 Pseudocode

Most of the normative specification is defined by means of pseudocode. The syntax is intended to be both precise and descriptive; the pseudocode is not intended to form the basis for the implementation of a Dirac decoder.

All processing defined by this standard is precise and the entire specification can be implemented using only the data types, functions and operators defined herein. That is, no operations on ”real” or ”floating point” numbers are required. All operations shall be implemented with sufficiently large integers so that overflow cannot occur.

The type of variables in the pseudocode is not explicitly declared. A variable assumes a type when it is assigned a value, which shall always have a defined type.

### 6.5.1 Processes and functions

Decoding and parsing operations are specified by means of processes – a series of operations acting on input data and global variable data. A process can also be a function, which means it returns a value, but it need not do so. So a process taking in variables $in1$ and $in2$ looks like:

| $foo(in1, in2)$ : | **Ref** |
|---|---|
| $op1(in1)$ | |
| $op2(in2)$ | |
| ... | |

Whilst a function process looks like:

| $bar(in1, in2)$ : | **Ref** |
|---|---|
| $op1(in1)$ | |
| $foo(in1, in2)$ | 6.5.1 |
| ... | |
| **return** $out1$ | |

The right-hand column in the pseudocode representation contains a cross-reference to the section in the specification containing the definition of other processes used at that line.

### 6.5.2 Variables

All input variables are deemed to be passed *by reference* in this specification. This means that any modification to a variable value that occurs within a process also applies to that variable within the calling process *even if it has a different name* in the calling process. One way to understand this is to envisage variable names as pointers to workspace memory.

For example, if we define $foo$ and $bar$ by

| $foo()$ : | **Ref** |
|---|---|
|     $num = 0$ | |
|     $bar(num)$ | |
|     **state**$[var\_name] = num$ | |

and

| $bar(val)$ : | **Ref** |
|---|---|
|     $val = val + 1$ | |

then at the end of $foo$, **state**$[var\_name]$ has been set to 1.

The only global variables are the state variables encapsulated in **state**. If a variable is not declared as an input to the process and is not a state variable, then it is local to the function.

If a process is particularly complex, it may be broken into a number of steps with intermediate discussion. This is signalled by appending and prepending "..." to the parts of the pseudocode specification:

| $foo()$ : | **Ref** |
|---|---|
|     $code$ | |
|     ... | |

[text]

| | |
|---|---|
| ... | |
|     $morecode$ | |
|     ... | |

[text]

| | |
|---|---|
| ... | |
|     $evenmorecode$ | |

The intervening text may define or modify variables used in the succeeding pseudocode, and must be considered as a normative part of the specification of the process. This is done as it is sometimes much more clear to split up a long and complicated process into a number of steps.

### 6.5.3 Control flow

The pseudocode comprises a series of statements, linked by functions and flow control statements such as **if**, **while**, and **for**.

The statements do not have a termination character, unlike the ; in C for example. Blocks of statements are indicated by indentation: indenting in begins a block, indenting out ends one.

Statements that expect a block (and hence a following indentation) end in a colon.

**if** The if control evaluates a boolean or boolean function, and if true, passes the flow to the block of following statement or block of statements. If the control evaluates as false, then there is an option to include one or more else if controls which offer alternative responses if some other condition is true. If none of the preceding controls evaluate to true, then there is the option to include an else control which catches remaining cases.

```
...
  **if** ($control1$):
    $block1$
  **else if** ($control2$):
    $block2$
  **else if** ($control3$):
    $block3$
  **else**:
    $block4$
```

The if and else if conditions are evaluated in the order in which they are presented. In particular, if *control*1 or *control*2 is true in the preceding example, *block*3 will not be executed even if *control*3 is true; neither will *block*4.

**for** The for control repeats a loop over an integer range of values. For example,

```
...
  **for** $i = 0$ **to** $n - 1$:
    $foo(i)$
```

calls $foo()$ with value $i$, as $i$ steps through from 0 to $n - 1$ inclusive.

**for each** The for each control loops over the elements in a list:

```
...
  **for each** $c$ **in** $Y, C1, C2$:
    $block$
```

**for such that** The for such that control loops over elements in a set which satisfy some condition:

```
...
  **for** $a \in A$ **such that** $control$:
    $block$
```

This may only be used when the order in which elements are processed is immaterial.

**while** The while control repeats a loop so long as a switch variable is true. When it is false, the loop breaks to the next statement(s) outside the block.

```
...
  **while** ($condition$):
    $block1$
  $block2$
```

## 7 Overall specification

[TBC - this will contain a summary of contents]

# 8 Video formats

This section defines the video formats supported by this specification.

A selection of widely used video formats are defined in normative Annex C. These video formats are characterized by their widespread use in television, cinema and multimedia applications.

This list is not exhaustive, however, and Dirac is a general purpose video compression system. These predefined formats are base formats that may be modified element by element to support a much larger range of possible video formats. Support is provided by the sequence parameters of the bitstream (Section 10) for signalling both the base video format and any modifications for complete characterization of the video format metadata.

## 8.1 Colour model

Dirac supports any video format that codes the raw image colors in a luma (grey-level) component with two associated chroma (color difference) components. These components are referred to as $Y$, $C1$ and $C2$.

In ITU defined systems (including ITU-BT.709, ITU-R BT.1361 and ITU-BT.1700), the $Y$, $C1$ and $C2$ values shall relate to the $E_Y$, $E_U$ and $E_V$ video components respectively. These video components are also widely referred to as $Y, U, V$ and $Y, C_B, C_R$.

In the ITU-T H.264 reversible color transform, the $Y$, $C1$ and $C2$ values shall correspond to the video components $Y, C_O, C_G$.

**Note:** Coding using $Y, C_O, C_G$ provides a simple reversible conversion to and from RGB components by using lossless integer transforms. The use of $Y, C_O, C_G$ supports lossless coding of RGB video and allows Dirac to be treated as an RGB compression system for applications that require this feature.

## 8.2 Interlace

Dirac supports both interlace and progressive formats. Interlace formats may be either top field first or bottom field first.

Dirac codes pictures where a picture may be a frame or a field. Fields consist of sets of alternate lines of video frames (even and odd lines). A pair of fields constituting a frame may correspond to distinct time intervals (true interlace scanning) or to the same time interval (progressive segmented frames). Hence the configuration of frame/field coding is independent of whether the video format is interlaced or progressive.

## 8.3 Component sampling

Chroma components $C1$ and $C2$ may be coded with the same dimensions as the Y component (4:4:4) sampling, or with half-width (4:2:2) or half-dimension (4:2:0) sampling.

$Y$, $C1$ and $C2$ picture components shall be sampled at the same temporal instant.

**Note:** All pictures are considered as individual entities whether or not all lines were sampled at the same instant. In video sequences that are not frame-based, such as 30fps interlaced video carrying 24fps progressive images in a 3:2 pull-down sequence, the compression performance may not be optimum. In such cases, a pre-processor may provide an encoder with a more easily compressed source such as the original 24fps source pictures. Such pre-processing does not form any part of this specification.

## 8.4 Bit resolution

The bit depth of each component sample is, in principle, unrestricted. Application-specific codecs may restrict the supported bit depth to a single value or a limited range of values.

Video is represented internally within the decoder specification as a bipolar (signed integer) signal. Video is presented at the video interface as an unsigned integer value by addition of an offset to these values (Section 15.10). Metadata concerning black level and white level is transmitted within the data stream (Section 10.3.8), but is not enforced at the decoder video interface: output video may undershoot or overshoot these values.

## 8.5 Picture frame size and rate

The frame size and frame rate is, in principle, unrestricted. Application-specific codecs may restrict the supported frame size and frame rate to a single value or a limited range of values, and compliance to a given level implies constraints on the values as specified in Annex D.

# 9 Stream syntax

This section specifies the overall structure of Dirac streams. Subsequent sections define the processes for parsing pictures, and Section 15 specifies how pictures are decoded.

## 9.1 Pseudocode

The parsing process is normatively defined using pseudocode and/or mathematical formulae. The definitions of stream syntax operations and pseudocode shall be as defined in Section 6.

The Dirac stream syntax uses a state model to express the stream in a way that can be parsed and used for decoding operations. The parsing and decoding operations are specified in terms of modifying the decoder state according to the data extracted from the Dirac stream. The state of the decoder is stored in the global variable **state**. This is a map (Section 6.3) and individual elements are accessed by means of named labels, e.g. **state**[VAR_NAME]. The state variables comprise the parameters that shall be used in parsing and decoding a picture. The variable **state** is a global variable and shall be accessible to all decoder functions and processes. All other variables shall be local to the function or process in which they are defined.

Decoder state variables (that is elements of state) may not directly correspond to parts of the stream, but may be calculated from them taking into account the decoder state as a whole. For example, a state variable value may be differentially encoded with respect to another value, with the difference, not the variable itself, encoded in the stream. Some parameters are encoded in the stream as indices to tables of values. The indices are coded as variable length integers. This allows the tables to be extended to contain new entries, in future versions of this specification, without changing the syntax.

## 9.2 Stream

A stream is a concatenation of Dirac sequences. The process for parsing a stream is to parse all sequences it contains. A Dirac sequence shall be decoded as a separate entity.

## 9.3 Sequence

The data contained in a Dirac sequence corresponds to a single video sequence with constant video parameters as defined in Sections 10.3. A Dirac sequence can be excised from a Dirac stream and decoded entirely independently.

A Dirac sequence shall comprise an alternating sequence of parse info headers and data units. The first data unit shall be a sequence header, and further sequence headers may be inserted at any data unit point in the sequence. The process for parsing a Dirac sequence shall be as defined below:

| *parse_sequence*() : | **Ref** |
|---|---|
|   **state** = {} | |
|   **state**[REF_PICTURES] = {} | |
|   *parse_info*() | 9.6 |
|   **video_params** = *sequence_header*() | 10 |
|   *parse_info*() | 9.6 |
|   **while** (*is_end_of_sequence*() == **False**): | 9.6.1 |
|     **if** (*is_seq_header*() == **True**): | 9.6.1 |
|       **video_params** = *sequence_header*() | 10 |
|     **else if** (*is_picture*()): | 9.6.1 |
|       *picture_parse*() | 11.1 |
|     **else if** (*is_auxiliary_data*()): | 9.6.1 |
|       *auxiliary_data*() | 9.5.1 |
|     **else if** (*is_padding*()): | 9.6.1 |
|       *padding*() | 9.5.2 |
|     *parse_info*() | 9.6 |

Each Dirac sequence shall start and end with a parse info header.

## 9.4 Parse Info headers

Parse info headers shall contain a 32 bit code so that the decoder can be synchronized with the stream. They are defined in Section 9.6. The parse info headers support navigating through the stream without the need to decode any data units. Each parse info header contains pointers to the location of the next and previous parse info headers within the stream. The stream may thus be thought of as a doubly linked list of data units. Each parse info headers contains a code that identifies the type of data held in the following data unit. This is the only information contained within the parse info headers that is needed to decode the sequence.

## 9.5 Data units

Data units may be one of:

- a sequence header,
- a picture,
- auxiliary data, and
- padding data.

A sequence header shall contain metadata describing the coded sequence and metadata needed to decode the stream. The sequence header is defined in Section 10. The first data unit in a sequence shall be a sequence header. To support reverse-parsing applications, the last data unit in a sequence should also be a sequence header.

Each sequence shall contain at least one picture and at least one sequence header. The first picture after each sequence header (if there is one) shall be an intra picture.

If a sequence contains more than one sequence header, the data in each sequence header shall be the same (byte-for-byte identical) within the sequence.

Each picture, whether a frame or field, may be coded with a dependency on prior pictures in the stream (reference pictures).

A picture data unit shall contain sufficient data to decode a single picture (frame or field of video), subject to having parsed a sequence header within the sequence and decoded any reference pictures.

Pictures within a sequence shall either all be fields or all be frames. Where pictures are fields, a sequence shall contain an even number of pictures, comprising a whole number of frames.

Auxiliary data and padding data do not contribute to the decoding process and so may be discarded.

Auxiliary and padding data units comprise undefined data for the purposes of this standard. These data units (together with the correct preceding parse info header) may be interposed at any point in the stream, but may safely be skipped by a compliant decoder. For the purposes of subsequent parts of this standard, the potential presence of auxiliary and padding data shall be ignored.

Padding data units shall not be used for any form of auxiliary data service or content. They may be used by an encoder, where required, to insert additional data to assist in complying with constant or constrained bit rate requirements.

### 9.5.1 Auxiliary data

The *auxiliary_data*() process for reading auxiliary data shall be as follows:

| *auxiliary_data*() : | **Ref** |
|---|---|
| *byte_align*() | |
| **for** $i = 1$ **to state**[NEXT_PARSE_OFFSET] − 13: | |
| *read_byte*() | |

### 9.5.2 Padding data

The *padding*() process for reading padding data shall be as follows:

| *padding*() : | **Ref** |
|---|---|
| *byte_align*() | |
| **for** $i = 1$ **to state**[NEXT_PARSE_OFFSET] − 13: | |
| *read_byte*() | |

## 9.6 Parse info header syntax

The parse info header provides information identifying the subsequent data unit type and length codes determining the number of bytes from the current parse info header to the next and previous parse info headers.

The parse info header shall be byte-aligned. It shall occur:

- at the beginning of a sequence
- at the end of a sequence
- before each data unit

The parse info header shall consist of 13 whole bytes. Thus subsequent data elements shall be byte aligned.

The value of the parse code, which is a component of the parse info header, shall be used to determine the type and format of the subsequent data unit.

The *parse_info*() process for reading parse info headers shall be as follows:

| *parse_info*() : | **Ref** |
|---|---|
| *byte_align*() | |
| **state**[PARSE_INFO_PREFIX] = *read_uint_lit*(4) | |
| **state**[PARSE_CODE] = *read_uint_lit*(4) | |
| **state**[NEXT_PARSE_OFFSET] = *read_uint_lit*(4) | |
| **state**[PREVIOUS_PARSE_OFFSET] = *read_uint_lit*(4) | |

The Parse Info parameters shall satisfy the following constraints:

- **state**[PARSE_INFO_PREFIX] shall be set to be 0x42 0x42 0x43 0x44, which is the character string “BBCD” as expressed by ISO/IEC 646.
- **state**[PARSE_CODE] shall be one of the supported values set out in Table 9.1
- **state**[NEXT_PARSE_OFFSET] shall be the number of bytes from the first byte of the current Parse Info header to the first byte of the next Parse Info header, if there is one. If there is no subsequent Parse Info header, it shall be be zero.
- **state**[PREVIOUS_PARSE_OFFSET] shall be the number of bytes from the first byte of the current Parse Info header to the first byte of the previous Parse Info header, if there is one. If there is no subsequent Parse Info header, it shall be be zero.

Consequently, the previous parse offset value of the current parse info header shall equal the next parse offset value of the previous parse info header, if there is one.

**Note:**

1. The parse info prefix, next parse offset and previous parse offset values are provided to support navigation and are not required to decode the sequence. See Section 14.1.
2. The parse offset values will normally be non-zero. However at the beginning and end of a stream there is no preceding or following parse info header respectively. In these circumstances the value of the offset is zero: these are the only places where zero values can occur.

### 9.6.1 Parse code values

Parse code values shall be divided into three sets: generic, core syntax and low delay syntax.

The value of parse codes allowed within the Dirac syntax shall be as shown in Table 9.1

| **state**[PARSE_CODE] | **Bits** | **Description** | **Number of Reference Pictures** |
|---|---|---|---|
| | | **Generic** | |
| 0x00 | 0000 0000 | Sequence header | – |
| 0x10 | 0001 0000 | End of Sequence | – |
| 0x20 | 0010 0000 | Auxiliary data | – |
| 0x30 | 0011 0000 | Padding data | – |
| | | **Core syntax** | |
| 0x0C | 0000 1100 | Intra Reference Picture (arithmetic coding) | 0 |
| 0x08 | 0000 1000 | Intra Non Reference Picture (arithmetic coding) | 0 |
| 0x4C | 0100 1100 | Intra Reference Picture (no arithmetic coding) | 0 |
| 0x48 | 0100 1000 | Intra Non Reference Picture (no arithmetic coding) | 0 |
| 0x0D | 0000 1101 | Inter Reference Picture (arithmetic coding) | 1 |
| 0x0E | 0000 1110 | Inter Reference Picture (arithmetic coding) | 2 |
| 0x09 | 0000 1001 | Inter Non Reference Picture (arithmetic coding) | 1 |
| 0x0A | 0000 1010 | Inter Non Reference Picture (arithmetic coding) | 2 |
| | | **Low-delay syntax** | |
| 0xCC | 1100 1100 | Intra Reference Picture | 0 |
| 0xC8 | 1100 1000 | Intra Non Reference Picture | 0 |

Table 9.1: Parse codes

Future versions of this specification may introduce new parse codes. In order that decoders complying with this version of the specification may decode future versions of the coded stream, the decoder shall discard data units that immediately follow parse info blocks containing unknown parse codes.

The parse codes shall be associated with a group of functions, listed below, which shall determine the type of subsequent data and the parsing and decoding processes which shall be used. All functions shall return a boolean, except for *num_refs*() which shall returns an integer:

| *is_seq_header*() : | **Ref** |
|---|---|
| **return state**[PARSE_CODE] == 0x00 | |

| *is_end_of_sequence*() : | **Ref** |
|---|---|
| **return state**[PARSE_CODE] == 0x10 | |

| *is_auxiliary_data*() : | **Ref** |
|---|---|
| **return** (**state**[PARSE_CODE]&0xF8) == 0x20 | |

| *is_padding*() : | **Ref** |
|---|---|
| **return state**[PARSE_CODE] == 0x30 | |

| *is_picture*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x08) == 0x08) | |

| *is_low_delay*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x88) == 0x88) | |

| *is_core_syntax*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x88) == 0x08) | |

| *using_ac*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x48) == 0x08) | |

| *is_reference*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x0C) == 0x0C) | |

| *is_non_reference*() : | **Ref** |
|---|---|
| **return** ((**state**[PARSE_CODE]&0x0C) == 0x08) | |

| *num_refs*() : | **Ref** |
|---|---|
| **return** (**state**[PARSE_CODE]&0x03) | |

| *is_intra*() : | **Ref** |
|---|---|
| **return** *is_picture*() and (*num_refs*() == 0) | |

| *is_inter*() : | **Ref** |
|---|---|
| **return** *is_picture*() and (*num_refs*() > 0) | |

### 9.6.2 Parse code value rationale (Informative)

The rationale for the parse code values in Table 9.1 is as follows:

- The MS bit (bit 7) is used to indicate the picture syntax (core or low delay syntax) and only applies to pictures. Core syntax codes whole frames rather than slices. Low delay syntax codes slices not frames.
- The second MS bit (bit 6) is used to indicate whether arithmetic coding is used and only applies to pictures. Core syntax may optionally use arithmetic coding. Low delay syntax does not use arithmetic coding. The permutation of the two bits which might indicate low delay syntax with arithmetic coding is reserved. Only arithmetic coding is supported on Inter pictures.

- The next three MS bits (bits 5, 4 and 3) indicate the type of data unit following the parse info unit. Bit 3 indicates whether it is a picture or non-picture data unit. Bits 5 and 4 indicate the 4 other parse codes.
- The three LS bits (bits 2, 1 and 0) indicate picture types. Bit 2 indicates whether a picture is a reference picture or not. Bits 0 and 1 indicate the number of references a picture has for motion compensation purposes: if these are both 0, the picture is an Intra picture.

# 10 Sequence header

This section defines the structure of the sequence header syntax. The sequence header shall be byte aligned. Parsing this header consists of reading the sequence parameters (parse parameters, base video format, source parameters and picture coding mode) and initializing the decoder parameters. The decoder parameters are initialized in the *set_coding_parameters*() process (Section 10.5).

The sequence header shall remain byte identical throughout a sequence.

The process for parsing the sequence header shall be as follows:

| *sequence_header*() : | **Ref** |
|---|---|
| *byte_align*() | |
| *parse_parameters*() | 10.1 |
| *base_video_format* = *read_uint*() | 10.2 |
| **video_params** = *source_parameters*(*base_video_format*) | 10.3 |
| *picture_coding_mode* = *read_uint*() | 10.4 |
| *set_coding_parameters*(**video_params**, *picture_coding_mode*) | 10.5 |
| **return video_params** | |

Parse parameters contain information a decoder may use to determine whether it is able to parse or decode the stream. Parse parameters are not used to decode the stream.

The base video format is a numerical index denoting a default set of parameters that describe the video source. For many common video formats the predefined values indicated by the base video format and defined in Annex C, will be sufficient without the need for further metadata to be present in the stream. However, to provide flexibility, source parameters may override the parameters indicated by the base video format (with the exception of the top field first flag).

Source parameters are parameters that describe the source video, not all of which are required to decode the stream. The source parameters are needed by applications that use the decoded video and so should be made available to them.

The picture coding mode indicates whether the video has been coded as a sequence of frames or fields.

Once the base video format, source parameters and picture coding mode have been read from the stream the information they contain may be decoded to provide the parameters used for decoding pictures. It is the purpose of the *set_coding_parameters*() process to initialize these parameters.

**Note:** Note that video parameters indicate whether the video sequence is interlaced or progressive. In particular a change from interlaced to progressive video, or vice-versa, necessitates that the Dirac sequence be terminated and a new sequence begun. The coding mode indicates whether the pictures within a Dirac sequence are fields or frames. Note that progressive video may still be encoded as fields, to provide backward compatibility with pseudo-progressive frame (PSF) video transmission.

The video parameters are not used by the Dirac decoder. Video parameter values should be made available using appropriate interfaces and standards to any downstream video processing device or display, but their use and interpretation by other devices is not specified in this standard. Neverthless, Annex F specifies the video systems model that should be used for the interpretation of video parameters.

## 10.1 Parse parameters

This section specifes the structure of the parse parameters, which is as follows:

| *parse_parameters*() : | **Ref** |
|---|---|
| **state**[VERSION_MAJOR] = *read_uint*() | |
| **state**[VERSION_MINOR] = *read_uint*() | |
| **state**[PROFILE] = *read_uint*() | |
| **state**[LEVEL] = *read_uint*() | |

Parse parameter data shall be constant (byte-for-byte identical) for all instances of the sequence header within a Dirac sequence. For stream interchange, parse parameter data should also be constant across all sequences within a stream.

### 10.1.1 Version number

The major version number shall define the version of the syntax with which the stream complies. A decoder complies with a major version number if it can parse all bit streams that comply with that version number. 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 fully all parts of a stream.

All minor versions of the specification shall be functionally compatible with earlier minor versions with the same major version number. Later minor versions may contain corrections, clarifications, and removal of ambiguities. Later minor version numbers shall not contain new features or new normative provisions.

Functional compatibility shall imply that a decode with the same major version number but a later minor version number than that contained in a stream, shall be capable of decoding the stream and producing pictures substantially equivalent to that of a decoder with the same version numbers as the stream.

The major version number of a stream compliant with this version of the Dirac specification shall be 2.

The minor version number of a stream compliant with this version of the Dirac specification shall be 2.

### 10.1.2 Profiles and levels

A profile shall define the toolset that is sufficient to decode a sequence.

A level shall determine decoder resources (picture and data buffers; computational resources) sufficient to decode a sequence, including the sizes **state**[RB_SIZE] and **state**[DPB_SIZE] of the reference picture and decoded picture buffers.

Applicable values of profile and level and the variables they set are specified in Annex D.

## 10.2 Base video format

The value of *base_video_format* decoded in parsing the sequence header shall be an index into table 10.1. For each entry in the table parameters are defined, in Annex C, indicating base video parameters corresponding to one of a set of predefined formats.

The selection of a base format represents an initial approximation to the video format which can then be refined to capture all the video format characteristics accurately by overriding parameters as necessary. In particular, the predefined video formats listed in table 10.1 do not represent all the video formats supported by Dirac; any video format parameters may in principle be defined and supported by Dirac sequence.

These base parameters may be modified by subsequent metadata present in the stream, with the exception of the top field first parameter which shall only be set by the base video format (see Section 10.3.4).

| Video format index | Video format description |
|---|---|
| 0 | Custom Format |
| 1 | QSIF525 |
| 2 | QCIF |
| 3 | SIF525 |
| 4 | CIF |
| 5 | 4SIF525 |
| 6 | 4CIF |
| 7 | SD 480I-60 (525 Line 59.94 Field/s Standard Definition) |
| 8 | SD 576I-50 (625 Line 50 Field/s Standard Definition) |
| 9 | HD 720P-60 (720 Line 59.94 Frame/s High Definition) |
| 10 | HD 720P-50 (720 Line 50 Frame/s High Definition) |
| 11 | HD 1080I-60 (1080 Line 60 Field/s High Definition) |
| 12 | HD 1080I-50 (1080 Line 50 Field/s High Definition) |
| 13 | HD 1080P-60 (1080 Line 59.94 Frame/s High Definition) |
| 14 | HD 1080P50 (1080 Line 50 Frame/s High Definition) |
| 15 | DC 2K-24 (2K D-Cinema, 24 Frame/s) |
| 16 | DC 4K-24 (4K D-Cinema, 24 Frame/s) |
| 17 | UHDTV 4K-60 (2160-line 59.94 Frame/s UHDTV) |
| 18 | UHDTV 4K-50 (2160-line 50 Frame/s UHDTV) |
| 19 | UHDTV 8K-60 (4320-line 59.94 Frame/s UHDTV) |
| 20 | UHDTV 8K-50 (4320-line 50 Frame/s UHDTV) |

Table 10.1: Dirac predefined video formats

**Note:**

1. The custom format is intended for use when no other suitable base video format is available from the table. Video format defaults will still be set as per Annex C, but these are token values which are expected to be almost wholly overridden by the subsequent source parameters.

2. The base video format ought to be as close as possible to the desired video format, especially in terms of picture dimensions and frame rate.

3. True 60Hz formats can be encoded by overriding the frame rate parameters (Section 10.3.5).

## 10.3 Source parameters

The source parameters are intended to indicate the format of the video that was originally encoded. They provide metadata that indicates how the decoded video should be displayed.

The source parameters shall comprise frame size, chroma sampling format, scan format, frame rate, pixel aspect ratio, clean area, signal range and colour specification. The frame size, chroma sampling format, scanning format and the signal range are required to decode the video. Display and downstream processing falls outside the scope of this specification, hence the interpretation of the other parameters (not required to decode the video) is not normatively defined, with the exception of frame rate (Section 10.3.5). The frame rate may impose requirements on compliant decoders for a given level and profile (Annex D).

Source parameter data shall remain constant throughout a Dirac sequence.

Default values for the source parameters shall be derived from the video format, as defined in Annex C. These default values shall be the source parameters unless they are overridden with alternative values encoded as part of the Source Parameters part of the stream.

The *source_parameters*() process shall return a structure defining the video source parameters. It shall be defined as follows:

| *source_parameters*(*base_video_format*) : | **Ref** |
|---|---|
| **video_params** = *set_source_defaults*(*base_video_format*) | 10.3.1 |
| *frame_size*(**video_params**) | 10.3.2 |
| *chroma_sampling_format*(**video_params**) | 10.3.3 |
| *scan_format*(**video_params**) | 10.3.4 |
| *frame_rate*(**video_params**) | 10.3.5 |
| *pixel_aspect_ratio*(**video_params**) | 10.3.6 |
| *clean_area*(**video_params**) | 10.3.7 |
| *signal_range*(**video_params**) | 10.3.8 |
| *colour_spec*(**video_params**) | 10.3.9 |
| **return video_params** | |

### 10.3.1 Setting source defaults

The function that sets the default values of the source video parameters shall take the video format index as an argument. That is, the signature of this function is: *set_source_defaults*(*base_video_format*) where *base_video_format* is an unsigned integer. The function returns a map of source video parameters.

The source video parameters shall be set, based on the video format index, as defined in Annex C. The parameters set by this function shall be: frame size, sampling format (4:4:4, 4:2:2 or 4:2:0), scan format (progressive or interlace), frame rate, pixel aspect ratio, clean area, signal range, colour specification. The labels used to access the map returned by the function shall be as defined in the subsequent sections that specify how to override the base video source parameters.

### 10.3.2 Frame size

The frame size decoding process shall be as follows:

| *frame_size*(**video_params**) : | **Ref** |
|---|---|
| *custom_dimensions_flag* = *read_bool*() | |
| **if** (*custom_dimensions_flag* == **True**): | |
|     **video_params**[FRAME_WIDTH] = *read_uint*() | |
|     **video_params**[FRAME_HEIGHT] = *read_uint*() | |

Thus is *custom_dimensions_flag* is **True**, the frame size determined by the base video format shall be overridden.

The frame width shall correspond to the width of the coded video, in pixels, that is coded in the stream. The frame height shall correspond to the number of lines per frame in the coded video, irrespective of whether the coded video is progressively scanned or is interlaced.

### 10.3.3 Chroma sampling format

The chroma sampling format decoding process shall be as follows:

| *chroma_sampling_format*(**video_params**) : | **Ref** |
|---|---|
| *custom_chroma_format_flag* = *read_bool*() | |
| **if** (*custom_chroma_format_flag* == **True**): | |
|     **video_params**[CHROMA_FORMAT_INDEX] = *read_uint*() | |

Thus if *custom_chroma_format_flag* is **True** then the base video format value is overridden.

The decoded value of **video_params**[CHROMA_FORMAT_INDEX] shall lie in the range 0 to 2 with values as defined in table 10.2:

| **video_params**[CHROMA_FORMAT_INDEX] | **Chroma format** |
|---|---|
| 0 | 4:4:4 |
| 1 | 4:2:2 |
| 2 | 4:2:0 |

Table 10.2: Supported chroma sampling formats

The chroma sampling format shall be used to determine the width and height of the chroma components of the coded video as described in Section 10.5.1 below.

### 10.3.4 Scan format

The scan format parameter shall indicate whether the source video represents progressive frames or interlaced fields.

The scan format decoding process shall be defined as follows:

| $scan\_format$(**video_params**) : | **Ref** |
|---|---|
| $custom\_scan\_format\_flag = read\_bool()$ | |
| **if** ($custom\_scan\_format\_flag$ == **True**): | |
|     **video_params**[SOURCE_SAMPLING] = $read\_uint()$ | |

If the custom scan format flag is set to **True**, the source sampling parameter defined by the base video format values shall be overridden by new values.

If **video_params**[SOURCE_SAMPLING] is set to 0, then the source video shall be progressively sampled. If it is 1, then the source video shall be interlaced. Values greater than 1 shall be reserved.

If the source video is interlaced, then **video_params**[TOP_FIELD_FIRST] shall be **True** if the top line of the frame is in the earlier field, else **video_params**[TOP_FIELD_FIRST] shall be **False**. This shall be set only by the base video format and cannot be overridden in the source parameters.

Both interlaced and progressive video may be coded as fields or frames.

### 10.3.5 Frame rate

The frame rate value (in frames per second) shall be **video_params**[FRAME_RATE_NUMER] divided by **video_params**[FRAME_RATE_DENOM]

The process for decoding the frame rate parameters shall be as follows:

| $frame\_rate$(**video_params**) : | **Ref** |
|---|---|
| $custom\_frame\_rate\_flag = read\_bool()$ | |
| **if** ($custom\_frame\_rate\_flag$ == **True**): | |
|     $index = read\_uint()$ | |
|     **if** ($index == 0$): | |
|         **video_params**[FRAME_RATE_NUMER] = $read\_uint()$ | |
|         **video_params**[FRAME_RATE_DENOM] = $read\_uint()$ | |
|     **else**: | |
|         $preset\_frame\_rate$(**video_params**, $index$) | |

If $custom\_frame\_rate\_flag$ is set to **True** the frame rate parameters set by the base video format shall be overridden by new values.

The decoded value of $index$ shall fall in the range 0 to 10.

If *index* is 0, then the frame rate numerator and denominator shall be individually defined by unsigned integer values.

For values greater than 0, the process $preset\_frame\_rate(\mathbf{video\_params}, index)$ shall set frame rate elements of **video_params** according to table 10.3.

| *index* | Numerator | Denominator |
|---|---|---|
| 1 | 24000 | 1001 |
| 2 | 24 | 1 |
| 3 | 25 | 1 |
| 4 | 30000 | 1001 |
| 5 | 30 | 1 |
| 6 | 50 | 1 |
| 7 | 60000 | 1001 |
| 8 | 60 | 1 |
| 9 | 15000 | 1001 |
| 10 | 25 | 2 |

Table 10.3: Available preset frame rate values

**Note:** Note that what is encoded is frame rate, not picture rate. If the video is coded as fields, then picture rate is twice the encoded frame rate.

### 10.3.6 Pixel aspect ratio

The pixel aspect ratio shall be defined as the ratio of the parameters:

**video_params**[PIXEL_ASPECT_RATIO_NUMER] : **video_params**[PIXEL_ASPECT_RATIO_DENOM]

The process for decoding the pixel aspect ratio parameters shall be defined as follows:

```
pixel_aspect_ratio(video_params) :                                        Ref
  custom_pixel_aspect_ratio_flag = read_bool()
  if (custom_pixel_aspect_ratio_flag == True):
    index = read_uint()
    if (index == 0):
      video_params[PIXEL_ASPECT_RATIO_NUMER] = read_uint()
      video_params[PIXEL_ASPECT_RATIO_DENOM] = read_uint()
    else:
      preset_pixel_aspect_ratio(video_params, index)
```

If *custom_pixel_aspect_ratio_flag* is set to **True**, the pixel aspect ratio defined by the default values shall be overridden by the new values defined by the index value.

The decoded value of *index* shall fall in the range 0 to 6.

If the value of *index* is 0, then the pixel aspect ratio numerator and denominator shall be individually defined by unsigned integer values.

If $index > 0$, the process $preset\_pixel\_aspect\_ratio(\mathbf{video\_params}, index)$ shall set the pixel aspect ratio according to table 10.4.

**Note:**

1. The pixel aspect ratio value defines the intended ratio of the pixel sampling such that the viewed picture has no geometric distortion. The pixel aspect ratio of an image is the ratio of the spacing of horizontal

| *index* | Numerator | Denominator |
|---|---|---|
| 1 (Square Pixels) | 1 | 1 |
| 2 (525-line systems) | 10 | 11 |
| 3 (625-line systems) | 12 | 11 |
| 4 (16:9 525-line systems) | 40 | 33 |
| 5 (16:9 625-line systems) | 16 | 11 |
| 6 (reduced horizontal resolution) | 4 | 3 |

Table 10.4: Available preset pixel aspect ratio values

samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios (PARs) are fundamental properties of sampled images because they determine the displayed shape of objects in the image. Failure to use the right PAR will result in distorted images, for example circles will be displayed as ellipses etc.

2. The pixel apect ratios shown in table 10.4 assume a 704x480 active picture for 525-line systems and a 704x576 active picture for 625-line systems.

3. Some video processing tools require an image aspect ratio. This can be derived from the pixel aspect ratio by multiplying the ratio of horizontal to vertical pixels by the pixel aspect ratio. So, for example, for a 704 x 480 line picture, with a pixel aspect ratio of 10:11 the image aspect ratio is (704 x 10)/(480 x 11) which is exactly 4:3.

### 10.3.7 Clean area

The process for decoding the clean area parameters shall be as follows:

| *clean_area*(**video_params**) : | **Ref** |
|---|---|
| *custom_clean_area_flag* = *read_bool*() | |
| **if** (*custom_clean_area_flag* == **True**): | |
|     **video_params**[CLEAN_WIDTH] = *read_uint*() | |
|     **video_params**[CLEAN_HEIGHT] = *read_uint*() | |
|     **video_params**[CLEAN_LEFT_OFFSET] = *read_uint*() | |
|     **video_params**[CLEAN_TOP_OFFSET] = *read_uint*() | |

The following restrictions shall apply:

- **video_params**[CLEAN_WIDTH]+**video_params**[CLEAN_LEFT_OFFSET] $\leq$ **video_params**[FRAME_WIDTH]
- **video_params**[CLEAN_HEIGHT]+**video_params**[CLEAN_TOP_OFFSET] $\leq$ **video_params**[FRAME_HEIGHT]

**Note:** The meaning and use of clean area are application defined: it might correspond to that picture which is to be displayed, or define a "container" within a picture of larger size.

### 10.3.8 Signal range

The signal range parameters indicate how the signal range of the picture component data, decoded by the Dirac decoder, should be adjusted prior to the colour matrixing operations (described in informative Annex F.1.3).

The signal range parameters shall also be used to determine the luma depth and chroma depth parameters (Section 10.5.2) and the resulting clipping levels applied to the decoded video (Section 15.9).

The process for decoding the signal range parameters is as follows:

| *signal_range*(**video_params**) : | **Ref** |
|---|---|
| *custom_signal_range_flag* = *read_bool*() | |
| **if** (*custom_signal_range_flag* == **True**): | |
|  *index* = *read_uint*() | |
|  **if** (*index* == 0): | |
|   **video_params**[LUMA_OFFSET] = *read_uint*() | |
|   **video_params**[LUMA_EXCURSION] = *read_uint*() | |
|   **video_params**[CHROMA_OFFSET] = *read_uint*() | |
|   **video_params**[CHROMA_EXCURSION] = *read_uint*() | |
|  **else**: | |
|   *preset_signal_ranges*(**video_params**, *index*) | |

If *custom_signal_range_flag* is set to **True** then the base video format signal range parameters shall be overridden by new values.

The decoded value of *index* shall fall in the range 0 to 4.

If *index* $> 0$ the process *preset_signal_ranges*(**video_params**, *index*) shall set the signal range elements of **video_params** according to table 10.5.

| *index* | Luma offset | Luma excursion | Chroma offset | Chroma excursion |
|---|---|---|---|---|
| 1 (8 Bit Full Range) | 0 | 255 | 128 | 255 |
| 2 (8 Bit Video) | 16 | 219 | 128 | 224 |
| 3 (10 Bit Video) | 64 | 876 | 512 | 896 |
| 4 (12 Bit Video) | 256 | 3504 | 2048 | 3584 |

Table 10.5: Available signal range presets

**Note:** Decoded video is represented within the decoder specification as bi-polar signals. An offset is added when video is output so that it is represented by unsigned integer values.

### 10.3.9 Color specification

The colour specification shall consist of three component parts:

- Color primaries
- Color matrix
- Transfer function

Defaults are available for all three parts collectively and individually.

The process for decoding the colour specification parameters shall be follows:

| *colour_spec*(**video_params**) : | **Ref** |
|---|---|
| *custom_colour_spec_flag* = *read_bool*() | |
| **if** (*custom_colour_spec_flag* == **True**): | |
|  *index* = *read_uint*() | |
|  *preset_colour_specs*(**video_params**, *index*) | |
|  **if** (*index* == 0): | |
|   *colour_primaries*(**video_params**) | 10.3.9.1 |
|   *colour_matrix*(**video_params**) | 10.3.9.2 |
|   *transfer_function*(**video_params**) | 10.3.9.3 |

The decoded value of *index* shall fall in the range 0 to 4.

*preset_colour_spec*(*index*) shall set the colour primaries, matrix and transfer function elements of **video_params** as specified in Table 10.6. If the value of *index* is 0, these values may be overridden as defined in the succeeding sections.

| *index* | Description | Primaries | Matrix | Transfer function |
|---|---|---|---|---|
| 0 | Custom | HDTV | HDTV | TV gamma |
| 1 | SDTV 525 | SDTV 525 | SDTV | TV gamma |
| 2 | SDTV 625 | SDTV 625 | SDTV | TV gamma |
| 3 | HDTV | HDTV | HDTV | TV gamma |
| 4 | D-Cinema | HDTV | HDTV | DCinema gamma |

Table 10.6: Color specification presets

#### 10.3.9.1 Color primaries

The colour primaries decoding process shall be defined as follows:

```
colour_primaries(video_params) :                                  Ref
  custom_colour_primaries_flag = read_bool()
  if (custom_colour_primaries_flag == True):
    index = read_uint()
    preset_colour_primaries(video_params, index)
```

The decoded value of *index* shall fall in the range 0 to 3.

*preset_colour_primaries*(**video_params**, *index*) shall set the colour primaries element of **video_params** as specified in Table 10.7.

| *index* | Description | Specification | Comment |
|---|---|---|---|
| 0 | HDTV | ITU-R BT.709 | Also Computer, Web, sRGB |
| 1 | SDTV 525 | SMPTE 170M | 525 primaries |
| 2 | SDTV 625 | EBU Tech 3213-E | 625 primaries |
| 3 | D-Cinema | SMPTE 428.1 | CIE XYZ |

Table 10.7: Color primaries presets

#### 10.3.9.2 Color matrix

The colour matrix decoding process shall be defined as follows:

```
colour_matrix() :                                                 Ref
  colour_matrix_flag = read_bool()
  if (colour_matrix_flag == True):
    index = read_uint()
    preset_colour_matrices(index)
```

The decoded value of *index* shall fall in the range 0 to 2.

The *preset_colour_matrices*(**video_params**, *index*) process shall set the colour matrix element in **video_params** as specified in Table 10.8.

| *index* | Description | Specification | Color matrix | Comment |
|---|---|---|---|---|
| 0 | HDTV | ITU-R BT.709 | $K_R = 0.2126$, $K_B = 0.0722$ | Also computer and web |
| 1 | SDTV | ITU-R BT.601 | $K_R = 0.299$, $K_B = 0.114$ | |
| 2 | Reversible | ITU-T H.264 | YCgCo | |

Table 10.8: Color matrix presets

#### 10.3.9.3 Transfer function

The transfer function decoding process shall be defined as follows:

| $transfer\_function($**video_params**$)$ : | **Ref** |
|---|---|
| $custom\_transfer\_function\_flag = read\_bool()$ | |
| **if** $(custom\_transfer\_function\_flag ==$ **True**$)$: | |
|   $index = read\_uint()$ | |
|   $preset\_transfer\_function($**video_params**$, index)$ | |

*index* shall fall in the range 0 to 3. The $preset\_transfer\_function($**video_params**$, index)$ process shall set the transfer function element of **video_params** as specified in Table 10.9.

| *index* | **Description** | **Specification** |
|---|---|---|
| 0 | TV gamm | ITU-R BT.1361 |
| 1 | Extended Gamut | ITU-R BT.1361 1998 Annex 1 |
| 2 | Linear | Linear |
| 3 | DCI Gamma | SMPTE 428.1 |

Table 10.9: Transfer function presets

## 10.4 Picture coding mode

The picture coding mode value in the sequence header shall determine whether source video is coded as frames or fields.

If the picture coding mode value is 1 then pictures shall correspond to fields. If it is 0 then pictures shall correspond to frames. Other picture coding mode values shall be reserved for future extensions.

If video is coded as fields then the earliest field in each frame shall have an even picture number (Section 11.1.1). That is the LSB of the picture number, expressed as a binary number, indicates field parity.

With field coding each frame shall be split into two fields as indicated by the scan format (Section 10.3.4).

An effect of field coding shall be to halve the vertical dimensions of coded pictures. Hence, once the picture coding mode is known, the picture dimensions, which shall be stored as part of the global state variable, shall be set (Section 10.5.1).

**Note:** It is possible to code progressive video as fields. In this case, the assignment of frame lines to fields will be determined by the value of the top field first parameter in the base video format (Annex C). Note that, according to Section 10.3.4, this base format default cannot be overridden for progressive video, as to do so would be artificial.

Sometimes progressive source video is conveyed as if it were interlaced (for example using interlaced SDI modes), and could be signaled as such. This is known as progressive segmented frames (PSF). A Dirac encoder could detect PSF, and signal video as progressive, yet still code the video as fields in order to introduce no additional buffering delay in the signal chain. Or it could take the signaled video format at face value.

## 10.5 Initializing coding parameters

The $set\_coding\_parameters()$ process shall initialize the dimensions of the coded picture (frame or field), and the video depth (the maximum number of bits in a decoded video sample), which are needed to decode pictures.

Picture dimensions and video depth shall remain constant throughout a Dirac sequence.

Initialization of the coding parameters shall be as defined in the table below:

| *set_coding_parameters*(**video_params**, *picture_coding_mode*) : | **Ref** |
|---|---|
| *picture_dimensions*(**video_params**, *picture_coding_mode*) | 10.5.1 |
| *video_depth*(**video_params**) | 10.5.2 |

### 10.5.1 Picture dimensions

The picture dimensions process, which determines the size of coded pictures, shall be defined as follows:

| *picture_dimensions*(**video_params**, *picture_coding_mode*) : | **Ref** |
|---|---|
| **state**[LUMA_WIDTH] = **video_params**[FRAME_WIDTH] | |
| **state**[LUMA_HEIGHT] = **video_params**[FRAME_HEIGHT] | |
| **state**[CHROMA_WIDTH] = **state**[LUMA_WIDTH] | |
| **state**[CHROMA_HEIGHT] = **state**[LUMA_HEIGHT] | |
| *chroma_format_index* = **video_params**[CHROMA_FORMAT_INDEX]] | |
| **if** (*chroma_format_index* == 1): | |
|   **state**[CHROMA_WIDTH]// = 2 | |
| **else if** (*chroma_format_index* == 2): | |
|   **state**[CHROMA_WIDTH]// = 2 | |
|   **state**[CHROMA_HEIGHT]// = 2 | |
| **if** (*picture_coding_mode* == 1): | |
|   **state**[LUMA_HEIGHT]// = 2 | |
|   **state**[CHROMA_HEIGHT]// = 2 | |

The parameter **video_params**[FRAME_HEIGHT] refers to the height of a frame. The parameter **state**[LUMA_HEIGHT] refers to the height of a picture. A picture may be either a frame or a field depending on whether it is being coded in an interlaced or progressive mode.

Frame height shall be an integer multiple of picture chroma height.

For convenience, the following utility functions shall be defined:

| *chroma_h_ratio*() : | **Ref** |
|---|---|
| **return state**[LUMA_WIDTH]//**state**[CHROMA_WIDTH] | |

| *chroma_v_ratio*() : | **Ref** |
|---|---|
| **return state**[LUMA_HEIGHT]//**state**[CHROMA_HEIGHT] | |

### 10.5.2 Video depth

The *video_depth*() process, which determines the maximum number of bits required to represent a sample of the decoded video, shall be defined as follows:

| *video_depth*(**video_params**) : | **Ref** |
|---|---|
| **state**[LUMA_DEPTH] = $\text{intlog}_2$(**video_params**[LUMA_EXCURSION] + 1) | |
| **state**[CHROMA_DEPTH] = $\text{intlog}_2$(**video_params**[CHROMA_EXCURSION] + 1) | |

Note that for YCoCg format the luma and chroma depths are different.

# 11 Picture syntax

This section specifies the structure of Dirac picture data units.

## 11.1 Picture parsing

This section specifies the operation of the *picture_parse*() process. The process for decoding and outputting pictures is specified in Section 15.

Picture data may be successfully parsed after parsing a sequence header within the same Dirac sequence. The picture parsing process shall be defined as follows:

| *picture_parse*() : | **Ref** |
|---|---|
| *byte_align*() | |
| *picture_header*() | 11.1.1 |
| **if** (*is_inter*()): | 9.6.1 |
|     *byte_align*() | |
|     *picture_prediction*() | 11.2 |
| *byte_align*() | |
| *wavelet_transform*() | 11.3 |

### 11.1.1 Picture header

The picture header shall immediately follow a parse info header with a picture parse code (Section 9.6). The picture header parsing process shall be defined as follows:

| *picture_header*() : | **Ref** |
|---|---|
| **state**[PICTURE_NUM] = $read\_uint\_lit(4)$ | |
| **if** (*is_inter*()): | 9.6.1 |
|     **state**[REF1_PICTURE_NUM] = (**state**[PICTURE_NUM] + *read_sint*())%$2^{32}$ | |
|     **if** ($num\_refs() == 2$): | 9.6.1 |
|         **state**[REF2_PICTURE_NUM] = (**state**[PICTURE_NUM] + *read_sint*())%$2^{32}$ | |
| **if** (*is_reference*()): | 9.6.1 |
|     **state**[RETD_PIC_NUM] = (**state**[PICTURE_NUM] + *read_sint*())%$2^{32}$ | |

Picture numbers shall be unique within a sequence and the set of all picture numbers within a sequence shall form a contiguous block of numbers.

Reference picture numbers shall encoded differentially with respect to the picture number.

The pictures corresponding to the reference picture numbers of a given picture shall occur before the given picture in the sequence.

The retired picture shall be a picture which shall be removed from the reference picture buffer before the current picture is decoded (Section 15.1). The rules for the use of the reference picture buffer shall be as defined in Section 15.4.

## 11.2 Picture prediction data

This section defines the picture prediction process that shall be used for decoding picture prediction parameters and motion vector fields for motion compensation.

The picture prediction process shall be defined as follows:

| *picture_prediction*() : | **Ref** |
|---|---|
| *picture_prediction_parameters*() | 11.2.1 |
| *byte_align*() | |
| *block_motion_data*() | 12 |

The decoding and generation of block motion vector fields shall be as defined in Section 12.

### 11.2.1 Picture prediction parameters

Picture prediction parameters consist of metadata required for successful parsing of the motion data and for performing motion compensation (Section 15.8).

The picture prediction parameters shall be defined as follows:

| *picture_prediction_parameters*() : | **Ref** |
|---|---|
| *block_parameters*() | 11.2.2 |
| *motion_vector_precision*() | 11.2.5 |
| *global_motion*() | 11.2.6 |
| *picture_prediction_mode*() | 11.2.7 |
| *reference_picture_weights*() | 11.2.8 |

### 11.2.2 Block parameters

This section specifies the operation of the process for setting motion compensation block parameters, which shall consist of the state variables **state**[LUMA_XBLEN], **state**[LUMA_YBLEN], **state**[LUMA_XBSEP], and **state**[LUMA_YBSEP] defining luma blocks, and **state**[CHROMA_XBLEN], **state**[CHROMA_YBLEN], **state**[CHROMA_XBSEP], and **state**[CHROMA_YBSEP] defining chroma blocks.

| *block_parameters*() : | **Ref** |
|---|---|
| *index* = *read_uint*() | |
| **if** (*index* == 0): | |
|   **state**[LUMA_XBLEN] = *read_uint*() | |
|   **state**[LUMA_YBLEN] = *read_uint*() | |
|   **state**[LUMA_XBSEP] = *read_uint*() | |
|   **state**[LUMA_YBSEP] = *read_uint*() | |
| **else**: | |
|   *preset_block_params*(*index*) | |
| *chroma_block_params*() | 11.2.3 |
| *motion_data_dimensions*() | 11.2.4 |

*index* shall lie in the range 0 to 4.

The *preset_block_params*(*index*) shall set the block parameters as specified in Table 11.1.

Chroma block parameter values shall be determined from luma values as defined in Section 11.2.3).

The dimensions of motion data arrays (numbers of blocks and superblocks) shall be as defined in Section 11.2.4.

| *index* | **state**[LUMA_XBLEN] | **state**[LUMA_YBLEN] | **state**[LUMA_XBSEP] | **state**[LUMA_YBSEP] |
|---|---|---|---|---|
| 1 | 8 | 8 | 4 | 4 |
| 2 | 12 | 12 | 8 | 8 |
| 3 | 16 | 16 | 12 | 12 |
| 4 | 24 | 24 | 16 | 16 |

Table 11.1: Luma block parameter presets

Block parameters shall satisfy the following constraints:

1. **state**[LUMA_XBLEN], **state**[LUMA_YBLEN], **state**[LUMA_XBSEP], and **state**[LUMA_YBSEP] shall all be positive multiples of 4
2. **state**[LUMA_XBLEN] $\geq$ **state**[LUMA_XBSEP] and **state**[LUMA_YBLEN] $\geq$ **state**[LUMA_YBSEP]
3. **state**[LUMA_XBLEN] $\leq$ 2$*$**state**[LUMA_XBSEP] and **state**[LUMA_YBLEN] $\leq$ 2$*$**state**[LUMA_YBSEP]

**Note:** Note that these requirements do not preclude length from equalling separation, i.e. motion compensation blocks that are not overlapped.

### 11.2.3 Setting chroma block parameters

This section defines how chroma block parameters shall be derived from luma block dimensions.

Chroma block parameters shall be equal to the corresponding luma block parameters scaled according to the chroma vertical and horizontal subsampling ratios. In this way chroma blocks and luma blocks are co-located in the video picture.

| *chroma_block_params*() : | **Ref** |
|---|---|
| **state**[CHROMA_XBLEN] = **state**[LUMA_XBLEN]//*chroma_h_ratio*() | 10.5.1 |
| **state**[CHROMA_YBLEN] = **state**[LUMA_YBLEN]//*chroma_v_ratio*() | 10.5.1 |
| **state**[CHROMA_XBSEP] = **state**[LUMA_XBSEP]//*chroma_h_ratio*() | 10.5.1 |
| **state**[CHROMA_YBSEP] = **state**[LUMA_YBSEP]//*chroma_v_ratio*() | 10.5.1 |

### 11.2.4 Numbers of blocks and superblocks

The number of blocks and superblocks horizontally and vertically shall be set as follows:

| *motion_data_dimensions*() : | **Ref** |
|---|---|
| **state**[SUPERBLOCKS_X] = **state**[LUMA_WIDTH] + 4 ∗ **state**[LUMA_XBSEP] − 1 | |
| **state**[SUPERBLOCKS_X]// = 4 ∗ **state**[LUMA_XBSEP] | |
| **state**[SUPERBLOCKS_Y] = **state**[LUMA_HEIGHT] + 4 ∗ **state**[LUMA_YBSEP] − 1 | |
| **state**[SUPERBLOCKS_Y]// = 4 ∗ **state**[LUMA_YBSEP] | |
| **state**[BLOCKS_X] = 4 ∗ **state**[SUPERBLOCKS_X] | |
| **state**[BLOCKS_Y] = 4 ∗ **state**[SUPERBLOCKS_Y] | |

These values shall determine the size of motion data arrays as per Section 12.2.1.

**Note:** The number of superblocks is set so that the dimensions of the picture are entirely covered by superblocks at a separation of 4 ∗ **state**[LUMA_XBSEP] horizontally and 4 ∗ **state**[LUMA_YBSEP] vertically.

### 11.2.5 Motion vector precision

The motion vector precision process shall be as follows:

| *motion_vector_precision*() : | **Ref** |
|---|---|
| **state**[MV_PRECISION] = *read_uint*() | |

**state**[MV_PRECISION] shall lie in the range 0 (pixel-accurate) to 3 (1/8th-pixel accurate).

### 11.2.6 Global motion

Global motion parameters shall be encoded if the **state**[USING_GLOBAL] flag is set to **True**. Up to two sets shall be encoded, depending upon the number of references.

The global motion process shall be as follows:

| *global_motion*() : | Ref |
|---|---|
| **state**[USING_GLOBAL] = *read_bool*() | |
| **if** (**state**[USING_GLOBAL] == **True**): | |
|   *global_motion_parameters*(**state**[GLOBAL_PARAMS][1]) | |
|   **if** (*num_refs*() == 2): | |
|     *global_motion_parameters*(**state**[GLOBAL_PARAMS][2]) | |

Each of the global motion parameters shall consist of three elements:

- an integer pan/tilt vector **state**[GLOBAL_PARAMS][*n*][PAN_TILT]
- an integer 2x2 matrix element **state**[GLOBAL_PARAMS][*n*][ZRS] capturing zoom, rotation and shear, together with a scaling exponent **state**[GLOBAL_PARAMS][*n*][ZRS_EXP]
- an integer perspective vector **state**[GLOBAL_PARAMS][*n*][*pespective*] capturing the effect of non-orthogonal projection onto the image plane, together with a scaling exponent **state**[GLOBAL_PARAMS][*n*][*pespective_exp*]

Their interpretation and the process for generating a global motion vector field shall be as defined in Section 15.8.8.

The global motion parameters process shall be defined as follows:

| *global_motion_parameters*(*gparams*) : | Ref |
|---|---|
| *pan_tilt*(*gparams*) | |
| *zoom_rotate_shear*(*gparams*) | |
| *perspective*(*gparams*) | |

The *pan_tilt*() process shall extracts horizontal and vertical translation elements and shall be defined as follows:

| *pan_tilt*(*gparams*) : | Ref |
|---|---|
| *gparams*[PAN_TILT] = **0** | |
| *nonzero_pan_tilt_flag* = *read_bool*() | |
| **if** (*nonzero_pan_tilt_flag* == **True**): | |
|   *gparams*[PAN_TILT][0] = *read_sint*() | |
|   *gparams*[PAN_TILT][1] = *read_sint*() | |

The *zoom_rotate_shear*() process shall extract a linear matrix element and shall be as defined as follows:

| *zoom_rotation_shear*(*gparams*) : | Ref |
|---|---|
| *nontrivial_zrs_flag* = *read_bool*() | |
| **if** (*nontrivial_zrs_flag* == **True**): | |
|   *gparams*[ZRS_EXP] = *read_uint*() | |
|   *gparams*[ZRS][0][0] = *read_sint*() | |
|   *gparams*[ZRS][0][1] = *read_sint*() | |
|   *gparams*[ZRS][1][0] = *read_sint*() | |
|   *gparams*[ZRS][1][1] = *read_sint*() | |
| **else**: | |
|   *gparams*[ZRS_EXP] = 0 | |
|   *gparams*[ZRS][0][0] = 1 | |
|   *gparams*[ZRS][0][1] = 0 | |
|   *gparams*[ZRS][1][0] = 0 | |
|   *gparams*[ZRS][1][1] = 1 | |

The *perspective*() process shall extract horizontal and vertical perspective elements and shall be defined as follows:

| *perspective*(*gparams*) : | **Ref** |
|---|---|
| $\quad$ *nonzero_perspective_flag* = *read_bool*() | |
| $\quad$ **if** (*nonzero_perspective_flag* == **True**): | |
| $\quad\quad$ *gparams*[PERSP_EXP] = *read_uint*() | |
| $\quad\quad$ *gparams*[PERSPECTIVE][0] = *read_sint*() | |
| $\quad\quad$ *gparams*[PERSPECTIVE][1] = *read_sint*() | |
| $\quad$ **else**: | |
| $\quad\quad$ *gparams*[PERSP_EXP] = 0 | |
| $\quad\quad$ *gparams*[PERSPECTIVE] = **0** | |

### 11.2.7 Picture prediction mode

The picture prediction mode encodes alternative methods of motion compensation and is present to support future extensions of this specification.

It shall be defined as follows:

| *picture_prediction_mode*() : | **Ref** |
|---|---|
| $\quad$ **state**[PICTURE_PRED_MODE] = *read_uint*() | |

In this specification, **state**[PICTURE_PRED_MODE] shall be 0.

### 11.2.8 Reference picture weight values

Reference picture weight values shall be determined as follows:

| *reference_picture_weights*() : | **Ref** |
|---|---|
| $\quad$ **state**[REFS_WT_PRECISION] = 1 | |
| $\quad$ **state**[REF1_WT] = 1 | |
| $\quad$ **state**[REF2_WT] = 1 | |
| $\quad$ *custom_weights_flag* = *read_bool*() | |
| $\quad$ **if** (*custom_weights_flag* == **True**): | |
| $\quad\quad$ **state**[REFS_WT_PRECISION] = *read_uint*() | |
| $\quad\quad$ **state**[REF1_WT] = *read_sint*() | |
| $\quad\quad$ **if** (*num_refs*() == 2): | |
| $\quad\quad\quad$ **state**[REF2_WT] = *read_sint*() | |

**Note:** For bi-directional prediction modes, reference 1 data will be weighted by

$$\frac{\textbf{state}[\text{REF1\_WT}]}{2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]}}$$

and reference 2 data by

$$\frac{\textbf{state}[\text{REF2\_WT}]}{2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]}}$$

(see Section 15.8.5).

The picture weights are signed integers and may be negative. In addition, they may not sum to $2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]}$, to accomodate fade prediction.

## 11.3 Wavelet transform data

The wavelet transform syntax shall provide metadata determining the wavelet transform parameters (including filter type, transform depth, and codeblock or slice structures) together with the transformed wavelet coefficients.

The wavelet transform process for parsing transform metadata and coefficients shall be defined as follows:

| *wavelet_transform*() : | **Ref** |
|---|---|
| **state**[ZERO_RESIDUAL] = **False** | |
| **if** (*is_inter*()): | 9.6.1 |
|     **state**[ZERO_RESIDUAL] = *read_bool*() | |
| **if** (**state**[ZERO_RESIDUAL] == **False**): | |
|     *transform_parameters*() | 11.3.1 |
|     *byte_align*() | |
|     *transform_data*() | 13 |

Parsing (unpacking) the wavelet transform data shall be as defined in Section 13.

Decoding the transformed wavelet transform data to produce decoded pictures shall be as defined in Section 15.

If **state**[ZERO_RESIDUAL] = **True** then all component pixels shall be set to zero (Section 15.1).

### 11.3.1 Transform parameters

The wavelet transform parameters shall define the metadata required to configure the inverse wavelet transform for both the low delay and core syntax.

The *transform_parameters*() process shall be defined as follows:

| *transform_parameters*() : | **Ref** |
|---|---|
| **state**[WAVELET_INDEX] = *read_uint*() | 11.3.1.1 |
| **state**[DWT_DEPTH] = *read_uint*() | 11.3.2 |
| **if** (*is_low_delay*() == **False**): | |
|     *codeblock_parameters*() | 11.3.3 |
| **else**: | |
|     *slice_parameters*() | 11.3.4 |
|     *quant_matrix*() | 11.3.5 |

#### 11.3.1.1 Wavelet filters

The wavelet filter parameter shall define the wavelet filter used by the Dirac stream. T The value of **state**[WAVELET_INDEX] shall lie in the range 0 to 6 with values as defined in Table 11.2:

| **state**[WAVELET_INDEX] | **Filter** |
|---|---|
| 0 | Deslauriers-Dubuc (9,7) |
| 1 | LeGall (5,3) |
| 2 | Deslauriers-Dubuc (13,7) |
| 3 | Haar with no shift |
| 4 | Haar with single shift per level |
| 5 | Fidelity filter |
| 6 | Daubechies (9,7) integer approximation |

Table 11.2: Wavelet filter presets

The implementation of the chosen wavelet filter shall be as defined in Section 15.6.3.

**Note:** For consistency, the filter nomenclature $(m, n)$ refers to the length of the analysis low-pass and high-pass filters in the conventional prefiltering (i.e. before subsampling) model of wavelet filtering. They do not reflect the length of lifting filters, which operate in the subsampled domain: see Section 15.6.3. Deslauriers-Dubuc filters are normally referred to in terms of the number of vanishing moments of their synthesis filters, so the (9,7) and (13,7) filters may be referred to in the literature as (2,2) and (4,2) filters respectively.

### 11.3.2 Transform depth

The transform depth parameter shall determine the number of stages in the wavelet transform.that the vertical and horizontal wavelet filters are applied.

Note: The transform depth determines the number of subbands and the the dimensions of the subband data array (Section 13.1.1).

### 11.3.3 Codeblock parameters (core syntax only)

In the core syntax only, each subband may be partitioned into a number of code blocks.

The process for extracting codeblock parameters shall be as follows:

| *codeblock_parameters*() : | **Ref** |
|---|---|
|   **state**[CODEBLOCK_MODE] = 0 | |
|   **for** *level* = 0 **to state**[DWT_DEPTH]: | |
|     **state**[CODEBLOCKS_X][*level*] = 1 | |
|     **state**[CODEBLOCKS_Y][*level*] = 1 | |
|   *spatial_partition_flag* = *read_bool*() | |
|   **if** (*spatial_partition_flag* == **True**): | |
|     **for** *level* = 0 **to state**[DWT_DEPTH]: | |
|       **state**[CODEBLOCKS_X][*level*] = *read_uint*() | |
|       **state**[CODEBLOCKS_Y][*level*] = *read_uint*() | |
|     **state**[CODEBLOCK_MODE] = *read_uint*() | |

The presence of codeblocks in subbands shall be indicated by setting *spatial_partition_flag* to **True**; otherwise it shall be **False**.

The number of codeblocks to be used for subbands at each transform depth level shall be encoded in **state**[CODEBLOCKS_Y][*level*] and **state**[CODEBLOCKS_X][*level*] for vertical and horizontal axes respectively.

The codeblock mode is encoded in **state**[CODEBLOCK_MODE], which shall have value 0 or 1, with meanings as defined in Table 11.3.

| **state**[CODEBLOCK_MODE] | **Description** |
|---|---|
| 0 | Single quantiser per subband, used for all codeblocks |
| 1 | Multiple Quantiser per subband, one for each codeblock |

Table 11.3: Codeblock modes

The operation of subband codeblock decoding shall be as defined in Section 13.4.3.

### 11.3.4 Slice coding parameters (low delay syntax only)

This slice parameters process shall be defined as follows:

| *slice_parameters*() : | **Ref** |
|---|---|
|   **state**[SLICES_X] = *read_uint*() | |
|   **state**[SLICES_Y] = *read_uint*() | |
|   **state**[SLICE_BYTES_NUMER] = *read_uint*() | |
|   **state**[SLICE_BYTES_DENOM] = *read_uint*() | |

### 11.3.5 Quantisation matrices (low-delay syntax)

The quantization matrix shall be used to modify the slice quantizer for each subband in a slice. The quantization matrix shall be encoded in the **state**[QMATRIX] decoder variable.

The *quant_matrix*() process shall be defined as follows:

```
quant_matrix() :                                              Ref
  custom_quant_matrix = read_bool()
  if (custom_quant_matrix == True):
    state[QMATRIX][0][LL] = read_uint()
    for level = 1 to state[DWT_DEPTH]:
      state[QMATRIX][level][HL] = read_uint()
      state[QMATRIX][level][LH] = read_uint()
      state[QMATRIX][level][HH] = read_uint()
  else:
    set_quant_matrix()
```

If **state**[DWT_DEPTH] $> 4$ then *custom_quant_matrix* shall be **True**.

If **state**[DWT_DEPTH] $\leq 4$, then custom quantization matrices may still be transmitted, for example to apply a different degree of perceptual weighting (see Annex E.2).

The function *set_quant_matrix*() shall set the quantization matrix based on the wavelet filter as per Annex E.1. These are unweighted matrices, whose values merely compensate for the differential power gain of the different subband ?lters. For perceptual weighting a custom quantisation matrix must be used.

# 12 Block motion data syntax

This section defines the operation of the *block_motion_data*() process for extracting block motion data from the Dirac stream.

Block motion data is aggregated into *superblocks*, consisting of a 4x4 array of blocks. The number of superblocks horizontally and vertically shall be determined so that there are sufficient superblocks to cover the picture area. Superblocks may overlap the right and bottom edge of the picture.

**Note:**

1. Since superblocks may overlap the right and bottom edge of the picture, blocks in such superblocks may also overlap the edges or even fall outside the picture area altogether. Motion data for blocks which fall outside the picture area is still decoded, but will not be used for motion compensation (Section 15.8).
2. Unlike macroblocks in MPEG standards, a superblock does not encapsulate all data within a given area of the picture. It is merely an aggregation device for motion data, and for this reason a different nomenclature has been adopted.

## 12.1 Prediction modes and splitting modes

### 12.1.1 Prediction modes

Two types of prediction mode shall be defined: a reference prediction mode, indicating which references are to be used for motion compensation, and a global motion mode flag, indicating how prediction is to be performed (using global motion or block motion for a given block).

Four reference prediction modes shall be defined and shall be denoted by integer constant values:

1. INTRA shall denote value 0, and shall indicate that DC values for a block shall be decoded and that no motion vectors shall be decoded.
2. REF1ONLY shall denote value 1 and shall indicate that a motion vector for the first reference picture shall be decoded, but no motion vector for the second reference picture shall be decoded.
3. REF2ONLY shall denote value 2 and shall indicate that a motion vector for the second reference picture shall be decoded, but no motion vector for the first reference picture shall be decoded.
4. REF1AND2 shall denote value 3 and shall indicate that motion vectors for both the first and second reference picture shall be decoded.

In addition, where global motion is used for a picture (i.e. **state**[USING_GLOBAL] is set), a global motion mode flag shall be encoded for each block. If **True**, global motion compensation shall be used for this block, and no block motion vectors or DC values shall be encoded. If **False**, block motion compensation shall be employed and one or more motion vectors shall be encoded.

### 12.1.2 Splitting modes

Block motion data shall be aggregated into superblocks, consisting of a 4x4 array of blocks, for each block motion data element.

Three superblock splitting levels shall be defined, numbered 0, 1, and 2.

When level 0 splitting is used, if a block motion data element is present for that superblock, only one value shall be coded. This value shall be applied to all blocks within the superblock.

When level 1 splitting is used, at most 4 values shall be coded for each block motion data element. Where present, each of these values shall be applied to the blocks in within the corresponding 2x2 sub-array of blocks within the superblock.

When level 2 splitting is used, if a block motion data element is present for that superblock, only four values shall be coded. Each of these values shall be applied to all the four blocks in one of the four 2x2 sub-arrays of blocks within the superblock.

## 12.2 Structure of block motion data arrays

For the purposes of this specification, block motion data shall be stored in the two dimensional array **state**[BLOCK_DATA]. Superblock splitting modes shall be stored in the two dimensional array **state**[SB_SPLIT].

For each block with coordinates $(i, j)$, a block motion data element **state**[BLOCK_DATA][$j$][$i$] shall be defined. It is a map (Section 6.3) and shall consist of up to five elements:

1. A motion vector for reference 1, **state**[BLOCK_DATA][$j$][$i$][VECTOR][1], consisting of integral horizontal and vertical elements **state**[BLOCK_DATA][$j$][$i$][VECTOR][1][0] and **state**[BLOCK_DATA][$j$][$i$][VECTOR][1][1].
2. A motion vector for reference 2, **state**[BLOCK_DATA][$j$][$i$][VECTOR][2], consisting of integral horizontal and vertical elements **state**[BLOCK_DATA][$j$][$i$][VECTOR][2][0] and **state**[BLOCK_DATA][$j$][$i$][VECTOR][2][1].
3. A set of integral DC values for each component, **state**[BLOCK_DATA][$j$][$i$][DC][$Y$], **state**[BLOCK_DATA][$j$][$i$][DC][$C1$], and **state**[BLOCK_DATA][$j$][$i$][DC][$C2$].
4. A reference prediction mode, **state**[BLOCK_DATA][$j$][$i$][RMODE], taking values INTRA, REF1ONLY, REF2ONLY, or REF1AND2 and indicating which references (if any) are to be used for predicting block $(i, j)$.
5. A global motion mode flag, **state**[BLOCK_DATA][$j$][$i$][GMODE].

### 12.2.1 Block motion data initialisation

This section specifies the operation of the *initialise_motion_data*() process. It shall set the dimensions of the block motion parameter arrays according to the numbers of blocks and superblocks defined in Section motiondatadimensions.

The array **state**[BLOCK_DATA] shall be set to have horizontal dimension **state**[BLOCKS_X] and vertical dimension **state**[BLOCKS_Y].

The array **state**[SB_SPLIT] shall be set to have horizontal dimension **state**[SUPERBLOCKS_X] and vertical dimension **state**[SUPERBLOCKS_Y].

## 12.3 Motion data decoding process

This section defines the *block_motion_data*() process for extracting block motion data elements.

This process depends upon the picture prediction parameters (Section 11.2.1).

Block motion data elements shall be coded differentially with respect to a spatial prediction. The spatial prediction processes for the block motion elements are defined in Section 12.3.6

The decoding process for the block motion data shall consist of:

1. decoding the superblock split modes,
2. decoding the prediction modes in each superblock according to the split mode, and
3. decoding the motion vectors and DC values according to the split mode and the decoded mode for each block.

The motion vector elements are further decomposed into horizontal and vertical components which are encoded as separate parts. The DC values are further decomposed into the the components which are encoded as separate parts.

The coded data for each part (splitting mode, prediction mode, vector component, or DC component values) shall consist of an entropy coded block preceded by a length code.

| $\mathit{block\_motion\_data}()$ : | **Ref** |
|---|---|
| $\quad \mathit{initialise\_motion\_data}()$ | 12.2.1 |
| $\quad \mathit{superblock\_split\_modes}()$ | 12.3.1 |
| $\quad \mathit{prediction\_modes}()$ | 12.3.3 |
| $\quad \mathit{vector\_elements}(1, 0)$ | 12.3.4 |
| $\quad \mathit{vector\_elements}(1, 1)$ | 12.3.4 |
| $\quad$ **if** $(\mathit{num\_refs}() == 2)$: | |
| $\qquad \mathit{vector\_elements}(2, 0)$ | 12.3.4 |
| $\qquad \mathit{vector\_elements}(2, 1)$ | 12.3.4 |
| $\quad \mathit{dc\_values}(Y)$ | 12.3.5 |
| $\quad \mathit{dc\_values}(C1)$ | 12.3.5 |
| $\quad \mathit{dc\_values}(C2)$ | 12.3.5 |

**Note:** The superblock splitting modes determine the number, and location, of prediction mode values to be decoded – there must be one for each 'prediction unit' (block, 2x2 array or blocks, or 4x4 array or blocks) within a superblock. Together, the split mode and the prediction mode determine the number and location of all other motion data parts, which can each then be decoded in parallel. Indeed, by attempting to decode the maximum possible number of prediction residue values for all motion data elements, the first two motion data elements may also be decoded in parallel with the others. Once all residue values are decoded, excess values can be discarded, the location of values determined and actual values reconstructed by prediction. This approach may be particularly valuable in hardware. Decoding may proceed in this way, as the arithmetic decoding engine allows bits to be read beyond the nominal end of an arithmetically-coded chunk by inserting 1s, hence allowing virtual values to be read.

### 12.3.1 Superblock splitting modes

This section defines the decoding of the superblock splitting mode values.

The superblock splitting mode shall determine the number of prediction modes coded for each superblock.

$\mathit{superblock\_split\_modes}()$ process shall be defined as follows:

| $\mathit{superblock\_split\_modes}()$ : | **Ref** |
|---|---|
| $\quad \mathit{length} = \mathit{read\_uint}()$ | |
| $\quad$ **state**[BITS_LEFT] $= 8 * \mathit{length}$ | |
| $\quad \mathit{byte\_align}()$ | |
| $\quad \mathit{ctx\_labels} = [\mathrm{SB\_F1}, \mathrm{SB\_F2}, \mathrm{SB\_DATA}]$ | |
| $\quad \mathit{initialise\_arithmetic\_decoding}(\mathit{ctx\_labels})$ | B.2.2 |
| $\quad$ **for** $\mathit{ysb} = 0$ **to state**[SUPERBLOCKS_Y] $- 1$: | |
| $\qquad$ **for** $\mathit{xsb} = 0$ **to state**[SUPERBLOCKS_X] $- 1$: | |
| $\qquad\quad \mathit{sb\_split\_residual} = \mathit{read\_uinta}(\mathit{sb\_split\_contexts}())$ | 12.3.7.1 |
| $\qquad\quad$ **state**[SB_SPLIT]$[\mathit{ysb}][\mathit{xsb}] = \mathit{sb\_split\_residual}$ | |
| $\qquad\quad$ **state**[SB_SPLIT]$[\mathit{ysb}][\mathit{xsb}] += \mathit{split\_prediction}(\mathit{xsb}, \mathit{ysb})$ | 12.3.6.2 |
| $\qquad\quad$ **state**[SB_SPLIT]$[\mathit{ysb}][\mathit{xsb}] \% = 3$ | |
| $\quad \mathit{flush\_inputb}()$ | A.3.1 |

### 12.3.2 Propagating data between blocks

The superblock splitting mode determines the maximum number of values to be decoded for each block motion data element: 0, 4, or 16. If the splitting mode is 0 or 1 and a value is decoded it applies to all 16 blocks or to one of the 4 2x2 sub-arrays of blocks within the superblock. So that prediction of values shall operate correctly, once decoded a value shall be propagated to all blocks to which it applies.

The $propagate\_data(xtl, ytl, k, idx)$ shall copy decoded block data from the top-left-most block $(xtl, ytl)$ of an array of $k \times k$ blocks, where $k$ shall be 4 if the splitting mode is 0 and $k$ shall be 2 if the splitting mode is 1. It shall be defined as follows:

| $propagate\_data(xtl, ytl, k, label)$ : | **Ref** |
|---|---|
| $\quad$ **for** $y = ytl$ **to** $ytl + k - 1$: | |
| $\quad\quad$ **for** $x = xtl$ **to** $xtl + k - 1$: | |
| $\quad\quad\quad$ **state**[BLOCK_DATA][$y$][$x$][$label$] = **state**[BLOCK_DATA][$ytl$][$xtl$][$label$] | |

### 12.3.3 Block prediction modes

The prediction mode process shall decode global motion and reference prediction modes required for each superblock according to the the superblock splitting mode.: 16 values shall be decoded for split mode 2, 4 values shall be decoded for split mode 1, and 1 value for split mode 0.

For split modes 0 and 1, decoded values shall placed in the top-left corner block of the array (4x4 or 2x2) of blocks to which they apply, and then propagated to the other blocks.

The $prediction\_modes()$ process shall be defined as follows:

| $prediction\_modes()$ : | **Ref** |
|---|---|
| $\quad length = read\_uint()$ | |
| $\quad$ **state**[BITS_LEFT] $= 8 * length$ | |
| $\quad byte\_align()$ | |
| $\quad ctx\_labels =$ [PMODE_REF1, PMODE_REF2, GLOBAL_BLOCK] | |
| $\quad initialise\_arithmetic\_decoding(ctx\_labels)$ | B.2.2 |
| $\quad$ **for** $ysb = 0$ **to** **state**[SUPERBLOCKS_Y] $- 1$: | |
| $\quad\quad$ **for** $xsb = 0$ **to** **state**[SUPERBLOCKS_X] $- 1$: | |
| $\quad\quad\quad block\_count = 2^{\textbf{state}[\text{SB\_SPLIT}][ysb][xsb]}$ | |
| $\quad\quad\quad step = 4//block\_count$ | |
| $\quad\quad\quad$ **for** $q = 0$ **to** $block\_count - 1$: | |
| $\quad\quad\quad\quad$ **for** $p = 0$ **to** $block\_count - 1$: | |
| $\quad\quad\quad\quad\quad block\_ref\_mode(4 * xsb + p * step, 4 * ysb + q * step)$ | 12.3.3.1 |
| $\quad\quad\quad\quad\quad propagate\_data(4 * xsb + p * step, 4 * ysb + q * step, step,$ RMODE) | 12.3.2 |
| $\quad\quad\quad\quad\quad block\_global\_mode(4 * xsb + p * step, 4 * ysb + q * step)$ | 12.3.3.2 |
| $\quad\quad\quad\quad\quad propagate\_data(4 * xsb + p * step, 4 * ysb + q * step, step,$ GMODE) | 12.3.2 |
| $\quad flush\_inputb()$ | A.3.1 |

#### 12.3.3.1 Block prediction mode

The $block\_ref\_mode()$ process shall be defined as follows:

| $block\_ref\_mode(x, y)$ : | **Ref** |
|---|---|
| $\quad$ **state**[BLOCK_DATA][$y$][$x$][RMODE] $= 0$ | |
| $\quad$ **if** ($read\_boola$(PMODE_REF1) $==$ **True**): | |
| $\quad\quad$ **state**[BLOCK_DATA][$y$][$x$][RMODE] $= 1$ | |
| $\quad$ **if** ($num\_refs() == 2$): | |
| $\quad\quad$ **if** ($read\_boola$(PMODE_REF2) $==$ **True**): | |
| $\quad\quad\quad$ **state**[BLOCK_DATA][$y$][$x$][RMODE]$+= 2$ | |
| $\quad$ **state**[BLOCK_DATA][$y$][$x$][RMODE]$\wedge = ref\_mode\_prediction(x, y)$ | 12.3.6.3 |

#### 12.3.3.2 Block global mode

The $block\_global\_mode()$ process shall be defined as follows:

| $block\_global(x, y)$ : | **Ref** |
|---|---|
| $\quad$ **state**[BLOCK_DATA][$y$][$x$][GMODE] $=$ **False** | |
| $\quad$ **if** (**state**[USING_GLOBAL] $==$ **True**): | |
| $\quad\quad$ **if** (**state**[BLOCK_DATA][$y$][$x$][RMODE]$! =$ INTRA): | |
| $\quad\quad\quad$ $block\_global\_residue = read\_boola$(GLOBAL_BLOCK) | |
| $\quad\quad\quad$ **state**[BLOCK_DATA][$y$][$x$][GMODE] $= block\_global\_residue$ | |
| $\quad\quad\quad$ **state**[BLOCK_DATA][$y$][$x$][GMODE]$\wedge = block\_global\_prediction(x, y)$ | 12.3.6.4 |

### 12.3.4 Block motion vector elements

The vector element process shall decode the set of horizontal, or the set of vertical motion vector elements associated with one of the reference pictures.

$vector\_elements()$ process shall be defined as follows:

| $vector\_elements(ref, dirn)$ : | **Ref** |
|---|---|
| $\quad$ $length = read\_uint()$ | |
| $\quad$ **state**[BITS_LEFT] $= 8 * length$ | |
| $\quad$ $byte\_align()$ | |
| $\quad$ $ctx\_labels =$ [VECTOR_F1, VECTOR_F2, VECTOR_F3, VECTOR_F4, VECTOR_F5+, VECTOR_DATA, VECTOR_SIGN] | |
| $\quad$ $initialise\_arithmetic\_decoding(ctx\_labels)$ | B.2.2 |
| $\quad$ **for** $ysb = 0$ **to** **state**[SUPERBLOCKS_Y] $- 1$: | |
| $\quad\quad$ **for** $xsb = 0$ **to** **state**[SUPERBLOCKS_X] $- 1$: | |
| $\quad\quad\quad$ $block\_count = 2^{\mathbf{state}[\text{SB\_SPLIT}][ysb][xsb]}$ | |
| $\quad\quad\quad$ $step = 4//block\_count$ | |
| $\quad\quad\quad$ **for** $q = 0$ **to** $block\_count - 1$: | |
| $\quad\quad\quad\quad$ **for** $p = 0$ **to** $block\_count - 1$: | |
| $\quad\quad\quad\quad\quad$ $block\_vector(4 * xsb + p * step, 4 * ysb + q * step, ref, dirn)$ | |
| $\quad\quad\quad\quad\quad$ $propagate\_data(4 * xsb + p * step, 4 * ysb + q * step, step, \text{VECTOR})$ | 12.3.2 |
| $\quad$ $flush\_inputb()$ | A.3.1 |

The block vector proces shall decode an individual motion vector element. It shall be defined as follows:

| $block\_vector(x, y, ref, dirn)$ : | Ref |
|---|---|
| **if** (**state**[BLOCK_DATA][$y$][$x$][RMODE][$ref$] == **True**): | |
|   **if** (**state**[BLOCK_DATA][$y$][$x$][GMODE] == **False**): | |
|     $mv\_residual = read\_sinta(mv\_contexts())$ | 12.3.7.2 |
|     **state**[BLOCK_DATA][$y$][$x$][VECTOR][$ref$][$dirn$] $= mv\_residual$ | |
|     **state**[BLOCK_DATA][$y$][$x$][VECTOR][$ref$][$dirn$]$+= mv\_prediction(x, y, ref, dirn)$ | |

### 12.3.5 DC values

The DV value process shall decode the DC values for a intra blocks for a given video component (Y, C1 or C2). It shall be defined as follows:

| $dc\_values(c)$ : | Ref |
|---|---|
| $length = read\_uint()$ | |
| **state**[BITS_LEFT] $= 8 * length$ | |
| $byte\_align()$ | |
| $ctx\_labels =$ [DC_F1, DC_F2+, DC_DATA, DC_SIGN] | |
| $initialise\_arithmetic\_decoding(ctx\_labels)$ | B.2.2 |
| **for** $ysb = 0$ **to state**[SUPERBLOCKS_Y] $- 1$: | |
|   **for** $xsb = 0$ **to state**[SUPERBLOCKS_X] $- 1$: | |
|     $block\_count = 2^{\mathbf{state}[\text{SB\_SPLIT}][ysb][xsb]}$ | |
|     $step = 4//block\_count$ | |
|     **for** $q = 0$ **to** $block\_count - 1$: | |
|       **for** $p = 0$ **to** $block\_count - 1$: | |
|         $block\_dc(4 * xsb + p * step, 4 * ysb + q * step, c)$ | |
|         $propagate\_data(4 * xsb + p * step, 4 * ysb + q * step, step, \text{DC})$ | 12.3.2 |
| $flush\_inputb()$ | A.3.1 |

The block DC process shall decode an individual component DC value. It shall be defined as follows:

| $block\_dc(x, y, c)$ : | Ref |
|---|---|
| **if** (**state**[BLOCK_DATA][$y$][$x$][RMODE] = INTRA): | |
|   $dc\_residual = read\_sinta(dc\_contexts())$ | 12.3.7.3 |
|   **state**[BLOCK_DATA][$y$][$x$][DC][$c$] $= dc\_residual$ | |
|   **state**[BLOCK_DATA][$y$][$x$][DC][$c$]$+= dc\_prediction(x, y, c)$ | 12.3.6.6 |

### 12.3.6 Spatial prediction of motion data elements

#### 12.3.6.1 Prediction apertures

A consistent convention for prediction apertures is used. The nominal prediction aperture for block motion data is defined to be the relevant data to the left, top and top-left of the data element in question (Figure 12.1). For the superblock split mode of the superblock with index $(i, j)$ this means the superblocks with indices $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. For the block motion data itself, the same applies where these indices are *block* indices.

This is the nominal prediction aperture. Not all data elements in this prediction aperture may be available, either because they would require negative indices, or because the data is not available - for example a block to the left of a block with reference mode REF2ONLYmay have reference mode REF1ONLYand so can furnish no contribution for a prediction to the Reference 2 motion vector.

When superblocks have split level 1 or 0, block data shall be propagated (Section 12.3.2) across 4 or 16 blocks so as to furnish a prediction. The effect is illustrated in Figure 12.2.

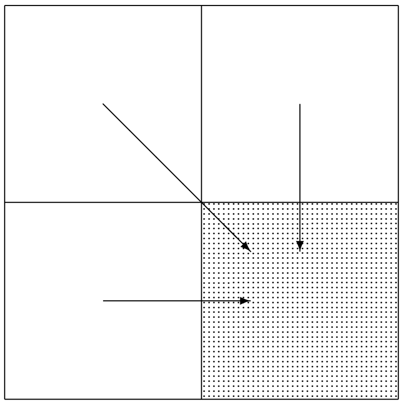

Figure 12.1: Basic prediction aperture

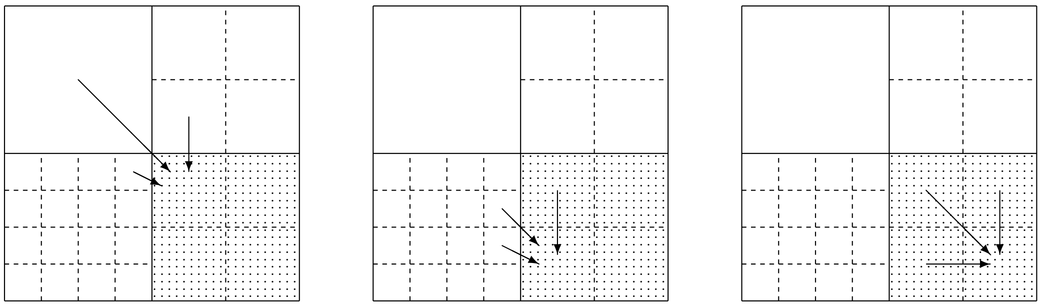

Figure 12.2: Effect of splitting modes on spatial prediction

### 12.3.6.2 Superblock split prediction

*split_prediction* returns the mean of the the neighbouring split values:

| *split_prediction*$(x, y)$ : | **Ref** |
|---|---|
| **if** $(x == 0$ **and** $y == 0)$: | |
|   **return** 0 | |
| **else if** $(y == 0)$: | |
|   **return state**[SB_SPLIT][0][$x - 1$] | |
| **else if** $(x == 0)$: | |
|   **return state**[SB_SPLIT][$y - 1$][0] | |
| **else**: | |
|    **return** mean(**state**[SB_SPLIT][$y - 1$][$x - 1$], **state**[SB_SPLIT][$y$][$x - 1$], **state**[SB_SPLIT][$y - 1$][$x$]) | |

### 12.3.6.3 Block mode prediction

The *ref_mode_prediction*() function shall return a value that represents a majority verdict for the presence of each of the references individually. It shall be defined as follows:

| $ref\_mode\_prediction(x, y)$ : | **Ref** |
|---|---|
| **if** $(x == 0$ **and** $y == 0)$: | |
|   **return** INTRA | |
| **else if** $(y == 0)$: | |
|   **return state**[BLOCK_DATA][0][$x-1$][RMODE] | |
| **else if** $(x == 0)$: | |
|   **return state**[BLOCK_DATA][$y-1$][0][RMODE] | |
| **else**: | |
|   $num\_ref1\_nbrs =$ **state**[BLOCK_DATA][$y-1$][$x$][RMODE]&1 | |
|   $num\_ref1\_nbrs+=$ **state**[BLOCK_DATA][$y-1$][$x-1$][RMODE]&1 | |
|   $num\_ref1\_nbrs+=$ **state**[BLOCK_DATA][$y$][$x-1$][RMODE]&1 | |
|   $pred = num\_ref1\_nbrs//2$ | |
|   $num\_ref2\_nbrs =$ (**state**[BLOCK_DATA][$y-1$][$x$][RMODE] $\gg 1$)&1 | |
|   $num\_ref2\_nbrs+=$ (**state**[BLOCK_DATA][$y-1$][$x-1$][RMODE] $\gg 1$)&1 | |
|   $num\_ref2\_nbrs+=$ (**state**[BLOCK_DATA][$y$][$x-1$][RMODE] $\gg 1$)&1 | |
|   $pred\wedge = (num\_ref2\_nbrs//2) \ll 1$ | |
|   **return** $pred$ | |

#### 12.3.6.4 Block global flag prediction

The $block\_global\_prediction()$ function shall return a value that represents a majority verdict of the neighbouring blocks. It shall be defined as follows:

| $block\_global\_prediction(x, y)$ : | **Ref** |
|---|---|
| **if** $(x == 0$ **and** $y == 0)$: | |
|   **return False** | |
| **else if** $(y == 0)$: | |
|   **return state**[BLOCK_DATA][0][$x-1$][GMODE] | |
| **else if** $(x == 0)$: | |
|   **return state**[BLOCK_DATA][$y-1$][0][GMODE] | |
| **else**: | |
|   **return** majority(**state**[BLOCK_DATA][$y-1$][$x-1$][GMODE], **state**[BLOCK_DATA][$y-1$][$x$][GMODE], **state**[BLOCK_DATA][$y$][$x-1$][GMODE]) | |

#### 12.3.6.5 Motion vector prediction

Motion vectors shall be predicted using the median of available block vectors in the aperture. A vector shall be available for prediction if:

1. its block falls within the picture area,
2. its prediction mode allows it to be defined, and
3. it is not a global motion block.

The $mv\_prediction(x, y, ref, dirn)$ shall return motion values according to the following rules:

**Case 1.** If $x == 0$ and $y == 0$, the value 0 shall be returned.

**Case 2.** If $x > 0$ and $y == 0$ then:

1. If **state**[BLOCK_DATA][0][$x-1$][GMODE] == **False** and (**state**[BLOCK_DATA][0][$x-1$][RMODE]&$ref$)! = 0 then vector element **state**[BLOCK_DATA][0][$x-1$][VECTOR][$ref$][$dirn$] shall be returned,
2. otherwise, 0 shall be returned

**Case 3.** If $x == 0$ and $y > 0$ then:

1. If **state**[BLOCK_DATA][$y-1$][0][GMODE] == **False** and (**state**[BLOCK_DATA][$y-1$][0][RMODE]&$ref$)! = 0 then vector element **state**[BLOCK_DATA][$y-1$][0][VECTOR][$ref$][$dirn$] shall be returned,
2. otherwise, 0 shall be returned

**Case 4.** If both $x > 0$ and $y > 0$ then all 3 blocks in the prediction aperture may potentially contribute to the prediction. Define the set $values = \{\}$. The prediction shall be the median of the available vector elements, as defined in the following pseudocode:

```
...
  if ($x > 0$ and $y > 0$):
    if (state[BLOCK_DATA][$y$][$x-1$][GMODE] == False):
      if ((state[BLOCK_DATA][$y$][$x-1$][RMODE]&$ref$)! = 0):
        $values = values \cup$ {state[BLOCK_DATA][$y$][$x-1$][VECTOR][$ref$][$dirn$]}
    if (state[BLOCK_DATA][$y-1$][$x$][GMODE] == False):
      if ((state[BLOCK_DATA][$y-1$][$x$][RMODE]&$ref$)! = 0):
        $values = values \cup$ {state[BLOCK_DATA][$y-1$][$x$][VECTOR][$ref$][$dirn$]}
    if (state[BLOCK_DATA][$y-1$][$x-1$][GMODE] == False):
      if ((state[BLOCK_DATA][$y-1$][$x-1$][RMODE]&$ref$)! = 0):
        $values = values \cup$ {state[BLOCK_DATA][$y-1$][$x-1$][VECTOR][$ref$][$dirn$]}
    return median($values$)                                              6.4.3
```

(Note that the median of an empty set is zero.)

#### 12.3.6.6 DC value prediction

DC values shall be predicted using the unbiased mean of available values in the prediction aperture.

The process *dc_prediction*$(x, y, c)$ shall return values according to the following rules:

**Case 1.** If $x == 0$ and $y == 0$, 0 shall be returned.

**Case 2.** If $x > 0$ and $y == 0$ then:

1. If **state**[BLOCK_DATA][0][$x-1$][RMODE] == INTRA, **state**[BLOCK_DATA][0][$x-1$][DC][$c$] shall be returned,
2. otherwise, 0 shall be returned.

**Case 3.** If $x == 0$ and $y > 0$ then:

1. If **state**[BLOCK_DATA][$y-1$][0][RMODE] == INTRA, **state**[BLOCK_DATA][$y-1$][0][DC][$c$] shall be returned,
2. otherwise, 0 shall be returned.

**Case 4.** If both $x > 0$ and $y > 0$ then all 3 blocks in the prediction aperture may potentially contribute to the prediction. Define a set $values = \{\}$. The prediction shall be the unbiased mean of available values, as defined in the following pseudocode:

```
...
  if ($x > 0$ and $y > 0$):
    if (state[BLOCK_DATA][$y$][$x-1$][RMODE] == INTRA):
      $values = values \cup \{$state[BLOCK_DATA][$y$][$x-1$][DC][$c$]$\}$
    if (state[BLOCK_DATA][$y-1$][$x$][RMODE] == INTRA):
      $values = values \cup \{$state[BLOCK_DATA][$y-1$][$x$][$ref$][DC][$c$]$\}$
    if (state[BLOCK_DATA][$y-1$][$x-1$][RMODE] == INTRA):
      $values = values \cup \{$state[BLOCK_DATA][$y-1$][$x-1$][$ref$][DC][$c$]$\}$
    if ($values != \{\}$):
      return $pred = \mathrm{mean}(values)$
    else:
      return 0
```

### 12.3.7 Block motion parameter contexts

#### 12.3.7.1 Superblock splitting mode

The *sb_split_contexts*() function shall return a context label map $c$ with the following values:

- $c[FOLLOW] = [\mathrm{SB\_F1}, \mathrm{SB\_F2}]$
- $c[DATA] = \mathrm{SB\_DATA}$

#### 12.3.7.2 Motion vectors

The *mv_contexts*() function shall return a context label map $c$ with the following values:

- $c[FOLLOW] = [\mathrm{VECTOR\_F1}, \mathrm{VECTOR\_F2}, \mathrm{VECTOR\_F3}, \mathrm{VECTOR\_F4}, \mathrm{VECTOR\_F5+}]$
- $c[DATA] = \mathrm{VECTOR\_DATA}$
- $c[SIGN] = \mathrm{VECTOR\_SIGN}$

#### 12.3.7.3 DC values

The *dc_contexts*() function shall return a context label map $c$ with the following values:

- $c[FOLLOW] = [\mathrm{DC\_F1}, \mathrm{DC\_F2+}]$
- $c[DATA] = \mathrm{DC\_DATA}$
- $c[SIGN] = \mathrm{DC\_SIGN}$

# 13 Transform data syntax

This section defines the process for unpacking (parsing, entropy decoding and inverse quantizing) wavelet transform coefficient data from the Dirac stream. Wavelet coefficients shall be signed integer values and shall be extracted using the integer VLC and (optionally) arithmetic decoding functions defined in annex A. The use of arithmetic coding within a picture shall be as signaled by the parse codes defined in Section 9.6. The result of this process shall be a set of fully populated wavelet subband data arrays, as defined in Section 13.1.

Wavelet coefficients shall be packed in the bitstream in one of two possible formats:

1. In the core syntax, coefficients shall be grouped within individual subbands, representing a range of spatial frequencies, from the lowest to the highest. A full set of subbands shall be encoded for each video component in turn.
2. In the low delay syntax, coefficients shall be grouped into slices that represent coefficients pertaining to an area of the picture. Each slice shall contain data for all video components and spatial frequency bands. Unpacking a slice allows an area of picture to be extracted without extracting (or even receiving) the remaining picture data.

The overall process for unpacking transform data shall be as follows:

| $transform\_data()$ : | **Ref** |
|---|---|
| **if** $(is\_low\_delay() ==$ **False**$)$: | |
| **state**[Y_TRANSFORM] $= core\_transform\_data(Y)$ | 13.4 |
| **state**[C1_TRANSFORM] $= core\_transform\_data(C1)$ | 13.4 |
| **state**[C2_TRANSFORM] $= core\_transform\_data(C2)$ | 13.4 |
| **else**: | |
| $low\_delay\_transform\_data()$ | 13.5.1 |

Unpacked wavelet coefficient data shall be stored in the state variables **state**[Y_TRANSFORM], $COneTransform$ and $CTwoTransform$ for the IDWT process (Section 15.6).

## 13.1 Subband data structures

Subband data shall be ordered by level (0, 1, 2, 3 etc) and orientation (LL, HL, LH and HH). In level 0, only the LL orientation shall be available (known also as the DC band); in the other levels only the HL, LH and HH orientations shall be available.

The 0-LL subband shall be presented first in each slice (low delay coding) or component (core syntax coding). Within each subband depth level, the orientation order shall be x-HL, x-LH, x-HH (where x is the transform depth level).

The subbands partition the spatial frequencies by orientation and level so that a four-level subband array is as illustrated in Figure 13.1.

### 13.1.1 Wavelet data initialisation

This section defines the $initialise\_wavelet\_data(comp)$ process, which returns a structure which will contain the wavelet coefficients for the component (Y, C1 or C2) indicated by $comp$.

The coefficient data shall comprise the four dimensional array $coeff\_data$, where individual subbands shall be two-dimensional arrays accessed by level $level$ and orientation $orient$: e.g.

$$band = coeff\_data[level][orient]$$

Valid levels shall be integer values in the range 0 to **state**[DWT_DEPTH] inclusive.

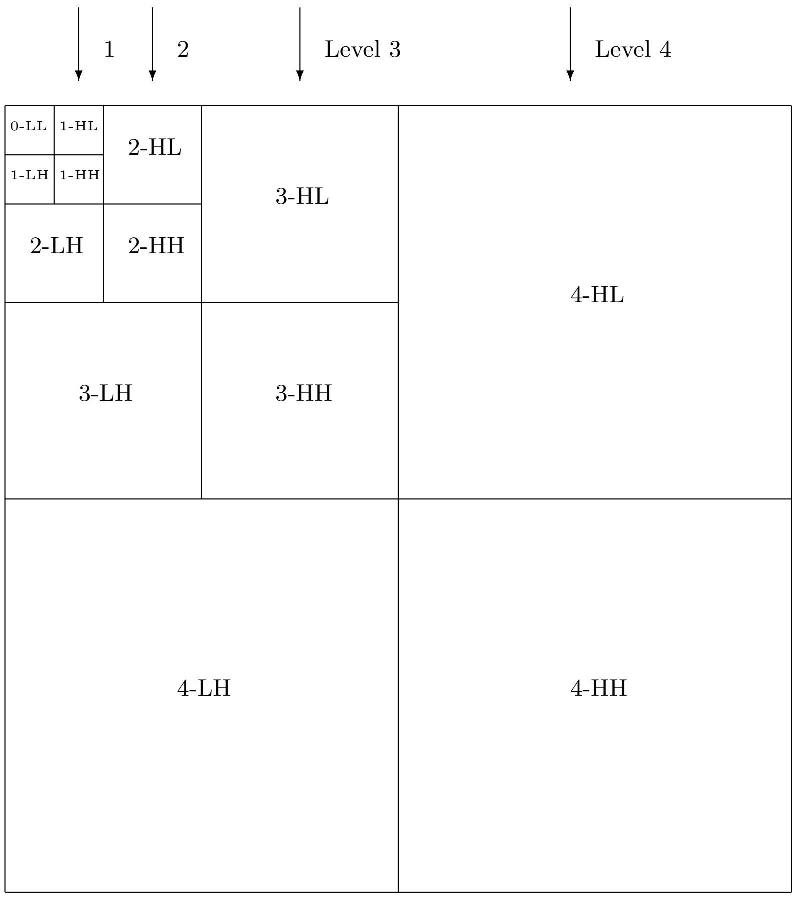

Figure 13.1: Subband decomposition of the spatial frequency domain showing subband numbering, for a 4-level wavelet decomposition

Level 0 shall consists of a single subband with orientation $LL$.

All other levels shall consist of 3 subbands of orientation $HL$, $LH$ and $HH$ in that order within the Dirac stream.

The orientations correspond to either low- or high-pass filtering horizontally and vertically: so e.g. the $LH$ band consists of coefficients derived from horizontal low-pass filtering and vertical high-pass filtering.

Each subband array shall be initialised so that:

$$\begin{aligned}\text{width}(coeff\_data[level][orient]) &= subband\_width(level, comp)\\ \text{height}(coeff\_data[level][orient]) &= subband\_height(level, comp)\end{aligned}$$

as specified in Section 13.1.2. These dimensions correspond to a wavelet transform being performed on a copy of the component data which has been padded (if necessary) so that its dimensions are a multiple of $2^{\textbf{state}[\text{DWT\_DEPTH}]}$.

Individual subband coefficients shall be signed integers accessed by vertical and horizontal coordinates within the subband, e.g.:

$$c = coeff\_data[level][orient][y][x]$$

for coordinates $(x, y)$ such that

$$\begin{aligned}0 \leq x < subband\_width(level, comp)\\ 0 \leq y < subband\_height(level, comp)\end{aligned}$$

### 13.1.2 Subband dimensions

This section defines the values of the $subband\_width(level, comp)$ and $subband\_height(level, comp)$ functions, giving the width and height of subbands at a given level for a given component, and hence the range of subband vertical and horizontal indices.

If $comp == Y$, set

$$\begin{aligned} w &= \textbf{state}[\text{LUMA\_WIDTH}] \\ h &= \textbf{state}[\text{LUMA\_HEIGHT}] \end{aligned}$$

Otherwise, set

$$\begin{aligned} w &= \textbf{state}[\text{CHROMA\_WIDTH}] \\ h &= \textbf{state}[\text{CHROMA\_HEIGHT}] \end{aligned}$$

The padded dimensions of the component shall be defined by:

$$\begin{aligned} scale &= 2^{\textbf{state}[\text{DWT\_DEPTH}]} \\ pw &= scale * ((w + scale - 1)//scale) \\ ph &= scale * ((h + scale - 1)//scale) \end{aligned}$$

If $level == 0$,

$$\begin{aligned} subband\_width(level) &= pw//2^{\textbf{state}[\text{DWT\_DEPTH}]} \\ subband\_height(level) &= ph//2^{\textbf{state}[\text{DWT\_DEPTH}]} \end{aligned}$$

If $level > 0$

$$\begin{aligned} subband\_width(level) &= pw//2^{\textbf{state}[\text{DWT\_DEPTH}]-level+1} \\ subband\_height(level) &= ph//2^{\textbf{state}[\text{DWT\_DEPTH}]-level+1} \end{aligned}$$

**Note:** In encoding, these padded dimensions may be achieved by padding the component data up to the padded dimensions and applying the forward Discrete Wavelet Transform (the inverse of the operations specified in Section 15.6). Any values may be used for the padded data, although the choice will affect wavelet coefficients at the right and bottom edges of the subbands. Good results, in compression terms, may be obtained by using edge extension for intra pictures and zero extension for inter pictures. A poor choice of padding may cause visible artefacts near the bottom and right edges at high levels of compression.

## 13.2 Inverse quantisation

This section defines the operation of inverse quantisation, which scales the dynamic range of unpacked wavelet coefficients according to a pre-determined factor. The inverse quantisation operation is common to both the low-delay and core syntax.

The $inverse\_quant()$ function shall be defined as follows:

| $inverse\_quant(quantised\_coeff, quant\_index)$ : | **Ref** |
|---|---|
| $magnitude = \|quantised\_coeff\|$ | |
| **if** $(magnitude != 0)$: | |
|     $magnitude *= quant\_factor(quant\_index)$ | 13.2.1 |
|     $magnitude += quant\_offset(quant\_index)$ | 13.2.1 |
|     $magnitude += 2$ | |
|     $magnitude = magnitude//4$ | |
| **return** $\text{sign}(quantised\_coeff) * magnitude$ | |

**Note:**

1. Dirac quantisation is an integer approximation of dead-zone quantisation, in which a value is quantised as
$$|x|//qf$$
for $x \geq 0$ or
$$-|x|//qf$$
Since this process involves rounding down, the inverse quantisation process adds an offset to reconstructed values after multiplying by $qf$. This produces a value on average closer to the original value.

2. The pseudocode description separates inverse quantisation from coefficient unpacking. However, since dead-zone quantisation is used, the $inverse\_quant()$ function must compute the magnitude. Hence it is more efficient to first extract the coefficient magnitude, then inverse quantise, and then extract the coefficient sign.

3. In the low delay syntax, the quantisation index is limited to 6 bits, i.e. a maximum value of 63. In the core syntax this limit will also suffice for 8 bit data and a 4-level transform, but the maximum value will in general depend upon the video bit depth, the type of wavelet filter and the transform depth. A value of 127 will account for most practical situations.

### 13.2.1 Quantisation factors and offsets

This section defines the operation of the $quant\_factor()$ and $quant\_offset()$ functions for performing inversion quantisation.

Quantisation factors shall be determined as follows:

```
quant_factor(index) :                                  Ref
  $base = 2^{index//4}$
  if (($index\%4$) == 0):
    return $4 * base$
  else if (($index\%4$) == 1):
    return $(503829 * base + 52958)//105917$
  else if (($index\%4$) == 2):
    return $(665857 * base + 58854)//117708$
  else if (($index\%4$) == 3):
    return $(440253 * base + 32722)//65444$
```

For intra pictures, offsets are approximately 1/2 of the quantisation factors, and for inter pictures they are 3/8 - these mark the reconstruction point within the quantisation interval:

```
quant_offset(index) :                                  Ref
  if ($index == 0$):
    $offset = 1$
  else:
    if ($is\_intra()$):
      if ($index == 1$):
        $offset = 2$
      else:
        $offset = (quant\_factor(index) + 1)//2$
    else:
      $offset = (quant\_factor(index) * 3 + 4)//8$
  return $offset$
```

The value of *index* passed to both functions shall be greater than or equal 0.

**Note:** The quantisation offsets have been selected so as to make inverse quantisation and re-quantisation by the same quantisation factor transparent. This requires that

$$3 \leq quant\_offset + 2 < quant\_factor$$

– hence the special conditions for quantisation indexes 0 and 1.

## 13.3 Intra DC subband prediction

This section defines the operation of the *intra_dc_prediction*(*band*) function for reconstructing values within Intra picture DC subbands using spatial prediction.

This function may be applied once all coefficients within the DC band have been unpacked, although it may be applied progressively to each coefficient as soon as it has been unpacked.

Intra DC values shall be derived by spatial prediction using the mean of the three values to the left, top-left and above a coefficient (where available).

The Intra DC subband prediction process shall be defined as follows:

```
$intra\_dc\_prediction(band)$ :                                                        Ref
  $prediction = 0$
  for $v = 0$ to $\mathrm{height}(band) - 1$:
    for $h = 0$ to $\mathrm{width}(band) - 1$:
      if ($h > 0$ and $v > 0$):
        $prediction = \mathrm{mean}(band[v][h-1], band[v-1][h-1], band[v-1][h])$
      else if ($h > 0$ and $v == 0$):
        $prediction = band[0][h-1]$
      else if ($h == 0$ and $v > 0$):
        $prediction = band[v-1][0]$
      else:
        $prediction = 0$
      $band[v][h] += prediction$
```

## 13.4 Core syntax wavelet coefficient unpacking

This section defines the overall operation of the *core_transform_data*(*comp*) process for unpacking the set of coefficient subbands corresponding to a video component (Y, C1 or C2) of a picture in the core Dirac syntax, according to the conventions set out in Section 13.1.

In the Dirac core syntax, subband data shall be entropy coded. It shall be arranged by level and orientation, from level 0 up to level **state**[DWT_DEPTH]. Coefficients may be VLC or arithmetic coded. Where arithmetic coding is used, the unpacking process for each subband is contingent on data from subbands of the same orientation in the next lower level. This is the *parent* subband; the subband of the same orientation in the next higher level is the *child* subband.

Unpacking an individual subband therefore requires prior unpacking of the parent subband, and of its parent, and so on until level 1 is reached (unpacking level 1 subbands does not depend upon the single level 0 DC band).

**Note:** The data for each subband consists of a subband header and a block of coded coefficient data. The subband header contains a length code giving the number of bytes of the block of coded data. The transform data can therefore be parsed without invoking entropy decoding at all, since the length codes allow a parser to skip from one subband header to the next.

### 13.4.1 Overall process

The overall *core_transform_data*() process shall be defined as follows:

| *core_transform_data*(*comp*) : | **Ref** |
|---|---|
| $\quad$ *coeff_data* = *initialise_wavelet_data*(*comp*) | 13.1.1 |
| $\quad$ *byte_align*() | |
| $\quad$ *subband*(*coeff_data*, 0, *LL*) | 13.4.2 |
| $\quad$ **for** *level* = 1 **to state**[DWT_DEPTH]: | |
| $\quad\quad$ **for each** *orient* **in** *HL*, *LH*, *HH*: | |
| $\quad\quad\quad$ *byte_align*() | |
| $\quad\quad\quad$ *subband*(*coeff_data*, *level*, *orient*) | 13.4.2 |
| $\quad$ **return** *coeff_data* | |

### 13.4.2 Subbands

This section defines the process for unpacking coefficients of a specified level and orientation *orient*.

The overall process shall consist of reading a byte-aligned header for each subband, including a length code for the subsequent arithmetically-coded data. Subband data shall be initialised to 0. If the length code is 0, the subband shall be skipped and all data within it shall remain set to zero.

Intra DC bands are predicted, and so must additionally be reconstructed.

The subband unpacking process shall be defined as follows:

| *subband*(*coeff_data*, *level*, *orient*) : | **Ref** |
|---|---|
| $\quad$ *length* = *read_uint*() | |
| $\quad$ *zero_subband_data*(*coeff_data*[*level*][*orient*]) | 13.4.2.1 |
| $\quad$ **if** (*length* == 0): | |
| $\quad\quad$ *byte_align*() | |
| $\quad$ **else**: | |
| $\quad\quad$ *quant_index* = *read_uint*() | |
| $\quad\quad$ *byte_align*() | |
| $\quad\quad$ *subband_coeffs*(*coeff_data*, *level*, *orient*, *length*, *quant_index*) | 13.4.2.2 |
| $\quad$ **if** (*is_intra*() and *level* == 0): | |
| $\quad\quad$ *intra_dc_prediction*(*coeff_data*[*level*][*orient*]) | 13.3 |

#### 13.4.2.1 Zero subband

The *zero_subband*() process shall sets all coefficients in a given subband to 0.

It shall be defined as follows:

| *zero_subband_data*(*band*) : | **Ref** |
|---|---|
| $\quad$ **for** $y = 0$ **to** height(*band*) − 1: | |
| $\quad\quad$ **for** $x = 0$ **to** width(*band*) − 1: | |
| $\quad\quad\quad$ *band*[*y*][*x*] = 0 | |

#### 13.4.2.2 Non-skipped subbands

Data within subbands may be split into one or more rectangular codeblocks (Figure 13.4.3). Codeblocks shall be scanned in raster order across the subband and coefficients shall be scanned in raster order within each codeblock.

The $subband\_coeffs()$ process shall be defined as follows:

| $subband\_coeffs(coeff\_data, level, orient, length, quant\_index)$ : | **Ref** |
|---|---|
| **state**[BITS_LEFT] $= 8 * length$ | |
| **if** $(using\_ac() ==$ **True**): | |
| $ctx\_labels =$ [SIGN_ZERO, SIGN_POS, SIGN_NEG, ZPZN_F1, ZPNN_F1, ZP_F2, ZP_F3, ZP_F4, ZP_F5, ZP_F6+, NPZN_F1, NPNN_F1, NP_F2, NP_F3, NP_F4, NP_F5, NP_F6+, COEFF_DATA, ZERO_BLOCK, Q_OFFSET_FOLLOW, Q_OFFSET_DATA, Q_OFFSET_SIGN] | |
| $initialise\_arithmetic\_decoding(ctx\_labels)$ | B.2.2 |
| **for** $y = 0$ **to state**[CODEBLOCKS_Y][$level$] $- 1$: | |
| **for** $x = 0$ **to state**[CODEBLOCKS_X][$level$] $- 1$: | |
| $codeblock(coeff\_data, level, orient, x, y, quant\_index)$ | 13.4.3 |
| $flush\_inputb()$ | A.3.1 |

The key to the context labels is explained in Section 13.4.4.4.

### 13.4.3 Subband codeblocks

This section defines the operation of the process:

$codeblock(band, parent, level, orient, cx, cy, quant\_index)$

This process shall unpack coefficients within the codeblock at position $(cx, cy)$. The dimensions of the codeblock shall be as defined in Section 13.4.3.1. The process for unpacking coefficients within the codeblock given these dimensions shall be as defined in Section 13.4.3.2.

#### 13.4.3.1 Codeblock dimensions

Each codeblock shall cover coefficients in the horizontal range $cb\_left$ to $cb\_right - 1$ and in the vertical range $cb\_top$ to $cb\_bottom - 1$ where these values shall be defined by the functions:

$$\begin{aligned}
cb\_left(x, band, level) &= (\text{width}(band) * x)//\textbf{state}[\text{CODEBLOCKS\_X}][level] \\
cb\_right(x, band, level) &= (\text{width}(band) * (x+1))//\textbf{state}[\text{CODEBLOCKS\_X}][level] \\
cb\_top(y, band, level) &= (\text{height}(band) * y)//\textbf{state}[\text{CODEBLOCKS\_Y}][level] \\
cb\_bottom(y, band, level) &= (\text{height}(band) * (y+1))//\textbf{state}[\text{CODEBLOCKS\_Y}][level]
\end{aligned}$$

where $x$ and $y$ are the codeblock coordinates within the subband.

#### 13.4.3.2 Codeblock unpacking loop

The codeblock unpacking process shall be defined as follows:

| $codeblock(coeff\_data, level, orient, cx, cy, quant\_index)$ : | **Ref** |
|---|---|
| $skipped = zero\_flag(level)$ | 13.4.3.3 |
| **if** $(skipped ==$ **False**$)$: | |
|  $band = coeff\_data[level][orient]$ | |
|  $quant\_index += codeblock\_quant\_offset()$ | 13.4.3.4 |
|  **for** $y = cb\_top(cy, band, level)$ **to** $cb\_bottom(cy, band, level) - 1$: | 13.4.3.1 |
|   **for** $x = cb\_left(cx, band, level)$ **to** $cb\_right(cx, band, level) - 1$: | 13.4.3.1 |
|    $coeff\_unpack(coeff\_data, level, orient, x, y, quant\_index)$ | 13.4.4 |

If the codeblock is skipped, then coefficients within that codeblock shall remain zero.

The function $codeblock\_quant\_offset()$ returns a signed value, but the quantiser offset values coded in the stream shall be constrained so that the reconstructed value of $quant\_index$ shall be non-negative.

**Note:** Codeblock quantisers are encoded differentially in the stream, and the value of $quant\_index$ is modified by this function (all variables) are passed by reference). A decoder ought to check that the reconstructed value of $quant\_index$ falls within the bounds it supports.

#### 13.4.3.3 Skipped codeblock flag

The skipped codeblock flag process shall be as follows:

| $zero\_flag(level)$ : | **Ref** |
|---|---|
| $num\_blocks =$ **state**$[\text{CODEBLOCKS\_X}][level] *$ **state**$[\text{CODEBLOCKS\_Y}][level]$ | |
| **if** $(num\_blocks == 1)$: | |
|  **return False** | |
| **else if** $(using\_ac() ==$ **True**$)$: | |
|  **return** $read\_boola(\text{ZERO\_BLOCK})$ | |
| **else**: | |
|  **return** $read\_boolb()$ | |

If the number of codeblocks is 1, then $zero\_flag()$ shall return **False**.

If arithmetic coding is employed, then the zero flag shall be decoded using the context probability indicated by the ZERO_BLOCK label, as defined in Annex B.2.4.

#### 13.4.3.4 Codeblock quantiser offset

The $codeblock\_quant\_offset()$ process shall be defined as follows:

| $codeblock\_quant\_offset()$ : | **Ref** |
|---|---|
| **if** $($**state**$[\text{CODEBLOCK\_MODE}] == 0)$: | |
|  **return** $0$ | |
| **else if** $(using\_ac() ==$ **True**$)$: | |
|  **return** $read\_sinta(quant\_context\_probs())$ | |
| **else**: | |
|  **return** $read\_sintb()$ | |

where $quant\_context\_probs()$ shall return the context probability label set:

$$\{[\text{Q\_OFFSET\_FOLLOW}], \text{Q\_OFFSET\_DATA}, \text{Q\_OFFSET\_SIGN}\}$$

### 13.4.4 Subband coefficients

This section describes the operation of the $coeff\_unpack(coeff\_data, level, orient, quant\_index, x, y)$ process for unpacking an individual coefficient in position $(x, y)$ in the subband $coeff\_data[level][orient]$.

Unpacking a coefficient shall make use of entropy decoding, inverse quantisation and, in the case of DC (level 0) bands of Intra pictures, neighbourhood prediction.

Arithmetic coding uses a highly compact set of contexts, with magnitudes contextualised on whether parent values and neighbouring values are zero or non-zero. See Annex A.4 for a definition of the Dirac arithmetic decoder.

The process for coefficient unpacking shall comprise up to four stages:

1. (for arithmetic coding only) determination of the magnitude context, based on whether the parent or neighbouring values are zero,
2. (for arithmetic coding only) determination of the sign context, based on the predicted sign value,
3. entropy decoding of the quantized coefficient value, and
4. inverse quantization of the quantized value.

The $coeff\_unpack()$ process shall be defined as follows:

| $coeff\_unpack(coeff\_data, level, orient, quant\_index, x, y)$ : | **Ref** |
|---|---|
| **if** $(using\_ac() ==$ **True**$)$: | |
|     $parent = zero\_parent(coeff\_data, level, orient, x, y)$ | 13.4.4.1 |
|     $nhood = zero\_nhood(coeff\_data[level][orient], x, y)$ | 13.4.4.2 |
|     $sign\_pred = sign\_predict(coeff\_data[level][orient], orient, x, y)$ | 13.4.4.3 |
|     $context\_prob\_set = select\_coeff\_ctxs(nhood, parent, sign\_pred)$ | 13.4.4.4 |
|     $quant\_coeff = read\_sinta(context\_prob\_set)$ | |
| **else**: | |
|     $quant\_coeff = read\_sintb()$ | |
| $coeff\_data[level][orient][y][x] = inverse\_quant(quant\_coeff, quant\_index)$ | 13.2 |

#### 13.4.4.1 Zero parent

The function $zero\_parent(coeff\_data, level, orient, v, h)$ shall return a boolean flag indicating whether the parent value of a coefficient in a subband is zero. The parent coefficient shall be the co-located coefficient in the parent subband, if there is one. There is deemed to be a parent if $level \geq 2$. If a parent coefficient does not exist, **True** shall be returned.

Note: Levels 0 and 1 have the same number of coefficients. Thus the first level that can be used as a parent is level 1, with level 2 as its child.

The parent value shall be determined as follows:

| $zero\_parent(data, level, orient, x, y)$ : | **Ref** |
|---|---|
| **if** $(level >= 2)$: | |
|     $parent = data[level - 1][orient][y//2][x//2]$ | |
| **else**: | |
|     $parent = 0$ | |
| **return** $parent == 0$ | |

#### 13.4.4.2 Zero neighbourhood

The $zero\_nhood()$ function shall return a boolean flag indicating whether the neighbouring values of a given subband coefficient are all zero.

The zero neighbourhood value shall be determined as follows:

| $zero\_nhood(band, x, y)$ : | **Ref** |
|---|---|
| **if** $(y > 0$ and $x > 0)$: | |
| $\quad$ **if** $((band[y-1][x-1]! = 0$ or $band[y][x-1]! = 0)$ or $band[y-1][x]! = 0)$: | |
| $\quad\quad$ **return False** | |
| **else if** $(y > 0$ and $x == 0)$: | |
| $\quad$ **if** $(band[y-1][0]! = 0)$: | |
| $\quad\quad$ **return False** | |
| **else if** $(y == 0$ and $x > 0)$: | |
| $\quad$ **if** $(band[0][x-1]! = 0)$: | |
| $\quad\quad$ **return False** | |
| **return True** | |

### 13.4.4.3 Sign prediction

The *sign_predict*() function shall return a prediction for the sign of the current pixel.

Correlation within subbands depends upon orientation, and so this is taken into account in forming the prediction.

For vertically-oriented (HL) bands, the predictor shall be the sign of the coefficient above the current coefficient; for horizontally-oriented (LH) bands, the predictor shall be the sign of the coefficient to the left.

The predictions shall be used only for the conditioning of the sign contexts.

The sign prediction value shall be determined as follows:

| $sign\_predict(band, orient, x, y)$ : | **Ref** |
|---|---|
| **if** $(orient == HL$ and $y == 0)$: | |
| $\quad$ **return** 0 | |
| **else if** $(orient == HL$ and $y > 0)$: | |
| $\quad$ **return** sign$(band[y-1][x])$ | |
| **else if** $(orient == LH$ and $x == 0)$: | |
| $\quad$ **return** 0 | |
| **else if** $(orient == LH$ and $x > 0)$: | |
| $\quad$ **return** sign$(band[y][x-1])$ | |
| **else**: | |
| $\quad$ **return** 0 | |

### 13.4.4.4 Coefficient context selection

This section defines the coefficient context selection function which shall return a map of context probability labels for decoding a coefficient value. Context probabilities shall be as defined in Annex A.4.1.

The map $m$ returned shall comprise three elements, accessed by the labels FOLLOW, DATA, and SIGN, where:

- $m$[FOLLOW] is an array of labels
- $m$[DATA] is a label
- $m$[SIGN] is a label

The elements of the map returned by

$$select\_coeff\_contexts(zero\_nhood, parent, sign\_pred)$$

shall be as defined in Table 13.1, for values of *zero_parent*, *zero_nhood* and *sign_pred*, where the context labels (e.g. ZPZN_F1, ZP_F2, COEFF_DATAand SIGN_ZERO) correspond to context probabilities (i.e. 16 bit unsigned integers) stored in the decoded state as defined in annex A.4.1.

The three columns on the left of table 13.1 represent the three inputs to the coefficient context selection function. The output of the function is a map with three elements. These elements are accessed by the labels FOLLOW, DATA and SIGN. The values of the three elements of the map are defined in the rightmost column for each set of inputs.

**Note:** The follow context probability sets are arrays indexed from zero as per Annex A.4.3.1. Note also that the parent values affect the context of all follow bits, and that neighbour values only affect the context of the first follow bit. A common data context probability is used for all coefficients.

Key to interpretation of the label names:

**ZP** zero parent

**NP** non-zero parent

**ZN** zero neighbour

**NN** non-zerro neighbour

**Fn** follow bit, bin N (n+ means bin n and higher)

| *zero_parent* | *zero_nhood* | *sign_pred* | **Context map** | |
|---|---|---|---|---|
| **True** | **True** | 0 | FOLLOW | [ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_ZERO |
| **True** | **True** | $< 0$ | FOLLOW | [ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_NEG |
| **True** | **True** | $> 0$ | FOLLOW | [ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_POS |
| **True** | **False** | 0 | FOLLOW | [ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_ZERO |
| **True** | **False** | $< 0$ | FOLLOW | [ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_NEG |
| **True** | **False** | $> 0$ | FOLLOW | [ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_POS |
| **False** | **True** | 0 | FOLLOW | [NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_ZERO |
| **False** | **True** | $< 0$ | FOLLOW | [NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_NEG |
| **False** | **True** | $> 0$ | FOLLOW | [NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_POS |
| **False** | **False** | 0 | FOLLOW | [NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_ZERO |
| **False** | **False** | $< 0$ | FOLLOW | [NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_NEG |
| **False** | **False** | $> 0$ | FOLLOW | [NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+] |
| | | | DATA | COEFF_DATA |
| | | | SIGN | SIGN_POS |

Table 13.1: Subband coefficient context sets

## 13.5 Low delay wavelet coefficient unpacking

This section defines the stream syntax that shall be used for low delay profiles. Only the syntax and parsing operations are defined here; picture decoding operations are defined in Section 15.

In low delay operation, the Dirac syntax shall partition the wavelet coefficients into a number of slices, from all subbands, corresponding to localized areas of the picture

A slice shall meet the following requirements:

1. A single quantizer, weighted for each subband by a quantization matrix, shall be used for quantization of the coefficients in each slice.
2. All wavelet coefficients shall be entropy-coded using variable-length coding alone. Arithmetic coding shall not be used.
3. The number of bytes used per slice shall be the same, to within one byte, for each slice in a picture.
4. Each picture may change the slice parameters within the picture by setting the relevant wavelet transform parameters (Section 11.3.4).

**Note:**

1. The slice structure implies that in practice incremental picture decoding can be easily achieved without accumulating an entire picture data set, yielding a decoding delay proportional to the height of the slices. (The actual achievable delay may be more than one slice height due to vertical filtering delay).
2. Using a fixed number of bits per slice does impact on compression efficiency but simplifies both encoder and decoder hardware, and assists a chain of multiple encoders and decoders using the same slice parameters in producing identical coding decisions and hence no cascading loss. These factors are of great significance in a professional environment.

### 13.5.1 Overall process

The low delay transform data process shall be defined as follows:

| $low\_delay\_transform\_data()()$ : | **Ref** |
|---|---|
| **state**[Y_TRANSFORM] $= initialise\_wavelet\_data(Y)$ | 13.1.1 |
| **state**[C1_TRANSFORM] $= initialise\_wavelet\_data(C1)$ | 13.1.1 |
| **state**[C2_TRANSFORM] $= initialise\_wavelet\_data(C2)$ | 13.1.1 |
| **for** $sy = 0$ **to state**[SLICES_Y] $- 1$: | |
|     **for** $sx = 0$ **to state**[SLICES_X] $- 1$: | |
|         $slice(sx, sy)$ | 13.5.2 |
| $intra\_dc\_prediction($**state**[Y_TRANSFORM][0][$LL$]$)$ | 13.3 |
| $intra\_dc\_prediction($**state**[C1_TRANSFORM][0][$LL$]$)$ | 13.3 |
| $intra\_dc\_prediction($**state**[C2_TRANSFORM][0][$LL$]$)$ | 13.3 |

DC values at the top and left edges of a slice shall be predicted from DC values in previously decoded slices, which must therefore be retained.

### 13.5.2 Slices

This section defines the operation of the $slice(sx, sy)$ process for unpacking coefficients within the slice with coordinates $(sx, sy)$.

Each slice shall contain the relevant coefficients from all subbands and for all components. Luma data shall be unpacked rst, and shall be followed by the chroma data, in which the chroma component coefficients shall be interleaved. A length code shall allow the luma and chroma coefficients each to be terminated early, with remaining values set to zero by means of bounded read operations.

The overall slice unpacking process shall be defined as follows:

| $slice(sx, sy)$ : | **Ref** |
|---|---|
| $slice\_bits\_left = 8 * slice\_bytes(sx, sy)$ | 13.5.3 |
| $qindex = read\_nbits(7)$ | |
| $slice\_bits\_left -= 7$ | |
| $slice\_quantisers(qindex)$ | 13.5.4 |
| $length\_bits = \text{intlog}_2(8 * slice\_bytes(sx, sy) - 7)$ | |
| $slice\_y\_length = read\_nbits(length\_bits)$ | |
| $slice\_bits\_left -= length\_bits$ | |
| **state**[BITS_LEFT] $= slice\_y\_length$ | |
| $luma\_slice\_band(0, LL, sx, sy)$ | 13.5.5.2 |
| **for** $level = 1$ **to state**[DWT_DEPTH]: | |
|     **for each** $orient$ **in** $HL, LH, HH$: | |
|         $luma\_slice\_band(level, orient, sx, sy)$ | 13.5.5.2 |
| $flush\_inputb()$ | A.3.1 |
| $slice\_bits\_left -= slice\_y\_length$ | |
| **state**[BITS_LEFT] $= slice\_bits\_left$ | |
| $chroma\_slice\_band(0, LL, sx, sy)$ | 13.5.5.3 |
| **for** $level = 1$ **to state**[DWT_DEPTH]: | |
|     **for each** $orient$ **in** $HL, LH, HH$: | |
|         $chroma\_slice\_band(level, orient, sx, sy)$ | 13.5.5.3 |
| $flush\_inputb()$ | A.3.1 |

$slice\_y\_length$ shall satisfy the following condition:

$$slice\_y\_length = 8 * slice\_bytes(sx, sy) - 7 - length\_bits$$

**Note:** Slice decoding makes use of bounded read functions, which return 1 when **state**[BITS_LEFT] is zero. This means that remaining coefficients are set to 0, since a solitary 1 is the VLC for 0. The logic of slice decoding applies this twice in each slice: once for luma coefficients, initializing the bits left to $slice\_y\_length$, and a second time to the chroma coefficients, initializing to the number of bits remaining.

### 13.5.3 Determining the number of bytes in a slice

The $slice\_bytes(sx, sy)$ shall be defined as follows:

| $slice\_bytes(sx, sy)$ : | **Ref** |
|---|---|
| $slice\_num = sy *$ **state**[SLICES_X] $+ sx$ | |
| $bytes = ((slice\_num + 1) *$ **state**[SLICE_BYTES_NUMER])// **state**[SLICE_BYTES_DENOM] | |
| $bytes -= ((slice\_num) *$ **state**[SLICE_BYTES_NUMER])// **state**[SLICE_BYTES_DENOM] | |
| **return** $bytes$ | |

**Note:** This function produces an integer value which will on average be the ratio of **state**[SLICE_BYTES_NUMER] to **state**[SLICE_BYTES_DENOM]. In many applications this ratio will not be an integer number, and the number of bytes will vary from slice to slice by one byte from time to time. This allows the low delay syntax to support any compression ratio exactly.

### 13.5.4 Setting slice quantisers

This section defines how quantisers for individual subbands are determined from the quantisation matrix and the quantisation index. The $slice\_quantisers()$ function shall be defined as follows:

| $slice\_quantisers(qindex)$ : | **Ref** |
|---|---|
| **state**[QUANTISER][0][$LL$] $= \max(qindex -$ **state**[QMATRIX][0][$LL$]$, 0)$ | |
| **for** $level = 1$ **to state**[DWT_DEPTH]: | |
|   **for each** $orient$ **in** $HL, LH, HH$: | |
|     $qval = \max(qindex -$ **state**[QMATRIX][$level$][$orient$]$, 0)$ | |
|     **state**[QUANTISER][$level$][$orient$] $= qval$ | |

**Note:** The non-negative quantisation matrix values are subtracted from the slice quantiser value, and so a higher value in the quantisation matrix represents a lower quantisation index and thus a lower degree of quantisation. The quantisation index value is also clipped to 0 so that it is non-negative. This ensures that as many subbands as possible within the slice can be coded losslessly. The quantisation matrix values are set as part of decoding the transform parameters (Section 11.3.5).

### 13.5.5 Slice subbands

This section defines the operation of the $luma\_slice\_band(level, orient, sx, sy)$ for unpacking individual luma slice subbands, and $chroma\_slice\_band(level, orient, sx, sy)$. for unpacking individual chroma slice subbands.

#### 13.5.5.1 Slice subband dimensions

The rectangular set of coefficients covered by a slice component ($Y$, $C1$ and $C2$) is demarcated by the values $slice\_left, slice\_right, slice\_top, slice\_bottom$, defined as fractions of the subband dimensions (Section 13.1.2) by the following functions:

$$\begin{aligned}
slice\_left(sx, level, c) &= (subband\_width(level, c) * sx)//\textbf{state}[\text{SLICES\_X}] \\
slice\_right(sx, level, c) &= (subband\_width(level, c) * (sx + 1))//\textbf{state}[\text{SLICES\_X}] \\
slice\_top(sy, level, c) &= (subband\_height(level, c) * sy)//\textbf{state}[\text{SLICES\_Y}] \\
slice\_bottom(sy, level, c) &= (subband\_height(level, c) * (sy + 1))//\textbf{state}[\text{SLICES\_Y}]
\end{aligned}$$

where $c = Y$ for luma coefficients, and $C1$ or $C2$ for chroma coefficients.

**Note:**

- The slice subband area formulae correspond to the codeblock area formulae for the core syntax (Section 13.4.3.1).
- Slice subbands may change dimension by 1 from one slice to another if **state**[SLICES_X] or **state**[SLICES_Y] do not divide the horizontal or vertical dimensions exactly.
- Chroma slice subbands might not have exactly the scaled dimensions of the luma slice subband, since the **state**[SLICES_X] and $SlicesY$ values may exactly divide luma dimensions but not chroma dimensions; and chroma components may receive different padding, depending on the transform depth.
- These issues may be easily avoided in particular applications by choosing suitable values for the transform depth and **state**[SLICES_X] and $SlicesY$.

#### 13.5.5.2 Luma slice subband data

The process for unpacking luma slice coefficients shall be defined as follows:

| *luma_slice_band*(*level*, *orient*, *sx*, *sy*) : | **Ref** |
|---|---|
| **for** $y = slice\_top(sy, level, Y)$ **to** $slice\_bottom(sy, level, Y) - 1$: | 13.5.5.1 |
|   **for** $x = slice\_left(sx, level, Y)$ **to** $slice\_right(sx, level, Y) - 1$: | 13.5.5.1 |
|     $val = read\_sintb()$ | A.3.3 |
|     $q =$ **state**[QUANTISER][$Y$][$level$][$orient$] | |
|     **state**[Y_TRANSFORM][$level$][$orient$][$y$][$x$] $= inverse\_quant(val, q)$ | |

### 13.5.5.3 Chroma slice subband data

Chroma slice subband coefficients shall follow luma coefficients within each slice. The two chroma components shall be interleaved coefficient-by-coefficient. The process for unpacking chroma slice coefficients shall be defined as follows:

| *luma_slice_band*(*level*, *orient*, *sx*, *sy*) : | **Ref** |
|---|---|
| **for** $y = slice\_top(sy, level, C1)$ **to** $slice\_bottom(sy, level, C1) - 1$: | 13.5.5.1 |
|   **for** $x = slice\_left(sx, level, C1)$ **to** $slice\_right(sx, level, C1) - 1$: | 13.5.5.1 |
|     $q =$ **state**[QUANTISER][$level$][$orient$] | |
|     $val = read\_sintb()$ | A.3.3 |
|     **state**[C1_TRANSFORM][$level$][$orient$] $= inverse\_quant(val, q)$ | 13.2 |
|     $val = read\_sintb()$ | A.3.3 |
|     **state**[C2_TRANSFORM][$level$][$orient$] $= inverse\_quant(val, q2)$ | 13.2 |

## 14 Sequence decoding (Informative)

There is no one unique way of describing a Dirac decoder. However the pseudocode below is illustrative code for a sample decoder. It emphasizes which parts of the decoding process generate decoded output data. Note that the potential presence of padding or auxiliary data is ignored for clarity.

```
decode_sequence() :                                                      Ref
    state = {}
    decoded_pictures = {}
    state[REF_PICTURES] = {}
    parse_info()                                                         9.6
    video_params = sequence_header()                                     10
    parse_info()                                                         9.6
    while (is_end_of_sequence() == False):                               9.6.1
        if (is_seq_header() == True):                                    9.6.1
            video_params = sequence_header()                             10
        else if (is_picture() == True):                                  9.6.1
            picture_parse()                                              11.1
            decoded_pictures[state[PICTURE_NUM]] = picture_decode()      15
        parse_info()                                                     9.6
    return {video_params, decoded_pictures}
```

The process returns the video parameters, consisting of the essential metadata required for display and interpretation of the video data, and the array of decoded pictures. Each decoded picture contains the three video component data arrays together with a picture number.

The pseudocode describes the decoding process. Decoding starts by clearing the decoder state and the decoder output. Thus video sequences may be decoded as independent entities. The first data extracted from the Dirac stream is parse information. The parse info header indicates what type of data unit follows, and this information is stored in the decoder state. The decoder continues to read pairs of data unit and parse info headers until the end of the sequence is reached. The end of sequence is indicated by data in the final parse info header. If a data unit is a sequence header the decoded video parameters are updated with the information contained in the header. If the data unit is a picture then:

- the picture is parsed, then decoded
- the picture is placed in the correct position in the output array

Note that since Dirac supports inter as well as intra picture coding, picture numbers within the stream may not be sequential, and the decoded output pictures will not be placed in the output buffer in order. Annex D defines the constraints which may be placed upon re-ordering depth.

Sequences need not be decoded from the start: decoding can start from any sequence header according to the provisions of Section 15.3, although some pictures might not be (completely) decodeable due to a chain of references reaching back earlier in the stream than the sequence header, introducing dependencies on unavailable pictures. The behaviour of a decoder when confronted with such pictures is application-specific.

### 14.1 Non-sequential picture decoding

The ability to decode pictures in a non-sequential manner is important for many applications, such as video editing. Non-sequential access means decoding a stream in any manner other than decoding pictures sequentially from the beginning of the stream to the end: this may include decoding only intra pictures, decoding backwards, or decoding pictures in random parts of the stream.

Stream navigation, including non-sequential access is supported by the information in the parse info headers in the stream (Section 9.6).

In order to start decoding, other than at the start of a sequence, the decoder must first synchronize to the stream. The parse info prefix is present to support such synchronization. A decoder would first search for the parse info prefix to locate the start of a parse info header. The parse info prefix is not guaranteed to occur uniquely within parse info headers (the entropy coding used in Dirac precludes this). However, the probability of a spurious prefix occuring is low: 1 in $2^{32}$, since the prefix is 4 bytes long. The probability of finding two spurious prefix sequences separated by the value of the next (or previous) parse offset is 1 in $2^{64}$.

Having synchronized with the stream the decoder now needs to locate a sequence header in order to find parameters needed to decode pictures. This may be done by moving back (or forward) through the stream, using the parse offsets.

The Dirac stream also supports seeking to a particular picture number, since this is contained in each picture header.

## 15 Picture decoding

This section defines the processes for decoding a picture from a Dirac stream.

Picture decoding depends upon correctly parsing the stream, and decoding operations are dependent on decoding the sequence header and picture metadata (Section 10 and 11) and unpacking the coefficient and motion data (Sections 13 and 12).

### 15.1 Overall picture decoding process

Picture data from the current picture being decoded is stored in the **state**[CURRENT_PICTURE] state variable, which is a map with labels PIC_NUM, and $Y$, $C1$ and $C2$ representing luma and chroma data.

After decoding the decoded picture is returned to the decoding application.

The *picture_decode*() process shall be invoked after parsing the *picture_parse*() process and shall be defined as follows:

```
picture_decode() :                                                          Ref
    state[CURRENT_PICTURE] = {}
    state[CURRENT_PICTURE][PIC_NUM] = state[PICTURE_NUM]
    if (is_ref(()):
        ref_picture_remove()                                                15.4
    if (state[ZERO_RESIDUAL] == False):
        inverse_wavelet_transform()                                         15.6
    else:
        state[CURRENT_PICTURE][Y] = 0
        state[CURRENT_PICTURE][C1] = 0
        state[CURRENT_PICTURE][C2] = 0
    if (is_inter()):
        ref1 = get_ref(state[REF1_PICTURE_NUM])                             15.4
        if (num_refs() == 2):                                               9.6.1
            ref2 = get_ref(state[REF2_PICTURE_NUM])                         15.4
        motion_compensate(ref1[Y], ref2[Y], state[CURRENT_PICTURE][Y], Y)   15.8
        motion_compensate(ref1[C1], ref2[C1], state[CURRENT_PICTURE][C1], C1)   15.8
        motion_compensate(ref1[C2], ref2[C2], state[CURRENT_PICTURE][C2], C2)   15.8
    clip_picture()                                                          15.9
    if (is_ref()):
        ref_picture_add()                                                   15.4
    offset_output_picture(state[CURRENT_PICTURE])                           15.10
    return state[CURRENT_PICTURE]
```

### 15.2 Picture reordering

Picture numbers within the stream may not be in numerical order, and subsequent reordering may be required: the size of the decoded picture buffer required to perform any such reordering may be specified as part of the application profile and level (Annex D).

### 15.3 Random access

Sequence headers represent safe entry points for decoding a sequence.

An accessible picture (with reference to a given sequence header) shall be defined as a picture decodeable without dependence on to data prior to the sequence header in coded order.

Accessibility should normally imply that each accessible picture has no reference picture prior to the sequence header, and no chain of references leading to a reference picture prior to the sequence header. A given level may allow this condition to be relaxed (for example, in P-only coding where unavailable references may be substituted for by zero pictures), but where no specific provision to the contrary is specified in an applicable level or profile, it shall apply.

The first picture data unit after a sequence header shall be called the access picture and shall be accessible with respect to the sequence header. It should normally be an intra picture. If the sequence contains inter pictures it should normally be an intra reference picture.

All picture data units subsequent to the sequence header in coded order shall also be accessible with respect to the sequence header if their picture numbers are greater than or equal to that of the access picture. The access picture therefore represents a temporal access point into the sequence.

**Note:**

If a sequence satisfies a maximum reordering depth constraint (Annex D.2.1.2) of size $N$ all pictures more than $N$ pictures later than the sequence header will have larger picture numbers than the first picture after the sequence header, and hence will be accessible. A reordering depth constraint thus implies that after a sequence header at most $N$ pictures will need to be discarded before all pictures are decodeable.

## 15.4 Reference picture buffer management

This section specifies how the Dirac stream data shall be used to manage the reference picture buffer **state**[REF_PICTURES]. The reference picture buffer has a maximum size of **state**[RB_SIZE] elements, as set in the applicable level (Annex D).

The *ref_picture_remove*() process shall be defined as follows:

```
ref_picture_remove() :                                        Ref
  n = state[RETD_PIC_NUM]
  for k = 0 to state[RB_SIZE] − 1:
    if (state[REF_PICTURES][k][PIC_NUM] == n):
      for j = k to state[RB_SIZE] − 2:
        state[REF_PICTURES][j] = state[REF_PICTURES][j + 1]
      state[RB_SIZE] −= 1
```

The *get_ref*($n$) function shall returns the reference picture in the buffer with picture number $n$. If there is no such picture it shall return an all-zero picture.

The *ref_picture_add*() process for adding pictures to the reference picture buffer shall proceed according to the following rules:

**Case 1.** If the reference picture buffer is not full i.e. has fewer than **state**[MAX_RB_SIZE] elements, then add **state**[CURRENT_PICTURE] to the end of the buffer.

**Case 2.** If the reference picture is full i.e. it has **state**[MAX_RB_SIZE] elements, then remove the first (i.e. oldest) element of the buffer, **state**[REF_PICTURES][0], set

$$\textbf{state}[\text{REF\_PICTURES}][i] = \textbf{state}[\text{REF\_PICTURES}][i+1]$$

for $i = 0$ to **state**[RB_SIZE] $- 2$, and set the last element **state**[REF_PICTURES][**state**[RB_SIZE] $- 1$] equal to a copy of **state**[CURRENT_PICTURE].

## 15.5 Picture IDWT

The inverse discrete wavelet transform process shall consist of transforming the wavelet coefficients for each of the video components. It shall be defined as follows:

| $inverse\_wavelet\_transform()$ : | **Ref** |
|---|---|
| **state**[CURRENT_PICTURE][$Y$] = $idwt$(**state**[Y_TRANSFORM]) | 15.6 |
| **state**[CURRENT_PICTURE][$C1$] = $idwt$(**state**[C1_TRANSFORM]) | 15.6 |
| **state**[CURRENT_PICTURE][$C2$] = $idwt$(**state**[C2_TRANSFORM]) | 15.6 |
| **for each** $c$ **in** $Y, C1, C2$: | |
|   $idwt\_pad\_removal$(**state**[CURRENT_PICTURE][$c$], $c$) | 15.7 |

## 15.6 Component IDWT

This section defines the $idwt(coeff\_data)$ process for reconstructing picture component data from decoded subband data $coeff\_data$ using the inverse discrete wavelet transform (IDWT). The IDWT shall be invoked in the picture decoding process only after successful unpacking of the subband coefficient data (Section 13.4).

The IDWT process shall return a pixel array from the subband wavelet coefficients representing a reconstructed video component (Y, C1 or C2) for a single picture.

Since wavelet filtering operates on both rows and columns of two-dimensional arrays independently it is useful to define operators row$(a, k)$ and column$(a, k)$ for extracting rows and columns with index $k$ from a 2-dimensional array $a$:

If $b = \text{row}(a, k)$ then $b[r]$ is a *reference* to the value of $a[k][r]$. This means that modifying the value of $b[r]$ modifies the value of $a[k][r]$.

If $b = \text{column}(a, k)$ then $b[r]$ is a *reference* the value of $a[r][k]$. This means that modifying the value of $b[r]$ modifies the value of $a[r][k]$.

The $idwt()$ process shall be an iterative procedure operating on four subbands ($LL$, $HL$,$LH$and $HH$) at each iteration stage to produce a new subband. The procedure shall be as follows

| $idwt(coeff\_data)$ : | **Ref** |
|---|---|
| $LL\_band = coeff\_data[0][LL]$ | |
| **for** $n = 1$ **to state**[DWT_DEPTH]: | |
|   $new\_LL = vh\_synth(LL\_band, coeff\_data[n][HL], coeff\_data[n][LH], coeff\_data[n][HH])$ | 15.6.1 |
|   $LL\_band = new\_LL$ | |
| **return** $LL\_band$ | |

Note that at each stage, the input dimensions of the input $LL\_band$ will be the same as those of the other input bands, whereas the output dimensions are double those of the input bands.

### 15.6.1 Vertical and horizontal synthesis

This section specifies the operation of the vertical and horizontal synthesis process:

$vh\_synth(LL\_data, HL\_data, LH\_data, HH\_data)$

$vh\_synth$ shall return an array of twice the dimensions of each of the input argument arrays.

$vh\_synth$ is repeatedly invoked by the IDWT synthesis process and operates on four subband data arrays of identical dimensions to produce a new array $synth$, which shall be returned as the result of the process.

**Step 1.** $synth$ is a temporary two-dimensional array that shall be initialised so that:

$$\begin{aligned}\text{width}(synth) &= 2 * \text{width}(LL\_data)\\ \text{height}(synth) &= 2 * \text{height}(LL\_data)\end{aligned}$$

**Step 2.** The data from the four arrays shall be interleaved as follows:

```
...
    for $y = 0$ to $(\text{height}(synth)//2) - 1$:
        for $x = 0$ to $(\text{width}(synth)//2) - 1$:
            $synth[2 * y][2 * x] = LL\_data[y][x]$
            $synth[2 * y][2 * x + 1] = HL\_data[y][x]$
            $synth[2 * y + 1][2 * x] = LH\_data[y][x]$
            $synth[2 * y + 1][2 * x + 1] = HH\_data[y][x]$
    ...
```

Note: This enables in-place calculation during the inverse filter process.

**Step 3.** Data shall be synthesised vertically by operating on each column of data using a one-dimensional filter, and then horizontally by operating on each row. The one-dimensional filters used shall be determined by the value of **state**[WAVELET_INDEX] according to Tables 15.1–15.7. The process shall be as follows:

```
...
    for $x = 0$ to $\text{width}(synth) - 1$:
        $1d\_synthesis(\text{column}(synth, x))$                15.6.2
    for $y = 0$ to $\text{height}(synth) - 1$:
        $1d\_synthesis(\text{row}(synth, y))$                   15.6.2
    ...
```

**Step 4.** Finally, the synthesised subband data shall implement a bitshift to remove any accuracy bits. The bit shift value $filtershift()$ used shall be determined by the value of **state**[WAVELET_INDEX] according to Tables 15.1–15.7. The process shall be as follows:

```
...
    $shift = filtershift()$
    if $(shift > 0)$:
        for $y = 0$ to $\text{height}(synth) - 1$:
            for $x = 0$ to $\text{width}(synth) - 1$:
                $synth[y][x] = (synth[y][x] + (1 << (shift - 1))) \gg shift$
```

**Note:** Accuracy bits are added in the encoder by shifting up all coefficients in the LL band prior to applying any filtering (this includes an initial shift of all values in the component data). Adding a small shift before each decomposition stage is the most efficient way of providing additional resolution mitigating aliasing through non-linear rounding effects.

### 15.6.2 One-dimensional synthesis

This section specifies the one-dimensional synthesis process $1d\_synthesis()$, which shall apply to a 1-dimensional array of coefficients of even length, consisting of either a row or a column of a 2-dimensional integral data array.

The one-dimensional synthesis process shall comprise the application of a number of reversible integer lifting filter operations.

Lifting filtering operations shall be one of four types, Type 1, Type 2, Type 3 and Type 4. Each type shall be characterised by four elements:

- a filter length value $L$
- a filter offset value $D$
- an array of taps of length $L$: $taps[0], \dots, taps[L-1]$
- a scale factor $S$

The four types of lifting operations shall be defined by the functions:

$lift1(A, L, D, taps, S)$,

$lift2(A, L, D, taps, S)$,

$lift3(A, L, D, taps, S)$, and

$lift4(A, L, D, taps, S)$.

respectively and shall act upon the values in a one-dimensional array $A$.

The Type 1 lifting process $lift1(A, L, D, taps, S)$ shall be defined as follows:

```
$lift1(A, L, D, taps, S)$ :                                         Ref
  for $n = 0$ to $(\text{length}(A)//2) - 1$:
    $sum = 0$
    for $i = D$ to $L + D - 1$:
      $pos = 2 * (n + i) - 1$
      $pos = \min(pos, \text{length}(A) - 1)$
      $pos = \max(pos, 1)$
      $sum += taps[i - D] * A[pos]$
    if $(S > 0)$:
      $sum += (1 \ll (S - 1))$
    $A[2 * n] += (sum \gg S)$
```

The Type 2 lifting process $lift2(A, L, D, taps, S)$ shall be defined as follows:

```
$lift2(A, L, D, taps, S)$ :                                         Ref
  for $n = 0$ to $(\text{length}(A)//2) - 1$:
    $sum = 0$
    for $i = D$ to $L + D - 1$:
      $pos = 2 * (n + i) - 1$
      $pos = \min(pos, \text{length}(A) - 1)$
      $pos = \max(pos, 1)$
      $sum += taps[i - D] * A[pos]$
    if $(S > 0)$:
      $sum += (1 \ll (S - 1))$
    $A[2 * n] -= (sum \gg S)$
```

The Type 3 lifting process $lift3(A, L, D, taps, S)$ shall be defined as follows:

```
$lift3(A, L, D, taps, S)$ :                                         Ref
  for $n = 0$ to $(\text{length}(A)//2) - 1$:
    $sum = 0$
    for $i = D$ to $L + D - 1$:
      $pos = 2 * (n + i)$
      $pos = \min(pos, \text{length}(A) - 2)$
      $pos = \max(pos, 0)$
      $sum += taps[i - D] * A[pos]$
    if $(S > 0)$:
      $sum += (1 \ll (S - 1))$
    $A[2 * n + 1] += (sum \gg S)$
```

The Type 4 lifting process $lift4(A, L, D, taps, S)$ shall be defined as follows:

| $lift4(A, L, D, taps, S)$ : | **Ref** |
|---|---|
| **for** $n = 0$ **to** $(\text{length}(A)//2) - 1$: | |
|     $sum = 0$ | |
|     **for** $i = D$ **to** $L + D - 1$: | |
|         $pos = 2 * (n + i)$ | |
|         $pos = \min(pos, \text{length}(A) - 2)$ | |
|         $pos = \max(pos, 0)$ | |
|         $sum += taps[i - D] * A[pos]$ | |
|     **if** $(S > 0)$: | |
|         $sum += (1 \ll (S - 1))$ | |
|     $A[2 * n + 1] -= (sum \gg S)$ | |

$1d\_synthesis$ shall apply the sequence of lifting filters specified in Section 15.6.3 corresponding to the value of **state**[WAVELET_INDEX],and shall invoke the corresponding lifting processes with the parameters defined.

#### 15.6.2.1 Mathematical formulation of lifting processes (Informative)

The lifting processes defined in the previous section are extremely similar, and careful attention should be paid to the detail of their operation in any implementation. The four different variants arise from two factors: the phase (odd or even) of the lifting operation, and their implementation using integer-only operations, which introduces rounding errors and makes addition and subtraction subtly different.

A lifting operation either modifies the odd coefficients by a linear combination of the even coefficients, or vice-versa. Mathematically, the four types of filter may be described as follows.

Type 1 and Type 2 lifting filtering operations modify the even coefficients by the odd coefficients:

$$\begin{aligned} A[2*n] \quad &+= \left(\sum_{i=-N}^{M} t_i * A[2*(n+i)+1] + (1 \ll (s-1))\right) \gg s \text{ (Type 1)} \\ A[2*n] \quad &-= \left(\sum_{i=-N}^{M} t_i * A[2*(n+i)+1] + (1 \ll (s-1))\right) \gg s \text{ (Type 2)} \end{aligned}$$

Type 3 and Type 4 lifting filtering operation modify the odd coefficients by the even coefficients:

$$\begin{aligned} A[2*n+1] \quad &+= \left(\sum_{i=-N}^{M} t_i A[2*(n+i)] + (1 \ll (s-1))\right) \gg s \text{ (Type 3)} \\ A[2*n+1] \quad &-= \left(\sum_{i=-N}^{M} t_i A[2*(n+i)] + (1 \ll (s-1))\right) \gg s \text{ (Type 4)} \end{aligned}$$

The distinctions between Type 1 and Type 2 and between Type 3 and Type 4 filters are necessary because integer division (bit-shifting) is being used, and so the filters are non-linear: a Type 1 or Type 3 filter with taps $t_i$ is *not* equivalent to an Type 2 or Type 4 filter with taps $-t_i$.

Edge extension is used where the filter would otherwise extend beyond the boundaries of the array. This is slightly different between Types 1 and 2 on the one hand and Types 3 and 4 on the other. This is because even values and odd values must be extended separately to maintain the correct phase (and hence invertibility of the filter). For example, a Type 1 filter must use the values 1 and $\text{length}(A) - 1$ at the edges because (as the length is even) these are the odd values nearest the edges.

Further information on wavelet filtering and lifting is provided in Annex G.

### 15.6.3 Lifting filter parameters

The lifting filters and filter bit-shift operations that apply for each value **state**[WAVELET_INDEX] shall be as specified in Tables 15.1 to 15.7 below.

Lifting steps:

1. Type 2, $L = 2, D = 0, taps = [1, 1], S = 2$ i.e.
   $A[2 * n] - = (A[2 * n - 1] + A[2 * n + 1] + 2) \gg 2$
2. Type 3, $L = 4, D = -1, taps = [-1, 9, 9, -1], S = 4$ i.e.
   $A[2 * n + 1] + = (-A[2 * n - 2] + 9 * A[2 * n] + 9 * A[2 * n + 2] - A[2 * n + 4] + 8) \gg 4$

$filtershift()$ returns 1

Table 15.1: **state**[WAVELET_INDEX] == 0: Deslauriers-Dubuc (9,7) lifting stages and shift values

Lifting steps:

1. Type 2, $L = 2, D = 0, taps = [1, 1], S = 2$ i.e.
   $A[2 * n] - = (A[2 * n - 1] + A[2 * n + 1] + 2) \gg 2$
2. Type 3, $L = 2, D = 0, taps = [1, 1], S = 1$ i.e.
   $A[2 * n + 1] + = (A[2 * n] + A[2 * n + 2] + 1) \gg 1$

$filtershift()$ returns 1

Table 15.2: **state**[WAVELET_INDEX] == 1: LeGall (5,3) lifting stages and shift values

Lifting steps:

1. Type 2, $L = 4, D = -1, taps = [-1, 9, 9, -1], S = 5$ i.e.
   $A[2 * n] - = (-A[2 * n - 3] + 9 * A[2 * n - 1] + 9 * A[2 * n + 1] - A[2 * n + 3] + 16) \gg 5$
2. Type 3, $L = 4, D = -1, taps = [-1, 9, 9, -1], S = 4$ i.e.
   $A[2 * n + 1] + = (-A[2 * n - 2] + 9 * A[2 * n] + 9 * A[2 * n + 2] - A[2 * n + 4] + 8) \gg 4$

$filtershift()$ returns 1

Table 15.3: **state**[WAVELET_INDEX] == 2: Deslauriers-Dubuc (13,7) lifting stages and shift values

Lifting steps:

1. Type 2, $L = 1, D = 1, taps = [1], S = 1$ i.e.
   $A[2 * n] - = (A[2 * n + 1] + 1) \gg 1$
2. Type 3, $L = 1, D = 0, taps = [1], S = 0$ i.e.
   $A[2 * n + 1] + = A[2 * n]$

$filtershift()$ returns 0

Table 15.4: **state**[WAVELET_INDEX] == 3: Haar filter with no shift

Lifting steps:

1. Type 2, $L = 1, D = 1, taps = [1], S = 1$ i.e.
   $A[2 * n] - = (A[2 * n + 1] + 1) \gg 1$
2. Type 3, $L = 1, D = 0, taps = [1], S = 0$ i.e.
   $A[2 * n + 1] + = A[2 * n]$

$filtershift()$ returns 1

Table 15.5: **state**[WAVELET_INDEX] == 4: Haar filter with single shift

Lifting steps:

1. Type 3, $L = 8, D = -3, taps = [] - 2, 10, -25, 81, 81, -25, 10, -2], S = 8$ i.e.

$$A[2*n+1] \quad += \quad (-2*(A[2*n-6]+A[2*n+8]) + 10*(A[2*n-4]+A[2*n+6]) \\ -25*(A[2*n-2]+A[2*n+4]) + 81*(A[2*n]+A[2*n+2]) + 128) \gg 8$$

1. Type 2, $L = 8, D = -3, taps = [-8, 21, -46, 161, 161, -46, 21, -8], S = 8$ i.e.

$$A[2*n] \quad -= \quad (-8*(A[2*n-7]+A[2*n+7]) + 21*(A[2*n-5]+A[2*n+5]) \\ -46*(A[2*n-3]+A[2*n+3]) + 161*(A[2*n-1]+A[2*n+1]) + 128) \gg 8$$

$filtershift()$ returns 0

Table 15.6: **state**[WAVELET_INDEX] == 5: Fidelity filter for improved downconversion and anti-aliasing

Lifting steps:

1. Type 2, $L = 2, D = 0, taps = [1817, 1817], S = 12$ i.e.
   $A[2*n] -= (1817*A[2*n-1] + 1817*A[2*n+1] + 2048) \gg 12$
2. Type 4, $L = 2, D = 0, taps = [3616, 3616], S = 12$ i.e.
   $A[2*n+1] -= (3616*A[2*n] + 3616*A[2*n+2] + 2048) \gg 12$
3. Type 1, $L = 2, D = 0, taps = [217, 217], S = 12$ i.e.
   $A[2*n] += (217*A[2*n-1] + 217*A[2*n+1] + 2048) \gg 12$
4. Type 3, $L = 2, D = 0, taps = [6497, 6497], S = 12$ i.e.
   $A[2*n+1] += (6497*A[2*n] + 6497*A[2*n+2] + 2048) \gg 12$

$filtershift()$ returns 1

Table 15.7: **state**[WAVELET_INDEX] == 6: Integer lifting approximation to Daubechies (9,7)

## 15.7 Removal of IDWT pad values

This section defines the decoding process $idwt\_pad\_removal(pic, c)$ for removing extraneous values after performing the IDWT.

Section 13.1.1 requires that subband coefficient data arrays are padded to ensure that the reconstructed data array $pic$ has dimensions divisible by $2^{\textbf{state}[\text{DWT\_DEPTH}]}$.

Values $width$ and $height$ are defined to be the appropriate dimensions of the component data:

- If $c = Y$, then

$$\begin{aligned} width &= \textbf{state}[\text{LUMA\_WIDTH}] \\ height &= \textbf{state}[\text{LUMA\_HEIGHT}] \end{aligned}$$

- else if $c = C1$ or $c = C2$,

$$\begin{aligned} width &= \textbf{state}[\text{CHROMA\_WIDTH}] \\ height &= \textbf{state}[\text{CHROMA\_HEIGHT}] \end{aligned}$$

All component data $pic[j][i]$ with

- $i \geq width$, or
- $j \geq height$

shall be discarded and $pic$ shall be resized to have width $width$ and height $height$.

## 15.8 Motion compensation

This section defines the operation of the process $motion\_compensate(ref1, ref2, pic, c)$ for motion-compensating a picture component array $pic$ of type $c = Y, U$ or $V$ from reference component arrays $ref1$ and $ref2$ of the same type.

This process shall be invoked for each component in a picture, subsequent to the decoding of coefficient data, specified in Section 13.4, and the Inverse Wavelet Transform (IWT), specified in Section 15.6.

Motion compensation shall use the motion block data **state**[BLOCK_DATA] and optionally may use the global motion parameters **state**[GLOBAL_PARAMS].

### 15.8.1 Overlapped Block Motion Compensation (OBMC) (Informative)

Motion compensated prediction methods provide methods for determining predictions for pixels in the current picture by using motion vectors to define offsets from those pixels to pixels in previously decoded pictures. Motion compensation techniques vary in how those pixels are grouped together, and how a prediction is formed for pixels in a given group. In conventional block motion compensation, as used in MPEG2, H.264 and many other codecs, the picture is divided into *disjoint* rectangular blocks and the motion vector or vectors associated with that block defines the offset(s) into the reference pictures.

In OBMC, by contrast, the predicted picture is divided into a regular overlapping blocks of dimensions $xblen$ by $yblen$ that cover at least the entire picture area as shown in figure 15.1. Overlapping is ensured by starting each block at a horizontal separation $xbsep$ and a vertical separation $ybsep$ from its neighbours, where these values are less than the corresponding block dimensions.

The overlap between blocks horizontally is $xoffset = (xblen - xbsep)/2$ both on the left and on the right, and vertically is $yoffset = (yblen - ybsep)/2$ both on the top and the bottom. As a result pixels in the overlapping areas lie in more than one block, and so more than one motion vector set (and set of associated predictions)

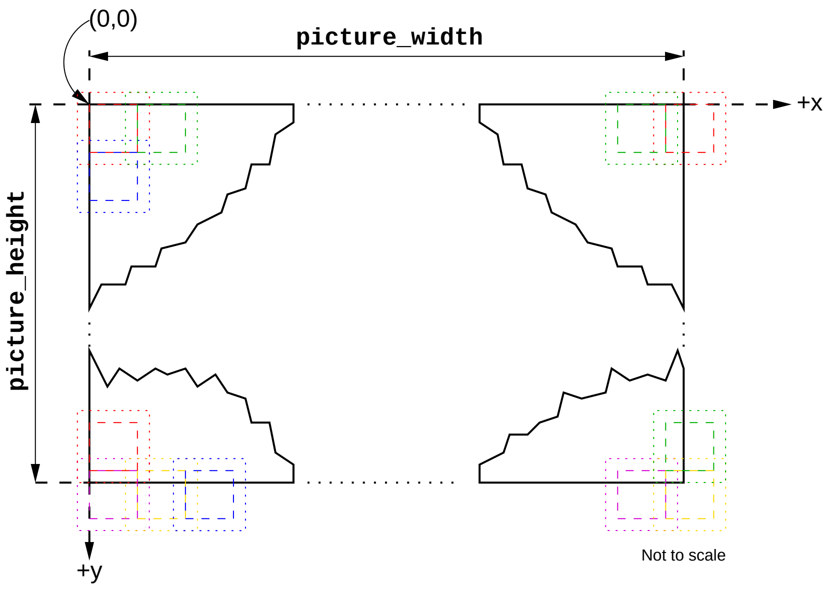

Figure 15.1: Block coverage of the predicted picture

applies to them. Indeed, a pixel may have up to eight predictions, as it may belong to up to four blocks, each of which may have up to two motion vectors. These are combined into a single prediction by using weights, which are so constructed so as to sum to 1. In the Dirac integer implementation, fractional weights are achieved by insisting that weights sum to a power of 2, which is then shifted out once all contributions have been summed.

In Dirac blocks are positioned so that blocks will overspill the left and top edges by ($xoffset$) and ($yoffset$) pixels. The number of blocks has been determined (Section 11.2.4) so that the picture area is wholly covered, and the overspill on the right hand and bottom edges will be at least the amount on the left and top edges. Indeed, the number of blocks has been set so that the blocks divide into whole superblocks (sets of 4x4 blocks), which mean that some blocks may fall entirely out of the picture area. Any predictions for pixels outside the actual picture area are discarded.

### 15.8.2 Overall motion compensation process

The motion compensation process shall form an integer prediction for each pixel in the predicted picture component *pic*, which shall be added to the pixel value, and then clipped to keep it in range.

The *motion_compensate*() process is defined by means of a temporary data array *mc_tmp* for storing the motion-compensated prediction for the current picture.

The *motion_compensate*() process shall be defined as follows:

| $motion\_compensate(ref1, ref2, pic, c)$ : | **Ref** |
|---|---|
| **if** $(c == Y)$: | |
| $\quad bit\_depth =$ **state**[LUMA_DEPTH] | |
| **else**: | |
| $\quad bit\_depth =$ **state**[CHROMA_DEPTH] | |
| $init\_dimensions(c)$ | 15.8.3 |
| $mc\_tmp = init\_temp\_array()$ | 15.8.4 |
| **for** $j = 0$ **to state**[BLOCKS_Y] $- 1$: | |
| $\quad$ **for** $i = 0$ **to state**[BLOCKS_X] $- 1$: | |
| $\quad\quad block\_mc(mc\_tmp, i, j, ref1, ref2, c)$ | 15.8.5 |
| **for** $y = 0$ **to state**[LEN_Y] $- 1$: | |
| $\quad$ **for** $x = 0$ **to state**[LEN_X] $- 1$: | |
| $\quad\quad pic[y][x] += (mc\_tmp[y][x] + 32) \gg 6$ | |
| $\quad\quad pic[y][x] = \text{clip}(pic[y][x], -2^{bit\_depth-1}, 2^{bit\_depth-1} - 1)$ | |

**Note:** Six bits are used for the overlapped-block weighting matrix. This ensures that 10-bit data may normally be motion compensated using 16-bit words as per Section 15.8.5.

### 15.8.3 Dimensions

Since motion compensation shall apply to both luma and (potentially subsampled) chroma data, for simplicity a number of variables are defined by the *init_dimensions*() function, which is as follows:

| $init\_dimensions(c)$ : | **Ref** |
|---|---|
| **if** $(c == Y)$: | |
| $\quad$ **state**[LEN_X] = **state**[LUMA_WIDTH] | |
| $\quad$ **state**[LEN_Y] = **state**[LUMA_HEIGHT] | |
| $\quad$ **state**[XBLEN] = **state**[LUMA_XBLEN] | |
| $\quad$ **state**[YBLEN] = **state**[LUMA_YBLEN] | |
| $\quad$ **state**[XBSEP] = **state**[LUMA_XBSEP] | |
| $\quad$ **state**[YBSEP] = **state**[LUMA_YBSEP] | |
| **else**: | |
| $\quad$ **state**[LEN_X] = **state**[CHROMA_WIDTH] | |
| $\quad$ **state**[LEN_Y] = **state**[CHROMA_HEIGHT] | |
| $\quad$ **state**[XBLEN] = **state**[CHROMA_XBLEN] | |
| $\quad$ **state**[YBLEN] = **state**[CHROMA_YBLEN] | |
| $\quad$ **state**[XBSEP] = **state**[CHROMA_XBSEP] | |
| $\quad$ **state**[YBSEP] = **state**[CHROMA_YBSEP] | |
| **state**[XOFFSET] = (**state**[XBLEN] − **state**[XBSEP])//2 | |
| **state**[YOFFSET] = (**state**[YBLEN] − **state**[YBSEP])//2 | |

**Note:** The subband data that makes up the IWT coefficients is padded in order that the IWT may function correctly. For simplicity, in this specification, padding data is removed after the IWT has been performed so that the picture data and reference data arrays have the same dimensions for motion compensation. However, it may be more efficient to perform all operations prior to the output of pictures using padded data, i.e. to discard padding values subsequent to motion compensation. Such a course of action is equivalent, so long as it is realised that blocks must be regarded as edge blocks if they overlap the actual picture area, not the larger area produced by padding.

### 15.8.4 Initialising the motion compensated data array

The *init_temp_array*() function shall return a two-dimensional data array with horizontal size **state**[LEN_X] and vertical size **state**[LEN_Y], such that each element of the two dimensional array shall be set to zero.

### 15.8.5 Motion compensation of a block

This section defines the *block_mc*() process for motion-compensating a single block.

Each block shall be motion-compensated by applying a weighting matrix to a block prediction and adding the weighted prediction into the motion-compensated prediction array.

The *block_mc*() process shall be defined as follows:

```
$block\_mc(mc\_pred, i, j, ref1, ref2, c)$ :                                                          Ref
  $xstart = i * \textbf{state}[\text{XBSEP}] - \textbf{state}[\text{XOFFSET}]$
  $ystart = j * \textbf{state}[\text{XBSEP}] - \textbf{state}[\text{XOFFSET}]$
  $xstop = (i+1) * \textbf{state}[\text{XBSEP}] + \textbf{state}[\text{XOFFSET}]$
  $ystop = (j+1) * \textbf{state}[\text{YBSEP}] + \textbf{state}[\text{YOFFSET}]$
  $mode = \textbf{state}[\text{BLOCK\_DATA}][j][i][\text{RMODE}]$
  $W = spatial\_wt(i, j)$                                                                              15.8.7
  **for** $y = \max(ystart, 0)$ **to** $\min(ystop, \textbf{state}[\text{LEN\_Y}]) - 1$:
    **for** $x = \max(xstart, 0)$ **to** $\min(xstop, \textbf{state}[\text{LEN\_X}]) - 1$:
      $p = x - xstart$
      $q = y - ystart$
      **if** $(mode == \text{INTRA})$:
        $val = \textbf{state}[\text{BLOCK\_DATA}][j][i][dc][c]$
      **else if** $(mode == \text{REF1ONLY})$:
        $val = pixel\_pred(ref1, 1, i, j, x, y, c)$                                                    15.8.7
        $val* = \textbf{state}[\text{REF1\_WT}] + \textbf{state}[\text{REF2\_WT}]$
        $val = (val + 2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]-1}) \gg \textbf{state}[\text{REFS\_WT\_PRECISION}]$
      **else if** $(mode == \text{REF2ONLY})$:
        $val = pixel\_pred(ref2, 2, i, j, x, y, c)$                                                    15.8.7
        $val* = \textbf{state}[\text{REF1\_WT}] + \textbf{state}[\text{REF2\_WT}]$
        $val = (val + 2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]-1}) \gg \textbf{state}[\text{REFS\_WT\_PRECISION}]$
      **else if** $(mode == \text{REF1AND2})$:
        $val1 = pixel\_pred(ref1, 1, i, j, x, y, c)$                                                   15.8.7
        $val1* = \textbf{state}[\text{REF1\_WT}]$
        $val2 = pixel\_pred(ref2, 2, i, j, x, y, c)$                                                   15.8.7
        $val2* = \textbf{state}[\text{REF2\_WT}]$
        $val = (val1 + val2 + 2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]-1}) \gg \textbf{state}[\text{REFS\_WT\_PRECISION}]$
      $val* = W[q][p]$
      $mc\_tmp[y][x] + = val$
```

**Note:** Note that if the two reference weights are 1 and **state**[REFS_WT_PRECISION] is 1, then reference weighting is transparent and

```
...
  $val = pixel\_pred(ref1, 1, i, j, x, y, c)$
  $val* = \textbf{state}[\text{REF1\_WT}] + \textbf{state}[\text{REF2\_WT}]$
  $val = (val + 2^{\textbf{state}[\text{REFS\_WT\_PRECISION}]-1}) \gg \textbf{state}[\text{REFS\_WT\_PRECISION}]$
  ...
```

reduces to

```
...
  $val = pixel\_pred(ref1, 1, i, j, x, y, c)$
  ...
```

In this case, therefore, the normal reference weighting produces no additional dynamic range for internal processing and 10 bit video can be motion compensated with 16 bit unsigned internal values.

In general, however, the worst case internal bit widths consist of the video bit depth plus the maximum of: 6 (the spatial matrix bit width) and the value of **state**[REFS_WT_PRECISION]. 6 bits should be sufficient for most fading compensation applications, and so 16 bit internals will suffice for all practical motion compensation scenarios for 8 and 10 bit video.

### 15.8.6 Spatial weighting matrix

This section specifies the function $spatial\_wt(i, j)$ for deriving the 6-bit spatial weighting matrix that shall be applied to the block with coordinates $(i, j)$.

Note that other weights shall be applied to the prediction as a result of the weights applied to each reference.

The same weighting matrix shall be returned for all blocks within the interior of the picture component array. Suitably modified weighting matrices shall be returned for blocks at the edges of the picture component data array.

The function shall return a two-dimensional spatial weighting matrix. This shall apply a linear roll-off in both horizontal and vertical directions.

The spatial matrix returned shall be the product of a horizontal and a vertical weighting matrix. It shall be defined as follows:

```
$spatial\_wt(i, j)$ :                                              Ref
  for $y = 0$ to state[YBLEN] $- 1$:
    for $x = 0$ to state[XBLEN] $- 1$:
      $W[y][x] = h\_wt(i)[x] * v\_wt(j)[y]$
  return $W$
```

The horizontal weighting function shall be defined as follows:

```
$h\_wt(i)$ :                                                       Ref
  if (state[XOFFSET] $!= 1$):
    for $x = 0$ to $2 *$ state[XOFFSET] $- 1$:
      $hwt[x] = 1 + (6 * x +$ state[XOFFSET] $- 1)//(2 *$ state[XOFFSET] $- 1)$
      $hwt[x +$ state[XBSEP]$] = 8 - hwt[x]$
  else:
    $hwt[0] = 3$
    $hwt[1] = 5$
    $hwt[$state[XBSEP]$] = 5$
    $hwt[$state[XBSEP]$ + 1] = 3$
  for $x = 2 *$ state[XOFFSET] to state[XBSEP] $- 1$:
    $hwt[x] = 8$
  if $(i == 0)$:
    for $x = 0$ to $2 *$ state[XOFFSET] $- 1$:
      $hwt[x] = 8$
  else if $(i ==$ state[BLOCKS_X] $- 1)$:
    for $x = 0$ to $2 *$ state[XOFFSET] $- 1$:
      $hwt[x +$ state[XBSEP]$] = 8$
  return $hwt$
```

The vertical weighting function shall be defined as follows:

```
$v\_wt(j)$ :                                                          Ref
  if ($\mathbf{state}[\text{YOFFSET}] != 1$):
    for $y = 0$ to $2 * \mathbf{state}[\text{YOFFSET}] - 1$:
      $vwt[y] = 1 + (6 * y + \mathbf{state}[\text{YOFFSET}] - 1)//(2 * \mathbf{state}[\text{YOFFSET}] - 1)$
      $vwt[y + \mathbf{state}[\text{YBSEP}]] = 8 - vwt[y]$
  else:
    $vwt[0] = 3$
    $vwt[1] = 5$
    $vwt[\mathbf{state}[\text{YBSEP}]] = 5$
    $vwt[\mathbf{state}[\text{YBSEP}] + 1] = 3$
  for $y = 2 * \mathbf{state}[\text{YOFFSET}]$ to $\mathbf{state}[\text{YBSEP}] - 1$:
    $vwt[y] = 8$
  if ($j == 0$):
    for $y = 0$ to $2 * \mathbf{state}[\text{YOFFSET}] - 1$:
      $vwt[y] = 8$
  else if ($j == \mathbf{state}[\text{BLOCKS\_Y}] - 1$):
    for $y = 0$ to $2 * \mathbf{state}[\text{YOFFSET}] - 1$:
      $vwt[y + \mathbf{state}[\text{YBSEP}]] = 8$
  return $vwt$
```

**Note:** The horizontal and vertical weighting arrays satisfy the perfect reconstruction property across block overlaps by construction:

$$\begin{aligned} hwt[x + \mathbf{state}[\text{XBSEP}]] &= 8 - hwt[x] \\ vwt[y + \mathbf{state}[\text{YBSEP}]] &= 8 - vwt[y] \end{aligned}$$

In addition, it can be shown they are always symmetric (except at picture edges), or equivalently the leading edges have skew-symmetry about the half-way point:

$$\begin{aligned} hwt[\mathbf{state}[\text{XBLEN}] - 1 - x] &= hwt[x] \\ vwt[\mathbf{state}[\text{YBLEN}] - 1 - y] &= vwt[y] \end{aligned}$$

The horizontal and vertical weighting matrix components for various block overlaps are shown in Table 15.8. These encompass all the default values listed in Table 11.1 for both luma and chroma.

| Overlap (length-separation) | Offset | Leading edge |
|---|---|---|
| 2 | 1 | 3,5 |
| 4 | 2 | 1,3,5,7 |
| 8 | 4 | 1,2,3,4,4,5,6,7 |
| 16 | 8 | 1,1,2,2,3,3,3,4,4,5,5,5,6,6,7,7 |

Table 15.8: Leading and trailing edge values for different block overlaps

### 15.8.7 Pixel prediction

This section defines the operation of the $pixel\_pred(ref, ref\_num, i, j, x, y, c)$ process which shall be used for forming the prediction for a pixel with coordinates $(x, y)$ in component $c$, belonging to the block with coordinates $(i, j)$.

The pixel prediction process shall consist of two stages. In the first stage, a motion vector to be applied to pixel $(x, y)$ shall be derived. For block motion, this shall be a block motionvector that shall apply to all pixels in a block. For global motion the motion vector shall be computed from the global motion parameters and may vary pixel-by-pixel.

In the second stage, the motion vector shall be used to derive coordinates in an reference picture.

| $pixel\_pred(ref, ref\_num, i, j, x, y, c)$ : | **Ref** |
|---|---|
| **if** (**state**[BLOCK_DATA][$j$][$i$][GMODE] == **False**): | |
| $mv$ = **state**[BLOCK_DATA][$j$][$i$][VECTOR][$ref\_num$] | |
| **else**: | |
| $mv = global\_mv(ref\_num, x, y)$ | 15.8.8 |
| **if** $(c! = Y)$: | |
| $mv = chroma\_mv\_scale(mv)$ | 15.8.9 |
| $px = (x \ll$ **state**[MV_PRECISION]$) + mv[0]$ | |
| $py = (y \ll$ **state**[MV_PRECISION]$) + mv[1]$ | |
| **if** (**state**[MV_PRECISION] $> 0$): | |
| **return** $subpel\_predict(ref, c, px, py)$) | 15.8.10 |
| **else**: | |
| **return** $ref[\text{clip}(py, 0, \text{height}(ref) - 1)][\text{clip}(px, 0, \text{width}(ref) - 1)]$ | |

### 15.8.8 Global motion vector field generation

This section specifies the operation of the $global\_mv(ref\_num, x, y)$ process for deriving a global motion vector for a pixel at location $(x, y)$.

The function shall be defined as follows:

| $global\_mv(ref\_num, x, y)$ : | **Ref** |
|---|---|
| $ez$ = **state**[GLOBAL_PARAMS][$ref\_num$][ZRS_EXP] | |
| $ep$ = **state**[GLOBAL_PARAMS][$ref\_num$][PERSP_EXP] | |
| $b$ = **state**[GLOBAL_PARAMS][$ref\_num$][PAN_TILT] | |
| $A$ = **state**[GLOBAL_PARAMS][$ref\_num$][ZRS] | |
| $c$ = **state**[GLOBAL_PARAMS][$ref\_num$][PERSPECTIVE] | |
| $m = 2^{ep} - (c[0] * x + c[1] * y)$ | |
| $v[0] = m * ((A[0][0] * x + A[0][1] * y) + 2^{ez} * b[0])$ | |
| $v[1] = m * ((A[1][0] * x + A[1][1] * y) + 2^{ez} * b[1])$ | |
| $v[0] = (v[0] + (1 \ll (ez + ep))) \gg (ez + ep)$ | |
| $v[1] = (v[1] + (1 \ll (ez + ep))) \gg (ez + ep)$ | |
| **return** $v$ | |

**Note:** Write $\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$. Mathematically, we wish the global motion vector $\mathbf{v}$ to be defined by:

$$\mathbf{v} = \frac{\mathbf{Ax} + \mathbf{b}}{1 + \mathbf{c}^T\mathbf{x}}$$

where: $\mathbf{A}$ is a matrix describing the degree of zoom, rotation or shear; $\mathbf{b}$ is a translation vector; and $\mathbf{c}$ is a perspective vector which expresses the degree to which the global motion is not orthogonal to the axis of view.

In Dirac, this formula is adjusted in two ways in order to get an implementable result. Firstly, the perspective element is adjusted to remove a division, changing the formula to:

$$\mathbf{v} = (1 - \mathbf{c}^T\mathbf{x})(\mathbf{Ax} + \mathbf{b})$$

which is valid for small $\mathbf{c}$. Secondly, the formula is re-cast in terms of integer arithmetic by giving the matrix element an accuracy factor $\alpha$ and the perspective element an accuracy factor $\beta$:

$$\mathbf{v} = (1 - 2^{-\beta}\mathbf{c}^T\mathbf{x})(2^{-\alpha}\mathbf{A}\mathbf{x} + \mathbf{b})$$

where the parameters $\mathbf{A}, \mathbf{b}, \mathbf{c}$ are now integral. (No accuracy bits are required for the translation, since it must be an integral number of sub-pixels.)

This reduces to

$$2^{\alpha+\beta}\mathbf{v} = (2^{\beta} - \mathbf{c}^T\mathbf{x})(\mathbf{A}\mathbf{x} + 2^{\alpha}\mathbf{b})$$

and this formula is used for the computation of values.

### 15.8.9 Chroma subsampling

When motion compensating chroma components, motion vectors shall be scaled by the $chroma\_mv\_scale()$ function. This produces chroma vectors in units of **state**[MV_PRECISION] with respect to the chroma samples, as follows:

| $chroma\_mv\_scale(v)$ : | **Ref** |
|---|---|
| $sv[0] = v[0]//chroma\_h\_ratio()$ | 10.5.1 |
| $sv[1] = v[1]//chroma\_v\_ratio()$ | 10.5.1 |
| **return** $sv$ | |

.

**Note:** Recall that division in this specification rounds towards -infinity. This division can be achieved by a bit-shift in C/C++ as chroma dimension ratios are 1 or 2.

### 15.8.10 Sub-pixel prediction

This section defines the operation of the $subpel\_predict(ref, c, u, v)$ function for producing a sub-pixel accurate value at location $(u, v)$ from an upconverted picture reference component of type $c$ (Y, C1 or C2).

Upconversion shall be defined by means of a half-pixel interpolated reference array $upref$. $upref$ shall have dimensions $(2W - 1)$x$(2H - 1)$ where the original reference picture component array has dimensions $W$x$H$, as per Section 15.8.11.

Motion vectors shall be permitted to extend beyond the edges of reference picture data, where values lying outside shall be determined by edge extension.

If **state**[MV_PRECISION] == 1, upconverted values shall be derived directly from the the half-pixel interpolated array $upref$, which shall be calculated as per Section 15.8.11.

If **state**[MV_PRECISION] == 2 or **state**[MV_PRECISION] == 3, upconverted values shall be derived by linear interpolation from the half-pixel interpolated array.

The sub-pixel prediction process shall be defined as follows:

| $subpel\_predict(ref, c, u, v)$ : | **Ref** |
|---|---|
| $upref = interp2by2(ref, c)$ | 15.8.11 |
| $hu = u \gg (\textbf{state}[\text{MV\_PRECISION}] - 1)$ | |
| $hv = v \gg (\textbf{state}[\text{MV\_PRECISION}] - 1)$ | |
| $ru = u - (hu \ll (\textbf{state}[\text{MV\_PRECISION}] - 1))$ | |
| $rv = v - (hv \ll (\textbf{state}[\text{MV\_PRECISION}] - 1))$ | |
| $w00 = (2^{\textbf{state}[\text{MV\_PRECISION}]-1} - rv) * (2^{\textbf{state}[\text{MV\_PRECISION}]-1} - ru)$ | |
| $w01 = (2^{\textbf{state}[\text{MV\_PRECISION}]-1} - rv) * ru$ | |
| $w10 = rv * (2^{\textbf{state}[\text{MV\_PRECISION}]-1} - ru)$ | |
| $w11 = rv * ru$ | |
| $xpos = \text{clip}(hu, 0, \text{width}(upref) - 1)$ | |
| $xpos1 = \text{clip}(hu + 1, 0, \text{width}(upref) - 1)$ | |
| $ypos = \text{clip}(hv, 0, \text{height}(upref) - 1)$ | |
| $ypos1 = \text{clip}(hv + 1, 0, \text{height}(upref) - 1)$ | |
| $val = w00 * upref[ypos][xpos] + w01 * upref[ypos][xpos1] + w10 * upref[ypos1][xpos] + w11 * upref[ypos1][xpos1]$ | |
| **if** $(\textbf{state}[\text{MV\_PRECISION}] > 1)$: | |
|   **return** $(val + 2^{2*\textbf{state}[\text{MV\_PRECISION}]-3}) \gg (2 * \textbf{state}[\text{MV\_PRECISION}] - 2)$ | |
| **else**: | |
|   **return** $val$ | |

**Note:** $hu$ and $hv$ represent the half-pixel part of the sub-pixel position $(u, v)$.

$ru$ and $rv$ represent the remaining sub-pixel component of the position. $ru$ and $rv$ satisfy

$$0 \leq ru, rv < 2^{\textbf{state}[\text{MV\_PRECISION}]-1}$$

The four weights $w00, w01, w10$ and $w11$ sum to $2^{2*\textbf{state}[\text{MV\_PRECISION}]-2}$, and hence the upconverted value is returned to the initial pixel ranges in the pseudocode above.

Note that the remainder values $ru$ and $rv$, and hence the four weight values, only depend on the motion vectors. This is because $u$ and $v$ have been computed by scaling the picture coordinates by $2^{\textbf{state}[\text{MV\_PRECISION}]}$ and adding the motion vector.

In particular constant linear interpolation weights are applied throughout a block when block motion is used. Likewise, the necessity of clipping the ranges of $xpos$, $ypos$ etc can be determined in advance for each block by checking whether any corner of the reference block will fall outside of the reference picture area. In most cases it will not and clipping will not be required for motion compensating most blocks.

For half-pixel motion vectors (**state**[MV_PRECISION] is 1), the majority of the pseudocode is redundant, and the return value $val$ will merely be the value at position $(u, v)$, clipped to the ranges of the upconverted reference.

### 15.8.11 Half-pixel interpolation

This section defines the $interp2by2(ref, c)$ process for generating an upconverted reference array $upref$ representing a half-pixel interpolation of the reference array $ref$ for component $c$ (Y, C1, or C2).

$upref$ shall be created in two stages. The first stage shall upconvert vertically. The second stage shall upconvert horizontally.

$upref$ shall have width $2 * \text{width}(ref) - 1$ and height $2 * \text{height}(ref) - 1$, so that all edge values shall be copied from the original array and not interpolated.

The interpolation filter shall be the 8-tap symmetric filter with taps as defined in Figure 15.2.

Where coefficients used in the filtering process fall outside the bounds of the reference array, values shall be

| Tap | $t[0]$ | $t[1]$ | $t[2]$ | $t[3]$ |
|---|---|---|---|---|
| Value | 21 | -7 | 3 | -1 |

Figure 15.2: Interpolation filter coefficients

supplied by edge extension.

The overall process shall be defined as follows:

```
$interp2by2(ref, c)$ :                                                        Ref
  **if** $(c == Y)$:
    $bit\_depth = \textbf{state}[\text{LUMA\_DEPTH}]$
  **else**:
    $bit\_depth = \textbf{state}[\text{CHROMA\_DEPTH}]$
  **for** $q = 0$ **to** $2 * \text{height}(ref) - 2$:
    **if** $(q\%2 == 0)$:
      **for** $p = 0$ **to** $\text{width}(ref) - 1$:
        $ref2[q][p] = ref[q//2][p]$
    **else**:
      **for** $p = 0$ **to** $\text{width}(ref) - 1$:
        $ref2[q][p] = 16$
        **for** $i = 0$ **to** $3$:
          $ypos = (q - 1)//2 - i$
          $ref2[q][p] += t[i] * ref[\text{clip}(ypos, 0, \text{height}(ref) - 1)][p]$
          $ypos = (q + 1)//2 + i$
          $ref2[q][p] += t[i] * ref[\text{clip}(ypos, 0, \text{height}(ref) - 1)][p]$
        $ref2[q][p] \gg= 5$
        $ref2[q][p] = \text{clip}(ref2[q][p], -2^{bit\_depth-1}, 2^{bit\_depth-1} - 1)$
  **for** $q = 0$ **to** $2 * \text{height}(ref) - 2$:
    **for** $p = 0$ **to** $2 * \text{width}(ref) - 2$:
      **if** $(p\%2 == 0)$:
        $upref[q][p] = ref2[q][p//2]$
      **else**:
        $upref[q][p] = 16$
        **for** $i = 0$ **to** $3$:
          $xpos = (p - 1)//2 - i$
          $upref[q][p] += t[i] * ref2[q][\text{clip}(xpos, 0, \text{width}(ref) - 1)]$
          $xpos = (p + 1)//2 + i$
          $upref[q][p] += t[i] * ref2[q][\text{clip}(xpos, 0, \text{width}(ref) - 1)]$
        $upref[q][p] \gg= 5$
        $upref[q][p] = \text{clip}(upref[q][p], -2^{bit\_depth-1}, 2^{bit\_depth-1} - 1)$
```

**Note:** While this filter may appear to be variable separable, the integer rounding and clipping processes prevent this being so. Note also that the clipping process for filtering terms implies that the upconversion uses edge-extension at the array edges, consistent with the edge-extension used in motion-compensation itself.

## 15.9 Clipping

Picture data must be clipped prior to being output or being used as a reference:

| *clip_picture*() : | **Ref** |
|---|---|
| **for each** $c$ **in** $Y, C1, C2$: | |
|   *clip_component*(**state**[CURRENT_PICTURE][$c$]) | |

| *clip_component*(*comp_data*, $c$) : | **Ref** |
|---|---|
| **if** $(c == Y)$: | |
|   *bit_depth* = **state**[LUMA_DEPTH] | |
| **else**: | |
|   *bit_depth* = **state**[CHROMA_DEPTH] | |
| **for** $y = 0$ **to** height(*comp_data*) $- 1$: | |
|   **for** $x = 0$ **to** width(*comp_data*) $- 1$: | |
|     *data* = clip(*comp_data*[$y$][$x$], $-2^{bit\_depth-1}, 2^{bit\_depth-1} - 1$) | |

**Note:** Note that clipping is incorporated into motion compensation, so that strictly speaking additional clipping is only required for intra pictures.

## 15.10 Video output ranges

Video output data ranges are deemed to be non-negative, so that the offset and excursion values may be applied by subsequent processing. Since decoded video data is bipolar, it must be suitably offset before output:

| *offset_output_data*(*picture_data*) : | **Ref** |
|---|---|
| **for each** $c$ **in** $Y, C1, C2$: | |
|   **if** $(c == Y)$: | |
|     *bit_depth* = **state**[LUMA_DEPTH] | |
|   **else**: | |
|     *bit_depth* = **state**[CHROMA_DEPTH] | |
|   *comp* = *picture_data*[$c$] | |
|   **for** $y = 0$ **to** height(*comp*) $- 1$: | |
|     **for** $x = 0$ **to** width(*comp*) $- 1$: | |
|       *comp*[$y$][$x$] $+= 2^{bit\_depth-1}$ | |

# A Data encodings

Data shall be encoded in the Dirac bitstream using four basic methods:

- fixed-length bit-wise codings,
- fixed-length byte-wise codings,
- variable-length codes,
- arithmetic encoding.

This annex defines how data shall be encoded in the Dirac stream and how sequences of bits shall be extracted as values of various types using the aforementioned fundamental data coding types. The extraction of arithmetically encoded data shall require the use of the arithmetic decoding engine defined in Annex B.2.

## A.1 Bit-packing and data input

This section defines the operation of the *read_bit*(), *read_byte*() and *byte_align*() functions used for direct access to the Dirac stream.

The stream data shall be accessed byte by byte, and a decoder is deemed to maintain a copy of the current byte, **state**[CURRENT_BYTE], and an index to the next bit (in the byte) to be read, **state**[NEXT_BIT]. **state**[NEXT_BIT] shall be an integer from 0 (least-significant bit) to 7 (most-significant bit). Bits within bytes shall be accessed from the msb first to the lsb.

### A.1.1 Reading a byte

The *read_byte*() function shall perform the following steps:

1. Set **state**[NEXT_BIT] = 7
2. Set **state**[CURRENT_BYTE] to the next unread byte in the stream

### A.1.2 Reading a bit

The *read_bit*() function shall be defined as follows:

| *read_bit*() : | **Ref** |
|---|---|
| $bit = ($**state**[CURRENT_BYTE] $\gg$ **state**[NEXT_BIT]$)\&1$ | |
| **state**[NEXT_BIT]$-=1$ | |
| **if** (**state**[NEXT_BIT] $< 0$): | |
|     **state**[NEXT_BIT] $= 7$ | |
|     *read_byte*() | |
| **return** *bit* | |

### A.1.3 Byte alignment

The *byte_align*() function shall be used to discard data in the current byte and begin data access at the next byte, unless input is already at the beginning of a byte. It shall be defined as follows:

| *byte_align*() : | **Ref** |
|---|---|
| **if** (**state**[NEXT_BIT]$!=7$): | |
|     *read_byte*() | |

## A.2 Parsing of fixed-length data

Dirac defines three fixed length data encodings as follows:

### A.2.1 Boolean

The *read_bool*() function shall return **True** if 1 is read from the stream and **False** otherwise. The *read_bool*() function shall be defined as follows:

| *read_bool*() : | **Ref** |
|---|---|
| **if** (*read_bit*() == 1): | |
|     **return True** | |
| **else**: | |
|     **return False** | |

### A.2.2 n-bit literal

An $n$-bit number in literal format shall be decoded by extracting $n$ bits in order, using the *read_bit*() function (Section A.1.2) and placing the first bit in the leftmost position, the second bit in the next position and so on. The resulting value shall be interpreted as an unsigned integer.

The *read_nbits*() function shall be defined as follows:

| *read_nbits*($n$) : | **Ref** |
|---|---|
| $val = 0$ | |
| **for** $i = 0$ **to** $n - 1$: | |
|     $val \ll= 1$ | |
|     $val += read\_bit()$ | A.1.2 |
| **return** *val* | |

### A.2.3 $n$-byte unsigned integer literal

The *read_uint_lit*() function shall be defined as follows:

| *read_uint_lit*($n$) : | **Ref** |
|---|---|
| *byte_align*() | A.1.3 |
| **return** *read_nbits*($8 * n$) | A.2.2 |

## A.3 Variable-length codes

Variable-length codes shall be used in three ways in the Dirac stream:

1. The first use shall be for direct encoding of header values into the stream.
2. The second use shall be for entropy coding of motion data and coefficients, where arithmetic coding is used.
3. The third use shall be for binarisation in the arithmetic encoding/decoding process so that integer values may be coded and decoded using a binary arithmetic coding engine. This is defined in Annex A.4.

When used for coding motion data and coefficients, VLCs shall be employed within a data block of known length. It is possible to gain additional compression by early termination: maintaining a count of remaining bits, and returning default values when this length is exceeded. This shall be achieved by use of the *read_bitb*(), *read_boolb*(), *read_uintb*() and *read_sintb*() for reading values from data blocks.

**Note:** A similar early termination facility is a used for arithmetic decoding.

### A.3.1 Data input for bounded block operation

This section specifies the operation of the *read_bitb*() process for reading bits from a block of known size, and the *flush_inputb*() process for discarding the remainder of a block of data.

These processes shall use **state**[BITS_LEFT] to determine the number of bits left to the end of the block.

The *read_bitb*() function shall be defined as follows:

| *read_bitb*() : | **Ref** |
|---|---|
| **if** (**state**[BITS_LEFT] == 0): | |
| **return** 1 | |
| **else**: | |
| **state**[BITS_LEFT] $-=1$ | |
| **return** *read_bit*() | |

When all bits in the block have been read, then *read_bitb*() shall return 1 by default.

It is possible that not all data in a block is exhausted after a sequence of read operations. At the end of a sequence of read operations, the decoder shall flush the block.

The *flush_inputb*() process shall be defined as follows:

| *flush_inputb*() : | **Ref** |
|---|---|
| **while** (**state**[BITS_LEFT] $> 0$): | |
| *read_bit*() | |
| **state**[BITS_LEFT] $-=1$ | |

### A.3.2 Unsigned interleaved exp-Golomb codes

This section defines the unsigned interleaved exp-Golomb data format and the operation of the *read_uint*() and the *read_uintb*() functions.

Unsigned interleaved exp-Golomb data shall be decoded to produce unsigned integer values.The format shall consist of two interleaved parts, and each code shall be an odd number, $2K+1$ bits in length.

The $K+1$ bits in the even positions (counting from zero) shall be the “follow” bits, and the $K$ bits in the odd positions shall be the “data” bits $b_i$ that are used to construct the decoded value itself. A follow bit value of 0 shall indicate a subsequent data bit, whereas a follow bit value of 1 shall terminate the code, a typical sequence being

$$0 \quad x_{K-1} \quad 0 \quad x_{K-2} \ldots 0 \quad x_0 \quad 1$$

The data bits $x_{K-1}, x_{K-2}, \ldots, x_0$ shall form the binary representation of the first $K$ bits of the $(K+1)$-bit number $N+1$, where $N$ is the number to be decoded, i.e.:

$$N+1 = 1x_{K-1}x_{K-2}\ldots x_0 \text{ (base 2) } = 2^K + \sum_{i=0}^{K-1} 2^i * x_i$$

Table A.1 shows encodings of the values 0–9.

Although apparently complex, the interleaving ensures that the code has a very simple decoding loop.

The *read_uint*() function shall return an unsigned integer value and shall be defined as follows:

| Bit sequence | Decoded value |
|---|---|
| 1 | 0 |
| 0 0 1 | 1 |
| 0 1 1 | 2 |
| 0 0 0 0 1 | 3 |
| 0 0 0 1 1 | 4 |
| 0 1 0 0 1 | 5 |
| 0 1 0 1 1 | 6 |
| 0 0 0 0 0 0 1 | 7 |
| 0 0 0 0 0 1 1 | 8 |
| 0 0 0 1 0 0 1 | 9 |

Table A.1: Example conversions from unsigned interleaved exp-Golomb-coded values to unsigned integers

| *read_uint*() : | **Ref** |
|---|---|
| $value = 1$ | |
| **while** (*read_bit*() == 0): | |
|   $value \ll= 1$ | |
|   **if** (*read_bit*() == 1): | |
|     $value+ = 1$ | |
| $value- = 1$ | |
| **return** *value* | |

**Note:** Conventional exp-Golomb coding places all follow bits at the beginning as a prefix. This is easier to read, but requires that a count of the prefix length be maintained. Values can only be decoded in two loops – the prefix followed by the data bits. Interleaved exp-Golomb coding allows values to be decoded in a single loop, without the need for a length count.

The *read_uintb*() function is identical to *read_uint*() except that the block-bounded read operation is employed, and shall be defined as follows:

| *read_uintb*() : | **Ref** |
|---|---|
| $value = 1$ | |
| **while** (*read_bitb*() == 0): | |
|   $value \ll= 1$ | |
|   **if** (*read_bitb*() == 1): | |
|     $value+ = 1$ | |
| $value- = 1$ | |
| **return** *value* | |

**Note:** When **state**[BITS_LEFT] == 0, all subsequent values read by *read_uintb*() will be 0.

### A.3.3 Signed interleaved exp-Golomb

This section defines the signed interleaved exp-Golomb data format and the operation of the *read_sint*() and *read_sintb*() functions.

The code for the signed interleaved exp-Golomb data format consists of the unsigned interleaved exp-Golomb code for the magnitude, followed by a sign bit for non-zero values, as shown in the table below:

| Bit sequence | Decoded value |
|---|---|
| 0 0 0 1 1 1 | -4 |
| 0 0 0 0 1 1 | -3 |
| 0 1 1 1 | -2 |
| 0 0 1 1 | -1 |
| 1 | 0 |
| 0 0 1 0 | 1 |
| 0 1 1 0 | 2 |
| 0 0 0 0 1 0 | 3 |
| 0 0 0 1 1 0 | 4 |

The *read_sint*() function shall be defined as follows.

```
read_sint() :                                  Ref
  $value = read\_uint()$
  if ($value != 0$):
    if ($read\_bit() == 1$):
      $value = -value$
  return $value$
```

The *read_sintb*() function is identical to *read_sint*() except that the block-bounded read operation is employed, and shall be defined as follows:

```
read_sintb() :                                 Ref
  $value = read\_uintb()$
  if ($value != 0$):
    if ($read\_bitb() == 1$):
      $value = -value$
  return $value$
```

**Note:** When **state**[BITS_LEFT] == 0, all subsequent values read by *read_sintb*() will be 0.

## A.4 Parsing of arithmetic-coded data

This section defines the operations for reading arithmetic-coded data. These operations shall make use of the elementary arithmetic coding functions defined by the arithmetic decoding engine defined in Annex B.2.

Arithmetically-coded data is present in the Dirac stream in data blocks which shall consist of a whole number of bytes and which shall be byte aligned. Where arithmetic coding is used, each such block shall be preceded by data which includes a length code *length*, which shall be equal to the length in bytes of the data block.

The function *initialise_arithmetic_decoding*(*length*) (Section B.2.2) shall then initialises the arithmetic decoder. Once the arithmetic decoder is initialised, boolean and integer values may be extracted.

After all values in a particular arithmetic coded block have been parsed, any remaining data shall be flushed using the *flush_inputb*() process (Section A.3.1).

### A.4.1 Context probabilities

Values shall be extracted by using binary context probabilities. A context is a decoding state, representing the set of all data decoded so far.

A context probability shall be a 16 bit unsigned integer value representing the probability of a bit being 0 in a given context, where zero probability is represented by 0x0, and equal likelihood by 0x8000. The process for initializing and updating context probabilities shall be as defined in annex B.2.2 and B.2.6.

**Note:** Probability 1, or certainty, would be represented by the 17-bit number 0x10000. This value, and probability 0 (0x0), can never be attained due to the operation of the probability update process (Annex B.2.6).

Different context probabilities shall be employed for extracting binary values, based on the values of previously decoded data. Each context probability shall be updated by the arithmetic decoding engine to track statistics after it has been used to extract a value.

The set of contexts probabilities shall be defined by **state**[CONTEXT_PROBS], and an individual context shall be accessed via a keyword label i.e. **state**[CONTEXT_PROBS] is a map and the context value shall be **state**[CONTEXT_PROBS][$l$] for a label $l$.

The array of context probability labels to be used in arithmetic decoding shall be passed to the arithmetic decoding engine at initialization (annex B.2.2).

### A.4.2 Arithmetic decoding of boolean values

Given a context probability lable $l$, the arithmetic decoding engine shall support a function $read\_boola(l)$, specified in Section B.2.4, which shall returna boolean value.

### A.4.3 Arithmetic decoding of integer values

This section defines the operation of the $read\_sinta(context\_prob\_set)$ and $read\_uinta(context\_prob\_set)$ functions for extracting integer values from a block of arithmetically coded data.

#### A.4.3.1 Binarisation and contexts

Signed and unsigned integer values shall be coded by first converting to binary form by using interleaved exp-Golomb binarisation as per Section A.3. The $read\_sinta()$ and $read\_uinta()$ processes shall be identical to the $read\_sint()$ and $read\_uint()$ processes, except that instances of $read\_bit()$ shall be replaced by instances of $read\_boola()$ (Section B.2.4) using a suitable context for each bit.

$read\_sinta()$ and $read\_uinta()$ shall be provided with a map $context\_prob\_set$, which shall consist of three parts:

1. an array of follow context indices, $context\_prob\_set[FOLLOW]$
2. a single data context index, $context\_prob\_set[DATA]$
3. a sign context index, $context\_prob\_set[SIGN]$ (ignored for unsigned integer decoding)

Each follow context shall be used for decoding the corresponding follow bit, with the last follow context being used for all subsequent follow bits also (if any).

The follow context selection function $follow\_context()$ shall be defined as follows:

| $follow\_context(index, context\_prob\_set)$ : | **Ref** |
|---|---|
| $pos = \min(index, length(context\_prob\_set[FOLLOW]) - 1)$ | |
| $ctx\_label = context\_prob\_set[FOLLOW][pos]$ | |
| **return** $ctx\_label$ | |

#### A.4.3.2 Unsigned integer decoding

The $read\_uinta()$ function shall be defined as follows:

| $read\_uinta(context\_prob\_set)$ : | **Ref** |
|---|---|
| $value = 1$ | |
| $index = 0$ | |
| **while** $(read\_boola(follow\_context(index, context\_prob\_set)) ==$ **False**): | A.4.3.1 |
|  $value \ll= 1$ | |
|  **if** $(read\_boola(context\_prob\_set[DATA]) ==$ **True**): | |
|   $value+ = 1$ | |
|  $index+ = 1$ | |
| $value- = 1$ | |
| **return** $value$ | |

### A.4.3.3 Signed integer decoding

$read\_sinta()$ decodes first the magnitude then the sign, as necessary:

| $read\_sinta(context\_prob\_set)$ : | **Ref** |
|---|---|
| $value = read\_uinta(context\_prob\_set)$ | |
| **if** $(value! = 0)$: | |
|  **if** $(read\_boola(context\_prob\_set[SIGN]) ==$ **True**): | |
|   $value = -value$ | |
| **return** $value$ | |

# B Arithmetic Coding

This annex has three parts:

1. a description of the principles of arithmetic coding,
2. a specification of the arithmetic decoding engine used in Dirac, and
3. a description of a compatible arithmetic encoder.

## B.1 Arithmetic coding principles (Informative)

This section provides an introduction to the principles underlying arithmetic coding. It briefly describes binary arithmetic coding, that is the coding of binary symbols, which is used in Dirac.

Arithmetic coding is an extremely powerful form of entropy coding, which can closely approximate the Shannon information limit for given data. Arithmetic encoding consists of a state machine that is fed with a sequence of symbols together with an estimate of each symbols probability. For each input symbol the arithmetic coding engine updates its state and output a number of coded bits. The number of output bits for each input symbol depends on the internal state and on the current probabilities the symbols that are coded, and can range from zero to many bits.

The variable number of coded bits output for each input symbol complicates the implementation but is essential to the optimal nature of arithmetic coding. Consider a binary symbol $b$, with $p(b = \mathbf{False}) = p_0$ and $p(b = \mathbf{True}) = 1 - p_0$. The entropy of $b$ is the expected number of bits required to encode $b$, and is equal to

$$e(p_0) = p_0 \log_2(1/p_0) + (1 - p_0) \log_2(1/(1 - p_0))$$

If $e(p_0)$ is plotted against $p_0$, it can be seen that if $p_0$ is not equal to 0.5 exactly, $e(p_0) < 1$. This means that an optimal binary entropy encoder that operates symbol by symbol, cannot produce an output for every symbol.

### B.1.1 Interval division and scaling

The fundamental idea of arithmetic coding is interval division and scaling. An arithmetic code can be thought of as a single number lying in an interval determined by the sequence of values being coded. For simplicity, this discussion describes binary arithmetic coding, but larger symbol alphabets can be treated in an analogous manner.

Let us begin with the interval $[0, 1)$, and suppose that we know (or have some estimate of) the probability of **False**, $p_0$. Conceptually we divide the interval into the intervals $[0, p0)$ and $[p_0, 1)$. Suppose we code **False** as the first symbol. In this case the interval is changed to $[0, p_0)$. If we code **True**, then the interval is changed to $[p_0, 1)$. After coding a number of symbols we arrive at an interval $[low, high)$. To code the next symbol we partition this interval into $[low, low + p_0(high - low))$ and $[low + p_0(high - low), high)$, and if the symbol is **False** we choose the first interval, otherwise the second.

For any integer $N$, this process clearly partitions the interval $[0, 1)$ into a set of disjoint intervals that correspond to all the input sequences of $N$ bits. Identifying such a bit sequence is equivalent to choosing a value in the corresponding interval, and for an interval width $w$ that in general requires

$$\lceil \log_2(1/w) \rceil$$

bits. With static probabilities, on average,

$$w = p_0^{Np_0}(1 - p_0)^{N(1-p_0)}$$

resulting in

$$\lceil Ne(p_0) \rceil$$

being used, demonstrating the near-optimality of arithmetic coding. Moreover, it is clearly possible to create an adaptive arithmetic code by changing the estimate of $p_0$ based on previously coded data.

### B.1.2 Finite precision arithmetic

As it stands, the procedure outlined in the previous section has a number of drawbacks for practical application. Firstly, it requires unlimited precision to scale the interval, which is not available in real hardware or software. Secondly, it only produces an output when all values have been coded. These problems are addressed by renormalisation and progressive output: periodically rescaling the interval, and outputting the most significant bits of *low* and *high* whenever they agree.

For example, if we know that

$$\begin{aligned} low &= b0xyz... \\ high &= b0pqr... \end{aligned}$$

then we can output 0, since this must prefix any value lying in the interval, and shift *low* and *high* to get $low = bxyz...$ and $high = bpqr....$ This has the effect of doubling the interval from 0 ($x \mapsto 2x$). Likewise if

$$\begin{aligned} low &= b1xyz... \\ high &= b1pqr... \end{aligned}$$

we can output 1 and shift to get $low = bxyz...$ and $high = bpqr...$ again: this is equivalent to doubling the interval from 1 ($x \mapsto 2x - 1$).

One problem remains: suppose the interval resolutely sits on the fence, straddling $\frac{1}{2}$ whilst getting smaller and smaller, with the most significant bits of low and high staying as 0 and 1 respectively. In this case, when the straddle is finally resolved, *low* and *high* will both be of the form $b10000...xyz$ or $b01111...pqr$.

The resolution strategy is to again rescale *low* and *high*, but this time double from $\frac{1}{2}$ (i.e. $x \mapsto 2x - \frac{1}{2}$), and keep a count of the number $k$ of times this is done, as this is the number of carry bits that are required. When the straddle is resolved as 1, then 1 followed by $k$ zero bits is output, otherwise 0 followed by k 1s is output. This ensures that the output exactly represents the small straddling interval.

A decoder can determine a symbol as soon as it has sufficient bits to distinguish whether a value lies in one interval or another. If constraints are placed on the size of the smallest interval before renormalisation (for example, by renormalising often enough and by having a fixed smallest allowable probability), then this can be accomplished within a fixed word width.

### B.1.3 Symbol probability estimation

## B.2 Arithmetic decoding engine

This section is a normative specification of the operation of the arithmetic decoding engine and the processes for using it to extract binary values from coded streams.

The arithmetic decoding engine shall consist of two elements:

1. a collection of state variables representing the state of the arithmetic decoder (Section B.2.2)
2. a function for extracting binary values from the decoder and updating the decoder state (Section B.2.4)

### B.2.1 State and contexts

The arithmetic decoder state shall consist of the following decoder state variables:

- **state**[LOW], an integer representing the beginning of the current coding interval.
- **state**[RANGE], an integer representing the size of the current coding interval.
- **state**[CODE], an integer within the interval from **state**[LOW] to **state**[LOW] + **state**[RANGE] − 1, determined from the encoded bitstream.

- **state**[BITS_LEFT], a decrementing count of the number of bits yet to be read in
- **state**[CONTEXT_PROBS], a map of all the contexts used in the Dirac decoder.

A context *context* shal be a 16 bit unsigned interger value which encapsulates the probability of zero symbol in the stream, represented as such that

$$0 < context[prob0] \leq \text{0xFFFF}$$

Contexts shall be accessed by decoding functions via a context label passed to the function.

### B.2.2 Initialisation

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

| *initialise_arithmetic_decoding*(*ctx_labels*) : | **Ref** |
|---|---|
| **state**[LOW] = 0x0 | |
| **state**[RANGE] = 0xFFFF | |
| **state**[CODE] = 0x0 | |
| **for** $i = 0$ **to** 15: | |
|     **state**[CODE] <<= 1 | |
|     **state**[CODE]+ = *read_bitb*() | |
| *init_context_probs*(*ctx_labels*) | |

The *init_context_probs*() process shall be defined as follows:

| *init_context_probs*(*ctx_labels*) : | **Ref** |
|---|---|
| **for** $i = 0$ **to** length(*ctx_labels*) − 1: | |
|     **state**[CONTEXT_PROBS][*ctx_labels*[*i*]] = 0x8000 | |

**Note:** The value 0x8000 represents 1/2 or equal likelihood for binary values.

### B.2.3 Data input

The arithmetic decoding process shall access data in a contiguous block of bytes whose size is equal to **state**[BITS_LEFT], this value having been set prior to decoding the block. The bits in this block shall be sufficient to allow for the decoding of all coefficients. However, the specification of arithmetic decoding operations in this section may occasionally cause further bits to be read, even though they are not required for determining decoded values. For this reason the bounded-block read function *read_bitb*() (Section A.3.1) shall be used for data access.

Since the length of arithmetically coded data elements is given in bytes within the Dirac stream, there may be bits left unread when all values have been extracted. These shall be flushed as described in Section A.3.1. Since arithmetically coded blocks are byte-aligned and a whole number of bytes, this aligns data input with the beginning of the byte after the arithmetically coded data i.e. at the end of the data chunk. *flush_inputb*() shall always be called at the end of decoding an arithmetically coded data element.

### B.2.4 Decoding boolean values

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 calling decoder function shall determine which contexts are passed to the arithmetic decoding operations.

This section defines the operation of the *read_boola*() function for extracting a boolean value from the Dirac stream. The function shall be defined as follows:

| $read\_boola(context\_label)$ : | **Ref** |
|---|---|
| $prob\_zero = \textbf{state}[\text{CONTEXT\_PROBS}][context\_label]$ | |
| $count = \textbf{state}[\text{CODE}] - \textbf{state}[\text{LOW}]$ | |
| $range\_times\_prob = (\textbf{state}[\text{RANGE}] * prob\_zero) \gg 16$ | |
| **if** $(count >= range\_times\_prob)$: | |
|   $value =$ **True** | |
|   $\textbf{state}[\text{LOW}] += range\_times\_prob$ | |
|   $\textbf{state}[\text{RANGE}] -= range\_time\_prob$ | |
| **else**: | |
|   $value =$ **False** | |
|   $\textbf{state}[\text{RANGE}] = range\_times\_prob$ | |
| $update\_context(\textbf{state}[\text{CONTEXT\_PROBS}][context\_label], value)$ | B.2.6 |
| **while** $(\textbf{state}[\text{RANGE}] <= 0\text{x}4000)$: | |
|   $renormalise()$ | B.2.5 |
| **return** $value$ | |

**Note:** The function scales the probability of the symbol 0 or **False** from the decoding context so that if this probability were 1, then the interval would equal that between **state**[LOW] and

$$high = \textbf{state}[\text{LOW}] + \textbf{state}[\text{RANGE}] - 1$$

and *count* is set to the normalised cut-off between 0/**False** and 1/**True** within this range.

### B.2.5 Renormalisation

Renormalisation shall be applied to stop the arithmetic decoding engine from losing accuracy. Renormalisation shall be applied while the range is less than or equal to a quarter of the total available 16-bit range (0x4000).

Renormalisation shall double the interval and read a bit into the codeword. The $renormalise()$ function shall be defined as follows:

| $renormalise()$ : | **Ref** |
|---|---|
| **if** $(((\textbf{state}[\text{LOW}] + \textbf{state}[\text{RANGE}] - 1) \wedge \textbf{state}[\text{LOW}]) >= 0\text{x}8000)$: | |
|   $\textbf{state}[\text{CODE}] \wedge= 0\text{x}4000$ | |
|   $\textbf{state}[\text{LOW}] \wedge= 0\text{x}4000$ | |
| $\textbf{state}[\text{LOW}] <<= 1$ | |
| $\textbf{state}[\text{RANGE}] <<= 1$ | |
| $\textbf{state}[\text{LOW}] \&= 0\text{xFFFF}$ | |
| $\textbf{state}[\text{CODE}] <<= 1$ | |
| $\textbf{state}[\text{CODE}] += read\_bitb()$ | |
| $\textbf{state}[\text{CODE}] \&= 0\text{xFFFF}$ | |

**Note:** For convenience let $low = \textbf{state}[\text{LOW}]$ and $high = \textbf{state}[\text{LOW}] + \textbf{state}[\text{RANGE}] - 1$ represent the upper and lower bounds of the interval. If the range is $<= 0\text{x}4000$ then one of three possibilities must obtain:

1. the msbs of *low* and *high* are both 0
2. the msbs of *low* and *high* are both 1
3. $low = b01...$, $high = b10....$, and the interval straddles the half-way point 0x8000.

The renormalisation process has the effect that: in case 1, the interval $[low, high]$ is doubled from 0 (i.e. $x \mapsto 2 * x$); in case 2 it is doubled from 1 (i.e. $x \mapsto 2 * x - 1$); and in case 3 it is doubled from 1/2 (i.e. $x \mapsto 2x - 0.5$).

### B.2.6 Updating contexts

Context probabilities shall be updated according to a probability look-up table **state**[PROB_LUT], which supplies a value for decrementing or incrementing the probability of zero based on the first 8 bits of its current value, according to Table B.1.

The *update_context*() process shall be defined as follows:

| *update_context*(*ctx_prob*, *value*) : | **Ref** |
|---|---|
|   **if** ($value$ == **True**): | |
|     $ctx\_prob -=$ **state**[PROB_LUT][$ctx\_prob \gg 8$] | Table B.1 |
|   **else**: | |
|     $ctx\_prob +=$ **state**[PROB_LUT][$255 - (ctx\_prob \gg 8)$] | Table B.1 |

The lookup table used for updating context probabilities shall be as defined in Table B.1. below. The lookup table entries are arranged in raster scan order with rows of thirteen entries. The entry corresponding to index zero is in the top left hand corner, the index increments by one from left to right and by thirteen from top to bottom, the entry corresponding to index 255 is on the right hand side of the last row.

| **state[PROB_LUT][] (indexes 0 to 255)** | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0, | 2, | 5, | 8, | 11, | 15, | 20, | 24, |
| 29, | 35, | 41, | 47, | 53, | 60, | 67, | 74, |
| 82, | 89, | 97, | 106, | 114, | 123, | 132, | 141, |
| 150, | 160, | 170, | 180, | 190, | 201, | 211, | 222, |
| 233, | 244, | 256, | 267, | 279, | 291, | 303, | 315, |
| 327, | 340, | 353, | 366, | 379, | 392, | 405, | 419, |
| 433, | 447, | 461, | 475, | 489, | 504, | 518, | 533, |
| 548, | 563, | 578, | 593, | 609, | 624, | 640, | 656, |
| 672, | 688, | 705, | 721, | 738, | 754, | 771, | 788, |
| 805, | 822, | 840, | 857, | 875, | 892, | 910, | 928, |
| 946, | 964, | 983, | 1001, | 1020, | 1038, | 1057, | 1076, |
| 1095, | 1114, | 1133, | 1153, | 1172, | 1192, | 1211, | 1231, |
| 1251, | 1271, | 1291, | 1311, | 1332, | 1352, | 1373, | 1393, |
| 1414, | 1435, | 1456, | 1477, | 1498, | 1520, | 1541, | 1562, |
| 1584, | 1606, | 1628, | 1649, | 1671, | 1694, | 1716, | 1738, |
| 1760, | 1783, | 1806, | 1828, | 1851, | 1874, | 1897, | 1920, |
| 1935, | 1942, | 1949, | 1955, | 1961, | 1968, | 1974, | 1980, |
| 1985, | 1991, | 1996, | 2001, | 2006, | 2011, | 2016, | 2021, |
| 2025, | 2029, | 2033, | 2037, | 2040, | 2044, | 2047, | 2050, |
| 2053, | 2056, | 2058, | 2061, | 2063, | 2065, | 2066, | 2068, |
| 2069, | 2070, | 2071, | 2072, | 2072, | 2072, | 2072, | 2072, |
| 2072, | 2071, | 2070, | 2069, | 2068, | 2066, | 2065, | 2063, |
| 2060, | 2058, | 2055, | 2052, | 2049, | 2045, | 2042, | 2038, |
| 2033, | 2029, | 2024, | 2019, | 2013, | 2008, | 2002, | 1996, |
| 1989, | 1982, | 1975, | 1968, | 1960, | 1952, | 1943, | 1934, |
| 1925, | 1916, | 1906, | 1896, | 1885, | 1874, | 1863, | 1851, |
| 1839, | 1827, | 1814, | 1800, | 1786, | 1772, | 1757, | 1742, |
| 1727, | 1710, | 1694, | 1676, | 1659, | 1640, | 1622, | 1602, |
| 1582, | 1561, | 1540, | 1518, | 1495, | 1471, | 1447, | 1422, |
| 1396, | 1369, | 1341, | 1312, | 1282, | 1251, | 1219, | 1186, |
| 1151, | 1114, | 1077, | 1037, | 995, | 952, | 906, | 857, |
| 805, | 750, | 690, | 625, | 553, | 471, | 376, | 255 |

Table B.1: Look-up table for context probability adaptation

### B.2.7 Efficient implementation (Informative)

The decoding operations defined in the preceding sections correspond closely to the descriptions of arithmetic coding principles contained in the academic literature. More efficient implementations are certainly possible, both for hardware and software. This section describes some simple techniques.

#### B.2.7.1 Change of variables

There is in fact no need for the decoder to keep track of both **state**[LOW] and **state**[CODE], since the test is made against the difference of these values, i.e. be defined as:

$$\mathbf{state}[\text{CODE\_MINUS\_LOW}] = \mathbf{state}[\text{CODE}] - \mathbf{state}[\text{LOW}]$$

So only this difference variable need be tracked. Since **state**[LOW] is initialised to zero, **state**[CODE_MINUS_LOW] is initialised just like **state**[CODE]. The *read_boola* then is re-written as:

| $read\_boola(context\_label)$ : | **Ref** |
|---|---|
| $prob\_zero = \mathbf{state}[\text{CONTEXT\_PROBS}][context\_label]$ | |
| $range\_times\_prob = (\mathbf{state}[\text{RANGE}] * prob\_zero) \gg 16$ | |
| **if** $(\mathbf{state}[\text{CODE\_MINUS\_LOW}] >= range\_times\_prob)$: | |
| $value = \mathbf{True}$ | |
| $\mathbf{state}[\text{CODE\_MINUS\_LOW}] - = range\_times\_prob$ | |
| $\mathbf{state}[\text{RANGE}] - = range\_time\_prob$ | |
| **else**: | |
| $value = \mathbf{False}$ | |
| $\mathbf{state}[\text{RANGE}] = range\_times\_prob$ | |
| $update\_context(\mathbf{state}[\text{CONTEXT\_PROBS}][context\_index], value)$ | B.2.6 |
| **while** $(\mathbf{state}[\text{RANGE}] <= 0x4000)$: | |
| $renormalise()$ | B.2.5 |
| **return** $value$ | |

The *renormalise*() function is very greatly simplified, since all the masking and bit-twiddling is eliminated to leave:

| $renormalise()$ : | **Ref** |
|---|---|
| $\mathbf{state}[\text{CODE\_MINUS\_LOW}] <<= 1$ | |
| $\mathbf{state}[\text{RANGE}] <<= 1$ | |
| $\mathbf{state}[\text{CODE\_MINUS\_LOW}] + = read\_bitb()$ | |

#### B.2.7.2 Bytewise operation

Accessing data bit by bit is also inefficient, so it is useful to look ahead and read in bytes into **state**[CODE_MINUS_LOW] or **state**[CODE] in advance. So, for example, **state**[CODE_MINUS_LOW] could be initialised to the first 4 bytes of the bitstream and **state**[RANGE] initialised to 0xFFFF0000, and all calulations shifted up by 16 bits. Then *read_boola* can be re-written as:

| *read_boola*(*context_label*) : | **Ref** |
|---|---|
| *prob_zero* = **state**[CONTEXT_PROBS][*context_label*] | |
| *range_times_prob* = ((**state**[RANGE] $\gg$ 16) $*$ *prob_zero*)&0xFFFF0000 | |
| **if** (**state**[CODE_MINUS_LOW] >= *range_times_prob*): | |
| $\quad$ *value* = **True** | |
| $\quad$ **state**[CODE_MINUS_LOW]− = *range_times_prob* | |
| $\quad$ **state**[RANGE]− = *range_time_prob* | |
| **else**: | |
| $\quad$ *value* = **False** | |
| $\quad$ **state**[RANGE] = *range_times_prob* | |
| *update_context*(**state**[CONTEXT_PROBS][*context_index*], *value*) | |
| **while** (**state**[RANGE] <= 0x40000000): | |
| $\quad$ *renormalise*() | |
| **return** *value* | |

and the renormalisation loop uses a counter, starting at 16, to input bits in 2-byte chunks:

| *renormalise*() : | **Ref** |
|---|---|
| **state**[CODE_MINUS_LOW] <<= 1 | |
| **state**[RANGE] <<= 1 | |
| **state**[COUNTER]− = 1 | |
| **if** (**state**[COUNTER] == 0): | |
| $\quad$ **state**[CODE_MINUS_LOW]+ = *read_uint_lit*(2) | |
| $\quad$ **state**[COUNTER] = 16 | |

#### B.2.7.3 Look-up table

In software it makes sense to use a modified probability LUT containing 512 elements, in which each even element is the negative increment to *prob_zero* if 0 is coded, and each odd element is the positive increment to *prob_zero* is 1 is coded. This means that access to the LUT will always be in a very local area whatever value is coded, whereas the basic structure will require either position $p$ or $255 - p$ to be accessed depending on the value.

## B.3 Arithmetic encoding (Informative)

This document only normatively defines the decoding of arithmetic coded data. However whilst it is clearly vital that an encoding process matches the decoding process, it is not entirely straightforward to derive an implementation of the encoder by only looking only at the decoder specification. Therefore this informative section describes a possible implementation for an arithmetic encoder that will produce output that is decodeable by the Dirac arithmetic decoder. This section is best read in conjunction with Annex B.2.

### B.3.1 Encoder variables

An arithmetic encoder requires the following unsigned integer variables, or some mathematically equivalent set:

- *low*, a value indicating the bottom of the encoding interval,
- *range*, a value indicating the width of the encoding interval,
- *carry*, a value tracking the number of unresolved “straddle” conditions (described below), and
- a set of 16-bit probability context probabilities, as described in Annex A.4.

The process for updating context probabilities, used for coding values, is described in Annex B.2.6

A Dirac binary arithmetic encoder implementation codes a set of data in three stages:

1. initialisation,
2. processing of all values, and
3. flushing.

### B.3.2 Initialisation

Initialisation of the arithmetic encoder is very simple – the internal variables are set as:

$$\begin{aligned} low &= 0\text{x}0 \\ range &= 0\text{xFFFF} \\ carry &= 0 \end{aligned}$$

With 16 bit accuracy, 0xFFFF corresponds to an interval width value of (almost) 1. All context probabilities are initialised to probability 1/2 (0x8000).

### B.3.3 Encoding binary values

The encoding process for a binary value must precisely mirror that for the decoding process (Annex B.2.4). In particular the interval variables *low* and *range* must be updated in the same way.

Coding a boolean value consists of three sub-stages (in order):

1. scaling the interval $[low, low + range)$,
2. updating contexts, and
3. renormalising and outputting data.

#### B.3.3.1 Scaling the interval

The integer interval $[low, low + range)$ represents the real interval

$$[l, h) = [low/2^{16}, (low + range)/2^{16})$$

In a given context with label *label*, the probability of zero can be extracted as

$$prob\_zero = \textbf{state}[\text{CONTEXT\_PROBS}][label]$$

If 0 is to be encoded, the real interval $[l, h)$ should be rescaled so that $l$ is unchanged and the width $r = h - l = range/2^{16}$ is scaled to $r * p_0$ where $p_0 = prob\_zero/2^{16}$.

This operation is approximated by setting

$$range = (range * prob\_zero) \gg 16$$

If 1 is to be encoded, $[l, h)$ should be rescaled so that $h$ is unchanged and $r$ is scaled to $(1 - p0) * r$. This operation is approximated by setting

$$\begin{aligned} low &\mathrel{+}= (range * prob\_zero) \gg 16 \\ range &\mathrel{-}= (range * prob\_zero) \gg 16 \end{aligned}$$

#### B.3.3.2 Updating contexts

Contexts are updated in exactly the same way as the decoder (Annex B.2.6).

#### B.3.3.3 Renormalisation and output

Renormalisation must cause *low* and *range* to be modified exactly as in the decoder (Annex B.2.5). In addition, during renormalisation bits are output when *low* and *low* + *range* agree in their msbs, taking into account carries accumulated when a straddle condition is accumulated.

In pseudocode, this is as follows:

```
...
  **while** ($range <= 0x4000$):
    **if** ($(((low + range - 1) \wedge low) >= 0x8000)$):
      $low \wedge = 0x4000$
      $carry+ = 1$
    **else**:
      $write\_bit(low \& 0x8000)$
      **while** ($carry > 0$):
        $write\_bit(!low \& 0x8000)$
        $carry- = 1$
    $low <<= 1$
    $range <<= 1$
    $low \& = 0xFFFF$
```

#### B.3.3.4 Flushing the encoder

After encoding, there may still be insufficient bits for a decoder to determine the final few encoded symbols, partly because further renormalisation is required – for example, msbs may agree but the range may still be larger than 0x4000) – and partly because there may be unresolved carries.

A four-stage process will adequately flush the encoder:

1. output remaining resolved msbs,
2. resolve remaining straddle conditions,
3. flush carry bits, and
4. byte align the output with padding bits.

The remaining msbs are output as follows:

```
...
  **while** ($((low + range - 1) \wedge low < 0x8000)$):
    $write\_bit(low \& 0x8000 != 0x0)$
    **while** ($carry > 0$):
      $write\_bit((low \& 0x8000) == 0x0)$
      $carry- = 1$
    $low <<= 1$
    $low \& = 0xFFFF$
    $range <<= 1$
```

Remaining straddles can then be resolved by:

```
...
  **while** ($((low \& 0x4000)$ and $(((low + range - 1) \& 0x4000) != 0x0))$):
    $carry+ = 1$
    $low \wedge = 0x4000$
    $low <<= 1$
    $range <<= 1$
    $low \& = 0xFFFF$
```

Carry bits can be discharged by picking a resolution of the final straddles:

```
...
  $write\_bit(low \& 0x4000 != 0x0)$
  for $c = 0$ to $carry$:
    $write\_bit((low \& 0x4000) == 0x0)$
```

Finally, 0-7 padding bits are added to the encoded output to make a whole number of bytes. These are not necessary for decoding, but for stream compliance.

## B.4 Efficient implementation

Some similar techniques to those described in Section B.2.7 can be used in the encoder to speed up operation.

#### B.4.0.5 Bytewise operation

It is not necessary to output bits one by one. Instead, *low* may be allowed to accumulate bits at the lower end and output them when a byte has accumulated. If the last bit determined was a 1, this 1 must be carried to the previous byte, so renormalisation becomes:

```
...
  while ($range <= 0x4000$):
    $low \ll= 1$
    $range \ll= 1$
    $counter -= 1$
    if ($counter == 0$):
      if ($low <= 1 \ll 24$ and $low + range > 1 \ll 24$):
        $carry += 1$
      else:
        if ($low < 1 \ll 24$):
          while ($carry$):
            $data[pos] = 0xFF$
            $carry -= 1$
            $pos += 1$
        else:
          $data[pos - 1] += 1$
          while ($carry$):
            $data[pos] = 0x00$
            $carry -= 1$
            $pos += 1$
        $data[pos](low \gg 16$
        $pos += 1$
        $low \& = 0xFFFF$
      $counter = 8$
```

#### B.4.0.6 Overlap and add

Renormalisation can be simplified still further by observing that carries occur if and only if the top byte of *low* becomes 0xFF. In this case a carried 1 would propagate up multiple bytes, turning 0xFFs into 0x00s. So it is possible to store the top two bytes of *low* (i.e. bit 24 containing the carry bit and the next byte) and do an overlap and add at the end to correctly propagate values back to the beginning. I.e. renormalisation becomes:

```
...
  **while** ($range <= 0x4000$):
    $low \ll= 1$
    $range \ll= 1$
    $counter- = 1$
    **if** ($counter == 0$):
      $low\_list[pos] = low \gg 16$
      $pos+ = 1$
      $counter = 8$
      $low\& = 0xFFFF$
```

At the end of coding all values, a flush function will complete *low_list* for remaining values and perform the overlap and add:

```
...
  $low+ = 1$
  $low \ll= counter$
  $low\_list[pos] = (low \gg 16)\&0xFFFF$
  $pos+ = 1$
  $low\_list[pos] = (low \gg 8)\&0xFFFF$
  $pos+ = 1$
  $data[pos - 1] = low\_list[pos - 1]\&0xFF$
  $data[pos - 2] = low\_list[pos - 2]\&0xFF$
  **for** $i = 0$ **to** $pos - 3$:
    $low\_list[pos - 3 - i]+ = low\_list[pos - 2 - i] \gg 8$
    $data[pos - 3 - i] = low\_list[pos - 3 - i]\&0xFF$
```

## C Predefined video formats

This annex defines the default values of video parameters that are determined by the value of the base video format. These defaults reduce overhead by allowing a large number of parameters to be set without explicit signaling.

The collection of default values for each value of the base video format constitutes a map, which shall be returned by the $set\_source\_defaults(base\_video\_format)$ function and used as a basis for defining the source video format in the sequence header as per Section 10.3.1.

All source parameters for any of the predefined video formats may be overridden as required in the sequence header.

| Video Formats | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Base video format index value** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Name (informative)** | Custom | QSIF525 | QCIF | SIF525 | CIF | 4SIF525 | 4CIF | SD480 -60I | SD576 -50I |
| **Frame Width:** | 640 | 176 | 176 | 352 | 352 | 704 | 704 | 720 | 720 |
| **Frame Height:** | 480 | 120 | 144 | 240 | 288 | 480 | 576 | 480 | 576 |
| **Chroma Sampling Format:** | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:0 | 4:2:2 | 4:2:2 |
| **Source Sampling:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **Top Field First:** | **False** | **False** | **True** | **False** | **True** | **False** | **True** | **False** | **True** |
| **Frame Rate Index** | 1 | 9 | 10 | 9 | 10 | 9 | 10 | 4 | 3 |
| **Numerator** | 24000 | 15000 | 25 | 15000 | 25 | 15000 | 25 | 30000 | 25 |
| **Denominator** | 1001 | 1001 | 2 | 1001 | 2 | 1001 | 2 | 1001 | 1 |
| **Aspect Ratio Index** | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| **Numerator** | 1 | 10 | 12 | 10 | 12 | 10 | 12 | 10 | 12 |
| **Denominator** | 1 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| **Clean Width:** | 640 | 176 | 176 | 352 | 352 | 704 | 704 | 704 | 704 |
| **Clean Height:** | 480 | 120 | 144 | 240 | 288 | 480 | 576 | 480 | 576 |
| **Clean Left Offset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 8 |
| **Clean Top Offset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Signal Range Index** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 3 |
| **Luma Offset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 64 | 64 |
| **Luma Excursion** | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 876 | 876 |
| **Chroma Offset** | 128 | 128 | 128 | 128 | 128 | 128 | 128 | 512 | 512 |
| **Chroma Excursion** | 255 | 255 | 255 | 255 | 255 | 255 | 255 | 896 | 896 |
| **Colour Specification Index** | 0 Custom | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 |
| **Colour Primaries Index** | 0 HDTV | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 | 1 SDTV 525 | 2 SDTV 625 |
| **Colour Matrix Index** | 0 HDTV | 1 SDTV | 1 SDTV | 1 SDTV | 1 SDTV | 1 SDTV | 1 SDTV | 1 SDTV | 1 SDTV |
| **Transfer Function Index** | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma |

Table C.1: Predefined video format parameters for video formats 0–8

| Video Formats | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Base video format index value** | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **Name (informative)** | HD720P-60 | HD720P-50 | HD1080I-60 | HD1080I-50 | HD1080P-60 | HD1080P-50 | DC2K | DC4K |
| **Frame Width:** | 1280 | 1280 | 1920 | 1920 | 1920 | 1920 | 2048 | 4096 |
| **Frame Height:** | 720 | 720 | 1080 | 1080 | 1080 | 1080 | 1080 | 2160 |
| **Chroma Sampling Format:** | 4:2:2 | 4:2:2 | 4:2:2 | 4:2:2 | 4:2:2 | 4:2:2 | 4:4:4 | 4:4:4 |
| **Source Sampling:** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **Top Field First:** | **True** | **True** | **True** | **True** | **True** | **True** | **True** | **True** |
| **Frame Rate Index** | 7 | 6 | 4 | 3 | 7 | 6 | 2 | 2 |
| **Numerator** | 60000 | 50 | 30000 | 25 | 60000 | 50 | 24 | 24 |
| **Denominator** | 1001 | 1 | 1001 | 1 | 1001 | 1 | 1 | 1 |
| **Pixel Aspect Ratio Index** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Numerator** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Denominator** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Clean Width** | 1280 | 1280 | 1920 | 1920 | 1920 | 1920 | 2048 | 4096 |
| **Clean Height** | 720 | 720 | 1080 | 1080 | 1080 | 1080 | 1080 | 2160 |
| **Clean Left Offset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Clean Top Offset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Signal Range Index** | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 |
| **Luma Offset** | 64 | 64 | 64 | 64 | 64 | 64 | 256 | 256 |
| **Luma Excursion** | 876 | 876 | 876 | 876 | 876 | 876 | 3504 | 3504 |
| **Chroma Offset** | 512 | 512 | 512 | 512 | 512 | 512 | 2048 | 2048 |
| **Chroma Excursion** | 896 | 896 | 896 | 896 | 896 | 896 | 3584 | 3584 |
| **Colour Specification Index** | 3<br>HDTV | 3<br>HDTV | 3<br>HDTV | 3<br>HDTV | 3<br>HDTV | 3<br>HDTV | 4<br>Cinema | 4<br>Cinema |
| **Colour Primaries Index** | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 3<br>Cinema | 3<br>Cinema |
| **Colour Matrix Index** | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV | 0<br>HDTV |
| **Transfer Function Index** | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma | 0<br>TV gamma |

Table C.2: Predefined video format parameters for video formats 9–16

| Video Formats | | | | |
|---|---|---|---|---|
| **Base video format index value** | 17 | 18 | 19 | 20 |
| **Name (informative)** | UHDTV 4K-60 | UHDTV 4K-50 | UHDTV 8K-60 | UHDTV 8K-50 |
| **Frame Width:** | 3840 | 3840 | 7680 | 7680 |
| **Frame Height:** | 2160 | 2160 | 4320 | 4320 |
| **Chroma Sampling Format:** | 4:2:2 | 4:2:2 | 4:2:2 | 4:2:2 |
| **Source Sampling:** | 0 | 0 | 0 | 0 |
| **Top Field First:** | **True** | **True** | **True** | **True** |
| **Frame Rate Index** | 7 | 6 | 7 | 6 |
| **Numerator** | 60000 | 50 | 60000 | 50 |
| **Denominator** | 1001 | 1 | 1001 | 1 |
| **Pixel Aspect Ratio Index** | 1 | 1 | 1 | 1 |
| **Numerator** | 1 | 1 | 1 | 1 |
| **Denominator** | 1 | 1 | 1 | 1 |
| **Clean Width** | 3840 | 3840 | 7680 | 7680 |
| **Clean Height** | 2160 | 2160 | 4320 | 4320 |
| **Clean Left Offset** | 0 | 0 | 0 | 0 |
| **Clean Top Offset** | 0 | 0 | 0 | 0 |
| **Signal Range Index** | 3 | 3 | 3 | 3 |
| **Luma Offset** | 64 | 64 | 64 | 64 |
| **Luma Excursion** | 876 | 876 | 876 | 876 |
| **Chroma Offset** | 512 | 512 | 512 | 512 |
| **Chroma Excursion** | 896 | 896 | 896 | 896 |
| **Colour Specification Index** | 3 HDTV | 3 HDTV | 3 HDTV | 3 HDTV |
| **Colour Primaries Index** | 0 HDTV | 0 HDTV | 0 HDTV | 0 HDTV |
| **Colour Matrix Index** | 0 HDTV | 0 HDTV | 0 HDTV | 0 HDTV |
| **Transfer Function Index** | 0 TV gamma | 0 TV gamma | 0 TV gamma | 0 TV gamma |

Table C.3: Predefined video format parameters for video formats 17–20

# D Profiles and levels

A Dirac decoder shall support one or more different profiles and levels. Profiles and levels determine which tools, syntax elements and structures shall be supported, and what decoder resources (computational and memory) are required.

## D.1 Profiles

A given profile requires that particular syntax/syntax elements shall be used and that decoder variables or functions shall be set to particular values.

Dirac defines four profiles, Main (Long GOP), Main (Intra), Simple and Low Delay, corresponding to different picture types. The Main (Intra), Simple and Low Delay profiles shall correspond to the Main, Simple and Low Delay profiles of VC-2.

The profiles shall satisfy the following conditions:

- A Low Delay profile Dirac sequence shall set **state**[PROFILE] equal to a value of 0 in the parse parameters (Section 10.1) for each sequence header in the sequence.
- A Simple profile sequence shall set **state**[PROFILE] equal to a value of 1 in the parse parameters for each sequence header in the sequence.
- A Main (Intra) profile sequence shall set **state**[PROFILE] equal to a value of 2 in the parse parameters for each sequence header in the sequence.
- A Main (Long GOP) profile sequence shall set **state**[PROFILE] equal to a value of 8 in the parse parameters for each sequence header in the sequence.

Further VC-2 compatible profiles may be added in future revisions of this specification with profile number less than 8; other profiles may be added with profile number greater than 8.

A Dirac sequence shall comply with one of the supported profiles.

### D.1.1 Low Delay profile

A Low Delay profile sequence shall contain only those data units whose parse codes are listed in Table D.1.

| **state**[PARSE_CODE] | **Bits** | **Description** | **Number of Reference Pictures** |
|---|---|---|---|
| 0x00 | 0000 0000 | Sequence header | – |
| 0x10 | 0001 0000 | End of Sequence | – |
| 0x20 | 0010 0000 | Auxiliary data | – |
| 0x30 | 0011 0000 | Padding data | – |
| 0xC8 | 1100 1000 | Intra Non Reference Picture | 0 |

Table D.1: Parse code values for Low Delay profile sequences

### D.1.2 Simple profile

A Simple profile sequence shall contain only those data units whose parse codes are listed in Table D.2.

### D.1.3 Main (Intra) profile

A Main (Intra) profile sequence shall contain only those data units whose parse codes are listed in Table D.3.

| **state**[PARSE_CODE] | **Bits** | **Description** | **Number of Reference Pictures** |
|---|---|---|---|
| 0x00 | 0000 0000 | Sequence header | – |
| 0x10 | 0001 0000 | End of Sequence | – |
| 0x20 | 0010 0000 | Auxiliary data | – |
| 0x30 | 0011 0000 | Padding data | – |
| 0x48 | 0100 1000 | Intra Non Reference Picture (no arithmetic coding) | 0 |

Table D.2: Parse code values for Simple profile sequences

| **state**[PARSE_CODE] | **Bits** | **Description** | **Number of Reference Pictures** |
|---|---|---|---|
| 0x00 | 0000 0000 | Sequence header | – |
| 0x10 | 0001 0000 | End of Sequence | – |
| 0x20 | 0010 0000 | Auxiliary data | – |
| 0x30 | 0011 0000 | Padding data | – |
| 0x08 | 0000 1000 | Intra Non Reference Picture (arithmetic coding) | 0 |

Table D.3: Parse code values for Main (Intra) profile sequences

### D.1.4 Main (Long GOP) profile [TBD]

A Main (Long GOP) profile sequence shall contain only those data units whose parse codes are listed in Table D.4.

| **state**[PARSE_CODE] | **Bits** | **Description** | **Number of Reference Pictures** |
|---|---|---|---|
| 0x00 | 0000 0000 | Sequence header | – |
| 0x10 | 0001 0000 | End of Sequence | – |
| 0x20 | 0010 0000 | Auxiliary data | – |
| 0x30 | 0011 0000 | Padding data | – |
| 0x0C | 0000 1100 | Intra Reference Picture (arithmetic coding) | 0 |
| 0x08 | 0000 1000 | Intra Non Reference Picture (arithmetic coding) | 0 |
| 0x0D | 0000 1101 | Inter Reference Picture (arithmetic coding) | 1 |
| 0x0E | 0000 1110 | Inter Reference Picture (arithmetic coding) | 2 |
| 0x09 | 0000 1001 | Inter Non Reference Picture (arithmetic coding) | 1 |
| 0x0A | 0000 1010 | Inter Non Reference Picture (arithmetic coding) | 2 |

Table D.4: Parse code values for Main (Long GOP) profile sequences

Low delay syntax pictures shall not be present in a Main (Long GOP) profile sequence.

## D.2 Levels

A given value of level shall define constraints on the decoder resources required to decode a compliant sequence. The values that are constrained are defined individually for each level.

A particular application domain may impose additional constraints on a decoder, for example the presence or absence of a suitable video interface. Such additional constraints are not covered by this specification.

This specification defines levels 1 and 128. Other levels are application specific and will be defined in future revisions of this specification. VC-2 compatible levels will have level number less than 128 and other levels will have level number greater than 128.

The value 0 shall be RESERVED and shall not be used for any defined level. Sequences may use the value 0 for streams that do not conform to any defined level.

For level 1, the video parameters shall correspond to one of the base video formats as defined in Annex C and the video parameters of the base video format shall not be overridden. Level 1 shall be available for Low Delay, Simple and Main (Intra) (VC-2 compatible) profiles only.

For level 128, the video parameters shall correspond to one of the base video formats as defined in Annex C, except that the signal range is restricted to 8 bit ranges and the chroma sampling format and frame dimensions may be overridden by suitable values. Level 128 shall be available for Main (Long GOP) profile only.

A sequence compliant with a given level $n$ shall set **state**[LEVEL] equal to $n$.

### D.2.1 Decoder data buffers[DRAFT-TBC]

A reference picture buffer **state**[REF_PICTURES] is defined for storing decoded reference pictures (if any). In addition, levels may define two additional buffers, and applicable parameters for them:

- a bit stream buffer **state**[STREAM_BUFFER] for buffering stream data prior to decoding
- a decoded picture buffer **state**[DECODED_PICTURES] for storing decoded pictures (reference or non-reference) for the purposes of picture reordering

For the purposes of this specification, the reference picture buffer shall be deemed to be additional to the decoded picture buffer (i.e. reference picture storage will be duplicated in the decoded picture buffer).

#### D.2.1.1 Bit stream buffer operation[TBC]

#### D.2.1.2 Picture reordering and decoded picture buffer[TBC]

Two parameters shall be defined for constraining picture reordering. The first is the size, **state**[DPB_SIZE], of the decoded picture buffer.

The second is the reordering depth applicable to a Dirac sequence. This shall be defined as the maximum number of picture data units that may occur between a picture with picture number $N$ and a picture with picture number $N+1$ or $N-1$.

### D.2.2 Buffer models [TBC]

### D.2.3 Level 1: VC-2 default level

This level is intended to provide minimal constraints on VC-2 compatible streams encoding one of the base video formats (Annex C). Data rates in this level are not bounded. This level is not intended to provide guarantees of real time decoding.

This level shall only be available if the profile is set to Low Delay, Simple, or Main (Intra).

#### D.2.3.1 Sequence header parameters

The following constraints shall apply to the sequence header parameters (Section 10):

- $base\_video\_format$ shall be between 1 and 20 inclusive

- *custom_dimensions_flag* shall be **False**
- *custom_chroma_format_flag* shall be **False**
- *custom_scan_format_flag* shall be **False**
- *custom_frame_rate_flag* shall be **False**
- *custom_pixel_aspect_ratio_flag* be **False**
- *custom_clean_area_flag* shall be **False**
- *custom_signal_range_flag* shall be **False**
- *custom_color_spec_flag* shall be **False**
- *picture_coding_mode* shall be as per Section 10.4

#### D.2.3.2 Picture header parameters

The following constraints shall apply to picture header parameters (Section 11.1.1):

- **state**[WAVELET_INDEX] shall be as table 11.2
- **state**[DWT_DEPTH] shall be between 0 and 4 inclusive
- for core syntax pictures:
  - **state**[CODEBLOCKS_X] and **state**[CODEBLOCKS_Y] shall be such that there is at least one coefficient in each codeblock.
- for low delay syntax pictures:
  - **state**[SLICES_X] and **state**[SLICES_Y] shall be such that there is at least one DC (0-LL) coefficient per slice.
  - values in the quantizer matrix shall lie between 0 and 127 inclusive

#### D.2.3.3 Transform data

The following constraints shall apply to transform data elements (Section 11.3):

- quantized and inverse quantized wavelet coefficients shall lie between $-2^{19}$ and $2^{19}$ inclusive
- for core syntax pictures:
  - quantization indices for subbands and codeblocks shall lie between 0 and 68 inclusive
  - quantization offsets encoded in codeblocks shall lie between -68 and 68 inclusive

### D.2.4 Level 128: Long-GOP default level [DRAFT-TBD]

This level is intended to provide minimal constraints on long GOP streams encoding simple variants of the base video formats (Annex C). Data rates in this level are not bounded. This level is not intended to provide guarantees of real time decoding.

This level shall only be available if the profile is set to Main (Long GOP).

#### D.2.4.1 Sequence header parameters

The following constraints shall apply to the sequence header parameters (Section 10):

- *base_video_format* shall be between 1 and 20 inclusive
- *custom_dimensions_flag* may be **False** or **True**; if **True** then the frame width and frame height shall be less than the values set in the base video format
- *custom_chroma_format_flag* may be **False** or **True**
- *custom_scan_format_flag* may be **False** or **True**
- *custom_frame_rate_flag* shall be **False**
- *custom_pixel_aspect_ratio_flag* be **False**
- *custom_clean_area_flag* shall be **False**
- *custom_signal_range_flag* shall be **True** and the signal range parameters set to 8 bit SDI ranges (index 2 in table 10.5)
- *custom_color_spec_flag* shall be **False**
- *picture_coding_mode* shall be as per Section 10.4

#### D.2.4.2 Picture header parameters

The following constraints shall apply to picture header parameters (Section 11.1.1):

- **state**[WAVELET_INDEX] shall be between 0 and 4 inclusive
- **state**[DWT_DEPTH] shall be between 0 and 4 inclusive
- **state**[CODEBLOCKS_X] and **state**[CODEBLOCKS_Y] shall be such that there is at least one coefficient in each codeblock

#### D.2.4.3 Transform data

The following constraints shall apply to transform data elements (Section 11.3):

- quantized and inverse quantized wavelet coefficients shall lie between $-2^{15}$ and $2^{15}$ inclusive
- quantization indices for subbands and codeblocks shall lie between 0 and 63 inclusive
- quantization offsets encoded in codeblocks shall lie between -63 and 63 inclusive

#### D.2.4.4 Reordering and reference buffers [DRAFT-TBD]

A decoder must maintain a reference picture buffer for the purposes of inter picture prediction, and (separately) a decoded picture buffer for the purposes of picture reordering, in addition to storage provided for decoding the current picture. The decoded picture buffer shall contain non-reference pictures only.

A bit stream buffer is not required for this level.

The following constraints shall apply:

- The reference buffer size **state**[MAX_RB_SIZE] shall be 3.

- The decoded picture buffer size shall be 2 in the case of frame coding or 4 in the case of field coding.
- The picture reordering depth shall be 5 in the case of frame coding or 11 in the case of field coding

# E Low delay quantisation matrices

This annex specifies the default quantisation matrices to be used in the low delay syntax and provides an informative description of quantisation matrix design principles and of quantiser selection in both the core and low-delay syntax.

## E.1 Quantisation matrices (low delay syntax)

This section defines default quantisation matrices to be used for the quantisation of slice coefficients in the low-delay syntax. The following tables define matrices for **state**[DWT_DEPTH] $\leq 4$. Values of **state**[DWT_DEPTH] not present in the tables in this section shall require a custom matrix to be encoded, as per Section 11.3.4. Informative advice for constructing quantisation matrices based on noise power conservation and perceptual weighting is given in Annex E.2.2.

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 5 | 5 | 5 | 5 |
| 1 | HL,LH, HH | - | 3, 3, 0 | 3, 3, 0 | 3, 3, 0 | 3, 3, 0 |
| 2 | HL,LH, HH | - | - | 4, 4, 1 | 4, 4, 1 | 4, 4, 1 |
| 3 | HL,LH, HH | - | - | - | 5, 5, 2 | 5, 5, 2 |
| 4 | HL,LH, HH | - | - | - | - | 6, 6, 3 |

Table E.1: Default quantisation matrices for **state**[WAVELET_INDEX] == 0 (Deslauriers-Dubuc (9,7))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 4 | 4 | 4 | 4 |
| 1 | HL,LH, HH | - | 2, 2, 0 | 2, 2, 0 | 2, 2, 0 | 2, 2, 0 |
| 2 | HL,LH, HH | - | - | 4, 4, 2 | 4, 4, 2 | 4, 4, 2 |
| 3 | HL,LH, HH | - | - | - | 5, 5, 3 | 5, 5, 3 |
| 4 | HL,LH, HH | - | - | - | - | 7, 7, 5 |

Table E.2: Default quantisation matrices for **state**[WAVELET_INDEX] == 1 (LeGall (5,3))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 5 | 5 | 5 | 5 |
| 1 | HL,LH, HH | - | 3, 3, 0 | 3, 3, 0 | 3, 3, 0 | 3, 3, 0 |
| 2 | HL,LH, HH | - | - | 4, 4, 1 | 4, 4, 1 | 4, 4, 1 |
| 3 | HL,LH, HH | - | - | - | 5, 5, 2 | 5, 5, 2 |
| 4 | HL,LH, HH | - | - | - | - | 6, 6, 3 |

Table E.3: Default quantisation matrices for **state**[WAVELET_INDEX] == 2 (Deslauriers-Dubuc (13,7)))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 8 | 12 | 16 | 20 |
| 1 | HL,LH, HH | - | 4, 4, 0 | 8, 8, 4 | 12, 12, 8 | 16, 16, 12 |
| 2 | HL,LH, HH | - | - | 4, 4, 0 | 8, 8, 4 | 12, 12, 8 |
| 3 | HL,LH, HH | - | - | - | 4, 4, 0 | 8, 8, 4 |
| 4 | HL,LH, HH | - | - | - | - | 4, 4, 0 |

Table E.4: Default quantisation matrices for **state**[WAVELET_INDEX] == 3 (Haar with no shift))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 8 | 8 | 8 | 8 |
| 1 | HL,LH, HH | - | 4, 4, 0 | 4, 4, 0 | 4, 4, 0 | 4, 4, 0 |
| 2 | HL,LH, HH | - | - | 4, 4, 0 | 4, 4, 0 | 4, 4, 0 |
| 3 | HL,LH, HH | - | - | - | 4, 4, 0 | 4, 4, 0 |
| 4 | HL,LH, HH | - | - | - | - | 4, 4, 0 |

Table E.5: Default quantisation matrices for **state**[WAVELET_INDEX] == 4 (Haar with single shift per level))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 0 | 0 | 0 | 0 |
| 1 | HL,LH, HH | - | 4, 4, 8 | 4, 4, 8 | 4, 4, 8 | 4, 4, 8 |
| 2 | HL,LH, HH | - | - | 8, 8, 12 | 8, 8, 12 | 8, 8, 12 |
| 3 | HL,LH, HH | - | - | - | 13, 13, 17 | 13, 13, 17 |
| 4 | HL,LH, HH | - | - | - | - | 17, 17, 21 |

Table E.6: Default quantisation matrices for **state**[WAVELET_INDEX] == 5 (Fidelity))

| | | **state**[DWT_DEPTH] | | | | |
|---|---|---|---|---|---|---|
| Level | Orientation | 0 | 1 | 2 | 3 | 4 |
| 0 | LL | 0 | 3 | 3 | 3 | 3 |
| 1 | HL,LH, HH | - | 1, 1, 0 | 1, 1, 0 | 1, 1, 0 | 1, 1, 0 |
| 2 | HL,LH, HH | - | - | 4, 4, 2 | 4, 4, 2 | 4, 4, 2 |
| 3 | HL,LH, HH | - | - | - | 6, 6, 5 | 6, 6, 5 |
| 4 | HL,LH, HH | - | - | - | - | 9, 9, 7 |

Table E.7: Default quantisation matrices for **state**[WAVELET_INDEX] == 6 (Daubechies (9,7))

## E.2 Quantisation matrix design and quantiser selection (Informative)

This section provides an informative guide to the principles used to design the default quantisation matrix

### E.2.1 Noise power normalisation

The quantisation matrices defined in the preceding section are designed to counteract the differential power gain of the various wavelet filters, so that quantisation noise from each subband is weighted equally in terms of its contribution to noise power when transformed back into the picture domain. Let $\alpha$ and $\beta$ represent the noise gain factors of the low-pass and high-pass wavelet filters used in wavelet decomposition. In a single level of wavelet decomposition, quantisation noise in each of the four subbands is therefore weighted by the factors shown in Figure E.1.

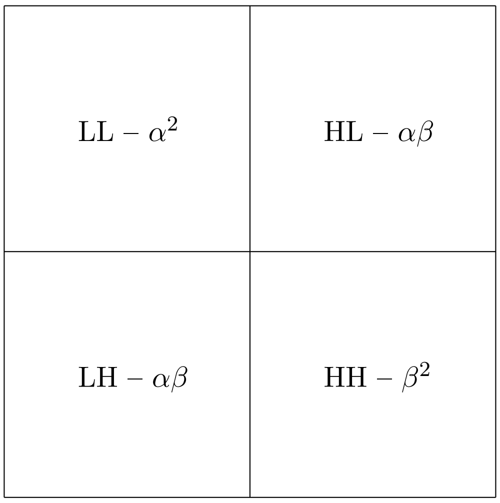

Figure E.1: Subband weights for a 1-level decomposition

For higher levels of decomposition, these subband weighting factors iterate in the same manner as the wavelet transform itself. For example, with a two-level decomposition, the first level LL band, with weight $\alpha^2$ is further decomposed to give four more bands with weights as for the 1-level decomposition, but multiplied by $\alpha^2$. This yields the weights shown in Figure E.2.

In this specification, wavelet synthesis filters have been defined in terms of lifting stages, which are filters operating on subsampled data. Wavelet filters are more traditionally represented in terms of an iterated binary polyphase filter bank: the relationship between these representation is described in Annex G.2. The factors $\alpha$ and $\beta$ are most easily computed from the filter bank representation. In this case $\alpha$ is either the RMS power gain of the low-pass synthesis filter, or the *reciprocal* of the RMS power gain of the low-pass analysis filter; and $\beta$ is the RMS power gain of the high-pass synthesis filter of the reciprocal of the RMS power gain of the high-pass analysis filter.

Thus, in the terminology of Annex G.2, $\alpha = \frac{1}{\left(\sum_n h(n)^2\right)^{\frac{1}{2}}}$ or $\alpha = \left(\sum_n \tilde{h}(n)^2\right)^{\frac{1}{2}}$

and $\beta = \frac{1}{\left(\sum_n g(n)^2\right)^{\frac{1}{2}}}$ or $\beta = \left(\sum_n \tilde{g}(n)^2\right)^{\frac{1}{2}}$

These alternative definitions arise because the wavelet filters defined in this specification are not orthogonal, but technically *biorthogonal* and so, strictly speaking, there is not power addition of the quantisation noise in

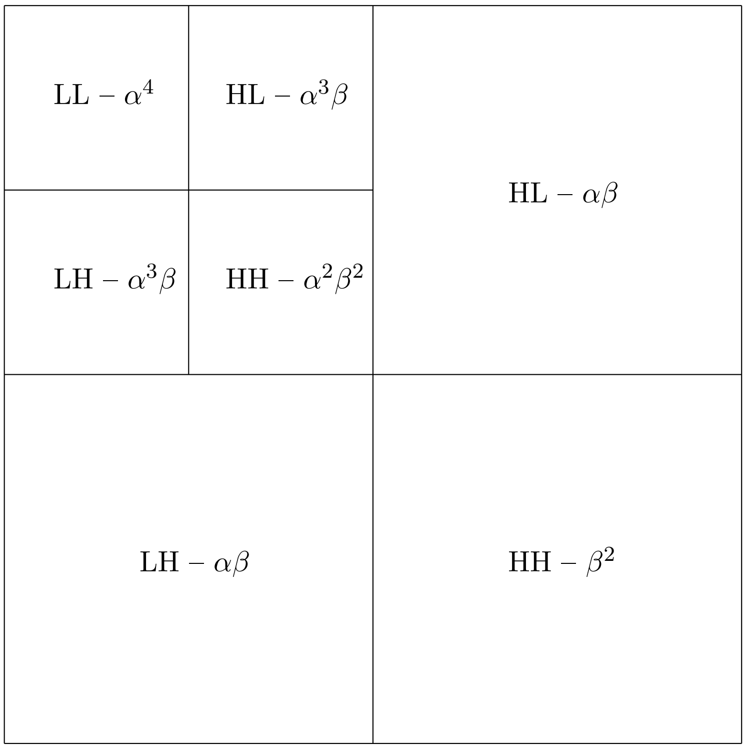

Figure E.2: Subband weights for a 2-level decomposition

each subband. The values used for quantisation matrices have been computed from the analysis rather than the synthesis filters, as this yields better compression results in practice.

Note also that these factors must also take into account the shift factors used to add accuracy bits prior to each wavelet decomposition stage. For a filter shift of $d$, $\alpha$ and $\beta$ are each multiplied by $2^{-d/2}$.

Given a subband weighting factor $w$, a quantisation offset for that subband may be defined as $4 * \log_2(w)$ rounded to the nearest integer. These offsets are then normalised so as to be non-negative, to produce the tables of the preceding section.

### E.2.2 Custom quantisation matrices

Custom matrices may be defined that take into account not only noise power normalisation but also perceptual weighting based on spatial frequency. Additional multiplicative factors may be computed for each subband, which produce a matrix of quantisation offsets which may then be added to the default unweighted quantisation matrices to produce a weighted quantisation matrix.

An example perceptual weighting may be constructed from the CCIR 959 Contrast Sensitivity Function (CSF). This is a function $csf(s)$ which produces a value representing the sensitivity to detail at a given normalised spatial frequency $s$. For luminance, it is defined by

$$csf(s) = 0.255 * (1 + 0.2561 * s^2)^{-0.75}$$

Assuming an isotropic response, we may form a 2-d perceptual weighting function on horizontal and vertical spatial frequencies $x_s, y_s$ by

$$\begin{aligned} c(x_s, y_s) &= \frac{1}{csf((xs^2 + ys^2)^{\frac{1}{2}})} \\ &= 0.255 * (1 + 0.2561 * (x_s^2 + y_s^2))^{0.75} \end{aligned}$$

Each subband in a wavelet decomposition represents a subset of spatial frequencies according to level and orientation, partitioning the spatial frequency domain as per Figure 13.1. Note that this partitioning is un-normalised, since output pictures (and their compression artefacts) may be viewed at a range of distances.

Accordingly we may pick a representative, un-normalised horizontal and vertical spatial frequency $(f_x(b), f_y(b))$ – perhaps the middle frequency of the band. For example, an LH band $b$ at level 1 in a 1-level decomposition will have mid frequency at $(pw/4, 3 * ph/4)$ where $ph$ and $pw$ are the padded width and height of the picture (Section 13.1.2). This may be turned into a true spatial frequency by normalising by the number of horizontal and vertical cycles per degree the output pictures will subtend at the target viewing distance and aspect ratio:

$$(f_x(b)/cpd_x, f_y(b)/cpd_y)$$

and this value may be fed into the weighting function to get a value $c(b)$. The appropriate quantisation offset for that subband is then $4 * \log_2(c(b))$, which may be used to define a modified quantisation matrix.

# F Video systems model (Informative)

## F.1 Colour models

All current video systems use a $Y, C1, C2$ form of coding for RGB source values. Although $Y, C_B, C_R$ is widely used, Dirac can support other colour systems such as $Y, C_O, C_G$ as defined by ITU-T H.264 AVC annex E. For this reason the non-luma components are generalized to the terms C1 and C2.

The R, G and B are tristimulus values (e.g. candelas/$m^2$). 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 $E_R$, $E_G$ and $E_B$ values are related to the linear RGB values by non-linear transfer functions. Normally, $E_R$, $E_G$ and $E_B$ also fall in the range $[0, 1]$, but in the case of extended gamut systems (such as ITU-R BT1361), negative values can also occur. The non-linear transfer function is typically performed in the camera and is specified in the transfer characteristic part of the appropriate colour specification. For aesthetic and psychovisual reasons the encoding transfer function is not always the inverse of the decoding transfer function. In fact the combined effect of the encoding and decoding transfer functions is such that the rendering intent or end-to-end gamma of the system can vary between about 1.1 and 1.6 depending on viewing conditions. The rationale for this is given in Digital Video and HDTV by Charles Poynton, (2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7).

The non-linear $E_R$, $E_G$ and $E_B$ values are subject to a matrix operation (known as 'non-constant luminance coding'), which transforms them into luma ($E_Y$) and colour difference (normally $E_{Cb}$ and $E_{Cr}$) values. $E_Y$ is normally limited to the range $[0, 1]$ and the colour difference values to the range $[-0.5, 0.5]$. In this specification, the color difference components are referred to as 'chroma components and are not to be confused with the chroma signals used by composite television systems where the colour difference signals are significantly reduced in both resolution and signal amplitude. The chroma components used in this specification can be sub-sampled, either horizontally, vertically or both horizontally and vertically.

### F.1.1 $YC_BC_R$ coding

The $E_Y$, $E_{Cb}$ and $E_{Cr}$ values are mapped to a range of integers denoted $Y$, $C_B$ and $C_R$, typically $[0, 255]$. 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.

### F.1.2 $YC_OC_G$ coding

In the case of YCoCg coding, the $E_R$, $E_G$ and $E_B$ values are directly linearly scaled to integer ranges before a lossless direct integer transform is applied to convert this data to $Y$, $C_O$ and $C_G$) data.

### F.1.3 Signal range

The output of the Dirac decoder consists of unsigned integer values. For $YC_BC_R$ coding, the offset and excursion values are used to linearly scale these values into intermediate vlues $E_Y$, $E_{Cb}$, and $E_{Cr}$. $E_Y$ is normally clipped to the range $[0, 1]$ and $E_{Cb}$, $E_{Cr}$ to the range $[-0.5, 0.5]$. The effect is to clip integer $Y$ values output by the decoder to the interval

$$\textbf{video\_params}[\text{LUMA\_OFFSET}], \textbf{video\_params}[\text{LUMA\_OFFSET}] + \textbf{video\_params}[\text{LUMA\_EXCURSION}]]$$

and $C1$, $C2$ values to

$$[\textbf{video\_params}[\text{CHROMA\_OFFSET}] - \textbf{video\_params}[\text{CHROMA\_EXCURSION}]/2, \textbf{video\_params}[\text{CHROMA\_OFFSET}] + \textbf{video}$$

However, maintaining an extended RGB gamut can 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 $Y$, $C1$, and $C2$ values.

In the case of $YCoCg$ coding, $E_Y$, $E_{CO}$, and $E_{CG}$ should not be calculated. Instead, direct integer conversion to RGB should be done (note: excursion values will be ignored in this integer conversion.)

### F.1.4 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 are the CIE (1931) XYZ chromaticity co-ordinates of the primaries and the white point of the device.

The color primary specification therefore allows exact color reproduction of decoded RGB values on different displays with different display primaries.

### F.1.5 Colour matrix

#### F.1.5.1 $YC_BC_R$ coding

Unit-scale luma and chroma values $E_Y$, $E_{Cb}$ and $E_{Cr}$ should be derived from decoded $Y$, $C1$ and $C2$ values using the signal range parameters as per Section F.1.3. Given these values, $E_R$, $E_G$ and $E_B$ are determined as follows:

$$\begin{aligned}
E_R &= E_Y + 2*(1-K_R)*E_{Cr} \\
E_G &= E_Y - \frac{2*K_R*(1-K_R)*E_{Cr}}{K_G} - \frac{2*K_B*(1-K_B)*E_{Cb}}{K_G} \\
E_B &= E_Y + 2*(1-K_R)*E_{Cb}
\end{aligned}$$

where $K_G = 1 - K_R - K_B$. This follows by inverting the equations

$$\begin{aligned}
K_R + K_G + K_B &= 1 \\
E_Y &= K_R*E_R + K_G*E_G + K_B*E_B \\
E_{Cb} &= \frac{E_B - E_Y}{2*(1-K_B)} \\
E_{Cr} &= \frac{E_R - E_Y}{2*(1-K_R)}
\end{aligned}$$

#### F.1.5.2 YCoCg coding

In the case of YCoCg coding, integer $I_R$, $I_G$, $I_B$ should be directly computed from the decoded $Y$, $C1$ ($C_O$) and $C2$ ($C_G$) values by

$$\begin{aligned}
Y &-= \textbf{video\_params}[\text{LUMA\_OFFSET}] \\
Co = C1 &-= \textbf{video\_params}[\text{CHROMA\_OFFSET}] \\
Cg = C2 &-= \textbf{video\_params}[\text{CHROMA\_OFFSET}] \\
t &= Y - (Cg \gg 1) \\
I_G &= t + Cg \\
I_B &= t - (Co \gg 1) \\
I_R &= I_B + Co
\end{aligned}$$

The integer values are converted to unit-scale $E_R$, $E_G$, $E_B$ by dividing by $2^{\textbf{state}[\text{LUMA\_DEPTH}]}$ and clipping to $[0,1]$. If the inverse transform has been correctly applied prior to coding and lossless coding employed, then clipping will be unnecessary, and reversing the above operations will reproduce $Y$, $C_O$ and $C_G$ losslessly from

$I_R$, $I_G$ and $I_R$ yielding a transparent RGB to RGB coding system:

$$\begin{aligned}
Co &= I_R - I_B \\
t &= I_B + (I_R - I_B) \gg 1 \approx (I_R + I_B)/2 \\
Cg &= I_G - t =\approx I_G - (I_R + I_B)/2 \\
Y &= t + (Cg \gg 1) \approx I_G/2 - (I_R + I_B)/4 + (I_R + I_B)/2 = I_R/4 + I_G/2 + I_B/4
\end{aligned}$$

Note that these matrix operations imply that the chroma data requires an additional bit, due to the subtractions used to create chroma components. So for 8-bit RGB ($I_R$, $I_G$, $I_B$) values, $Y$ will be 8 bits and $C_O$ and $C_G$ will be 9 bits.

## F.2 Transfer characteristics

### F.2.1 TV transfer characteristic

ITU-R BT.601-6 defines the 625-line and 525-line standard definition systems with an assumed receiver display gamma value of 2.8. SMPTE 170M defines the NTSC SDTV system with an assumed receiver display gamma value of 2.2.

High Definition systems for both 50Hz and 60Hz based systems use an encoding gamma value of 0.45 with a linear portion at the low end of the scale to avoid the need for infinite gain at the receiver. This gamma value is defined by ITU-R BT.709.

### F.2.2 Extended Colour Gamut

ITU-R BT 1361 (Worldwide Unified Colorimetry of Future TV Systems) defines a color system with an extended colour gamut. Refer to ITU-R BT 1361 (1998) for details.

ISO/IEC 61966-2 (Extended RGB Color Space) defines another colour system with an extended color gamut. Refer to IEC 61966-2-2:2003 for details.

In both cases, it should be noted that use of the full range of $Y, C1, C2$ values can create negative R, G or B values. The original color gamut equations were designed around the CRT (cathode ray tube) device. Some flat panel displays are capable of displaying a wider color gamut resulting in the desire to extend the color gamut to maximize the impact of these displays.

#### F.2.2.1 Linear

A linear transfer characteristic has $f(x) = x$ i.e. $E_X = X$.

## F.3 Frame rate

The ratio of the frame rate values **video_params**[FRAME_RATE_NUMER] and **video_params**[FRAME_RATE_DENOM] encodes the intended rate at which frames should be displayed subsequent to decoding. If **video_params**[SOURCE_SAMPLING] is 1 (interlaced sampling), then fields are displayed at double the frame rate, in the order specified by the **video_params**[TOP_FIELD_FIRST] flag.

## F.4 Aspect ratios and clean area

### F.4.1 Pixel aspect ratio

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 whole image. Failure to use the correct value of pixel aspect ratio will result in distorted images where circles will be displayed as ellipses.

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

For a number NH of pixels per unit length and NV pixels per unit height, this ratio is 1/NH : 1/NV or NV : NH. For a video standard of WxH pixels displayed at 4:3 picture aspect ratio, NH=W/4 and NV=H/3.

#### F.4.1.1 Using non-square pixel aspect ratios

The defined pixel aspect ratios are designed to give image aspect ratios for standard definition television operating with a standard 4:3 picture aspect ratio.

For 525-line video, defining a 704 x 480 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 10:11 (i.e. 480/3 : 704/4 ).

For 625-line video defining a 704 x 576 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 12:11 (i.e. 576/3 : 704/4 ).

If the intended image aspect ratio is 16:9, then the H:V pixel aspect ratios change accordingly to 40:33 for 525-line video and 16:11 for 625-line video.

The values specified above are widely, but not unanimously, agreed to be the correct values. Differences of viewpoint arise from how much of the available horizontal picture size of 720 Y pixels is intended for display.

You are strongly advised to use one of the default pixel aspect ratios. However, if you know what you are doing and dont like the default values the codec allows you to define your own ratio. You should be aware that many display devices could ignore your decision and may default to using different and unsuitable values.

### F.4.2 Clean area

The clean area is intended to define an area within which picture information is subjectively uncontaminated by all edge distortions and possible unintended picture content such as microphones appearing at the top of the picture. It could be appropriate to display the clean area rather than the whole picture, which can contain edge distortions or unintended content.

The top-left corner of the clean area has coordinates

$$(\textbf{video\_params}[\text{CLEAN\_LEFT\_OFFSET}], \textbf{video\_params}[\text{CLEAN\_TOP\_OFFSET}])$$

counting from the top-left corner of the picture data, and dimensions **video_params**[CLEAN_WIDTH] by **video_params**[CLEAN_HEIGHT].

Note that these dimensions refer to pixels within a picture, not a frame, so a change from interlaced to progressive picture coding will necessitate a change of clean area if a custom clean area is used.

The clean area and the pixel aspect ratio together determine the aspect ratio of the displayed image which is the ratio of the width of the intended display area to the height of the intended display area:

$$\frac{\textbf{video\_params}[\text{CLEAN\_WIDTH}] * \textbf{video\_params}[\text{PIXEL\_ASPECT\_RATIO\_NUMER}]}{\textbf{video\_params}[\text{CLEAN\_HEIGHT}] * \textbf{video\_params}[\text{PIXEL\_ASPECT\_RATIO\_DENOM}]}$$

Given two separate sequences, with identical image aspect ratio, if the top left corner and bottom right 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.

# G Wavelet transform and lifting (Informative)

This is an informative annex introducing the fundamentals of wavelet filtering and the lifting scheme.

## G.1 Wavelet filter banks

Figure G.1 below illustrates a single stage of a generalized wavelet decimation followed by reconstruction. The aim is to get perfect reconstruction of the output so that it is identical to the original input. The filters

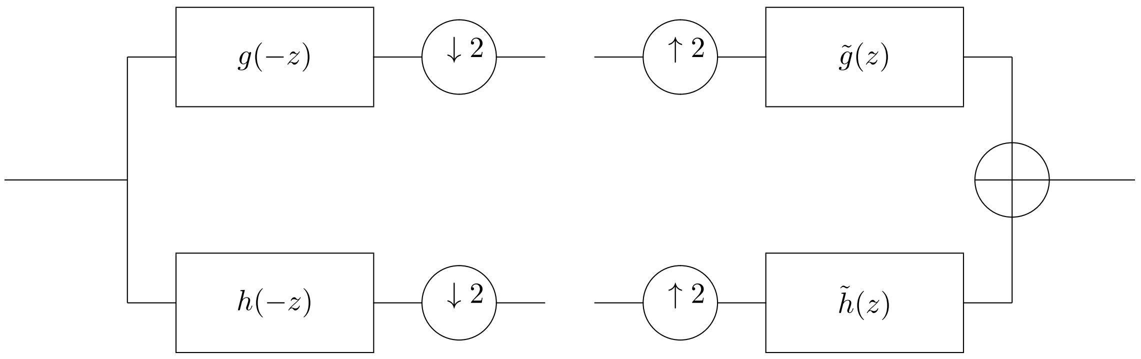

Figure G.1: Wavelet decimation and reconstruction

$h(z)$ and $g(z)$ are the low-pass and high-pass analysis filters, whilst $\tilde{h}(z)$ and $\tilde{g}(z)$ are the synthesis filters. The filters must satisfy the conditions

$$\begin{aligned} h(z)\tilde{h}(z^{-1}) + g(z)\tilde{g}(z^{-1}) &= 2 \text{ (Perfect reconstruction)} \\ h(z)\tilde{h}(-z^{-1}) + g(z)\tilde{g}(-z^{-1}) &= 0 \text{ (Alias cancellation)} \end{aligned}$$

These conditions imply that the synthesis filters are derived from the analysis filters and vice-versa:

$$\begin{aligned} \tilde{g}(z) &= z^{-1}h(-z^{-1}) \\ \tilde{h}(z) &= z^{-1}g(-z^{-1}) \end{aligned}$$

If we have an *orthogonal* wavelet decomposition, then additionally $H = \tilde{h}$ and $g = \tilde{g}$ and there is a single "mother" wavelet.

The next figure (F.2) illustrates how the frequency components are distributed both during decimation and reconstruction. This figures illustrates how the alias frequencies created during the decimation process are cancelled out during the reconstruction process. This feature of alias cancellation results from the wavelet process and is a specific attribute of wavelet coding. It is important to note that if the decoder receives imperfect signals (caused, for example, by quantisation errors) then the imperfections will result in distortion in the reconstructed output.

Figure F.2 -Illustration of the alias frequency generation and cancellation in a wavelet filter bank A single wavelet stage is insufficient for most video coding applications. The figure below illustrates how only the low-pass path is passed on to the next wavelet decimation step. Because each step of the wavelet decimation is self-contained, the reconstructed output is still identical to the input (barring quantisation errors).

Figure F.3 -Two-step wavelet processing filter bank The application of wavelet filter banks in picture coding results in a two-dimensional decimation process as illustrated in figure F.4 below.

Figure F.4 -Decomposition of a single image into 7 wavelet frequency bands The final figure illustrates how a real image is decimated to produce a low-frequency proxy in the top-left corner and a range of increasing

frequency band components extending to the right side for increasing horizontal frequencies and downwards for increasing vertical frequencies.

Figure F.5 -Decomposition of the EBU "Boats" picture into 7 wavelet frequency bands

## G.2 Lifting

For any set of filters, the analysis and synthesis filter banks shown in Figure G.1 can easily be re-expressed as polyphase filter banks by means of applying *matrices* of filters in the subsampled domain. This is shown in Figure G.2, where $A(z)$ is the $z-$transform of the analysis polyphase filter matrix, and $S(z)$ is the $z-$transform of the synthesis polyphase filter matrix (the entries of both matrices being Laurent polynomials). In this

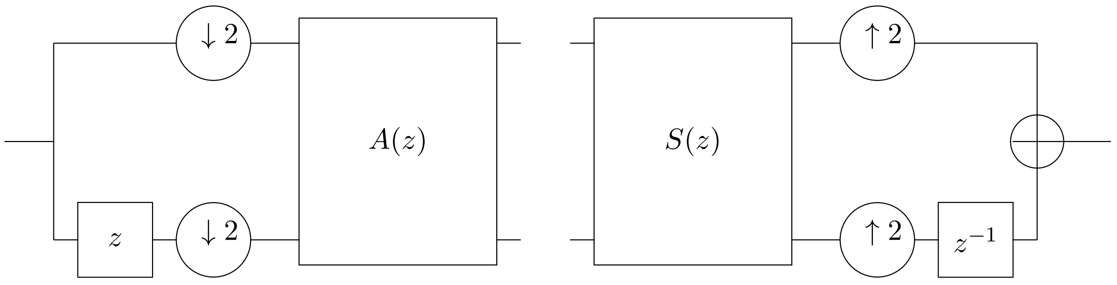

Figure G.2: Polyphase representation of wavelet filter banks

representation, linear combinations of filters operate on both even and odd samples to produce new even and odd samples:

$$\begin{pmatrix} x_e^{out}(z) \\ x_o^{out}(z) \end{pmatrix} = A(z) \begin{pmatrix} x_e^{in}(z) \\ x_o^{in}(z) \end{pmatrix}$$

Since the filter process is invertible, it can be shown that the analysis and synthesis matrices are related by $A(z) = (S(z^{-1})^T)^{-1}$. Hence, in particular both the analysis and synthesis matrices are invertible. It can be shown that this means that they are (up to gain factors and delays) factorisable into products of upper and lower triangular matrices:

$$A(z) = \begin{pmatrix} 1 & a_1(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b_1(z) & 1 \end{pmatrix} \begin{pmatrix} 1 & a_2(z) \\ 0 & 1 \end{pmatrix} \dots$$

Each upper- or lower-triangular polyphase matrix represents a so-called *lifting* stage whereby either even coefficients are modified solely by odd coefficients or odd coefficients solely by even coefficients. For example, if

$$\begin{pmatrix} x_e^{out}(z) \\ x_o^{out}(z) \end{pmatrix} = \begin{pmatrix} 1 & a(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_e^{in}(z) \\ x_o^{in}(z) \end{pmatrix}$$

then

$$\begin{aligned} x_e^{out}(z) &= x_e^{in}(z) + a(z)x_o^{in}(z) \\ x_o^{out}(z) &= x_o^{out}(z) \end{aligned}$$

and the filter $a(z)$ has been applied to the odd coefficients and then used to modify the even coefficients. Not only is this computationally efficient, breaking long filters into a number of shorter filter applied successively but the factorisation into such filter stages allows for all computations to be done in-place, without additional memory.