Dirac Specification

Version 0.10.1

Abstract

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

Dirac is a video compression system 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
- both intra-frame and motion-prediction coding
- gradual quality loss with increasing compression

Contents

1 Introduction **6**

1.1 Purpose 6

1.2 Scope 6

1.3 Status 6

1.4 Document structure 6

2 The conventions used in the specification **8**

2.1 State machine decoder representation 8

2.2 Numbers and arithmetic operations 8

2.2.1 Numbers 8

2.2.2 Arithmetic operations 8

2.3 Pseudocode 10

2.3.1 Processes 10

2.3.2 Variables and data types 11

2.3.3 Control flow 12

2.3.4 Logical (boolean) operations 13

I Stream access **15**

3 Data encodings **15**

3.1 Bit-packing and data input 15

3.2 Fixed-length code formats 15

3.2.1 Bool 15

3.2.2 n-byte unsigned integer literal 16

3.3 Variable-length code formats 16

3.3.1 Unsigned interleaved exp-Golomb 16

3.3.2 Signed interleaved exp-Golomb 17

4 Arithmetic decoding **18**

4.1 State and contexts 18

4.1.1 Rescaling contexts 18

4.2 Initialisation 18

4.3 Data input 19

4.4 Decoder functions 19

4.4.1 Shifting bits in 19

4.4.2 Decoding boolean values 20

4.4.3 Arithmetic decoding of integer values 20

4.5 Context indices 21

II Stream parsing 24

5 Dirac stream specification 24

5.1 Introduction 24
5.2 Stream 24
5.2.1 Sequence 24
5.3 Parse Info header 25
5.4 Access Units 27
5.5 Access Unit header 27
5.6 Access unit parse parameters 28
5.6.1 AU picture number 28
5.6.2 Version number 28
5.6.3 Profiles and levels 28
5.7 Access unit sequence parameters 28
5.7.1 Setting format defaults 29
5.7.2 Custom image dimensions 29
5.7.3 Chroma formats 29
5.7.4 Video depth 30
5.8 Access unit source parameters 30
5.8.1 Scan format 31
5.8.2 Frame rate 31
5.8.3 Aspect ratio 32
5.8.4 Clean area 32
5.8.5 Signal range 33
5.8.6 Colour specification 33
5.9 Picture 34
5.9.1 Initialising decoding parameters 34
5.9.2 Picture header 35
5.10 Picture prediction 35
5.10.1 Picture prediction parameters 36
5.10.2 Block parameters 36
5.10.3 Setting chroma block parameters 37
5.10.4 Motion vector precision 37
5.10.5 Global motion 37
5.10.6 Picture prediction mode 39

5.10.7 Reference picture weight values . . . 39

5.11 Wavelet transform . . . 39

5.11.1 Wavelet transform parameters . . . 40

5.11.2 Wavelet depth . . . 41

5.11.3 Spatial partition of wavelet data . . . 41

6 Motion data decoding 43

6.1 Motion data conventions . . . 43

6.2 Motion data decoding process . . . 43

6.2.1 Overall decoding loop . . . 44

6.2.2 Motion data initialisation . . . 44

6.2.3 Superblock decoding . . . 45

6.2.4 Block decoding . . . 45

6.2.5 Spatial prediction of motion data elements . . . 47

6.2.6 Block motion data contexts . . . 50

7 Wavelet coefficient decoding 52

7.1 Decoded subband data conventions . . . 52

7.1.1 Wavelet data initialisation . . . 52

7.1.2 Dimensions of wavelet subbands . . . 52

7.2 Wavelet data decoding process . . . 53

7.2.1 Summary . . . 53

7.2.2 Decoding loop . . . 54

7.3 Subband decoding process . . . 54

7.3.1 Subband header and codeblock loop . . . 54

7.3.2 Decoding subband codeblocks . . . 55

7.3.3 Intra DC band prediction . . . 56

7.4 Subband coefficient decoding process . . . 57

7.4.1 Overall coefficient decoding process . . . 57

7.4.2 Parent values . . . 57

7.4.3 Zero neighbourhood . . . 58

7.4.4 Sign prediction . . . 58

7.4.5 Coefficient context selection . . . 58

7.4.6 Inverse quantisation . . . 59

7.4.7 Quantisation factors and offsets . . . 60

7.4.8 Updating counts and resetting contexts . . . 60

III Decoding operations 61

8 Picture decoding process **61**

8.1 Introduction 61

8.2 Overall picture decoding process 61

8.2.1 Picture data initialisation 61

8.2.2 Initialisation 61

8.2.3 Decoding process 62

8.3 Seeking in the Dirac stream 62

8.4 Reference picture buffer management 62

8.5 Clipping 63

9 Inverse discrete wavelet transform **64**

9.1 IDWT synthesis operation 64

9.2 Vertical and horizontal synthesis 64

9.3 One-dimensional synthesis 66

9.4 Filters and shift values 67

9.5 Removal of IDWT pad values 70

10 Motion compensation **71**

10.1 Definitions and conventions 71

10.2 Overlapped Block Motion Compensation (OBMC) (Informative) 72

10.3 Overall motion compensation process 73

10.4 Pixel prediction 73

10.5 Spatial weighting matrix 76

10.6 Block prediction 77

10.7 Global motion vector field generation 77

10.7.1 Chroma scaling 77

10.7.2 Field generation 78

10.8 Upconversion 78

10.8.1 Pixel-accurate motion vectors 78

10.8.2 Half-pixel accurate motion vectors 79

10.8.3 Quarter- and eighth-pixel accurate motion vectors 79

10.9 Implementation 80

A Parse diagrams **81**

B Video systems model and source parameters **92**

B.1 Colour 92

B.2 Frame rate 92

B.3 Aspect ratios and clean area 93

B.4 Signal range 93
B.5 Colour primaries 94
B.6 Colour matrix 94
B.7 Transfer charactaristics 95
B.8 Source parameter presets 95

C Video format defaults **97**

D Profiles and levels **100**

1 Introduction

1.1 Purpose

Dirac was 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 began as an Open Source software project, and reference implementations of the decoder and encoder are available at http://sourceforge.net/projects/dirac.

1.2 Scope

This document specifies normative decoder operations ("semantics") and stream syntax. The stream syntax is primarily specified by means of pseudocode, the conventions of which are described in Section 2.3. The decoder semantics are specified by means of a mixture of pseudocode and conventional mathematical symbolism.

A number of other elements are also included for informative purposes. The specification is not an implementation guide, and in the interests of clarity many of the operations are specified in a way that would not be efficient to implement. However, we have attempted to indicate where this is so, and to suggest ways in which an efficient implementation may be achieved, but these are by no means exhaustive. An optimised Open Source software Dirac encoder and decoder system, named Schrödinger, is available at http://sourceforge.net/projects/schrodinger, and may be studied to aid implementation.

In addition, we are well aware that many users of this document may wish to make both encoders and decoders. There are many sources of information on how to design efficient compression algorithms, for example for entropy coding, motion estimation, frame-dropping, rate control, motion estimation and rate-distortion optimisation. This document does not attempt to address these issues in detail, but to provide supplementary information where appropriate to allow those reasonably "skilled in the art" to develop a Dirac encoder rapidly and accurately, and approach design compromises knowledgably.

1.3 Status

This is version 0.10.0 of the Dirac specification. The document includes a full description of the core Dirac stream syntax and decoder operations. It does not yet contain a specification of profiles and levels supported by Dirac, or the compatible extensions required to support the Dirac Pro toolset. These shall be added shortly.

1.4 Document structure

This document specifies the Dirac decoder and stream structure in terms of a layered model:

1. Stream data access
2. Parsing and interpretation of the Dirac stream
3. Picture decoding operations

Stream data access consists of the operations used to extract data values (of boolean and integer type) from a raw Dirac bitstream. These include data that has been encoded "literally" (i.e. according to

conventional bit-wise representations), variable-length codes, and data entropy coded using arithmetic encoding. Stream data access methods are used both by parsing and decoding operations.

Parsing and interpretation defines the structure of Dirac streams, and defines intermediate decoder data structures in which extracted data is stored, which encapsulate *both* meta-data used to control picture decoding processes (for example, motion compensation block sizes and overlaps, picture dimensions and so forth) *and* the blocks of (arithmetically coded) data used as input to these processes.

Picture decoding operations produce decoded pictures from these populated data structures by applying specified functions to them. They are not necessary for navigating the stream or reading any of the stream data, but only for outputting pictures.

Note in particular that the distinction between parsing and picture decoding is *not* exactly that between syntax and semantics: complex semantics are required for correct parsing of the stream as well as for decoding pictures.

It is perhaps unusual in a specification to separate these layers quite so distinctly, and our purpose in doing so is to provide much greater clarity. For implementors, we hope that the decoupling of the stream structure from the (computationally intensive) picture decoding processes will (we hope) avoid imposing implicit design decisions merely through the style of the specification. Many other users of the specification will not be interested in the precise format of stream elements but in how the underlying algorithm works - or vice-versa. It should be possible to construct a Dirac parsing engine, for example for frame skipping in video playback applications, extremely simply and without requiring comprehension of the entire specification.

This layered structure is reflected in the structure of the specification, which, after defining conventions used in the specification is divided into three corresponding parts: stream data access, defining functions for data types; accessing and parsing the Dirac bitstream and populating data structures (including the wavelet coefficients and motion data); and high-level decoder operations and picture output, specifically the inverse wavelet transform and motion compensation.

In addition to these parts, appendices deal with standard settings, parameter presets and levels and profiles.

2 The conventions used in the specification

2.1 State machine decoder representation

This specification uses a state-machine model to express parsing and decoding operations. The state of the decoder/parser is stored in the global variable **state**, and individual variable values are accessed by means of named tokens, e.g. **state**[*var_name*] (i.e. they are *maps* as defined in Section 2.3.2). All the individual variables are therefore also globally accessible from all decoder functions.

A default state variable, **default_state** is also defined, which is initialised on accessing the stream. State values revert to the default state values on beginning to parse each picture, as described in Section 5.

The parsing and decoding operations are specified in terms of modifying the decoder state. Decoder state variables may not directly correspond to elements of the stream, but are 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 stream structure itself is summarised in parse diagrams, which are presented in Appendix A. The parsing process is defined by means of pseudocode and/or mathematical formulae. The conventions for these elements are described in the succeeding sections. In the event of any conflict between the parse diagrams and the specified parsing processes, the latter shall be deemed to be correct.

2.2 Numbers and arithmetic operations

2.2.1 Numbers

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

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

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

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

2.2.2 Arithmetic operations

All arithmetic defined by this specification is exact: the entire specification can be implemented using only integer and logical operations. All operations are to be implemented with sufficiently large integers so that overflow cannot occur.

The following arithmetic operators are defined on numerical values:

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

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 $a * b$.

Real division The real number value of a divided by b is represented by a/b or $\frac{a}{b}$.

Exponentiation For integers a, b, $b > 00$ a^b is defined as $a * a * \ldots * a$ (b times). a^0 is 1.

Ceiling $\lceil a \rceil$, the smallest integer greater than or equal to a real number a

Floor $\lfloor a \rfloor$, the smallest integer greater than or equal to a real number a

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

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

Clip $\text{clip}(a, b, t)$ clips the value a to the range defined by b and t:

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

Integer division Integer division is defined for a and b integer values, $b > 0$ by

$$a//b = \left\lfloor \frac{a}{b} \right\rfloor$$

(i.e. always round down).

Remainder For integers a, b, with $b > 0$, $a\%b$ is equal to $a - (a//b) * b$. $a\%b$ always lies between 0 and $b - 1$.

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

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

Mean Given a set $S = \{s_0, s_1, \ldots, s_{n-1}$ of integer values, the integer mean $\text{mean}(S)$ 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 $\text{median}(S)$ 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

$(t_{(n-2)/2} + t_{n/2})//2$ if n is even.

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.

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

These operations are also defined on boolean values, interpreted as single-bit integers, where 0 is interpreted as **False** and 1 as **True** and vice-versa. Logical NOT is not defined bitwise, to avoid ambiguity concerning leading zeroes).

2.3 Pseudocode

The bulk of the normative specification is defined by means of pseudocode. The syntax used is an amalgam of Python and Basic. It is not intended to be executable code, but rather both precise and descriptive. Iin particular, all the arithmetic functions and operations defined in the preceding section may be applied to variables within a pseudocode process.

2.3.1 Processes

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:

$foo(in1, in2)$:	
$op1(in1)$	
$op2(in2)$	
...	

whilst a function process looks like

$bar(in1, in2)$:	
$op1(in1)$	
$foo(in1, in2)$	2.3.1
...	
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.

Note well: all input variables are deemed to 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 labels for pointers to workspace memory.

For example, if we define foo and bar by

$foo()$:	
$num = 0$	
$bar(num)$	
state[var_name] $= num$	

and

$bar(val)$:	
$val = val + 1$	

then at the end of foo, **state**[var_name] has been set to 1.

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:

$foo()$:	
$op1()$	
...	

[text]

...	
$op2()$	
...	

[text]

...	
$op3()$	

Note that 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.

2.3.2 Variables and data types

The only global variables are the state variables encapsulated in **state** and **default_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.

The following basic types are defined:

Boolean A boolean variable has two possible states, **True** and **False**.

Unsigned integer A non-negative (≥ 0) whole number, of arbitrary size.

Integer A whole number, of arbitrary size.

Set A collection of variables or values, with no particular indexing. The usual set-theoretic operations such as $\cup$ (union), $\cap$ (intersection), $\in$ (membership) and so on apply.

Map A map is a set accessed by token names. For example $p[Y]$, $p[U]$, $p[V]$ might give the value of the different video components (Y, U and V) of a pixel. The set of argument tokens of a map m can be accessed by $\text{args}(m)$, so that $\text{args}(p) = \{Y, U, V\}$.

Array A list is a set with an integer index or indices. All arrays are indexed from 0. Elements of a 1-dimensional array a are accessed by $a[n]$ for n in the range 0 to $\text{length}(a) - 1$. Elements of a 2-dimensional array are accessed by $a[n][m]$ for $0 \leq m \leq \text{width}(a) - 1$ and $0 \leq n \leq \text{height}(a) - 1$.

A 1-dimensional array can be explicitly defined by the syntax $a = [u, v, w, \ldots]$. Then $a[0] = u$, $a[1] = v$ and so on.

These basic variable types may be combined. For example, picture data may be considered to be a map of arrays pic, where $pic[Y]$ is a 2-dimensional array storing luma data, and $pic[U]$ and $pic[V]$ are two-dimensional arrays storing chroma data.

Variables within processes are not explicitly declared, and their type is determined from context or defined in the surrounding description.

Assignment between variables a and b is denoted by $a = b$, and is a copy operation between the data contained within b to a.

It is to be distinguished from the boolean identity operator $a == b$ (Section 2.3.4).

Occasionally the notation $a = \mathbf{0}$ will be used for an array of integer values: it means set all elements of the matrix to 0.

For integer variables, assignment can be combined with arithmetic and bit-wise operations in the usual programming manner: for example,

$$x+ = y$$

means $x = x + y$, and

$$x| = \text{0x4B}$$

means $x = x|0x4B$.

2.3.3 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 (*control*1):	
*block*1	
else if (*control*2):	
*block*2	
else if (*control*3):	
*block*3	
else:	
*block*4	

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, U, V:	
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*):	
*block*1	
*block*2	

2.3.4 Logical (boolean) operations

A logical operator takes a variable or pair of variables as arguments and returns the boolean values **True** or **False**.

The following logical operators are defined:

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

$<$ Less than

$\leq$ Less than or equal to

$>$ Greater than

$>=$ Greater than or equal to

! Not. $!a$ is **True** for a boolean value a if and only if a is **False**

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

When used in pseudocode conditions, the words “and” and “or” are used to denote logical AND and logical (inclusive) OR between boolean values, for example:

if (*condition*1 and *condition*2):	
...	

Majority Given a set $S = s_0, \ldots, s_{n-1}$ of boolean values, majority(S) returns mean(S) where the elements of S are intepreted as 0 if **False** and 1 if **True**. So 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**.

Part I

Stream access

3 Data encodings

Data is encoded in the Dirac bitstream in three basic ways: fixed-length bit-wise and byte-wise encodings; variable-length codes; and arithmetic encoding.

This section defines how data bits are extracted from the bitstream and how sequences of bits are interpreted as values of various types using fundamental data-reading functions, covering encodings of the first two sorts. The extraction of arithmetic-encoded data is defined in Section 4.

3.1 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.

Access to the Dirac stream is bytewise, and a decoder is deemed to maintain a copy of the current byte, **state**[current_byte], and an index to the next bit to be read, **state**[next_bit]. **state**[next_bit] is an integer from 0 (least-significant bit) to 7 (most-significant bit). Bits within bytes are accessed from the msb first to the lsb.

Each access unit and individual frame is a whole number of bytes. Decoding from the start of an access unit, **state**[next_bit] is set to 7.

The *read_byte*() function –

- returns **state**[current_byte] if **state**[next_bit] == 7
- sets **state**[next_bit] = 7 and returns the next byte in the Dirac stream otherwise

The *read_bit*() function is defined by

read_bit() :	
if (**state**[next_bit] == 7):	
state[current_byte] = *read_byte*()	
bit = (**state**[current_byte] ≫ **state**[next_bit])&1	
state[next_bit]− = 1	
if (**state**[next_bit] < 0):	
state[next_bit] == 7	
return *bit*	

The *byte_align*() function discards data in the current byte and begins data access at the next byte, unless input is already at the beginning of a byte:

byte_align() :	
state[next_bit] = 7	

This is used to ensure that a whole number of bytes are read before beginning reading a new stream element.

3.2 Fixed-length code formats

3.2.1 Bool

The *read_bool*() function returns –

- **True** if *read_bit*() is 1
- **False** if *read_bit*() is 0

3.2.2 n-byte unsigned integer literal

A single byte may be interpreted as an unsigned integer value from 0 to 255.

An n-byte number in literal format shall be decoded by the recipe:

read_uint_lit(n) :	
$val = 0$	
for $i = 0$ to $n - 1$:	
$val += read_byte()$	3.1
$val \ll= 8$	
return *val*	

3.3 Variable-length code formats

Variable-length codes are used in two ways in the Dirac stream. The first use is for direct encoding into the stream. The second use is for binarisation in the arithmetic encoding/decoding process so that integer values may be coded and decoded using a binary arithmetic coding engine.

3.3.1 Unsigned interleaved exp-Golomb

This section defines the unsigned interleaved exp-Golomb data format and the operation of the *read_uint*() function.

Unsigned interleaved exp-Golomb data is decoded to produce unsigned integer values.The format consists of two interleaved parts, and each code is an odd number, $2K + 1$ bits, in length.

The $K + 1$ bits in the even positions (counting from zero) are the "follow" bits, and the K bits in the odd positions are the "data" bits b_i which are used to construct the decoded value itself. A follow bit value of 0 indicates a subsequent data bit, whereas a follow bit value of 1 terminates the code:

$$0 \quad b_{K-1} \quad 0 \quad b_{K-2} \ldots 0 \quad b_0 \quad 1$$

The data bits $b_{K-1}, b_{K-2}, \ldots, b_0$ are the binary representation of the first K bits of the $(K + 1)$-bit number $N + 1$, where N is the number to be decoded:

$$N + 1 = 1b_{K-1}b_{K-2} \ldots b_0(\text{base } 2)$$

A table of encodings of the first 10 values is shown in Figure 1.

Although apparently complex, the interleaving ensures that the code has a very simple decoding loop. The *read_uint*() function returns an unsigned integer value and is defined by the recipe:

read_uint() :	
$value = 1$	
while (*read_bool*() == **False**):	
$value \ll= 1$	
if (*read_bool*() == **True**):	
$value += 1$	
$value -= 1$	
return *value*	

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

Figure 1: Example conversions from unsigned interleaved exp-Golomb-coded values to unsigned integers

Informative: 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.

3.3.2 Signed interleaved exp-Golomb

This section defines the signed interleaved exp-Golomb data format and the operation of the $read_sint()$ function.

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 (Figure 2).

Bit sequence	Decoded value
1	0
0 0 1 1	-1
0 0 1 0	1
0 1 1 1	-2
0 1 1 0	2
0 0 0 0 1 1	-3
0 0 0 0 1 0	3
0 0 0 1 1 1	-4
0 0 0 1 1 0	4

Figure 2: Example conversions from signed interleaved exp-Golomb-coded values to signed integers

The decoding operation is as follows.

$read_sint()$:	
$value = read_uint()$	
if ($read_bool()$ == **True**):	
$value = -value$	
return $value$	

4 Arithmetic decoding

This section describes the arithmetic decoding engine and processes for using it to extract data from the Dirac stream.

The arithmetic decoding engine consists of two elements:

- a collection of state variables representing the state of the arithmetic decoder (Section 4.2)
- a set of functions for extracting values from the decoder and updating the decoder state (Section 4.4)

4.1 State and contexts

The arithmetic decoder state consists of the following decoder state variables:

- **state**[low], an integer representing the beginning of the current coding interval
- **state**[high], an integer representing the end of the current coding interval
- **state**[code], an integer within the interval from **state**[low] to **state**[high], determined from the encoded bitstream
- **state**[bits_left], a decrementing count of the number of bits yet to be read in
- **state**[contexts], a map of all the contexts used in the Dirac decoder

A context *context* is an integer array consisting of two positive values, *context*[0], and *context*[1], representing counts of values 0 and 1 respectively. Contexts are accessed by decoding functions via the indices defined in Section 4.5.

4.1.1 Rescaling contexts

An individual context is rescaled by halving the counts of 0 and 1 and ensuring that these counts do not reach zero:

rescale_context(*context*) :	
$context[0] \gg= 1$	
$context[0] += 1$	
$context[1] \gg= 1$	
$context[1] += 1$	

4.2 Initialisation

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

initialise_arithmetic_decoding(*block_data_length*) :	
state[bits_left] $= 8 * block_data_length$	
state[low] = 0x0000	
state[high] = 0x0000	
state[code] = 0x0000	
init_contexts()	

Contexts are initialised by the *init_contexts*() function as follows:

init_contexts() :	
for $i = 0$ to $length$(**state**[contexts]) $- 1$:	
state[contexts][i][0] $= 1$	
state[contexts][i][1] $= 1$	

4.3 Data input

The arithmetic decoding process accesses data in a contiguous block of bytes whose size is set on initialisation (Section 4.2). The bits in this block are 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 a read function *read_bita*() is defined which returns 0 if the bounds of this block of data have been exceeded:

read_bita() :	
if (**state**[bits_left] == 0):	
return 0	
else:	
state[bits_left]$- = 1$	
return *read_bit*()	

Informative: The Dirac arithmetic decoding engine uses 16 bit words, and so no more than 16 additional bits can be read beyond the end of the block. Hence it is sufficient to read in the entire block of data and pad the end with two zero bytes to avoid a branch condition with each input bit.

4.4 Decoder functions

The arithmetic decoding engine is a multi-context, adaptive binary arithmetic decoder, performing binary renormalisation and producing binary outputs. For each bit decoded, the semantics of the relevant calling decoder function determine which contexts are passed to the arithmetic decoding operations.

4.4.1 Shifting bits in

This section defines the operation of the *shift_bit_in*() and *shift_all_bits*() functions for reading bits into the arithmetic decoding state variables.

shift_bit_in() :	
state[high] $\ll= 1$	
state[high]$\& =$ 0xFFFF	
state[high]$+ = 1$	
state[low] $\ll= 1$	
state[low]$\& =$ 0xFFFF	
state[code] $\ll= 1$	
state[code]$\& =$ 0xFFFF	
state[code]$+ =$ *read_bita*()	4.3

shift_all_bits() expands the interval between **state**[low] and **state**[high] until the msbs (bit 15) differ and the interval no longer straddles the half-way point 0x8000.

$shift_all_bits()$:	
while (**state**[high]&0x8000) == 0x0&&(**state**[low]&0x8000) == 0x0):	
$shift_bit_in()$	
while ((**state**[high]&0x4000) == 0x0and(**state**[low]&0x4000) == 0x4000):	
state[code]ˆ = 0x4000	
state[high]ˆ = 0x4000	
state[low]ˆ = 0x4000	
$shift_bit_in()$	

Informative: Note that if 16-bit words (unsigned shorts) are used for decoder state variables **state**[low], **state**[high] and **state**[code] then there is no need for &-ing with 0xFFFF. However, the operations specified here are defined in terms of integers, since intermediate calculations require higher dynamic range. In software, the efficiency of using short word lengths may or may not be offset by the requirement to cast to other data types for these calculations.

4.4.2 Decoding boolean values

This section specifies the operation of the $read_boola()$ function for extracting a boolean value from the Dirac stream. Before extracting any values, all possible bits are shifted in to ensure that the decoding state has maximum information.

$read_boola(context_index)$:	
$shift_all_bits()$	4.4.1
$context =$ **state**[contexts]$[context_index]$	
$weight = context[0] + context[1]$	
$scaler = (0x10000 + weight//2)//weight$	
$probability0 = context[0] * scaler$	
$count = code - low + 1$	
$range = high - low + 1$	
$range_times_prob = (range * probability0) >> 16$	
if $(count > range_times_prob)$:	
$value =$ **True**	
$low = low + range_times_prob$	
$context[1] += 1$	
else:	
$value = False$	
$high = low + range_times_prob - 1$	
$context[0] += 1$	
if $((context[0] + context[1]) > 255)$:	
$rescale_context($**state**[contexts]$[context_index])$	4.1.1
return $value$	

Informative: The function scales the probability of 0 from the decoding context so that a probability of 1 is commensurate with the interval between **state**[low] and **state**[high]. If **state**[code] is greater than this cut-off, then 1 (**True**) has been encoded, else 0 (**False**) has.

4.4.3 Arithmetic decoding of integer values

This section defines the operation of the $read_sinta(context_set)$ function for extracting integer values from a block of arithmetically coded data.

4.4.3.1 Binarisation and contexts

Signed and unsigned integer values are binarised using interleaved exp-Golomb binarisation as per Section 3.3: the $read_sinta()$ and $read_uinta()$ processes are essentially identical to the $read_sint()$ and $read_uint()$ processes, except that instances of $read_bool()$ are replaced by instances of $read_ba()$ (Section 4.4.2) using suitable contextualisation.

A choice of context depends upon whether the bit is a data bit, follow bit, or sign bit, and the position of the bit within the binarisation: $context_set$ consists of three parts -

- an array of follow contexts, $context_set[follow]$ (indexed from 0 to length($context_set[follow]$) $-$ 1)
- a single data context $context_set[data]$
- a sign context $context_set[sign]$ (ignored for unsigned integer decoding)

Each follow context is used for decoding the corresponding follow bit, with the last follow context being used for all subsequent follow bits (if any) also. The follow context selection function $follow_context()$ is defined by:

$follow_context(index, context_set)$:	
$pos = max(index, length(context_set[follow]) - 1$	
$ctx_index = context_set[follow][pos]$	
return **state**[contexts][ctx_index]	

So the last follow context is used for all the remaining follow bits also.

4.4.3.2 Unsigned integer decoding

Unsigned integers are decoded by:

$read_uinta(context_set)$:	
$value = 1$	
$index = 0$	
while ($read_ba(follow_context(index, context_set))$ == **False**):	
$value \ll= 1$	
if ($read_ba$(state[contexts][$context_set[data]$])):	
$value+ = 1$	
$index+ = 1$	
$value- = 1$	
return $value$	

4.4.3.3 Signed integer decoding

$read_sinta()$ decodes first the magnitude then the sign, as necessary:

$read_sinta(context_set)$:	
$value = read_uinta(context_set)$	
if ($value! = 0$):	
if ($read_ba$(**state**[contexts][$context_set[sign]$])) == **True**):	
$value = -value$	
return $value$	

4.5 Context indices

The following is a list of all the context indices used in Dirac arithmetic decoding operations:

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_INFO
Q_OFFSET_SIGN
SB_F1
SB_F2
SB_DATA
PMODE_REF1
PMODE_REF2
GLOBAL_BLOCK
REF1x_F1
REF1x_F2
REF1x_F3
REF1x_F4
REF1x_F5+
REF1x_DATA
REF1x_SIGN
REF1y_F1
REF1y_F2
REF1y_F3
REF1y_F4
REF1y_F5+
REF1y_DATA
REF1y_SIGN
REF1x_F1
REF1x_F2
REF1x_F3
REF1x_F4
REF1x_F5+
REF1x_DATA
REF1x_SIGN
REF2y_F1
REF2y_F2
REF2y_F3
REF2y_F4
REF2y_F5+

REF2y_DATA
REF2y_SIGN
YDC_F1
YDC_F2+
YDC_DATA
YDC_SIGN
UDC_F1
UDC_F2+
UDC_DATA
UDC_SIGN
VDC_F1
VDC_F2+
VDC_DATA
VDC_SIGN

Part II

Stream parsing

5 Dirac stream specification

This section specifies the Dirac stream and stream parsing operations, excepting the decoding of wavelet coefficients and motion data, which are deferred to Sections 7 and 6. The decoding operations for extracting decoded pictures from parsed data are specified in Section 8.

The stream parsing specification is augmented by the parse diagrams in Appending A, which summarise in graphical form the struture of the stream.

5.1 Introduction

A stream is a concatenation of Dirac sequences. A sequence is a concatenation of Access Units, comprised of Access Unit headers and a number of picture data units, together with data headers ("Parse Info") allowing for efficient navigation of the sequence.

The essential difference between a stream and a sequence is that a sequence corresponds to a single video sequence, meaning a stream of images of constant video parameters (picture dimensions, aspect ratio, frame rate and so on as defined in Sections 5.7 and 5.8). Any change in video parameters necessitates that a sequence be terminated and a new sequence started.

Default decoding parameters are computed based on the Access Unit header data. AU header data (excluding the AU picture number) is required to be constant throughout a sequence (Section 5.5): however the decoding parameters used for decoding pictures are *not* necessarily constant since they may be overridden within the picture data.

As a result, the parsing and decoding model used in this specification maintains two sets of state variables: the sequence or default state variable **default_state**, holding the defaults to be used throughout a sequence, and the state variable **state** holding the values to be used for decoding the current picture, which may override the defaults for many variables. The picture state variable is re-intialised from the default settings before each picture is decoded.

Informative: The requirement that default decoding parameters are overridden for each picture – rather than, for example, changing defaults for all subsequent pictures – potentially causes a little more overhead. However it greatly enhances random access: once the Access Unit header has been read, *any* picture within the Access Unit can be successfully parsed independently, and decoding may even be possible from a variety of points within the Access Unit. Since the AU data is constant throughout the sequence, reading the AU header once allows any picture in the sequence to be parsed.

5.2 Stream

A stream is a concatenation of Dirac sequences. The process for parsing a stream is to parse all sequences it contains.

5.2.1 Sequence

The data contained in a Dirac Sequence corresponds to a single video sequence with constant video parameters as defined in Sections 5.7 and 5.8. A sequence is preceded by a Parse Info header which indicates the beginning of the sequence with a parse code. A Dirac sequence can be excised from a Dirac stream and decoded entirely independently.

video_sequence() :	
parse_info()	5.3
while (*is_access_unit*()):	5.3
access_unit()	5.4

5.3 Parse Info header

This section specifies the operation of the *parse_info*() process for parsing Parse Info header data. This header is byte-aligned. It occurs at the beginning of a sequence, at the end of a sequence, before an Access Unit header, and before each set of picture data. It is used to navigate through the stream (Section 8). The values of Parse Info parameters determine the type and format of the subsequent data structures, in particular indicating whether a picture is Intra or Inter coded, and if Inter how many references it has.

parse_info() :	
byte_align()	
state[parse_info_prefix] = *read_uint_lit*(4)	
state[parse_code] = *read_byte*()	
state[next_parse_offset] = *read_uint_lit*(3)	
state[previous_parse_offset] = *read_uint_lit*(3)	

The Parse Info parameters shall satisfy the following constraints:

- **state**[parse_info_prefix] shall be set to be 0x42 0x42 0x43 0x44, which is ASCII for BBCD.
- **state**[parse_code] shall be one of the supported values set out in Table 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 0
- **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 0

state[parse_code]	**Bits**	**Description**	**Number of Reference Pictures**
0x00	0000 0000	Access Unit header	–
0x0C	0000 1100	Intra Reference Picture	0
0x08	0000 1000	Intra Non Reference Picture	0
0x0D	0000 1101	Inter Reference Picture	1
0x0E	0000 1110	Inter Reference Picture	2
0x09	0000 1001	Inter Non Reference Picture	1
0x10	0000 1010	Inter Non Reference Picture	2
0x10	0001 0000	End of Sequence	–

Table 1: Parse codes

A number of functions are defined based on the parse code value, considered as a bit-field, which shall be used to direct subsequent decoding operations. All are boolean, except for *num_refs*() which returns an integer:

is_AU() :	
return !(**state**[parse_code]&0x18)	

is_picture() :	
return ((**state**[parse_code]&0x18) == 0x08)	

is_end_of_sequence() :	
return ((**state**[parse_code]&0x18) == 0x10)	

is_reference() :	
return ((**state**[parse_code]&0x04) == 0x04)	

is_non_reference() :	
return ((**state**[parse_code]&0x04) == 0x00)	

num_refs() :	
return (**state**[parse_code]&0x03)	

is_intra() :	
return (*num_refs*() == 0)	

is_inter() :	
return (*num_refs*() > 0)	

Informative: Next Parse Offset and Previous Parse Offset are added to the byte stream to simplify parsing. Next Parse Offset represents the offset in bytes from the start of the current Parse Info to the start of the next Parse Info. So counting forward Next Parse Offset bytes from the first byte (0x42=B) of the current Parse Info should yield a byte of value 0x42=B corresponding to the start of the next Parse Info. The Previous Parse Offset is the number of bytes backwards to the start of the previous Parse Info header. The Previous Parse Offset of the current Parse Info therefore equals the Next Parse Offset of the previous Parse Info.

The 4 byte Parse Info Prefix is present to allow an application to find a point from which to start decoding. That is, the function of Parse Prefix Header is to synchronise the decoder with the byte stream. Parsing of the stream can start from any Access Unit Header, and successful decoding once the reference buffer has converged (see Section 8).

The decoder first needs to find a Parse Info structure. It should then check the Parse Code in the Parse Info. If the following parse unit is an Access Unit Header then the decoder can start decoding. If the it is a Picture then the decoder should skip forward by Next Parse Offset bytes (from the start of the Parse Info Prefix) to the next Parse Info. The decoder would continue skipping forward until it locates an Access Unit Header. Note that the decoder does not need to parse any intervening data in order to navigate through the stream to find an Access Unit Header. The Previous Parse Offset is provided to allow searching backwards through the byte stream.

Any particular instance of the Parse Info Prefix in the byte stream may not, necessarily, indicate the start of a Parse Info structure. This is because other parts of the byte stream may, by chance, introduce these bytes into the byte stream. In particular, the use of arithmetic coding in Dirac means that it is impossible to directly avoid accidentally introducing the Parse Info Prefix. When encoding a bytestream it is not necessary to avoid accidentally introducing Parse Info Prefix sequences. They are present to allow synchronisation of the bytes stream with the decoder and this can be ensured, even in the presence of spurious Parse Info Prefixes, as follows. When the decoder finds a Parse Info Prefix it should skip forward by Next Parse Offset (or back by Previous Parse Offset) and check whether the next three bytes are a Parse Info Prefix. If so the decoder can be reasonably certain that it has found a genuine Parse Info Prefix. If it does not find another Parse Info Prefix it was probably unlucky enough to have found a spurious Parse Info Prefix. In this case it should search for the next Prefix and repeat the test.

The probability of a spurious Parse Info Prefix is low: 1 in 2^{32} since the prefix is 4 bytes long. This is the probability of finding two Parse Info Prefix sequences separated by Next Parse Offset. The test

outlined in the previous paragraph is, therefore, more than adequate in practice. For the paranoid the test may of course be extended.

The test for two appropriately separated Parse Info Prefixes is, anyway, prudent in any channel subject to bit errors even in the absence of spurious Prefixes.

This defintion of Dirac only includes three types of Parse code: the AU header, those for different picture sorts, and that indicating the end of the sequence. It is envisaged that other codes may be introduced in future to indicate data such as user data or extension data.

5.4 Access Units

This section specifies the operation of the *access_unit*() process for parsing an Access Unit. Access Units provide points at which the stream may be randomly accessed. Specifically, a stream may be successfully parsed from any Access Unit Header (Section 5.5) without reference to prior data, and successfully decoded once the reference picture buffer has converged (Section 8).

The Access Unit parsing process is given by:

access_unit() :	
access_unit_header()	5.5
parse_info()	5.3
while (*is_picture*() == **True**):	5.3
picture()	5.9
parse_info()	5.3

Each Access Unit begins with the Access Unit header. Picture data may be read from that point until the next Access Unit or until the end of the sequence/stream. Data is read into the default parameter set by parsing the Access Unit header (Section 5.5).

5.5 Access Unit header

This section specifies the structure of the Access Unit header. This Access Unit header is byte aligned. Parsing this header consists in reading the Access Unit parameters (parse, source and sequence parameters) and initialising the default decoder parameters **default_state** as a result of these. Access Unit parameters remain constant throughout a sequence, so in theory the AU header may be skipped after being read once.

access_unit_header() :	
byte_align()	
parse_parameters()	5.6
sequence_parameters()	5.7
source_parameters()	5.8

Informative: Note that source 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 source parameters are not used by the Dirac decoder. Source and sequence 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, Appendix B specifies the video systems model that should be used for the interpretation of source and sequence parameters.

5.6 Access unit parse parameters

This section specifes the structure of the Access Unit Parse Parameters, which is as follows:

parse_parameters() :	
default_state[au_picture_number] = *read_uint_lit*(4)	
default_state[version_major] = *read_uint*()	
default_state[version_minor] = *read_uint*()	
default_state[profile] = *read_uint*()	
default_state[level] = *read_uint*()	

Access Unit Parse parameter data, with the exception of **default_state**[au_picture_number] shall remain constant (byte-for-byte identical) for all instances of the Access Unit header within a Dirac sequence.

5.6.1 AU picture number

default_state[au_picture_number] shall be equal to the picture number of the immediately succeeding picture, if there is one.

5.6.2 Version number

The version number of the Dirac syntax specification (this document) shall be used by the decoder to determine whether it can decode the sequence. It falls into two integer parts, the major and minor version, written as $M.m$, where $M =$ **default_state**[version_major] and $m =$ **default_state**[version_minor].

The major version defines 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. Such a compliant decoder must be able to parse all previous versions too. Decoders that comply with a major version of the specification may not be able to parse the bit stream corresponding to a later specification.

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

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

5.6.3 Profiles and levels

A profile determines a toolset that is sufficient to decode a sequence. A level determines decoder resources (picture and data buffers; computational resources) sufficient to decode a sequence. Applicable values of profile and level are specified in Appendix D.

5.7 Access unit sequence parameters

The AU sequence parameters consist of the video format, the image dimensions, the chroma format and the video depth. AU sequence parameter data shall remain constant throughout a Dirac sequence.

sequence_parameters() :	
default_state[video_format] = *read_uint*()	
set_format_defaults()	5.7.1
image_dimensions()	5.7.2
chroma_format()	5.7.3
video_depth()	5.7.4

5.7.1 Setting format defaults

Default parameter values are set based on the value of **default_state**[video_format], as specified in Appendix C. These cover sequence and source parameters, but also decoding parameters such as wavelet transform depth and motion compensation block sizes. For example, if **default_state**[video_format] == 4, CIF defaults are set, with picture size equal to 252 × 288, 4:2:0 chroma format, and 12 × 12 luma blocks.

Sequence and source parameters may be overridden by subsequent data in the AU header. For example, an image width of 360 may be encoded as per Section imagedimensions, overriding the CIF format defaults. Decoding parameters may be overridden by data in individual pictures.

5.7.2 Custom image dimensions

If a flag is set, the image dimensions specified by the video format defaults may be overridden:

image_size() :	
custom_dimensions_flag = *read_bool*()	
if (*custom_dimensions_flag* == **True**):	
default_state[luma_width] = *read_uint*()	
default_state[luma_height] = *read_uint*()	

5.7.3 Chroma formats

If a flag is set, the chroma format specified by the video format defaults is overridden.

chroma_format() :	
chroma_format_flag = *read_bool*()	
if (*chroma_format_flag* == **True**):	
default_state[chroma_format_index] = *read_uint*()	
chroma_dimensions()	

The supported chroma formats are specified in Table 2:

default_state[chroma_format_index]	**Chroma format**
0	4:4:4
1	4:2:2
1	4:2:0

Table 2: Supported chroma formats

Chroma dimensions are set according to the scaling implied by the chroma format:

chroma_dimensions() :	
if (**default_state**[chroma_format_index] == 0):	
default_state[chroma_width] = **default_state**[luma_width]	
default_state[chroma_height] = **default_state**[luma_height]	
else if (**default_state**[chroma_format_index] == 1):	
default_state[chroma_width] = **default_state**[luma_width]//2	
default_state[chroma_height] = **default_state**[luma_height]	
else:	
default_state[chroma_width] = **default_state**[luma_width]//2	
default_state[chroma_height] = **default_state**[luma_height]//2	

Utility functions returning the chroma subsampling factors are also defined:

chroma_h_ratio() :	
if (**default_state**[chroma_format_index] == 0):	
return 1	
else if (**default_state**[chroma_format_index] == 1):	
return 2	
else:	
return 2	

chroma_h_ratio() :	
if (**default_state**[chroma_format_index] == 0):	
return 1	
else if (**default_state**[chroma_format_index] == 1):	
return 1	
else:	
return 2	

5.7.4 Video depth

If a flag is set, the default video depth specified by the video format is overridden:

video_depth() :	
video_depth_flag = *read_bool*()	
if (*video_depth_flag* == **True**):	
default_state[video_depth] = *read_uint*()	

5.8 Access unit source parameters

The Access Unit source parameters consist of: the scan format, frame rate, aspect ratio, clean area, signal range and colour specification. These parameters have been grouped together as they directly influence how a downstream display device will display decoded pictures produced by a Dirac decoder. Access Unit source parameter data shall remain constant throughout a Dirac sequence. Default values are derived from the video format, as specified in Appendix C.

Display and downstream processing falls outside the scope of this specification, and hence the interpretation of these parameters is not normatively defined, with the exception of frame rate (Section 5.8.2) which imposes requirements on compliant decoders for a given level and profile (Appendix D). Appendix B describes how the source should be interpreted. Deviation from the description there will reduce video quality, perhaps significantly, and in embedding a Dirac decoder in a display or processing device manufacturers should take the greatest care in adhering to that description.

source_parameters() :	
scan_format()	5.8.1
frame_rate()	5.8.2
aspect_ratio()	5.8.3
clean_area()	5.8.4
signal_range()	5.8.5
colour_spec()	5.8.6

5.8.1 Scan format

Scan Format parameters are concerned with interlace. If **default_state**[interlaced] = **True**, then the video should be displayed as interlaced video. The process for parsing the Scan Format parameters is as follows:

scan_format() :	
scan_format_flag = *read_bool*()	
if (*scan_format_flag* == **True**):	
default_state[interlaced] = *read_bool*()	
if (**default_state**[interlaced]):	
field_dominance_flag = *read_bool*()	
if (*field_dominance_flag* == **True**):	
default_state[top_field_first] = *read_bool*()	
field_interleaving_flag = *read_bool*()	
if (*field_interleaving_flag* == **True**):	
default_state[sequential_fields] = *read_bool*()	

Informative: If we have an interlaced source the field lines can either be interleaved line by line (*pseudo-progressive* format) or interleaved field by field (*sequential field format*, required for low delay and low resource coding). Pseudo-progressive format is set as default for all the interlaced video formats. The field interleaving flag indicates non-default field interleaving, and the sequential fields (Boolean) parameter indicates whether the fields are interleaved as pseudo-progressive or sequential fields.

Note that if the video stream is in sequential field format, then picture dimensions refer to fields rather than frames. Even picture numbers will in this case refer to even fields, and odd picture numbers to odd fields. In pseudo-progressive mode, picture dimensions are frame dimensions.

scan_format() :	
scan_format_flag = *read_bool*()	
if (*scan_format_flag* == **True**):	
default_state[interlaced] = *read_bool*()	
if (**default_state**[interlaced]):	
field_dominance_flag = *read_bool*()	
if (*field_dominance_flag* == **True**):	
default_state[top_field_first] = *read_bool*()	
field_interleaving_flag = *read_bool*()	
if (*field_interleaving_flag* == **True**):	
default_state[sequential_fields] = *read_bool*()	

5.8.2 Frame rate

The process for parsing Frame Rate parameters is as follows:

$frame_rate()$:	
$frame_rate_flag = read_bool()$	
if ($frame_rate_flag ==$ **True**):	
$index = read_uint()$	
if ($index == 0$):	
default_state[frame_rate_numer] = $read_uint()$	
default_state[frame_rate_denom] = $read_uint()$	
else:	
$preset_frame_rate()$	uint

The decoded value of $index$ shall fall in the range 0 to 8.

$preset_frame_rate(index)$ sets frame rate values as specified in Table 15 (Appendix B.8).

The true frame rate is $\frac{\textbf{default_state}[\text{frame_rate_numer}]}{\textbf{default_state}[\text{frame_rate_denom}]}$.

Note that what is encoded is frame rate, not picture rate. If **default_state**[sequential_fields] = **True** then picture rate is twice the encoded frame rate. Supported frame rates in a given profile and level are specified in Appendix D.

5.8.3 Aspect ratio

The process for extracting aspect ratio parameters is as follows:

$aspect_ratio()$:	
$aspect_ratio_flag = read_bool()$	
if ($apect_ratio_flag ==$ **True**):	
$index = read_uint()$	
if ($index == 0$):	
default_state[aspect_ratio_numer] = $read_uint()$	
default_state[aspect_ratio_denom] = $read_uint()$	
else:	
$preset_aspect_ratio(index)$	

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

$preset_aspect_ratio(index)$ sets aspect ratio values as specified in Table 16 (Appendix B.8).

The true aspect ratio is $\frac{\textbf{default_state}[\text{aspect_ratio_numer}]}{\textbf{default_state}[\text{aspect_ratio_denom}]}$.

It is a pixel aspect ratio.

5.8.4 Clean area

The process for extracting the clean area parameters is as follows:

$clean_area()$:	
$clean_area_flag = read_bool()$	
if ($clean_area_flag ==$ **True**):	
default_state[clean_width] = $read_uint()$	
default_state[clean_height] = $read_uint()$	
default_state[left_offset] = $read_uint()$	
default_state[top_offset] = $read_uint()$	

The clean area determines the part of the picture that should be displayed. The following restrictions shall apply:

- **state**[clean_width] + **state**[left_offset] ≤ **default_state**[luma_width]
- **state**[clean_height] + **state**[top_offset] ≤ **default_state**[luma_height]

5.8.5 Signal range

Picture component data output by the Dirac decoder is in the range 0 to $2^{\textbf{default_state}[\text{video_depth}]} - 1$. The signal range parameters determine how these data ranges should be adjusted prior to matrixing operations (Appendix **??**).

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

signal_range() :	
signal_range_flag = *read_bool*()	
if (*signal_range_flag* == **True**):	
index = *read_uint*()	
if (*index* == 0):	
default_state[luma_offset] = *read_uint*()	
default_state[luma_excursion] = *read_uint*()	
default_state[chroma_offset] = *read_uint*()	
default_state[chroma_excursion] = *read_uint*()	
else:	
preset_signal_ranges(*index*)	

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

preset_signal_ranges(*index*) sets signal range values as specified in Tables 17 and 18 (Appendix B.8).

The offset and excursion values shall satisfy the following constraints:

- 0 ≤ **default_state**[luma_offset] < $2^{\textbf{default_state}[\text{video_depth}]}$
- 0 ≤ **default_state**[luma_excursion] < $2^{\textbf{default_state}[\text{video_depth}]}$
- 0 ≤ **default_state**[chroma_offset] < $2^{\textbf{default_state}[\text{video_depth}]}$
- 0 ≤ **default_state**[chroma_excursion] < $2^{\textbf{default_state}[\text{video_depth}]}$

5.8.6 Colour specification

This section specifies the *colour_spec*() parsing process. The colour specification consists of primaries, matrix and transfer function. Defaults are available for all three collectively and individually. The process is:

colour_spec() :	
$colour_spec_flag$ = *read_bool*()	
if (*colour_spec_flag* == **True**):	
index = *read_uint*()	
preset_colour_specs(*index*)	
if (*colour_spec_index* == 0):	
colour_primaries()	5.8.6.1
colour_matrix()	5.8.6.2
transfer_function()	5.8.6.3

index shall fall in the range 0 to 3.

preset_colour_spec(*index*) sets the colour primaries, matrix and transfer function as specified in Table 19 (Appendix B.8).

5.8.6.1 Colour primaries

The *colour_primaries*() process is as follows:

colour_primaries() :	
colour_primaries_flag = *read_bool*()	
if (*colour_primaries_flag* == **True**):	
index = *read_uint*()	
preset_colour_primaries(*index*)	

index shall fall in the range 0 to 3. *preset_colour_primaries*(*index*) sets the colour primaries as specified in Table 20 (Appendix B.8).

5.8.6.2 Colour matrix

The *colour_matrix*() process is as follows:

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

index shall fall in the range 0 to 2. *preset_colour_matrices*(*index*) sets the colour matrix as specified in Table 21 (Appendix B.8).

5.8.6.3 Transfer function

The *transfer_function*() process is as follows:

transfer_function() :	
transfer_function_flag = *read_bool*()	
if (*transfer_function_flag* == **True**):	
index = *read_uint*()	
preset_transfer_functions(*index*)	

index shall fall in the range 0 to 3. *preset_transfer_functions*(*index*) sets the transfer function as specified in Table 22 (Appendix B.8).

5.9 Picture

This Section specifies the operation of the *picture*() parsing process. The process for decoding and outputting pictures is specified in Section 8.

Picture data may be successfully parsed after parsing any Access Unit header within the same Dirac sequence. The parsing process is:

picture() :	
init_decode_params()	5.9.1
picture_header()	5.9.2
if (*is_inter*()):	5.3
picture_prediction()	5.10
wavelet_transform()	5.11

5.9.1 Initialising decoding parameters

The default parameters are used to initialise the decoder state prior to decoding each picture:

$init_decode_params()$:	
for each var in args(**default_state**):	
state[var] = **default_state**[var]	

State variables and default state variables are listed in the index.

5.9.2 Picture header

The picture header is byte aligned and follows a Parse Info header with a picture parse code. The process for parsing the picture header is as follows:

$picture_header()$:	
$bytealign()$	
state[picture_number] = $read_byte_lit(4)$	
if ($is_inter()$):	5.3
$reference_picture_numbers()$	
$retired_picture_list()$	

Picture numbers are not required to be unique within a sequence.

Reference picture numbers are encoded differentially with respect to the picture number:

$reference_picture_numbers()$:	
state[ref1_picture_number] = (**state**[picture_number] + $read_sint()$)%2^{32}	
if ($num_refs() == 2$):	5.3
state[ref2_picture_number] = (**state**[picture_number] + $read_sint()$)%2^{32}	

The retired picture list is a list of pictures to be removed from the reference picture buffer before the current picture is decoded. The rules for the use of the retired picture list are specified in Section 8.4. The list of retired picture numbers is also encoded differentially with respect to the picture number:

$retired_picture_list()$:	
$num_retired_pictures = read_uint()$	
for $i = 0$ to $num_retired_pictures - 1$:	
state[retired_picture_list][i] = (**state**[picture_number] + $read_sint()$)%2^{32}	

5.10 Picture prediction

This section specifies the $picture_prediction()$ process for parsing picture prediction data. The process consists of two parts: extracting picture prediction parameters and decoding and extracting motion vector fields for motion compensation, as follows:

$picture_prediction()$:	
$picture_prediction_parameters()$	5.10.1
$block_data()$	6

The decoding and generation of block motion vector fields is specified in Section 6. The remainder of this section is concerned with the overall structure of the picture prediction data and the process for parsing and setting picture prediction parameters, including global motion parameters.

The two elements of the picture prediction process correspond to two elements of the Dirac stream, the picture prediction parameters and the block motion data. Global motion parameters are used to construct a global motion vector field, and block motion data is used to provide motion vectors block-by-block. Both elements may not be present, depending upon flags signalled within the stream. Both elements are byte-aligned within the stream.

5.10.1 Picture prediction parameters

Picture prediction parameters consist of metadata required for successful parsing of the motion data and for performing motion compensation. It includes data indicating whether and how global motion is used, the size of blocks, and the weightings used for reference pictures in motion compensation (Section 10).

picture_prediction_parameters() :	
block_parameters()	5.10.2
motion_vector_precision()	5.10.4
global_motion()	5.10.5
picture_prediction_mode()	5.10.6
reference_picture_weights()	5.10.7

5.10.2 Block parameters

This Section specifies the operation of the *block_parameters*() process for setting the block parameters, consisting 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.

Before this process is invoked, default block parameters are set by the video format encoded in the Access Unit header (Section 5.5). These may be temporarily overridden for the current picture if a flag is set, either by a preset or by explicit signalling:

block_parameters() :	
block_params_flag = *read_bool*()	
if (*block_params_flag*):	
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()	5.10.3

index shall fall in the range 0 to 4. *preset_block_params*(*index*) sets the transfer function as specified in Table 3 (note that chroma block parameter values are computed from luma values in Section 5.10.3).

	Block parameters			
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 3: 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]$-$**state**[luma_xbsep] and **state**[luma_yblen]$-$**state**[luma_ybsep] shall be powers of 2 other than 1

4. **state**[chroma_xblen] $-$ **state**[chroma_xbsep] and **state**[chroma_yblen] $-$ **state**[chroma_ybsep] shall be powers of 2 other than 1

Informative: Note that these requirements do not preclude length from equalling separation, i.e. motion compensation blocks are not overlapped. This may improve quality at higher bitrates.

5.10.3 Setting chroma block parameters

This section specifies the operation of the *chroma_block_params*() process, which determines chroma block dimensions from luma block dimensions. Chroma block parameters are 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()() :	
state[chroma_xblen] = **state**[luma_xblen]//*chroma_h_ratio*()	
state[chroma_yblen] = **state**[luma_yblen]//*chroma_v_ratio*()	
state[chroma_xbsep] = **state**[luma_xbsep]//*chroma_h_ratio*()	
state[chroma_ybsep] = **state**[luma_ybsep]//*chroma_v_ratio*()	

[Note that all the stuff about recomputing the chroma block lengths is now redundant since we allow non-overlapping blocks as well]

5.10.4 Motion vector precision

This section specifies the *motion_vector_precision*() process for setting the precision (number of sub-pixel accuracy bits) used for motion compensation.

motion_vector_precision() :	
motion_vector_precision_flag = *read_bool*()	
if (*motion_vector_precision_flag* == **True**):	
state[mv_precision] = *read_uint*()	

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

5.10.5 Global motion

[Say something about global motion data structures?]

Global motion parameters are encoded if a flag is set. Up to two sets are encoded, depending upon the number of references:

global_motion() :	
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])	

Global motion parameters **state**[global_params] consist of three elements:

- an integer pan/tilt vector
$$\mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$$
- an integer matrix element
$$\mathbf{A} = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix}$$
capturing zoom, rotation and shear, together with a scaling exponent
- an integer perspective element
$$\mathbf{c} = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$
capturing the effect of non-orthogonal projection onto the image plane, together with a scaling exponent

Their interpretion and the process for generating a global motion vector field is specified in Section 10.7. These elements are parsed in turn:

$global_motion_parameters(gparams)$:	
$pan_tilt(gparams)$	
$zoom_rotate_shear(gparams)$	
$perspective(gparams)$	

The $pan_tilt()$ process extracts horizontal and vertical translation elements:

$pan_tilt(gparams)$:	
$gparams.\mathbf{b} = \mathbf{0}$	
$nonzero_pan_tilt_flag = read_bool()$	
if ($nonzero_pan_tilt_flag ==$ **True**):	
$gparams.\mathbf{b}_0 = read_sint()$	
$gparams.\mathbf{b}_1 = read_sint()$	

The $zoom_rotate_shear()$ process extracts a linear matrix element:

$zoom_rotation_shear(gparams)$:	
$nontrivial_zrs_flag = read_bool()$	
if ($nontrivial_zrs_flag ==$ **True**):	
$gparams[ZRS_exp] = read_uint()$	
$gparams.\mathbf{A}_{0,0} = read_sint()$	
$gparams.\mathbf{A}_{0,1} = read_sint()$	
$gparams.\mathbf{A}_{1,0} = read_sint()$	
$gparams.\mathbf{A}_{1,1} = read_sint()$	
else:	
$gparams[ZRS_exp] = 0$	
$gparams.\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	

The $perspective()$ process extracts horizontal and vertical perspective elements:

$perspective(gparams)$:	
$nonzero_perspective_flag = read_bool()$	
if ($nonzero_perspective_flag ==$ **True**):	
$gparams[perspective_exp] = read_uint()$	
$gparams.\mathbf{c}_0 = read_sint()$	
$gparams.\mathbf{c}_1 = read_sint()$	
else:	
$gparams[perspective_exp] = 0$	
$gparams.\mathbf{c} = \mathbf{0}$	

5.10.6 Picture prediction mode

The picture prediction mode encodes alternative methods of motion compensation.

$picture_prediction_mode()$:	
$pic_pred_mode_flag = read_bool()$	
if ($pic_pred_mode_flag ==$ **True**):	
state[picture_prediction_mode] = $read_uint()$	
else:	
state[picture_prediction_mode] = 0	

state[picture_prediction_mode] shall be 0.

5.10.7 Reference picture weight values

Alternative reference picture weight values may be defined to override the video format defaults:

$reference_picture_weights()$:	
$non_default_weights_flag = read_bool()$	
if ($non_default_weights_flag ==$ **True**):	
state[refs_weight_precision] = $read_uint()$	
state[ref1_weight] = $read_sint()$	
if ($num_refs == 2$):	
state[ref2_weight] = $read_sint()$	

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

$$\frac{\mathbf{state}[\text{ref1_weight}]}{2^{\mathbf{state}[\text{refs_weight_precision}]}}$$

and reference 2 data by

$$\frac{\mathbf{state}[\text{ref2_weight}]}{2^{\mathbf{state}[\text{refs_weight_precision}]}}$$

(Section 10.4).

5.11 Wavelet transform

The $wavelet_tranform()$ function parses metadata determining the wavelet transform (including filters, wavelet depth, and code block structures) together with the transformed wavelet coefficients.

Decoded wavelet transform coefficient data is stored in the state variables **state**[y_transform], **state**[u_transform] and **state**[v_transform] for subsequent processing using the inverse wavelet transform (Section 9).

wavelet_transform() :	
state[zero_residual] = **False**	
if (*is_inter*()):	5.3
state[zero_residual] = *read_bool*()	
if (**state**[zero_residual] = **False**):	
transform_parameters()	5.11.1
state[component_width] = **state**[luma_width]	
state[component_height] = **state**[luma_height]	
state[y_transform] = *transform_data*()	7.2.1
state[component_width] = **state**[chroma_width]	
state[component_height] = **state**[chroma_height]	
state[u_transform] = *transform_data*()	7.2.1
state[v_transform] = *transform_data*()	7.2.1

If **state**[zero_residual] = **True** then all component pixels will be set to zero prior to motion compensation (Section 8).

Wavelet coefficients have been encoded using entropy coding, and their decoding is specified in Section 7. The remainder of this section specifies the decoding of transform parameters.

5.11.1 Wavelet transform parameters

The wavelet transform parameters encode the filter to be used, the depth of filtering and how subbands are spatially partitioned:

transform_parameters(*state*) :	
wavelet_filter()	5.11.1.1
wavelet_depth()	5.11.2
spatial_partition()	5.11.3

5.11.1.1 Wavelet filters

A variety of preset wavelet filters are available, encoded as a values of **state**[wavelet_index], which is an index into Table 4. Default wavelet filters are the Deslauriers-Debuc (9,3) filter for intra pictures and the LeGall (5,3) filter for inter pictures. If a flag is set, other presets may be used. Their interpretation and lifting implementations are specified in Section 9.4.

wavelet_filter() :	
non_default_wavelet_flag = *read_bool*()	
if (*non_default_wavelet_flag* == **True**):	
state[wavelet_index] = *read_uint*()	
else:	
if (*is_intra*() == **True**):	
state[wavelet_index] = **default_state**[wavelet_index][INTRA]	
else:	
state[wavelet_index] = **default_state**[wavelet_index][INTER]	

state[wavelet_index] shall lie in the range 0 to 7.

Informative: 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 9.4. Deslauriers-Debuc filters are normally referred to in terms of the number of vanishing

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

Table 4: Wavelet filter presets

moments of their synthesis filters, so the (9,3) and (13,5) filters may be referred to in the literature as (2,2) and (4,2) filters respectively.

5.11.2 Wavelet depth

The wavelet depth determines the number of times the vertical and horizontal wavelet filters may be applied. The *wavelet_depth*() parsing process is as follows:

wavelet_depth() :	
$non_default_wavelet_depth_flag$ = *read_bool*()	
if ($non_default_wavelet_depth_flag$ == **True**):	
state[wavelet_depth] = *read_uint*()	

Allowable **state**[wavelet_depth] values are determined by the level and profile (Appendix D). The wavelet depth determines the number of subbands and the the dimensions of the subband data array (Section 7.1.1).

5.11.3 Spatial partition of wavelet data

Each subband may be partitioned into a number of code blocks. The number of codeblocks to be used for subbands at level *level* is encoded in **state**[codeblocks][*level*][*v*] and **state**[codeblocks][*level*][*h*] respectively.

spatial_partition() :	
$spatial_partition_flag$ = *read_bool*()	
if ($spatial_partition_flag$ == **True**):	
$nondefault_partition_flag$ = *read_bool*()	
if ($nondefault_partition_flag$ == **True**):	
depth = **state**[wavelet_depth]	
for *level* = 0 to *depth*:	
state[codeblocks][*level*][*h*] = *read_uint*()	
state[codeblocks][*level*][*v*] = *read_uint*()	
index = *read_uint*()	
state[codeblock_mode] = *codeblock_mode*(*index*)	
else:	
if (*is_intra*() == **True**):	
state[codeblocks] = **default_state**[codeblocks][INTRA]	
else:	
state[codeblocks] = **default_state**[codeblocks][INTER]	

index shall lie in the range 0 to 2, and *codeblock_mode* sets the codeblock mode according to Table 5.

The maximum number of codeblocks vertically shall be less than or equal $subband_height(level)//4$ and the number of codeblocks horizontally shall be less than or equal to $subband_width(level)//4$ (subband dimensions as specified in Section subbandwidthheight).

index	**state**[codeblock_mode]
0	SINGLE_QUANT
1	MULTI_QUANT
2	Undefined

Table 5: Codeblock modes

6 Motion data decoding

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

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

Informative: 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 **??**).

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.

6.1 Motion data conventions

For the purposes of this specification, block motion data is stored in a two dimensional array **state**[block_data] of block data structures. A block motion data element $\mathbf{b} = \mathbf{state}[\text{block_data}][j][i]$ consists of:

- A motion vector for reference 1, $\mathbf{b}[ref1]$, with integral horizontal and vertical components $\mathbf{b}[ref1].x$ and $\mathbf{b}[ref1].y$
- A motion vector for reference 2, $\mathbf{b}[ref2]$, with integral horizontal and vertical components $\mathbf{b}[ref2].x$ and $\mathbf{b}[ref2].y$
- A set of integral DC values, $\mathbf{b}[dc][Y]$, $\mathbf{b}[dc][U]$, and $\mathbf{b}[dc][V]$ for each component
- A prediction mode, $\mathbf{b}[mode]$, consisting of two flags $\mathbf{b}[mode][1]$ and $\mathbf{b}[mode][2]$ indicating whether the corresponding reference is to be used for predicting block (i, j)

Four prediction modes shall be supported by the decoder:

- INTRA- corresponding to $\mathbf{b}[mode][1] = \mathbf{False}$ and $\mathbf{b}[mode][2] = \mathbf{False}$, and using DC prediction only
- REF1ONLY- corresponding to $\mathbf{b}[mode][1] = \mathbf{True}$ and $\mathbf{b}[mode][2] = \mathbf{False}$, and using a prediction from Reference 1 only
- REF2ONLY- corresponding to $\mathbf{b}[mode][1] = \mathbf{False}$ and $\mathbf{b}[mode][2] = \mathbf{True}$, and using a prediction from Reference 2 only
- REF1AND2- corresponding to $\mathbf{b}[mode][1] = \mathbf{True}$ and $\mathbf{b}[mode][2] = \mathbf{True}$, and using a prediction from Reference 1 and Reference 2

In this way, Reference X is used for prediction if and only if $\mathbf{b}[mode][X] = \mathbf{True}$.

6.2 Motion data decoding process

This section specifies the overall operation of the *block_data*() process for extracting block motion data elements: motion vectors and block prediction modes. This process is called by the *picture_prediction*() process (Section 5.10) and depends upon the parameters that have been extracted and set in the *picture_prediction_parameters*() process (Section 5.10.1).

Block motion data elements are all coded differentially with respect to a spatial prediction. The spatial prediction processes for the block motion elements are specified in Section 6.2.5

6.2.1 Overall decoding loop

The decoding loop for block data iterates over all superblocks in raster order:

$block_data()$:	
$initialise_motion_data()$	6.2.2
$length = read_uint()$	
$byte_align()$	
$initialise_arithmetic_decoding(length)$	4.2
$superblock_count = 0$	
for $v = 0$ to **state**[superblocks_y]:	
for $h = 0$ to **state**[superblocks_x]:	
$superblock(4 * v, 4 * h)$	
$superblock_count += 1$	
if ($superblock_count == 32$):	
$superblock_count = 0$	
for $i = 0$ to len(**state**[contexts]) $- 1$:	
$rescale_context(i)$	4.1
$byte_align()$	

Informative: The specification for parsing and decoding block data is written indicating that a sequence of superblocks are read following the block data length. This is conceptually correct. However, in practice it may be more efficient to read the whole of the block data (excluding the block data length) into a buffer before parsing and decoding. To facilitate this the arithmetic coded superblock data is byte aligned, i.e. starts at the beginning of a byte boundary and occupies a whole number of bytes.

One reason for the improved efficiency afforded by first reading into a buffer is because the arithmetic coding engine may require (up to) two extra bytes of data for its output to converge. In this specification this feature is provided by testing that the amount of data requested has not exceeded $length$ bytes and, if it has, inputting a zero byte. By reading the data directly into a buffer of length $length + 2$ this test can be avoided.

6.2.2 Motion data initialisation

This section specifies the operation of the $initialise_motion_data()$ process. It sets the dimension variables determining the number of blocks and superblocks and hence the dimension of the **state**[block_data] array encapsulating block motion data.

The number of superblocks horizontally and vertically is set by:

$$\mathbf{state}[\text{superblocks_x}] = \left\lceil \frac{\mathbf{state}[\text{luma_width}]}{4 * \mathbf{state}[\text{luma_xbsep}]} \right\rceil$$

$$\mathbf{state}[\text{superblocks_y}] = \left\lceil \frac{\mathbf{state}[\text{luma_height}]}{4 * \mathbf{state}[\text{luma_ybsep}]} \right\rceil$$

The number of blocks horizontally and vertically is set by:

$$\mathbf{state}[\text{blocks_x}] = 4 * \mathbf{state}[\text{superblocks_x}]$$

$$\mathbf{state}[\text{blocks_y}] = 4 * \mathbf{state}[\text{superblocks_y}]$$

The array **state**[block_data] is set to have horizontal dimension **state**[blocks_x] and vertical dimension **state**[blocks_y].

The array **state**[sb_split] is set to have horizontal dimension **state**[superblocks_x] and vertical dimension **state**[superblocks_y].

6.2.3 Superblock decoding

This section specifies the process *superblock*(y, x) for decoding a superblock containing the blocks with horizontal indices $x, x+1, x+2, x+3$ and vertical indices $y, y+1, y+2, y+3$.

Data for blocks within each superblock is preceded by a superblock header containing a split mode.

There are three possible split modes: 0, 1 and 2. In split mode 0, a single set of block data is encoded, which is to be used for all blocks within the superblock. In split mode 1, four sets of block data is encoded, to be used for each of the four sets of 2x2 blocks within the superblock. In split mode 2, all sixteen sets of block data are encoded.

$superblock(y, x)$:	
$sb_split(y//4, x//4)$	6.2.3.1
$block_count = 2^{\mathbf{state}[\text{sb_split}][y//4][x//4]}$	
$step = 4//block_count$	
for $q = 0$ to $block_count - 1$:	
for $p = 0$ to $block_count - 1$:	
$block(y + q * step, x + p * step)$	6.2.4
$propagate_data(y + q * step, x + p * step, step)$	6.2.3.2

6.2.3.1 Superblock splitting mode The *sb_split*(y, x) decodes the superblock splitting mode at superblock coordinates (x, y).

$sb_split(ypos, xpos)$:	
state[sb_split][$ypos$][$xpos$] $= read_uinta(sb_split_contexts())$	
state[sb_split][$ypos$][$xpos$]$+= split_prediction(ypos, xpos)$	6.2.5.2
state[sb_split][$ypos$][$xpos$]$\% = 3$	

6.2.3.2 Propagating data between blocks The *propagate_data*(s, r, k) copies decoded block data from the top-left-most block of a set of $k \times k$ blocks:

$propagate_data(s, r, k)$:	
for $j = s$ to $s + k - 1$:	
for $i = r$ to $r + k - 1$:	
state[block_data][j][i] = **state**[block_data][s][r]	

6.2.4 Block decoding

The *block_decode*(y, x) process parses block motion data for the block at coordinates (x, y). If **state**[using_global] is set then a block will either be intra or will use global motion compensation according to the block mode, and for non-intra blocks a flag is decoded indicating whether this is so. If the block is not intra, and not globally motion compensated, then motion vectors are decoded.

$block_decode(y, x)$:	
state[block_data][y][x][$mode$] = $block_mode(y, x)$	6.2.4.1
if (**state**[block_data][y][x][$mode$] == INTRA):	
state[block_data][y][x][$global$] = **False**	
$block_dc(y, x)$	6.2.4.4
else:	
if (**state**[using_global] == **True**):	
$block_global(y, x)$	6.2.4.2
else:	
state[block_data][y][x][$global$] = **False**	
if (**state**[block_data][y][x][$global$] = **False**):	
$block_vectors(y, x)$	6.2.4.3

Informative: Note that if the picture is using global motion, block global motion vectors are never used: the

6.2.4.1 Block mode

The $block_mode(ypos, xpos)$ process parses the block prediction mode for the block at position $(xpos, ypos)$. Each bit of the block prediction mode is decoded separately.

$block_mode(ypos, xpos)$:	
state[block_data][$ypos$][$xpos$][$mode$][1] = $read_boola$(PMODE_REF1)	
state[block_data][$ypos$][$xpos$][$mode$][1] ˆ = $mode_prediction(ypos, xpos, 1)$	6.2.5.3
if (**state**[num_refs] == 2):	
state[block_data][$ypos$][$xpos$][$mode$][2] = $read_boola$(PMODE_REF1)	
state[block_data][$ypos$][$xpos$][$mode$][2] ˆ = $mode_prediction(ypos, xpos, 2)$	6.2.5.3

6.2.4.2 Block global motion flag

The $block_global(ypos, xpos)$ process parses the block global motion flag for the block at position $(xpos, ypos)$.

$block_mode(ypos, xpos)$:	
state[block_data][$ypos$][$xpos$][$global$] = $read_boola$(GLOBAL_BLOCK)	
state[block_data][$ypos$][$xpos$][$global$] ˆ = $block_global_prediction(ypos, xpos)$	6.2.5.4

6.2.4.3 Block motion vectors

The $block_motion(ypos, xpos)$ process parses the block motion vectors for the block at position $(xpos, ypos)$.

$block_motion(ypos, xpos)$:	
if (**state**[block_data][$ypos$][$xpos$][$mode$][1] == **True**):	
state[block_data][$ypos$][$xpos$][$ref1$].x = $read_sinta(ref1x_contexts())$	6.2.6.2
state[block_data][$ypos$][$xpos$][$ref1$].y = $read_sinta(ref1y_contexts())$	6.2.6.2
state[block_data][$ypos$][$xpos$][$ref1$]+ = $mv_prediction(ypos, xpos, 1)$	
if (**state**[block_data][$ypos$][$xpos$][$mode$][2] == **True**):	
state[block_data][$ypos$][$xpos$][$ref2$].x = $read_sinta(ref2x_contexts())$	6.2.6.2
state[block_data][$ypos$][$xpos$][$ref2$].y = $read_sinta(ref2y_contexts())$	6.2.6.2
state[block_data][$ypos$][$xpos$][$ref2$]+ = $mv_prediction(ypos, xpos, 2)$	

6.2.4.4 DC values

The $block_dc(ypos, xpos)$ process parses the block DC values for the three components for the block

at position $(xpos, ypos)$.

$block_motion(ypos, xpos)$:	
for each c in Y, U, V:	
state[block_data][$ypos$][$xpos$][dc][c] $= read_sinta(dc_contexts(c))$	6.2.6.3
state[block_data][$ypos$][$xpos$][dc][c]$+= dc_prediction(ypos, xpos, c)$	6.2.5.6

6.2.5 Spatial prediction of motion data elements

6.2.5.1 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 3). 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.

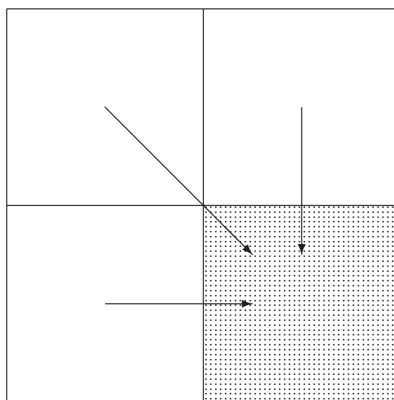

Figure 3: Basic prediction aperture

Note that 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 mode REF2ONLYmay have mode REF1ONLYand so can furnish no contribution for a prediction to the Reference 2 motion vector.

Note also that when superblocks have split level 1 or 0, block data has been propagated (Section 6.2.3.2) across 4 or 16 blocks so as to furnish a prediction. The effect is illustrated for a variety of splitting modes in Figure 4.

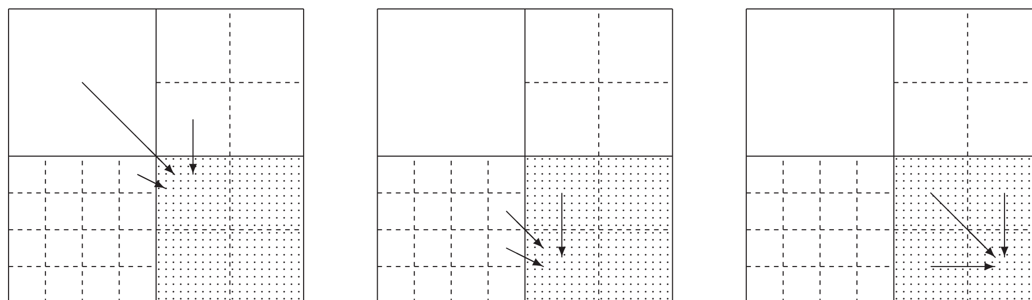

Figure 4: Effect of splitting modes on spatial prediction

6.2.5.2 Superblock split prediction

split_prediction returns the mean of the the neighbouring split values:

$split_prediction(ypos, xpos)$:	
if $(xpos == 0 \&\& ypos == 0)$:	
return 0	
else if $(ypos == 0)$:	
return **state**[sb_split][$ypos$][$xpos-1$]	
else if $(xpos == 0)$:	
return **state**[sb_split][$ypos-1$][$xpos$]	
return (**state**[sb_split][$ypos-1$][$xpos-1$] + **state**[sb_split][$ypos$][$xpos-1$]+ **state**[sb_split][$ypos-1$][$xpos$] + 1)//3	

6.2.5.3 Block mode prediction

mode_prediction returns a majority verdict for each of the references:

$mode_prediction(ypos, xpos, n)$:	
if $(xpos == 0 \&\& ypos == 0)$:	
return **False**	
else if $(ypos == 0)$:	
return **state**[block_data][$ypos$][$xpos-1$][$mode$][n]	
else if $(xpos == 0)$:	
return **state**[block_data][$ypos-1$][$xpos$][$mode$][n]	
return majority(**state**[block_data][$ypos-1$][$xpos-1$][$mode$][n], **state**[block_data][$ypos-1$][$xpos$][$mode$][n], **state**[block_data][$ypos$][$xpos-1$][$mode$][n])	

6.2.5.4 Block global flag prediction

block_global_prediction returns a majority verdict of the neighbouring blocks:

$block_global_prediction(ypos, xpos)$:	
if $(xpos == 0 \&\& ypos == 0)$:	
return **False**	
else if $(ypos == 0)$:	
return **state**[block_data][$ypos$][$xpos-1$][$global$]	
else if $(xpos == 0)$:	
return **state**[block_data][$ypos-1$][$xpos$][$global$]	
return majority(**state**[block_data][$ypos-1$][$xpos-1$][$global$], **state**[block_data][$ypos-1$][$xpos$][$global$], **state**[block_data][$ypos$][$xpos-1$][$global$])	

6.2.5.5 Motion vector prediction

Motion vectors are predicted using the median of available block vectors in the aperture. A vector is available for prediction if a) its block falls within the picture area and b) its prediction mode allows it to be defined and c) it is not a global motion block.

The process $mv_prediction(ypos, xpos, ref)$ returns motion values according to the following rules:

Case 1. If $xpos == 0$ and $ypos == 0$, there are no vectors in the prediction aperture and the zero vector $(0, 0)$ is returned.

Case 2. If $xpos > 0$ and $ypos == 0$ then:

1. If

$\textbf{state}[\text{block_data}][ypos][xpos-1][global] == \textbf{False}$

and

$\textbf{state}[\text{block_data}][ypos][xpos-1][mode][ref] == \textbf{True}$

then $\textbf{state}[\text{block_data}][ypos][xpos-1][ref]$ is returned.

2. Otherwise, $(0, 0)$ is returned

Case 3. If $xpos == 0$ and $ypos > 0$ then:

1. If

   $\textbf{state}[\text{block_data}][ypos-1][xpos][global] == \textbf{False}$

   and

   $\textbf{state}[\text{block_data}][ypos-1][xpos][mode][ref] == \textbf{True}$

   then $\textbf{state}[\text{block_data}][ypos-1][xpos][ref]$ is returned.

2. Otherwise, $(0, 0)$ is returned

Case 4. If both $xpos > 0$ and $ypos > 0$ then all 3 blocks in the prediction aperture may potentially contribute to the prediction. Define sets $valuesx = \emptyset$ and $valuesy = \emptyset$. The prediction is the median vector has horizontal and vertical components equal to the median of the horizontal and vertical components of available vectors:

if ($\textbf{state}[\text{block_data}][ypos][xpos-1][global] == \textbf{True}$):	
if ($\textbf{state}[\text{block_data}][ypos][xpos-1][mode][ref] == \textbf{False}$):	
$valuesx = valuesx \cup \{\textbf{state}[\text{block_data}][ypos][xpos-1][ref].x\}$	
$valuesy = valuesy \cup \{\textbf{state}[\text{block_data}][ypos][xpos-1][ref].y\}$	
if ($\textbf{state}[\text{block_data}][ypos-1][xpos][global] == \textbf{True}$):	
if ($\textbf{state}[\text{block_data}][ypos-1][xpos][mode][ref] == \textbf{False}$):	
$valuesx = valuesx \cup \{\textbf{state}[\text{block_data}][ypos-1][xpos][ref].x\}$	
$valuesy = valuesy \cup \{\textbf{state}[\text{block_data}][ypos-1][xpos][ref].y\}$	
if ($\textbf{state}[\text{block_data}][ypos-1][xpos-1][global] == \textbf{False}$):	
if ($\textbf{state}[\text{block_data}][ypos-1][xpos-1][mode][ref] == \textbf{True}$):	
$valuesx = valuesx \cup \{\textbf{state}[\text{block_data}][ypos-1][xpos-1][ref].x\}$	
$valuesy = valuesy \cup \{\textbf{state}[\text{block_data}][ypos-1][xpos-1][ref].y\}$	
return $(\text{median}(valuesx), \text{median}(valuesy))$	

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

6.2.5.6 DC value prediction

DC values are predicted using the unbiased mean available values in the aperture. The process $dc_prediction(ypos, xpos, comp)$ returns values according to the following rules:

Case 1. If $xpos == 0$ and $ypos == 0$, there are no blocks in the prediction aperture and the default prediction $2^{\textbf{state}[\text{video_depth}]-1}$ is returned.

Case 2. If $xpos > 0$ and $ypos == 0$ then:

1. If $\textbf{state}[\text{block_data}][ypos][xpos-1][mode] == \text{INTRA}$, $\textbf{state}[\text{block_data}][ypos][xpos-1][dc][comp]$ is returned

2. Otherwise, $2^{\textbf{state}[\text{video_depth}]-1}$ is returned

Case 3. If $xpos == 0$ and $ypos > 0$ then:

1. If **state**[block_data][$ypos-1$][$xpos$][$mode$] == INTRA, **state**[block_data][$ypos-1$][$xpos$][dc][$comp$ is returned

2. Otherwise, $2^{\mathbf{state}[\text{video_depth}]-1}$ is returned

Case 4. If both $xpos > 0$ and $ypos > 0$ then all 3 blocks in the prediction aperture may potentially contribute to the prediction. Define a set $values = \emptyset$. The prediction is the unbiased mean of available values:

...	
$pred = (0, 0)$	
if (**state**[block_data][$ypos$][$xpos - 1$][$mode$] == INTRA):	
$values = values \cup \{$**state**[block_data][$ypos$][$xpos - 1$][dc][$comp$]$\}$	
if (**state**[block_data][$ypos - 1$][$xpos$][$mode$] == INTRA):	
$values = valuesx \cup \{$**state**[block_data][$ypos - 1$][$xpos$][ref][dc][$comp$]$\}$	
if (**state**[block_data][$ypos - 1$][$xpos - 1$][$mode$] == INTRA):	
$values = values \cup \{$**state**[block_data][$ypos - 1$][$xpos - 1$][ref][dc][$comp$]$\}$	
if ($values! = \{\}$):	
return $pred = (unbiased_mean(values)$	
else:	
return $2^{\mathbf{state}[\text{video_depth}]-1}$	

6.2.6 Block motion data contexts

6.2.6.1 Superblock splitting mode

The $sb_split_contexts()$ function returns the following unsigned integer context set:

- Follow = [SB_F1, SB_F2]
- Data = SB_DATA

6.2.6.2 Motion vectors

There are four motion vector context sets, which are signed integer context sets as follows.

$ref1x_contexts$ returns the set with

- Follow = [REF1x_F1, REF1x_F2, REF1x_F3, REF1x_F4, REF1x_F5+]
- Data = REF1x_DATA
- Sign = REF1x_SIGN

$ref1y_contexts$ returns the set with

- Follow = [REF1y_F1, REF1y_F2, REF1y_F3, REF1y_F4, REF1y_F5+]
- Data = REF1y_DATA
- Sign = REF1y_SIGN

$ref2x_contexts$ returns the set with

- Follow = [REF1x_F1, REF1x_F2, REF1x_F3, REF1x_F4, REF1x_F5+]
- Data = REF1x_DATA

- Sign = REF1x_SIGN

ref1y_contexts returns the set with

- Follow = [REF2y_F1, REF2y_F2, REF2y_F3, REF2y_F4, REF2y_F5+]
- Data = REF2y_DATA
- Sign = REF2y_SIGN

6.2.6.3 DC values

There are three DC value context sets, which are signed integer context sets for each component.

dc_contexts(Y) returns the set:

- Follow = [YDC_F1, YDC_F2+]
- Data = YDC_DATA
- Sign = YDC_SIGN

dc_contexts(U) returns the set:

- Follow = [UDC_F1, UDC_F2+]
- Data = UDC_DATA
- Sign = UDC_SIGN

dc_contexts(V) returns the set:

- Follow = [VDC_F1, VDC_F2+]
- Data = VDC_DATA
- Sign = VDC_SIGN

7 Wavelet coefficient decoding

This section specifies how wavelet transform coefficients are parsed.

7.1 Decoded subband data conventions

7.1.1 Wavelet data initialisation

This section specifies the $initialise_wavelet_data()$ process, which returns a structure which will contain the decoded wavelet coefficients for the component.

For the purposes of this specification, this is a four-dimensional array $data$, where individual subbands are two-dimensional arrays accessed by level and depth:

$band = data[level][orientation]$

Valid levels are integers from in the range 0 to **state**[transform_depth]inclusive. Level 0 consists of a single subband with orientation LL. All other levels consist of 3 subbands of orientation LH, HL, and HH. The orientations correspond to either low- or high-pass filtering horizontally and vertically: so the LH band consists of coefficients derived from horizontal low-pass filtering and vertical high-pass filtering. The subbands partition the spatial frequency domain by orientation and level as illustrated in Figure 5.

Individual subband coefficients are signed integers accessed by vertical and horizontal coordinates within the subband:

$c = band[y][x], x \in \{0, ..., subband_width(level) - 1\}, \quad y \in \{0, ..., subband_height(level) - 1\}$

where the dimensions $subband_width(level)$ and $subband_height(level)$ of the subband are as defined in Section 7.1.2. 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 $2^{\mathbf{state}[\text{transform_depth}]}$.

7.1.2 Dimensions of wavelet subbands

This section defines the values of the $subband_width(level)$ and $subband_height(level)$ functions, giving the width and height of subbands at a given level, and hence the range of subband vertical and horizontal indices.

Define the padded dimensions of the component by

$$ph = 2^{\mathbf{state}[\text{transform_depth}]} * \left\lceil \frac{\mathbf{state}[\text{component_height}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil$$

$$pw = 2^{\mathbf{state}[\text{transform_depth}]} * \left\lceil \frac{\mathbf{state}[\text{component_width}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil$$

If $level == 0$,

$$\begin{aligned} subband_height(level) &= ph//2^{\mathbf{state}[\text{transform_depth}]} \\ &= \left\lceil \frac{\mathbf{state}[\text{component_height}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil \\ subband_width(level) &= pw//2^{\mathbf{state}[\text{transform_depth}]} \\ &= \left\lceil \frac{\mathbf{state}[\text{component_width}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil \end{aligned}$$

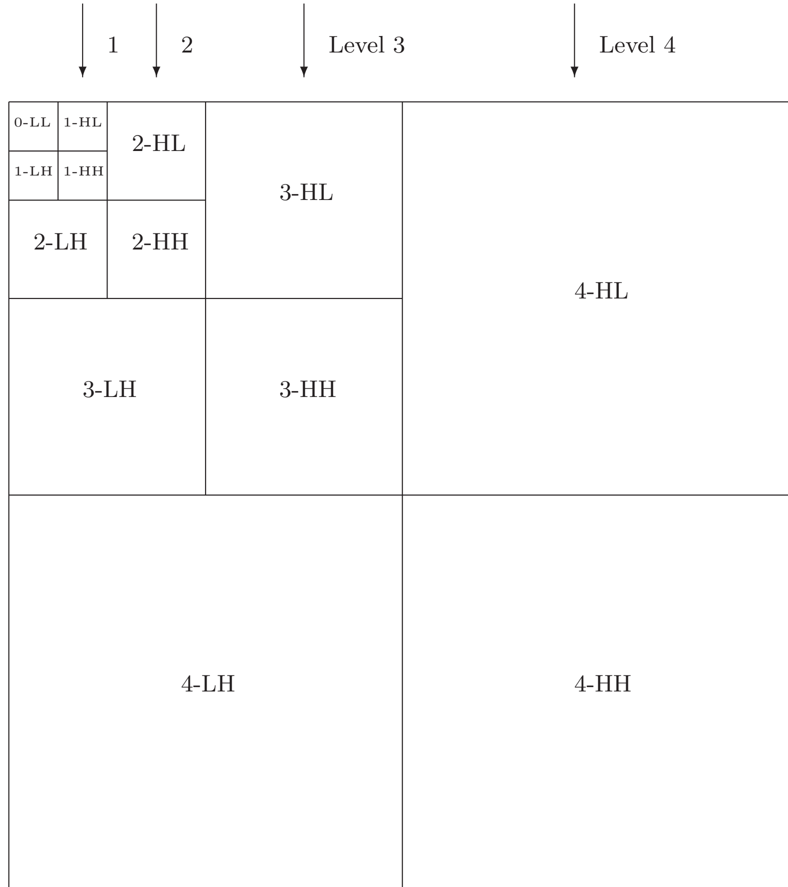

Figure 5: Subband decomposition of the spatial frequency domain showing subband numbering, for a 4-level wavelet decomposition

If $level > 0$

$$\begin{aligned}
subband_height(level) &= ph//2^{\mathbf{state}[\text{transform_depth}]-level+1} \\
&= 2^{level-1} * \left\lceil \frac{\mathbf{state}[\text{component_height}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil \\
subband_width(level) &= pw//2^{\mathbf{state}[\text{transform_depth}]-level+1} \\
&= 2^{level-1} * \left\lceil \frac{\mathbf{state}[\text{component_width}]}{2^{\mathbf{state}[\text{transform_depth}]}} \right\rceil
\end{aligned}$$

Informative: 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 9). 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.

7.2 Wavelet data decoding process

7.2.1 Summary

This section specifies the overall operation of the $transform_data()$ process for parsing and decoding the set of coefficient subbands corresponding to a video picture component (Y, U or V), according to the conventions set out in Section 7.1.

In the Dirac stream, subband data is arranged by level and orientation, from level 0 up to level

state[transform_depth]. The decoding 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.

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

Informative: The data for each subband consists of a subband header and a block of arithmetically coded coefficient data. The subband header contains a length code giving the number of bytes of the block of arithmetically-coded data. The transform data can therefore be parsed without invoking arithmetic decoding at all, since the length codes allow a parser to skip from one subband header to the next, similarly to the way that parse unit offsets allow frame skipping.

7.2.2 Decoding loop

The overall $transform_data()$ decoding process is as follows:

$transform_data()$:	
$data = initialise_wavelet_data()$	7.1.1
$subband_decode(data, 0, LL)$	7.3
for $level = 1$ to **state**[transform_depth]:	
for each $orient$ in LH, HL, HH:	
$subband_decode(data, level, orient)$	7.3
$byte_align()$	
return $data$	

7.3 Subband decoding process

This section specifies the process $subband_decode(level, orient)$ for coefficients within a subband at level $level$ (0 to **state**[transform_depth]) and of orientation $orient$ (LL, LH, HL, or HH).

7.3.1 Subband header and codeblock loop

This section specifies the operation of the $subband_decode(data, level, orient)$ function for decoding a subband at level $level$ and of orientation $orient$.

7.3.1.1 Initialisation

The $subband_decode()$ process begins by reading a length code. If this length is zero, then the subband is deemed to be skipped and all coefficients are set to zero before the process exits:

$subband_decode(data, level, orient)$:	
$length = read_unsigned()$	
if ($length == 0$):	
for $y = 0$ to $subband_height(level) - 1$:	
for $x = 0$ to $subband_width(level) - 1$:	
$band[y][x] = 0$	
return	
...	

If $length > 0$ then the subband coefficient decoding process is initialised by setting up arithmetic decoding, initialising the coefficient count, reading the quantisation index and byte-aligning the subsequent read operations.

...	
if ($length > 0$):	
$initialise_arithmetic_decoding(length)$	4.2
state[coefficient_count] = 0	
$vol = subband_height(level) * subband_width(level)$	
state[coefficient_reset] $= max(min(vol//32, 800), 25)$	
$quant_index = read_uint()$	
$byte_align()$	
...	

Note: byte alignment only occurs if $length > 0$: a skipped subband is **not** byte-aligned.

7.3.1.2 Codeblocks

Data within a subband is divided into code blocks, representing rectangular blocks of coefficients. The numbers of codeblocks in each subband are determined in decoding the transform header, as specified in Section 5.11.3.

The overall subband decoding process loops over all the code blocks after initialising the arithmetic decoding engine, and setting quantisers. There is a different code block decoding process for Intra DC bands, since values are coded using spatial prediction.

...	
$band = data[level][orient]$	
if ($level > 1$):	
$parent = data[level-1][orient]$	
else:	
$parent = \emptyset$	
for $y = 0$ to **state**[codeblocks]$[level][v] - 1$:	
for $x = 0$ to **state**[codeblocks]$[level][h] - 1$:	
$codeblock(band, parent, orient, quant_index, y, x)$	7.3.2
if ($is_intra()$ and $level == 0$):	
$intra_dc_prediction(band)$	7.3.3

7.3.2 Decoding subband codeblocks

This section defines the operation of the $codeblock(band, parent, orient, quant_index, y, x)$ function, which decodes a codeblock in position (x, y) and populates it with reconstructed wavelet coefficients.

7.3.2.1 Codeblock dimensions

The codeblock covers coefficients in the horizontal range $left$ to $right - 1$ and in the vertical range $bottom$ to $top - 1$ where these values are defined by:

$$\begin{aligned}
left &= (subband_width(level) * x)//\mathbf{state}[\text{codeblocks}][level][horizontal] \\
right &= (subband_width(level) * (x+1))//\mathbf{state}[\text{codeblocks}][level][horizontal] \\
bottom &= (subband_height(level) * y)//\mathbf{state}[\text{codeblocks}][level][vertical] \\
top &= (subband_height(level) * (y+1))//\mathbf{state}[\text{codeblocks}][level][vertical]
\end{aligned}$$

7.3.2.2 Codeblock decode process

The codeblock decoding process is defined as:

$codeblock(band, parent, orient, quant_index, y, x)$:	
if $(zero_flag())$:	7.3.2.3
for $v = bottom$ to $top - 1$:	
for $h = left$ to $right - 1$:	
$band[v][h] = 0$	
else:	
$quant_idx += quant_offset()$	7.3.2.4
for $v = bottom$ to $top - 1$:	
for $h = left$ to $right - 1$:	
$coeff_decode(band, parent, orient, quant_idx, v, h)$	7.4

7.3.2.3 Zero block flag

We may set the number of codeblocks in the subband as

num_blocks = **state**[codeblocks][$level$][$horizontal$] ∗ **state**[codeblocks][$level$][$vertical$]

If num_blocks is 1 or $level = 0$, then $zero_flag()$ returns **False**.

Otherwise, the flag is decoded from the stream: $read_boola$(ZERO_BLOCK) is returned.

7.3.2.4 Block quantiser offset

If **state**[codeblock_mode] = SINGLE_QUANT, $quant_offset()$ shall return 0.

If **state**[codeblock_mode] = MULTI_QUANT then the quantiser index offset is decoded from the stream – $read_sinta(quant_contexts())$ is returned, where $quant_contexts()$ returns the context set:

- Follow= {Q_OFFSET_FOLLOW}
- Data=Q_OFFSET_INFO
- Sign=Q_OFFSET_SIGN

7.3.3 Intra DC band prediction

This section defines the operation of the $intra_dc_prediction(band)$ function for reconstructing values within Intra picture DC bands using spatial prediction. Intra DC values are derived by spatial prediction using the mean of the three values to the left, top-left and above a coefficient (if available).

$intra_dc_prediction(band)$:	
$prediction = 0$	
for $v = 0$ to $subband_height(level) - 1$:	
for $h = 0$ to $subband_width(level) - 1$:	
$prediction = 0$	
$N = 0$	
if $(v > 0)$:	
$prediction+ = band[v-1][h]$	
$N+ = 1$	
if $(h > 0)$:	
$prediction+ = band[v-1][h-1] + band[v][h-1]$	
$N+ = 1$	
else:	
if $(h > 0)$:	
$prediction+ = band[0][h-1]$	
$N+ = 1$	
$prediction = prediction//N$	
$band[v][h]+ = prediction$	

7.4 Subband coefficient decoding process

This section describes the operation of the $coeff_decode(band, parent, orient, quant_idx, v, h)$ process for decoding a coefficient in position (h, v) in the subband *band* with parent band *parent* and orientation *orient*.

Decoding a coefficient makes use of arithmetic decoding, inverse quantisation and, in the case of DC (level 0) bands of Intra frames, neighbourhood prediction. The decoding process periodically refreshes the contexts, by halving the context counts every time a count is reached.

Arithmetic coding uses a highly compact set of contexts, with magnitudes contextualised on whether parent values and neighbouring values are zero or non-zero.

7.4.1 Overall coefficient decoding process

Two different processes are used for decoding coefficients, depending upon whether spatial prediction is required or not.

$coeff_decode(band, parent, orient, quant_index, v, h)$:	
$parent = parent_val(parent, v, h)$	7.4.2
$nhood = zero_nhood(band, v, h)$	7.4.3
$sign_pred = sign_predict(band, orient, v, h)$	7.4.4
$context_set = select_coeff_ctxs(parent, nhood, sign_pred)$	7.4.5
$quant_coeff = read_sina(context_set)$	
$band[v][h] = inverse_quant(quant_coeff, quant_index)$	7.4.6
$update_count()$	7.4.8

7.4.2 Parent values

The function $parent_val(v, h)$ returns the parent value of a coefficient in a subband, which is the co-located coefficient in the parent subband. If there is a parent band ($parent! = \emptyset$), then

$parent[v//2][h//2]$

is returned. If there is no parent ($parent = \emptyset$), then 0 is returned.

7.4.3 Zero neighbourhood

The $zero_nhood()$ function returns a boolean indicating whether neighbouring values are all zero.

$zero_nhood(band, v, h)$:	
if ($v > 0$):	
if ($band[v-1][h] != 0$):	
return **False**	
if ($h > 0$):	
if ($band[v-1][h-1]) != 0 \|\| band[v][h-1] != 0$):	
return **False**	
else:	
if ($h > 0$):	
if ($band[v][h-1] != 0$):	
return **False**	
return **True**	

7.4.4 Sign prediction

The $sign_predict()$ function returns 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 is the sign of the coefficient above the current coefficient; for horizontally-oriented (LH) bands, the predictor is the sign of the coefficient to the left.

The predictions are not used for differential encoding of the sign, but for conditioning of the sign contexts only.

$sign_predict(band, orient, v, h)$:	
if ($orient == HL$):	
if ($v == 0$):	
return 0	
else:	
return $sign(band[v-1][h])$	
else if ($orient == LH$):	
if ($h == 0$):	
return 0	
else:	
return $sign(band[v][h-1])$	
else:	
return 0	

7.4.5 Coefficient context selection

This section defines the $select_context(zero_nhood, parent, sign)$ function, which chooses a context index set for decoding a coefficient value.

Twelve possible coefficient index sets are defined, and are returned as specified in Table 6. Note that follow contexts are an array indexed from 0 as per Section 4.4.3.

Note that 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 is used for all coefficients.

parent	*zero_nhood*	*sign*	Context set	
0	**True**	0	Follow	[ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_ZERO
0	**True**	< 0	Follow	[ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_NEG
0	**True**	> 0	Follow	[ZPZN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_POS
0	**False**	0	Follow	[ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_ZERO
0	**False**	< 0	Follow	[ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_NEG
0	**False**	> 0	Follow	[ZPNN_F1,ZP_F2,ZP_F3, ZP_F4,ZP_F5,ZP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_POS
$\neq 0$	**True**	0	Follow	[NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_ZERO
$\neq 0$	**True**	< 0	Follow	[NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_NEG
$\neq 0$	**True**	> 0	Follow	[NPZN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_POS
$\neq 0$	**False**	0	Follow	[NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_ZERO
$\neq 0$	**False**	< 0	Follow	[NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_NEG
$\neq 0$	**False**	> 0	Follow	[NPNN_F1,NP_F2,NP_F3, NP_F4,NP_F5,NP_F6+]
			Data	COEFF_DATA
			Sign	SIGN_POS

Table 6: Subband coefficient context sets

7.4.6 Inverse quantisation

The inverse_quant() function is defined by:

$inverse_quantise(quantised_coeff, quant_index):$	
$magnitude = \lvert quantised_coeff \rvert$	
$magnitude *= quant_factor(quant_index)$	7.4.7
$magnitude += quant_offset(quant_index)$	7.4.7
$magnitude = magnitude//4$	
return $sign(quantised_coeff) * magnitude$	

Informative: The pseudocode description separates inverse quantisation from decoding. However, since dead-zone quantisation is used, the *inverse_quant*() function must compute the magnitude. Hence it is more efficient to first decode the coefficient magnitude, then inverse quantise, and then

decode the coefficient sign.

7.4.7 Quantisation factors and offsets

This section specifies the operation of the $quant_factor()$ and $quant_offset()$ functions for performing inversion quantisation.

Quantisation factors represent an approximation of quarter-bit values with two bits of accuracy (i.e. $round(2^{\frac{index}{4}+2})$:

$quant_factor(index)$:	
$base = 2 ** (index//4)$	
if $((x\%4) == 0)$:	
return $4 * base$	
else if $((x\%4) == 1)$:	
return $78892 * base + 8292)//16585$	
else if $((x\%4) == 2)$:	
return $228486 * base + 20195)//40391$	
else if $((x\%4) == 3)$:	
return $440253 * base + 32722)//65444$	

Offsets are approximately 3/8 of the quantisation factors - these mark the reconstruction point within the quantisation interval:

$quant_offset(index)$:	
return $(quant_factor(index) * 3 + 4)//8$	

7.4.8 Updating counts and resetting contexts

The $update_count()$ function updates a periodic count of subband coefficients and rescales arithmetic decoding contexts if **state**[coefficient_reset]has been reached.

$update_count()$:	
state[coefficient_count]+ = 1	
if (**state**[coefficient_count] == **state**[coefficient_reset]):	
state[coefficient_count] = 0	
for $i = 0$ to $len($**state**[contexts]$) - 1$:	
$rscale_context(i)$	4.1

Part III

Decoding operations

8 Picture decoding process

This section specifies the process for decoding a picture from the Dirac stream. Picture decoding depends upon correctly parsing the Dirac bitstream, and decoding operations are dependent upon the parsing operations set out in Sections 5, 6 and 7.

This section does not specify how pictures are encoded, nor how pictures are presented for display, which are outside the scope of this specification.

8.1 Introduction

Dirac supports both intra and inter picture coding, with forward and backward prediction. This means that pictures may be encoded in the stream in non-display order: reordering pictures will be required in order to display them correctly, and decoded picture buffer will be necessary to store pictures while temporally prior pictures are decoded. Note that the core Dirac specification does not encompass the operation of the decoded picture buffer: this is specified in conjunction with the level and profile values extracted from the stream (Section 5.6), in Appendix D.

Decoded pictures may, however, be reference pictures, used for the prediction of subsequent pictures in the Dirac stream. Reference pictures are stored in a reference picture buffer **state**[ref_buffer]. The operation of **state**[ref_buffer] does form part of the core Dirac specification, and the rules for management of the buffer are set out in Section 8.4.

8.2 Overall picture decoding process

8.2.1 Picture data initialisation

Picture data from the current picture being decoded is stored in the **state**[current_picture] state variable, which is a structure with indices *pic_num*, Y, U and V.

The *init_picture_data*() initialises the current picture data so that:

- **state**[current_picture][*pic_num*] = **state**[picture_number]
- **state**[current_picture][Y] is a 2-dimensional of width **state**[luma_width] and height**state**[luma_height], all values **state**[current_picture][Y][y][x] set to 0
- **state**[current_picture][U] and $CurrentPicture[V]$ are 2-dimensional arrays of width **state**[chroma_width] and height **state**[chroma_height], all values **state**[current_picture][U][y][x] and **state**[current_picture][V][y][x] set to 0

8.2.2 Initialisation

The process for decoding pictures within a Dirac sequence can commence once an Access Unit header has been located and parsed, and the default parameters set. This is achieved by searching for a Parse Info header within the sequence for which *is_au*() returns **True** and then initialising the default state parameters by parsing the subsequent Access Unit header as per Section 5.5.

At this point the reference picture buffer shall be initialised with no reference pictures.

This initialisation process need only occur once within a sequence, since apart from the AU picture number, all AU headers within a sequence are identical.

8.2.3 Decoding process

The process for decoding a picture with picture number n in a Dirac sequence consists of:

$decode_pic(n)$:	
$seek_picture_parse(n)$	8.3
$picture()$	5.9
$init_picture_data()$	8.2.2
$ref_buffer_remove()$	8.4
if (**state**[zero_residual] == **False**):	
state[current_picture][Y] = $idwt$(**state**[y_transform])	9
state[current_picture][U] = $idwt$(**state**[u_transform])	9
state[current_picture][V] = $idwt$(**state**[v_transform])	9
if ($is_inter()$):	
$ref1 = get_ref$(**state**[ref1_picture_number])	
if ($num_refs() == 2$):	5.3
$ref2 = get_ref$(**state**[ref2_picture_number])	
$motion_compensate(ref1[Y], ref2[Y],$ **state**[current_picture]$[Y], c)$	10
$motion_compensate(ref1[U], ref2[U],$ **state**[current_picture]$[U], c)$	10
$motion_compensate(ref1[V], ref2[V],$ **state**[current_picture]$[V], c)$	10
clip $_picture()$	8.5
if ($is_ref()$):	
$ref_buffer_add()$	8.4

When randomly accessing a sequence, a picture may not be decodeable because reference pictures may not be available in the buffer. In this case the current picture may be discarded, although some decoders may be designed to produce an output.

A Dirac sequence shall be so constructed so that if pictures are decoding commences from the beginning of the stream and pictures are decoded in stream order, there shall be no undecodeable pictures i.e. the reference pictures associated with any picture in the sequence stall have occurred prior to that picture in the sequence.

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

8.3 Seeking in the Dirac stream

The $seek_picture_parse(n)$ locates and parses a Parse Info header within the Dirac sequence for which $is_picture()$ is **True** and **state**[picture_number] $== n$. If pictures are being decoded sequentially from the stream, there is no need to search for parse codes. The parse offset values may be used to make the seeking process greatly more efficient, so that only a single parse code need be located initially (Section 1)

8.4 Reference picture buffer management

This section specifies how the Dirac stream data is used to manage the reference picture buffer **state**[ref_buffer]. The reference picture buffer has a maximum size of 5 elements.

The $ref_picture_remove()$ process operates as follows:

$ref_picture_remove()$:	
for $i = 0$ to length(**state**[retired_picture_list]) $- 1$:	
$n =$ **state**[retired_picture_list][i]	
for $k = 0$ to length(**state**[ref_buffer]) $- 1$:	
if (**state**[ref_buffer][k][pic_number] $== n$):	
delete **state**[ref_buffer][k]	
state[retired_picture_list] $= \emptyset$	

The $get_ref(n)$ function returns the (first) reference picture in the buffer with picture number n.

The $ref_picture_add()$ process for adding pictures to the reference picture buffer proceeds according to the following rules:

Case 1. If the reference picture buffer is not full i.e. has fewer than 5 elements, then add the **state**[current_picture] data to the end of the buffer.

Case 2. If the reference picture is full i.e. it has 5 elements, then remove the first (i.e. oldest) element of the buffer **state**[ref_buffer][0], and set

$$\textbf{state}[\text{ref_buffer}][4] = \textbf{state}[\text{current_picture}]$$

8.5 Clipping

Picture data must be clipped to the video range prior to being output or being used as a reference:

$clip_picture()$:	
for each c in Y, U, V:	
$clip_comp$(**state**[current_picture][c])	

$clip_component(data)$:	
for $y = 0$ to height($data$) $- 1$:	
for $x = 0$ to width($data$) $- 1$:	
$data = \text{clip}(data[y][x], 0, 2^{\textbf{state}[\text{video_depth}]} - 1)$	

9 Inverse discrete wavelet transform

This section defines the process $idwt(data)$ for reconstructing picture component data from decoded subband data $data$ using the Inverse Discrete Wavelet Transform (IDWT). $idwt()$ can be invoked in the picture decoding process only after successful parsing of the subband coefficient data (Section 7).

The $idwt()$ process consists of two sub-processes:

1. Synthesis, which returns a pixel array from the subband wavelet coefficients:
   $pic = idwt_synthesis(data)$ (Section 9.1)
2. Pad-removal, which removes padding values from the synthesised pixel array pic (Section 9.5)

The output returned by this process is a two-dimensional array pic of pixel data representing Y, U or V component data.

Informative: The IDWT can operate with a number of different wavelet filters, whose identity has been signalling in the transform data header. These allow trade-offs to be made between compression efficiency, implementation complexity and perceptual quality.

Different filters are applicable in different scenarios. Shorter filters are generally more suitable for motion-compensated residual data, and longer filters for intra picture data. The default filter set includes a "Fidelity" filter optimised for downconversion, so that a lower-resolution, but high-quality, proxy layer may be decoded for viewing.

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 $referenceto$ 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 $referenceto$ the value of $a[r][k]$. This means that modifying the value of $b[r]$ modifies the value of $a[r][k]$.

Informative: These definitions allow us to express the important feature that all filtering operations are specified as *in place* calculations, not involving data being copied at any stage.

9.1 IDWT synthesis operation

This section defines the process $idwt_synthesis(pic, data)$ invoked by $idwt()$.

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

$idwt_synthesis(data)$:	
$LL_band = data[0][LL]$	
for $n = 0$ to $\textbf{state}[\text{wavelet_depth}] - 1$:	
$LL_band = vh_synthesis(LL_band, data[n][HL], data[n][LH], data[n][HH])$	9.2
return LL_band	

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

9.2 Vertical and horizontal synthesis

This section specifies the operation of the vertical and horizontal synthesis process $vh_synthesis(LL_data, HL_data, LH_data, HH_d$

vh_synthesis is repeatedly invoked by *idwt_synthesis*(). It operates on four subband data arrays of identical dimensions to produce a new array *synth*, which is returned as the result of the process.

Step 1. *synth* is 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 is interleaved as follows:

...	
for $y = 0$ to $(\text{height}(synth)//2) - 1$:	
for $x = 0$ to $(\text{width}(synth)//2) - 1$:	
$synth[2*y][2*x] = LL_data[y][x]$	
$synth[2*y][2*x+1] = HL_data[y][x]$	
$synth[2*y+1][2*x] = LH_data[y][x]$	
$synth[2*y+1][2*x+1] = HH_data[y][x]$	
...	

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

Step 3. Data is next synthesised vertically by operating on each column of data using a one-dimensional filter, and then horizontally by operating on each row.

...	
for $x = 0$ to $\text{width}(synth) - 1$:	
$1d_synthesis(\text{column}(synth, x))$	??
for $y = 0$ to $\text{height}(synth) - 1$:	
$1d_synthesis(\text{row}(synth, y))$	??
...	

Step 4. Finally, the synthesised subband data receives a bitshift to remove any accuracy bits. The shift value $filtershift()$ used is as determined in Section 9.4 from the wavelet index **state**[wavelet_index].

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

Informative: 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 is the most efficient way of providing additional resolution where it is needed: the result is that the shift varies with the level to which a subband belongs.

Accuracy bits are required because the rounding stages in integer lifting introduce non-linearities that can aliase data between subbands. Quantising data in non-DC bands can then (for example) produce artefacts at DC band frequencies. This increases bit rate and the perceptual impact of quantisation, especially for 8-bit video. The accuracy bits have been computed so as to virtually eliminate these effects (with the exception of Haar0 which is included as it is suitable for low-delay low-complexity implementations, especially lossless coding). For example, Fidelity does not require a shift value since its filter gain is such as to add a bit of resolution with each level of decomposition.

9.3 One-dimensional synthesis

This section specifies the one-dimensional synthesis process $1d_synthesis()$ applied 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 comprises the application of a number of reversible integer lifting filter operations. An integral lifting filter is characterised by four elements:

- a set of taps $t_{-N}, \ldots, t_M$
- a scale factor s
- a parity (odd or even)
- whether it is an "update" or "predict" filter

An even lifting filtering operation modifies the even coefficients by the odd coefficients:

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

An odd lifting filtering operation modifies the odd coefficients by the even coefficients:

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

Informative: Note that the distinction between update and predict filters is necessary because integer rounding is being used, and the filters are non-linear. A predict filter with taps t_i is not equivalent to an update filter with taps $-t_i$.

These formulae are deemed to be applied for all applicable n before the next filtering stage is applied and to use reflection at the array edges where the filter would otherwise overlap. In pseudo-code, two functions $lifteven(A, t_i, s)$ and $liftodd(A, t_i, s)$ are specified by:

$lifteven(A, t_i, s, fsort)$:	
for $n = 0$ to (length(A)//2) $- 1$:	
$sum = 0$	
for $i = -N$ to M:	
$pos = 2 * (n + i) - 1$	
if ($pos < 0$):	
$sum+ = A[-pos]$	
else if ($pos \geq$ length(A)):	
$sum+ = A[2 *$ length$(A) - pos]$	
else:	
$sum+ = A[pos]$	
$sum+ = (1 \ll (s - 1))$	
if ($fsort == PREDICT$):	
$A[2 * n]- = (sum \gg s)$	
else:	
$A[2 * n]+ = (sum \gg s)$	

and

$liftodd(A, t_i, s, fsort)$:	
for $n = 0$ to (length(A)//2) $- 1$:	
$sum = 0$	
for $i = -N$ to M:	
$pos = 2 * (n + i)$	
if ($pos < 0$):	
$sum+ = A[-pos]$	
else if ($pos \geq$ length(A)):	
$sum+ = A[2 *$ length$(A) - pos]$	
else:	
$sum+ = A[pos]$	
$sum+ = (1 \ll (s - 1))$	
if ($fsort == PREDICT$):	
$A[2 * n + 1]- = (sum \gg s)$	
else:	
$A[2 * n + 1]+ = (sum \gg s)$	

1d_synthesis applies the sequence of lifting filters specified in Section 9.4, invoking *lifteven* for even parity filters and *liftodd* for odd parity filters.

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

9.4 Filters and shift values

The following the lifting filters and bitshift operations that apply for each value of **state**[wavelet_index] are specified in Tables 7 to 14.

Informative: This specification contains an implementation of the Daubechies (9,7) filter (**state**[wavelet_index] == 7). Daubechies (9,7), like any other FIR biorthogonal wavelet, possesses a lifting implementation. However, to produce a perfect reconstruction filter, the lifting stages require real-valued filter taps, and a final coefficient scaling stage. Any realisable implementation is therefore an approximation. The implementation specified here is fully integral, yet (other than omitting the final scaling stages), it is a very close approximation. These filters can therefore (unlike the JPEG2000 implementation) be used for lossless as well as lossy compression, whilst lossy compression performance is near-indentical to the real-valued filter. The integer lifting implementaiton also allows for much more efficient implementation.

Lifting steps	$filtershift()$
1. Even, Predict, $s = 2, t_0 = 1, t_1 = 1$ i.e. $A[2*n]- = (A[2*n-1] + A[2*n+1] + 2) \gg 2$ 2. Odd, Update, $s = 4$, $t_{-1} = -1, t_0 = 9, t_1 = 9, t_2 = -1$ i.e. $A[2*n+1]+ = (-A[2*n-2] + 9*A[2*n] + 9*A[2*n+2] - A[2*n+4] + 8) \gg 4$	1

Table 7: **state**[wavelet_index] == 0: Deslauriers-Debuc (9,3) lifting stages and shift values

Lifting steps	$filtershift()$
1. Even, Predict, $s = 2, t_0 = 1, t_1 = 1$ i.e. $A[2*n]- = (A[2*n-1] + A[2*n+1] + 2) \gg 2$ 2. Odd, Update, $s = 1$, $t_0 = 1, t_1 = 1$ i.e. $A[2*n+1]+ = (A[2*n] + A[2*n+2] + 1) \gg 1$	1

Table 8: **state**[wavelet_index] == 1: LeGall (5,3) lifting stages and shift values

Lifting steps	$filtershift()$
1. Even, Predict, $s = 5, t_{-1} = -1, t_0 = 9, t_1 = 9, t_2 = -1$ 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. Odd, Update, $s = 4, t_{-1} = -1, t_0 = 9, t_1 = 9, t_2 = -1$ i.e. $A[2*n+1]+ = (-A[2*n-2] + 9*A[2*n] + 9*A[2*n+2] + A[2*n+4] + 8) \gg 4$	1

Table 9: **state**[wavelet_index] == 2: Deslauriers-Debuc (13,5) lifting stages and shift values

Lifting steps	$filtershift()$
1. Even, Predict, $s = 1, t_1 = 1$ i.e. $A[2*n]- = (A[2*n+1] + 1) \gg 1$ 2. Odd, Update, $s = 0, t_0 = 1$ i.e. $A[2*n+1]+ = A[2*n]$	0

Table 10: **state**[wavelet_index] == 3: Haar filter with no shift

Lifting steps	$filtershift()$
1. Even, Predict, $s = 1, t_1 = 1$ i.e. $A[2*n]- = (A[2*n+1] + 1) \gg 1$ 2. Odd, Update, $s = 0, t_0 = 1$ i.e. $A[2*n+1]+ = A[2*n]$	1

Table 11: **state**[wavelet_index] == 4: Haar filter with single shift

Lifting steps	$filtershift()$
1. Even, Predict, $s = 1, t_1 = 1$ i.e. $A[2*n] -= (A[2*n+1]+1) \gg 1$ 2. Odd, Update, $s = 0, t_0 = 1$ i.e. $A[2*n+1] += A[2*n]$	2

Table 12: **state**[wavelet_index] == 5: Haar filter with double shift

Lifting steps	$filtershift()$
1. Even, Predict, $s = 8, t_{-1} = -11, t_0 = 36, t_1 = 36, t_2 = -11$ i.e. $A[2*n] -= (-11*A[2*n-3]+36*A[2*n-1]+36*A[2*n+1]-11*A[2*n+3]+128) \gg 8$ 2. Odd, Update, $s = 8, t_{-1} = -22, t_0 = 84, t_1 = 84, t_2 = -22$ i.e. $A[2*n+1] += (-22*A[2*n-2]+84*A[2*n]+84*A[2*n+2]-22*A[2*n+4]+128) \gg 8$ 3. Even, Predict, $s = 8, t_{-1} = -19, t_0 = 85, t_1 = 85, t_2 = -19$ i.e. $A[2*n] -= (-19*A[2*n-3]+85*A[2*n-1]+85*A[2*n+1]-19*A[2*n+3]+128) \gg 8$ 4. Odd, Update, $s = 8, t_{-1} = -8, t_0 = 41, t_1 = 41, t_2 = -8$ i.e. $A[2*n+1] += (-8*A[2*n-2]+41*A[2*n]+41*A[2*n+2]-8*A[2*n+4]+128) \gg 8$	0

Table 13: **state**[wavelet_index] == 6: Fidelity filter for improved downconversion and anti-aliasing

Lifting steps	$filtershift()$
1. Even, Predict, $s = 12, t_0 = 1817, t_1 = 1817$ i.e. $A[2*n] -= (1817*A[2*n-1]+1817*A[2*n+1]+2048) \gg 12$ 2. Odd, Predict, $s = 12, t_0 = 3616, t_1 = 3616$ i.e. $A[2*n+1] -= (3616*A[2*n]+3616*A[2*n+2]+2048) \gg 12$ 3. Even, Update, $s = 12, t_0 = 217, t_1 = 217$ i.e. $A[2*n] += (217*A[2*n-1]+217*A[2*n+1]+2048) \gg 12$ 4. Odd, Update, $s = 12, t_0 = 6497, t_1 = 6497$ i.e. $A[2*n+1] += (6497*A[2*n]+6497*A[2*n+2]+2048) \gg 12$	1

Table 14: **state**[wavelet_index] == 7: Integer lifting approximation to Daubechies (9,7)

9.5 Removal of IDWT pad values

This section defines the decoding process $idwt_pad_removal(pic, c)$. This is invoked subsequent to $idwt_synthesis$.

Subband data elements have been padded to ensure that the reconstructed data array *pic* has dimensions divisible by $2^{\mathbf{state}[\text{wavelet_depth}]}$.

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

- If $c = Y$, then

$$\begin{aligned} width &= \mathbf{state}[\text{luma_width}] \\ height &= \mathbf{state}[\text{luma_height}] \end{aligned}$$

- else if $c = U$ or $c = V$,

$$\begin{aligned} width &= \mathbf{state}[\text{chroma_width}] \\ height &= \mathbf{state}[\text{chroma_height}] \end{aligned}$$

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

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

are discarded and *pic* is resized to have width *width* and height *height*. [Need a bit somewhere about width and height conventions for 2-d arrays]

10 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 is invoked for each component in a picture, subsequent to the decoding of coefficient data, specified in Section 7, and the Inverse Wavelet Transform (IWT), specified in Section 9.

10.1 Definitions and conventions

Motion compensation uses the motion block data **state**[block_data] and (optionally) the global motion parameters **state**[global_params].

Since $motion_compensate()$ applies to both luma and (potentially subsampled) chroma data, for simplicity a number of local variables are defined. If $c = Y$ then:

$$\begin{aligned} lenX &= \textbf{state}[\text{luma_width}] \\ lenY &= \textbf{state}[\text{luma_height}] \\ xblen &= \textbf{state}[\text{luma_xblen}] \\ yblen &= \textbf{state}[\text{luma_yblen}] \\ xbsep &= \textbf{state}[\text{luma_xbsep}] \\ ybsep &= \textbf{state}[\text{luma_ybsep}] \end{aligned}$$

If $c = U$ or $c = V$, then likewise:

$$\begin{aligned} lenX &= \textbf{state}[\text{chroma_width}] \\ lenY &= \textbf{state}[\text{chroma_height}] \\ xblen &= \textbf{state}[\text{chroma_xblen}] \\ yblen &= \textbf{state}[\text{chroma_yblen}] \\ xbsep &= \textbf{state}[\text{chroma_xbsep}] \\ ybsep &= \textbf{state}[\text{chroma_ybsep}] \end{aligned}$$

Define the offsets $xoffset, yoffset$ by

$$\begin{aligned} xoffset &= (xblen - xbsep)//2 \\ yoffset &= (yblen - ybsep)//2 \end{aligned}$$

Informative: 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. The specification of this section fully supports such an interpretation.

Throughout this Section, the following conventions are used.

- x, y are co-ordinates in the predicted picture component
- u, v are co-ordinates in a potentially upconverted reference picture component

10.2 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 6. 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.

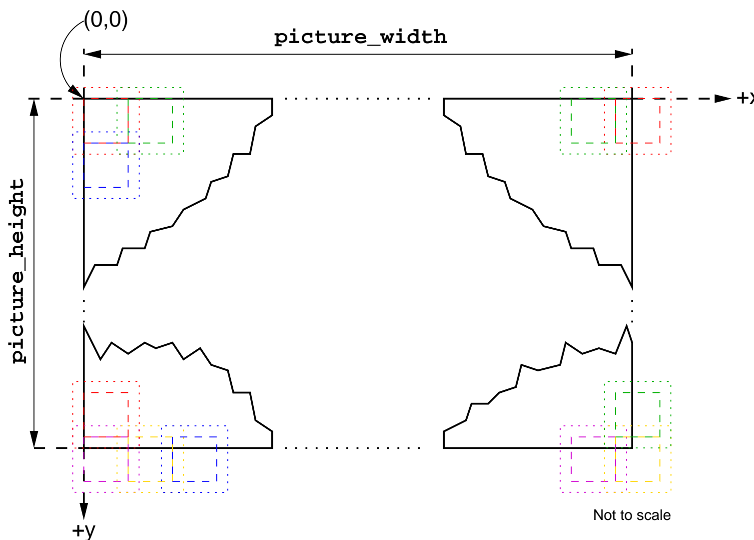

Figure 6: Block coverage of the predicted picture

The overlap between blocks horizontally is $xblen - xbsep$ and vertically is $yblen - ybsep$. 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) 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 ??) 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 picture area defined by $0 \leq x < lenX, 0 \leq y < lenY$ are discarded.

10.3 Overall motion compensation process

The motion compensation process forms an integer prediction $p[y][x]$ for each pixel in the predicted picture component *pic*, and adds it to the component data. This is then clipped to keep it in range. Note that this clipping is *in addition* to clipping performed on the output picture after motion compensation and/or the inverse wavelet transform (Section 8.5).

for $y = 0$ to $lenY - 1$:	
for $x = 0$ to $lenX - 1$:	
$pic[y][x] += pixel_predict(y, x, pic, ref1, ref2, c)$	10.4
$pic[y][x] = \text{clip}(pic[y][x], 0, 2^{\mathbf{state}[\text{video_depth}]} - 1)$	

10.4 Pixel prediction

In order to specify the *pixel_predict*() process, some definitions are required. For block indices (i, j), define the set of elements $B(i, j)$ in the corresponding block by:

$$\begin{aligned} xstart &= \max(i * xbsep - xoffset, 0) \\ ystart &= \max(j * xbsep - xoffset, 0) \\ xstop &= \min(xstart + xblen, lenX) = \min((i+1) * xbsep + xoffset, lenX) \\ ystop &= \min(ystart + yblen, lenY) = \min((j+1) * ybsep + yoffset, lenY) \\ B(i, j) &= \{(x, y) : xstart \leq x < xstop, ystart \leq y < ystop\} \end{aligned}$$

Define the total weight resolution *total_wt_bits* as follows:

$$\begin{aligned} hbits &= \log_2(xblen - xbsep) + 1 = \log_2(xoffset) + 2 \\ vbits &= \log_2(yblen - ybsep) + 1 = \log_2(yoffset) + 2 \\ total_wt_bits &= hbits + vbits + \mathbf{state}[\text{refs_weight_precision}] \end{aligned}$$

This is the number of bits added to pixel values in order to perform OBMC reversibly with integer arithmetic using the spatial specified in Sections 10.5 and the reference weights extracted in parsing the picture prediction header data (Section 5.10.7).

The $pixel_predict(y, x, ref1, ref2, c)$ function forms a prediction by adding together weighted predictions for all blocks containing the pixel (x, y). Weight contributions come both from a spatial matrix and from the weights assigned to references:

$pixel_pred(y, x, pic, ref1, ref2, c)$:	
$p = 0$	
for (i, j) such that $(x, y) \in B(i, j)$:	
$m =$ **state**[block_data][j][i][$mode$]	
if ($m ==$ INTRA):	
$val =$ **state**[block_data][j][i][dc][c]	
$val = val * 2^{\mathbf{state}[\text{refs_weight_precision}]}$	
else if ($m ==$ REF1ONLY):	
$val = block_pred(ref1, 1, i, j, x, y, c)$	
$val = val *$ (**state**[ref1_weight] + **state**[ref2_weight])	
else if ($m ==$ REF2ONLY):	
$val = block_pred(ref2, 2, i, j, x, y, c)$	
$val = val *$ (**state**[ref1_weight] + **state**[ref2_weight])	
else:	
$val1 = block_pred(ref1, 1, i, j, x, y, c)$	
$val1 = val1 *$ **state**[ref1_weight]	
$val2 = block_pred(ref2, 2, i, j, x, y, c)$	
$val2 = va2l *$ **state**[ref2_weight]	
$val = val * spatial_wt(i, j, x, y)$	10.5
$p = p + val$	
$p = (p + 2^{total_wt_bits-1}) \gg total_wt_bits$	
return p	

Informative:

1. Note that the number of bits $total_wt_bits$ used for the OBMC weighting matrix depends upon the block sizes - specifically the block overlaps - selected. A Dirac decoder level (**??**) specifies the maximum block overlaps allowable, and hence the word widths necessary for processing OBMC. If we assume that the picture weights are complementary (i.e. the weights for reference 1 and reference 2 sum to $2^{\mathbf{state}[\text{refs_weight_precision}]}$, then the number of bits required for performing motion compensation calculations is

$$\mathbf{state}[\text{video_depth}] + total_wt_bits + \mathbf{state}[\text{refs_weight_precision}]$$

unsigned bits. 8 bit video data encoded with block overlaps of 4 luminance pixels and the standard picture weights therefore requires 8+3+3+1=15 unsigned bits. The additional bit within a 16 bit word could be used to provide additional reference weighting.

2. The motion compensation process has been presented as double loop: first over all pixels in a given component and second over all blocks of which the pixel is a member. However, if an intermediate buffer is allowed, the loop order can be reversed. In this case one sets a picture buffer, consisting of a component of data (or a strip of component data lines) and add in the weighted predictions for each block. As the blocks overlap, the contributions sum and form a prediction for every pixel. This is the most natural implementation strategy:

$b[\ ][\ ]=0$	
for $j=0$ to **state**[blocks_y] -1:	
for $i=0$ to **state**[blocks_x] -1:	
$m=$ **state**[block_data]$[j][i][mode]$	
for each (x,y) in $B(i,j)$:	
if ($m==$ INTRA):	
$wt=2^{\textbf{state}[\text{refs_weight_precision}]}*spatial_wt(i,j,x,y)$	
$b[y][x]+=$ **state**[block_data]$[j][i][dc][c]*wt$	
else if ($m==$ REF1ONLY):	
$wt=(\textbf{state}[\text{ref1_weight}]+\textbf{state}[\text{ref2_weight}])*spatial_wt(i,j,x,y)$	
$b[y][x]+=block_pred(ref1,1,i,j,x,y,c)*wt$	
else if ($m==$ REF2ONLY):	
$wt=(\textbf{state}[\text{ref1_weight}]+\textbf{state}[\text{ref2_weight}])*spatial_wt(i,j,x,y)$	
$b[y][x]+=block_pred(ref2,2,i,j,x,y,c)*wt$	
else:	
$wt=\textbf{state}[\text{ref1_weight}]*spatial_wt(i,j,x,y)$	
$b[y][x]+=block_pred(ref1,1,i,j,x,y,c)*wt$	
$wt=\textbf{state}[\text{ref2_weight}]*spatial_wt(i,j,x,y)$	
$b[y][x]+=block_pred(ref2,2,i,j,x,y,c)*wt$	
for $y=0$ to $lenY-1$:	
for $x=0$ to $lenX-1$:	
$pic[y][x]+=(b[y][x]+2^{total_wt_bits-1})\gg total_wt_bits$	
$pic[y][x]=\text{clip}(pic[y][x],0,2^{\textbf{state}[\text{video_depth}]}-1)$	

The double multiplication by a spatial and by a reference weight can be avoided by using a set of spatial weighting matrices pre-multiplied by the applicable reference weights according to prediction mode. In the default case where the reference picture weights are one, as is the picture weight precision, this means a double-size spatial matrix for all modes other than REF1AND2.

3. The reference prediction weights used for each prediction mode may appear confusing. It is helpful to think of two cases for using reference picture weighting. The first is interpolative prediction, where the picture being predicted is, for example, a cross-fade and is closely approximated by some mixture of the reference pictures: $P \backsimeq \delta R_1 + (1-\delta)R_2$. Here the weights we'd like to use for each frame prediction add up to 1 (or $2^{\textbf{state}[\text{refs_weight_precision}]}$ for integer weights). The second case is scaling prediction, where the weights we'd like to use for the frame predictions don't add up to 1: for example, a fade to or from black $P \backsimeq \delta_1 R_1$ and $P \backsimeq \delta_2 R_2$. It is not possible to choose weights for each prediction mode which will be optimal both cases. The weighting factors chosen will give work with interpolative prediction (which is more common) but are not perfect for scaling prediction. It would have been possible to create a variety of prediction modes to cover all cases, however the potential savings do not justify the additional complexity.

For interpolative prediction, all data in the current picture will be of commensurate scale to that of the references. In forming the bi-directional prediction, a value $W_1p_1 + W_2p2_2$ is formed, so the prediction has "scale" $W_1 + W_2$. $W_1 + W_2$ is therefore the weighting value used to scale unidirectional prediction, in order to provide predictions of commensurate order. The unity weighting value $2^{\textbf{state}[\text{refs_weight_precision}]}$ is used for DC blocks as this gives the best prediction, and in the interpolative case this equals $W_1 + W_2$ so all predictions are of the same order.

The weighting factors we would like to use for unidirectionally predicted blocks in the scaling case are $2W_1$ and $2W_2$ - the factor 2 takes into account that we're only adding in one prediction value as against two for bidirectional prediction. These factors differ from W_1+W_2, and hence unidirectional prediction is incorrect when there are two references. Note, however, that we can still perform prediction with

the correct scaling values when we only have a single reference. Note also that the value of $W_1 + W_2$ was selected instead of $2^{\mathbf{state}[\text{refs_weight_precision}]}$, which would be equivalent in the interpolative case, as it gives a better approximation when the weights do not sum to $2^{\mathbf{state}[\text{refs_weight_precision}]}$.

10.5 Spatial weighting matrix

This section specifies the process $wt(i, j, x, y)$ for deriving a spatial weighting value for a pixel with coordinates (x, y) in the block with coordinates (i, j). Note that other weights are applied to the prediction as a result of the weights applied to each reference.

The two-dimensional spatial weighting matrix W applies a linear roll-off in both horizontal and vertical directions based on the position of the pixel (x, y) within the block $B(i, j)$. Define $xpos, ypos$ as the relative pixel coordinates from the top-left corner of the block:

$$\begin{aligned} xpos &= x - (i * xbsep - xoffset) \\ ypos &= x - (i * ybsep - yoffset) \end{aligned}$$

Define a horizontal weighting array WH by the recipe:

$max_x = 2(xblen - xbsep)$	
if $(i == 0 \text{and} xpos < xblen//2)$:	
$WH[xpos] = max_x$	
else if $(i == \mathbf{state}[\text{blocks_x}] \text{and} xpos \geq xblen//2)$:	
$WH[xpos] = max_x$	
else:	
$WH[xpos] = \text{clip}(xblen - 2 \lvert xpos - \frac{(xblen-1)}{2} \rvert, max_x)$	

Likewise define WV by

$max_y = 2(yblen - ybsep)$	
if $(j == 0 \text{and} xpos < yblen//2)$:	
$WH[ypos] = max_y$	
else if $(j == \mathbf{state}[\text{blocks_y}] \text{and} ypos \geq yblen//2)$:	
$WH[ypos] = max_y$	
else:	
$WV[ypos] = \text{clip}(yblen - 2 \lvert ypos - \frac{(yblen-1)}{2} \rvert, 0, max_y)$	

The overall spatial weighting matrix W is given by

$$W[ypos][xpos] = WH[xpos][WV[ypos]$$

and this is the value returned.

Note that blocks at the extremities of the block set receive maximum weight around their outward-facing edges. This is to compensate for the lack of blocks making weight contributions on these edges, and ensures that the total contribution for the pixels in the blocks is $2^a lpha$. In section $(i = 0)$, the profile of the matrix for interior blocks is:

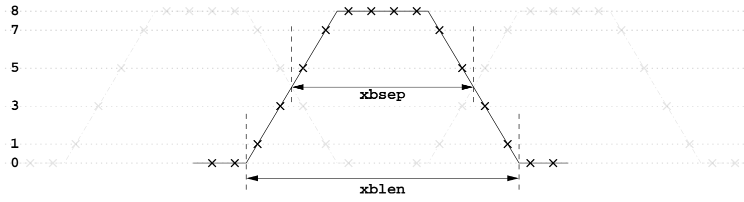

Figure 7: Profile of overlapped-block motion compensation matrix

10.6 Block prediction

This section specifies the operation of the $block_pred(ref, ref_num, i, j, x, y, c)$ process for forming a prediction for a pixel with coordinates (x, y) in component c, belonging to the block with coordinates (i, j).

Case 1: state[block_data]$[j][i][global]$ = **False**. In this case, the block motion vectors are used to form a prediction. Motion vectors for chroma components must be scaled according to the chroma scale factors. If $c = Y$, set

$$mv = \mathbf{state}[\text{block_data}][j][i][ref]$$

whereas if $c = U$ or $c = V$, set

$$\begin{aligned} mv.x &= (\mathbf{state}[\text{block_data}][j][i][ref].x + (chroma_h_factor()//2))//chroma_h_factor() \\ mv.y &= (\mathbf{state}[\text{block_data}][j][i][ref].y + (chroma_v_factor()//2))//chroma_v_factor() \end{aligned}$$

(chroma subsampling factors are as specified in Section 2)

Case 2: state[block_data]$[j][i][global]$ = **True**. In this case, a motion vector is determined from the global motion parameters as per Section 10.7:

$$mv = global_mv(ref, ref_num, x, y, c)$$

In both cases the value

$upconvert(ref, (x \ll \mathbf{state}[\text{mv_precision}]) + mv.x, (y \ll \mathbf{state}[\text{mv_precision}]) + mv.y)$

where *upconvert* is defined in Section 10.8 is returned.

10.7 Global motion vector field generation

This section specifies the operation of the $global_pred(ref, ref_num, x, y, c)$ process for deriving a global motion vector for a pixel at location (x, y), in a component of type c from a reference ref.

10.7.1 Chroma scaling

The global motion parameters are extracted from the state data. If the component is a chroma component, the parameters must be scaled appropriately. Set:

- $\tilde{\mathbf{A}} = \mathbf{state}[\text{global_params}][ref_num].\mathbf{A}$
- $\tilde{\mathbf{b}} = \mathbf{state}[\text{global_params}][ref_num].\mathbf{b}$
- $\tilde{\mathbf{c}} = \mathbf{state}[\text{global_params}][ref_num].\mathbf{c}$

If $c = Y$, set $\mathbf{A} = \tilde{\mathbf{A}}$, $\mathbf{b} = \tilde{\mathbf{b}}$ and $\mathbf{c} = \tilde{\mathbf{c}}$.

If $c = U$ or $c = V$, $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ are defined as follows. Scale $\mathbf{b}$ according to the chroma scale factors:

$$\begin{aligned} b_0 &= \left(\tilde{b}_0 + chroma_h_factor()//2\right) //chroma_h_factor() \\ b_1 &= \left(\tilde{b}_1 + chroma_h_factor()//2\right) //chroma_h_factor() \end{aligned}$$

Scale $\mathbf{A}$, taking into account vertical and horizontal factors:

$$\begin{aligned} A_{0,0} &= \tilde{A}_{0,0} \\ A_{1,1} &= \tilde{A}_{1,1} \\ A_{0,1} &= \left(\tilde{A}_{0,1} * chroma_h_factor() + chroma_v_factor()//2\right) //chroma_v_factor() \\ A_{1,0} &= \left(\tilde{A}_{1,0} * chroma_v_factor() + chroma_h_factor()//2\right) //chroma_h_factor() \end{aligned}$$

and give $\mathbf{c}$ the inverse scaling to $\mathbf{b}$:

$$\begin{aligned} c_0 &= \tilde{c}_0 * chroma_h_factor \\ c_1 &= \tilde{c}_1 * chroma_h_factor \end{aligned}$$

10.7.2 Field generation

Set $\alpha = \mathbf{state}[\text{global_params}][ref_num][ZRS_exp]$ and $\beta = \mathbf{state}[\text{global_params}][ref_num][perspective_exp]$.

Writing $\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$, set $\mathbf{y}$ to be the integer vector defined by:

$$\mathbf{y} = \left(2^{\alpha+\beta} - 2^{\alpha} * \mathbf{c}^T\mathbf{x}\right)\left(2^{\beta} * \mathbf{A}\mathbf{x} + 2^{\alpha+\beta} * \mathbf{b}\right)$$

Set

$$\begin{aligned} mv.x &= y_0 \gg (\alpha + \beta) \\ mv.x &= y_1 \gg (\alpha + \beta) \end{aligned}$$

and return the motion vector mv.

10.8 Upconversion

This section specifies the operation of the $upconvert(ref, u, v)$ function for producing a value from an upconverted picture reference. This allows for sub-pixel precision in motion compensation.

Motion vectors are allowed to extend beyond the edges of the upconverted reference picture component and values lying outside the range of the component are determined by edge extension, using the values:

$$\begin{aligned} cu &= \text{clip}(u, 0, 2^{\mathbf{state}[\text{mv_precision}]} * width(ref)) \\ cv &= \text{clip}(v, 0, 2^{\mathbf{state}[\text{mv_precision}]} * height(ref)) \end{aligned}$$

There are four cases, depending upon the motion vector precision selected.

10.8.1 Pixel-accurate motion vectors

If $\mathbf{state}[\text{mv_precision}] == 0$, no upconversion is actually required and the value $ref[cv][cu]$ is returned.

10.8.2 Half-pixel accurate motion vectors

If **state**[mv_precision] == 1 then the reference picture component ref is upconverted by a factor of 2 in each dimension to create an array $upref$. The value returned is $upref[cv][cu]$.

$upref$ is created in two stages, first upconverting vertically by a factor of 2, then horizontally. Define the interpolation filter h to be the 10-tap symmetric filter with taps as defined in figure 8.

Tap	t_0	t_1	t_2	t_3	t_4
Value	167	-56	25	-11	3

Figure 8: Interpolation filter coefficients

Define an array $ref2$ of height $2 * height(ref)$ and width $width(ref)$ by the recipe, for $0 \leq p < width(ref)$ and $0 \leq q < 2 * height(ref)$:

Case 1. If $q\%2 == 0$, set

$$ref2[q][p] = ref2[q//2][p]$$

Case 2. If $q\%2! = 0$, $ref2[q][p]$ is set by

$ref2[q][p] = \sum_{i=0}^{4} t_i * (ref[\text{clip}((q-1)//2 - i, 0, height(ref)-1)][p] + ref[\text{clip}((q+1)//2 + i, 0, height(ref)-1))][p$	
$ref2[q][p] = (ref2[q][p] + 128) \gg 8$	
$ref2[q][p] = \text{clip}(ref2[q][p], 0, 2^{\textbf{state}[\text{video_depth}]} - 1)$	

The full upconverted array is constructed from $ref2$ in the same way. For $0 \leq p < 2 * width(ref)$ and $0 \leq q < 2 * height(ref)$ we have:

Case 1. If $p\%2 == 0$, set

$$upref[q][p] = ref2[q][p//2]$$

Case 2. If $p\%2! = 0$, $upref[q][p]$ is set by

$upref[q][p] = \sum_{i=0}^{4} t_i * (ref2[q][\text{clip}((p-1)//2 - i, 0, width(ref2)-1)] + ref2[q][\text{clip}((p+1)//2 + i, 0, width(ref2) -$	
$upref[q][p] = (upref[q][p] + 128) \gg 8$	
$upref[q][p] = \text{clip}(upref[q][p], 0, 2^{\textbf{state}[\text{video_depth}]} - 1)$	

Informative: 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.

10.8.3 Quarter- and eighth-pixel accurate motion vectors

If **state**[mv_precision] == 2 or **state**[mv_precision] == 3, upconverted values are derived by linear interpolation from the half-pixel interpolation values $upref$, which is calculated as per Section 10.8.2. Given coordinates (u, v), their half-pixel part is extracted by:

$$\begin{aligned} hu &= u \gg (\textbf{state}[\text{mv_precision}] - 1) \\ hv &= v \gg (\textbf{state}[\text{mv_precision}] - 1) \end{aligned}$$

and their remainder (giving the residual subpixel accuracy) by

$$
\begin{aligned}
ru &= u - (hu \ll (\mathbf{state}[\text{mv_precision}] - 1)) \\
rv &= v - (hv \ll (\mathbf{state}[\text{mv_precision}] - 1))
\end{aligned}
$$

ru and rv satisfy $0 \leq ru, rv < 2^{\mathbf{state}[\text{mv_precision}]-1}$. Then define four weighting values by:

$$
\begin{aligned}
w00 &= (2^{\mathbf{state}[\text{mv_precision}]-1} - rv) * (2^{\mathbf{state}[\text{mv_precision}]-1} - ru) \\
w01 &= (2^{\mathbf{state}[\text{mv_precision}]-1} - rv) * ru \\
w10 &= rv * (2^{\mathbf{state}[\text{mv_precision}]-1} - ru) \\
w11 &= rv * ru
\end{aligned}
$$

and also define the clipped coordinates that we shall use for interpolation by:

$$
\begin{aligned}
cu &= \text{clip}(hu, 0, width(upref) - 1) \\
cu1 &= \text{clip}(hu + 1, 0, width(upref) - 1) \\
cv &= \text{clip}(hv, 0, width(upref) - 1) \\
cv1 &= \text{clip}(hv + 1, 0, width(upref) - 1)
\end{aligned}
$$

The value returned is:

$$
\left(
\begin{array}{l}
w00 * upref[cv][cu] + \\
w01 * upref[cv][cu1] + \\
w10 * upref[cv1][cu] + \\
w11 * upref[cv1][cu1] + 2^{\mathbf{state}[\text{mv_precision}]-2}
\end{array}
\right) \gg (\mathbf{state}[\text{mv_precision}] - 1)
$$

Informative: Note that the remainder values ru, rv are not determined from the clipped half-pixel values cu, cv, $cu1$, and $cv1$. This ensures the remainder values depend only on the motion vector, and hence are constant across each block, and allows a block-wise implementation. If the clipped values had been used, blocks whose reference block straddled the edge of a picture would use different remainders in different parts of the block. See Section 10.9.

10.9 Implementation

[TBC]

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

A Parse diagrams

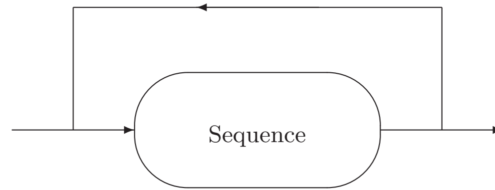

Figure 9: Stream

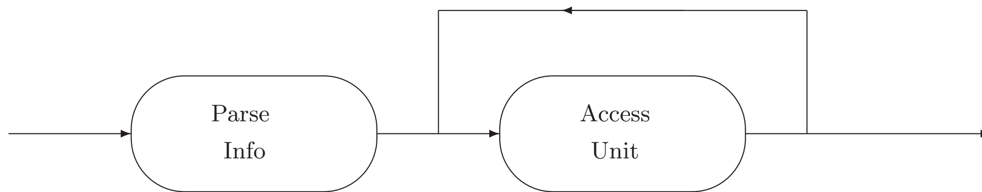

Figure 10: Sequence

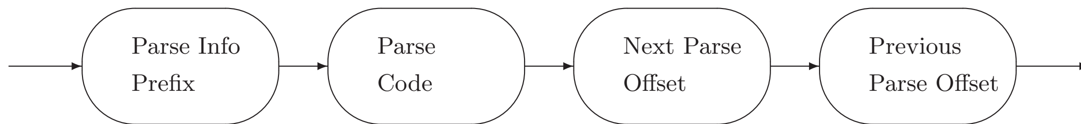

Figure 11: Parse Info

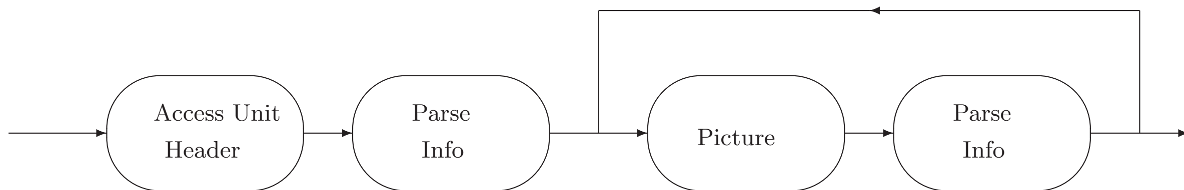

Figure 12: Access Unit

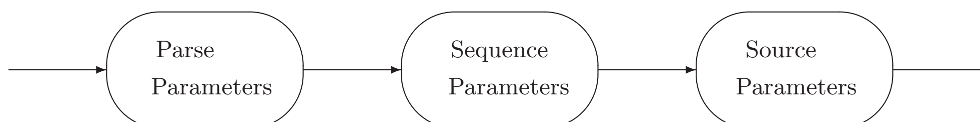

Figure 13: Access Unit header

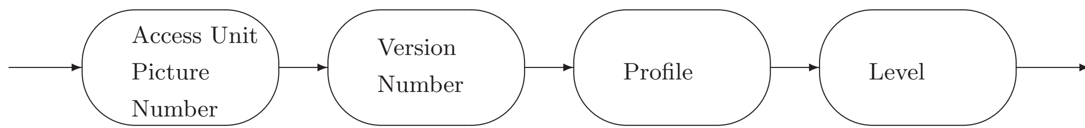

Figure 14: Access Unit Parse Parameters

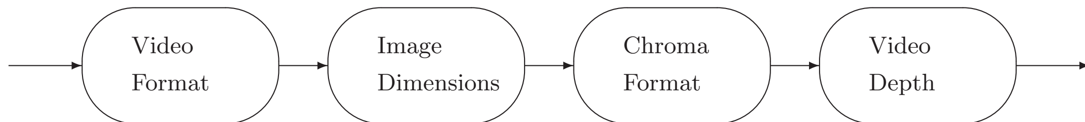

Figure 15: Sequence Parameters

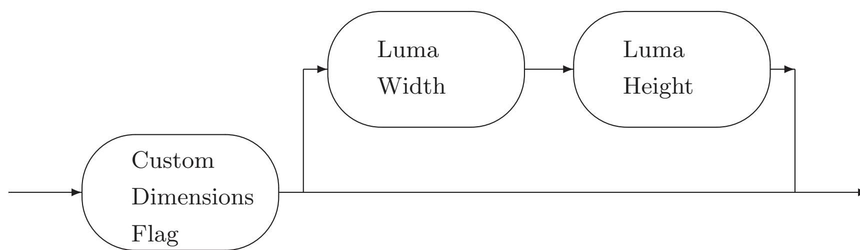

Figure 16: Image dimensions

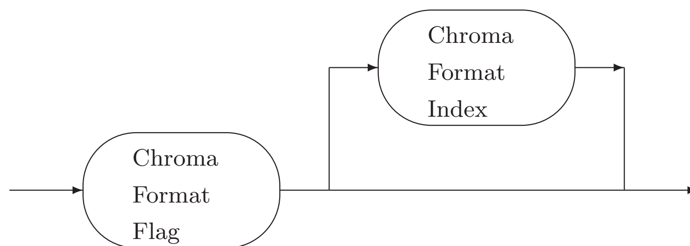

Figure 17: Chroma formats

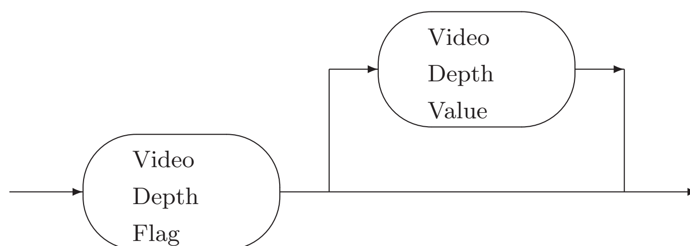

Figure 18: Video Depth

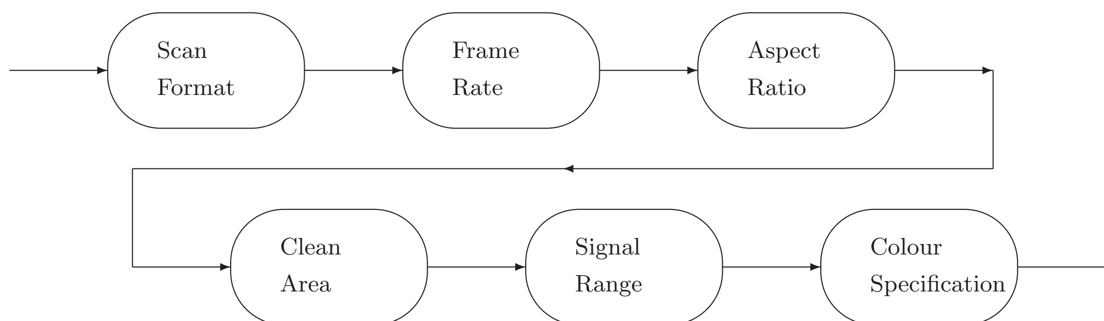

Figure 19: Access Unit Source Parameters

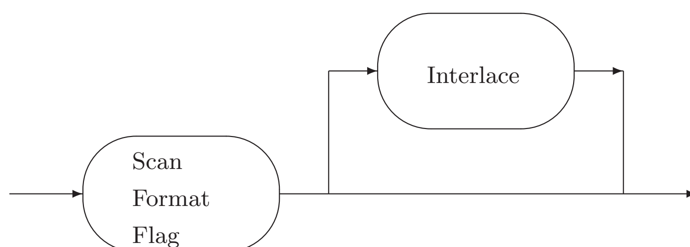

Figure 20: Scan Format

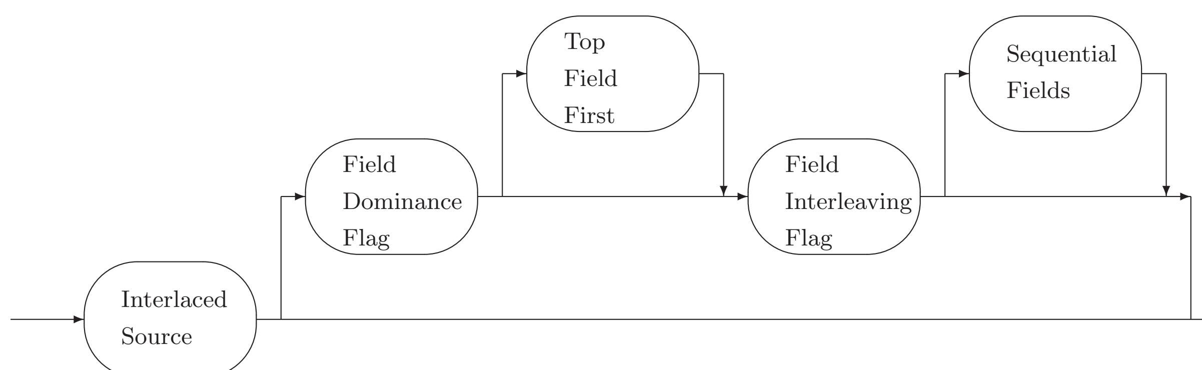

Figure 21: Interlace

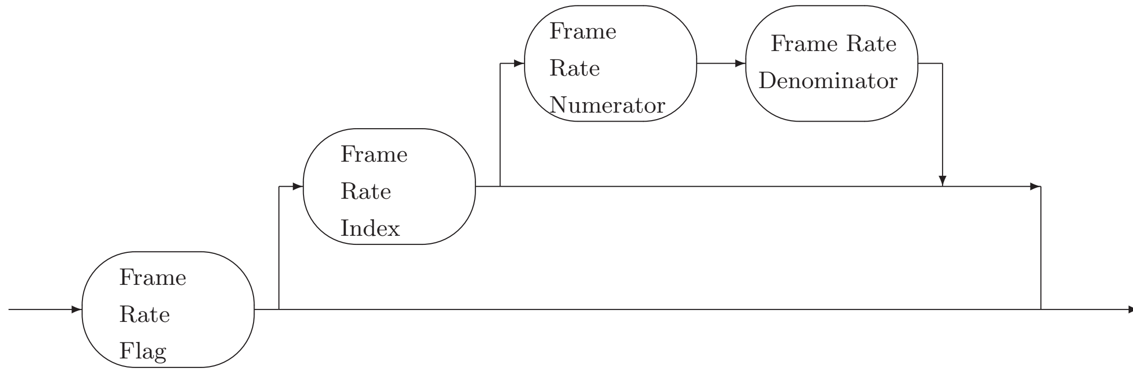

Figure 22: Frame Rate

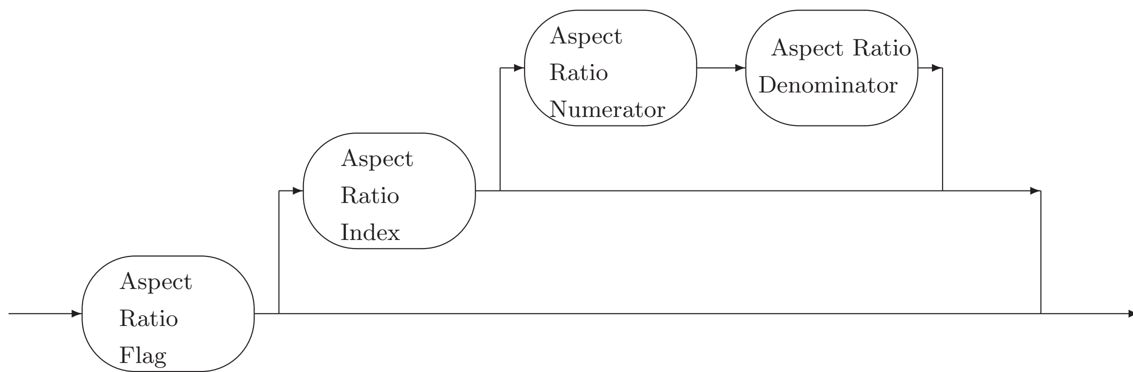

Figure 23: Aspect Ratio

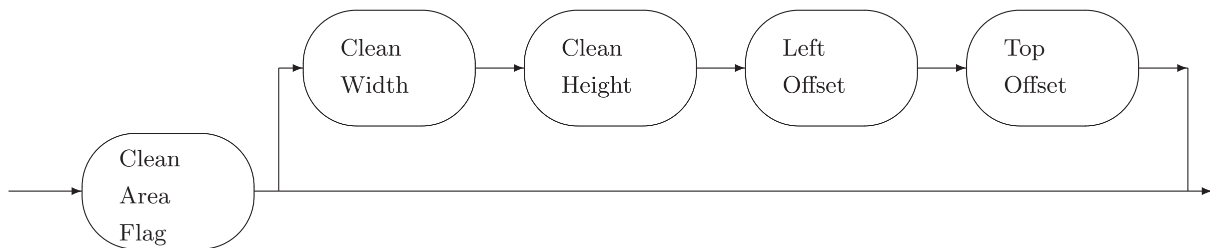

Figure 24: Clean Area

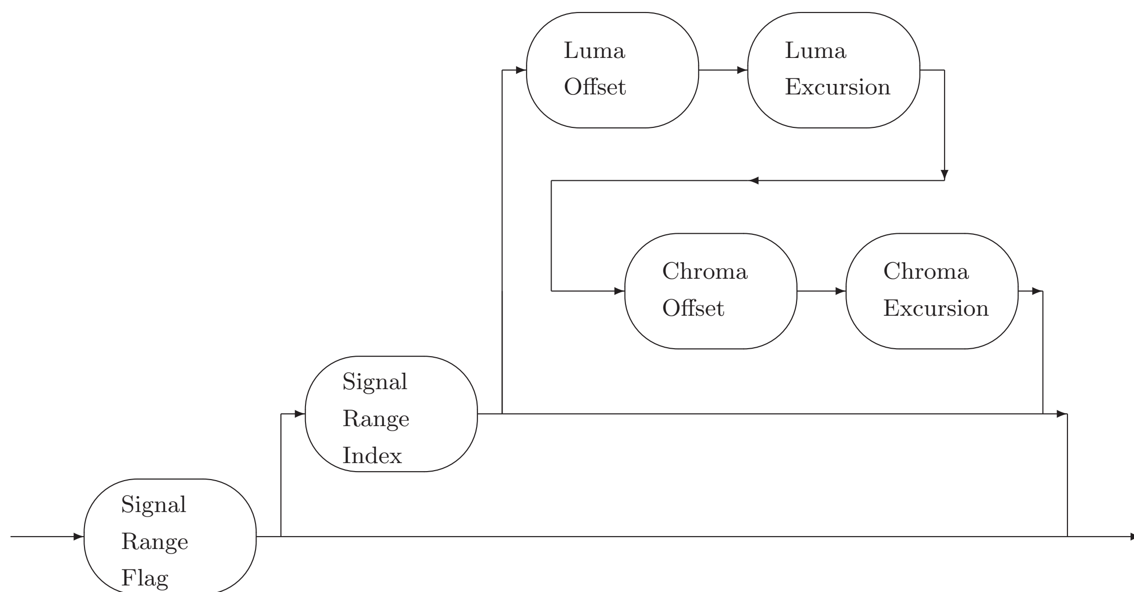

Figure 25: Signal Range

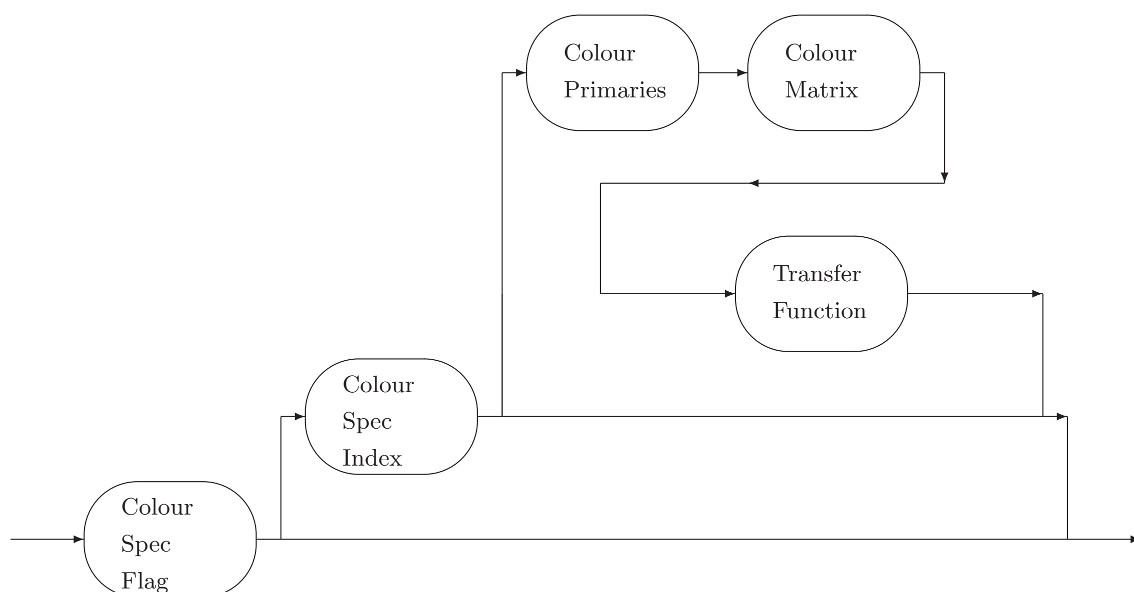

Figure 26: Colour Specification

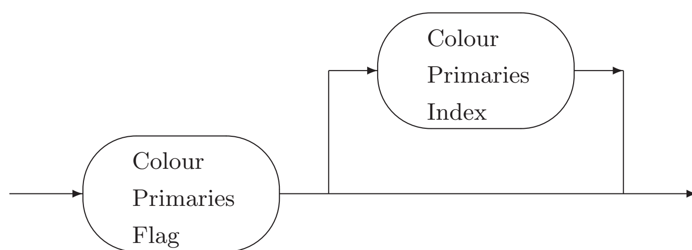

Figure 27: Colour Primaries

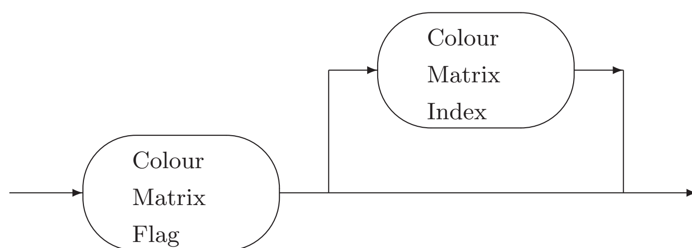

Figure 28: Colour Matrix

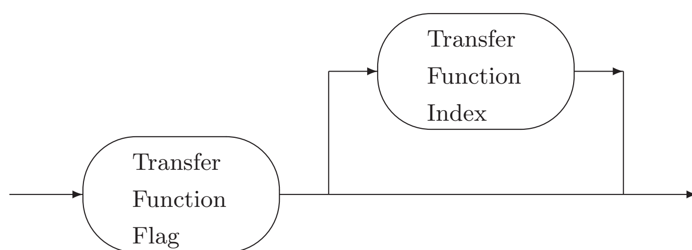

Figure 29: Transfer Function

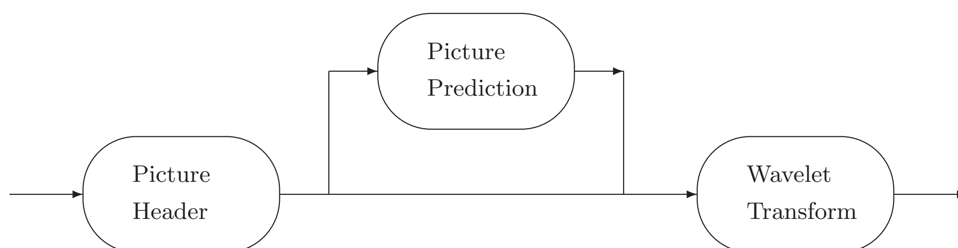

Figure 30: Picture

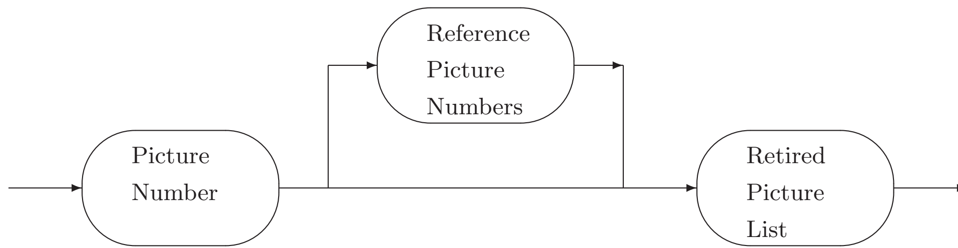

Figure 31: Picture header

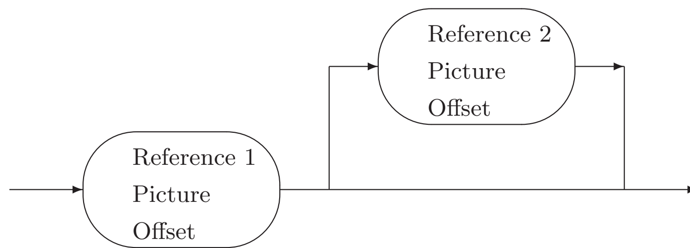

Figure 32: Reference Picture Numbers

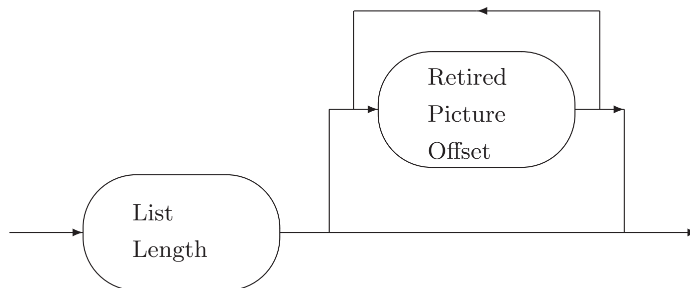

Figure 33: Retired Picture List

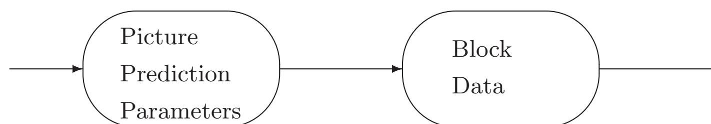

Figure 34: Picture Prediction

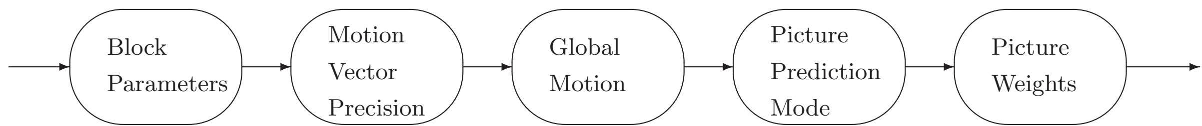

Figure 35: Picture prediction parameters

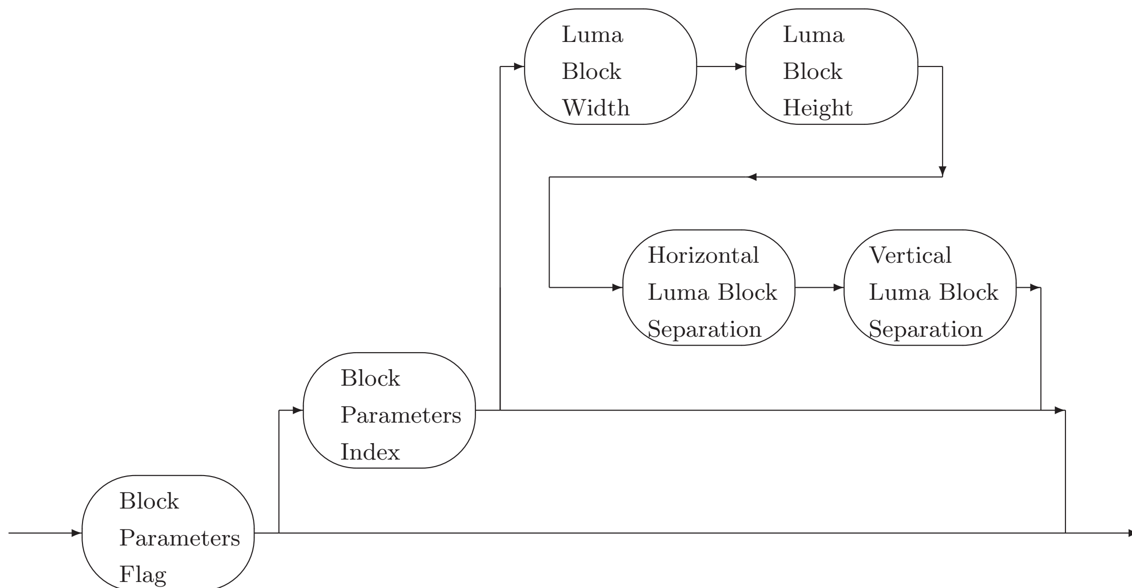

Figure 36: Block Parameters

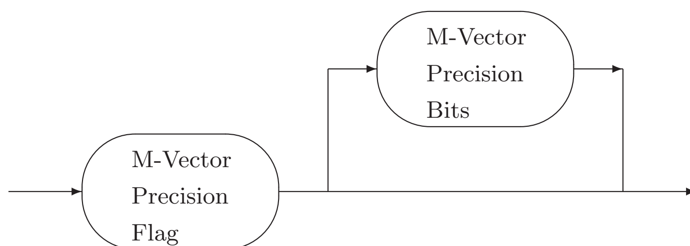

Figure 37: Motion Vector Precision

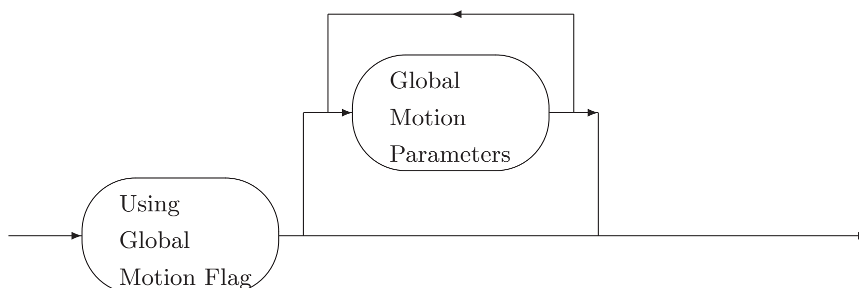

Figure 38: Global Motion

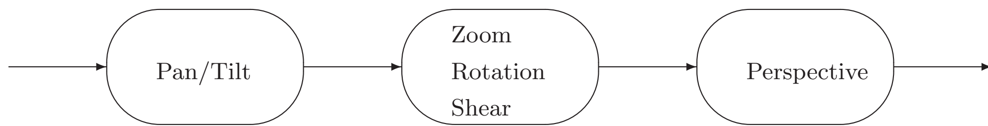

Figure 39: Global Motion Parameters

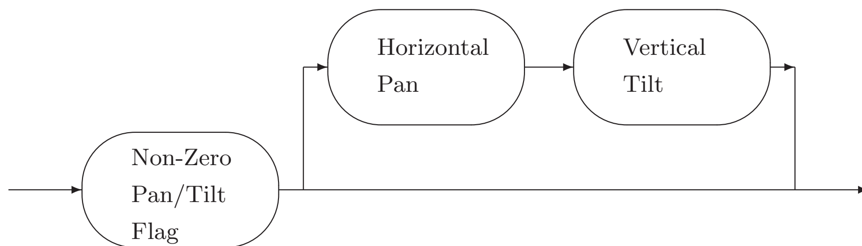

Figure 40: Pan/tilt

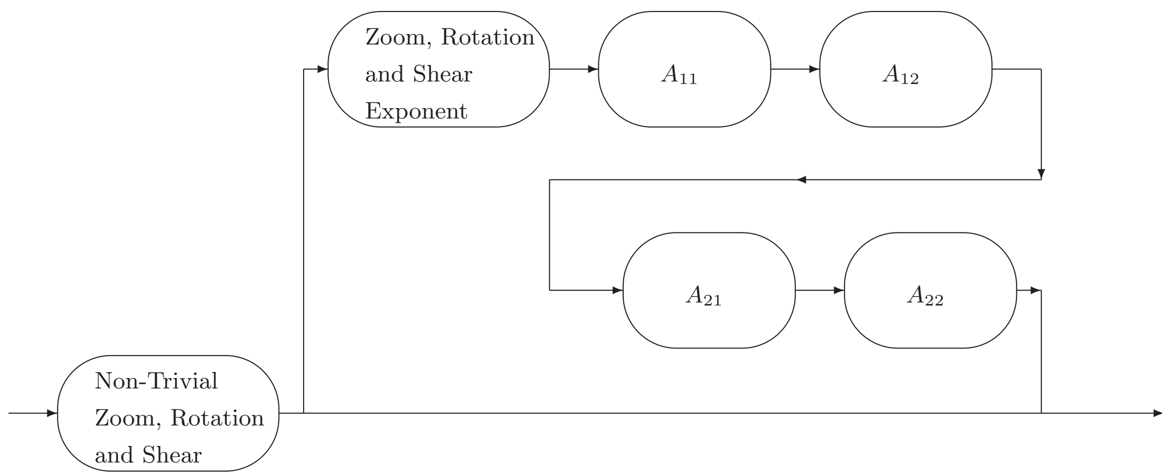

Figure 41: Zoom, Rotation and Shear

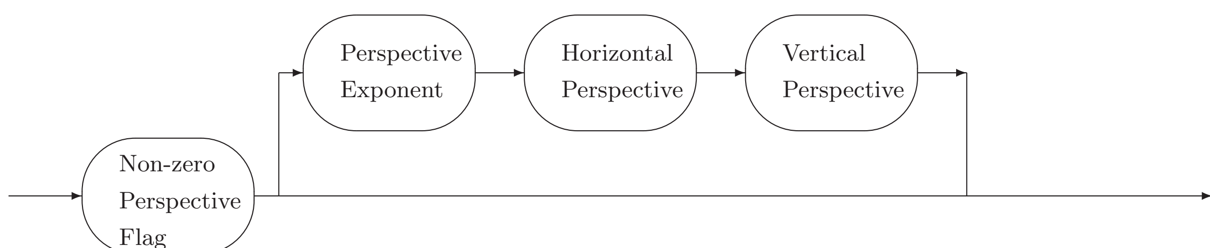

Figure 42: Perpective

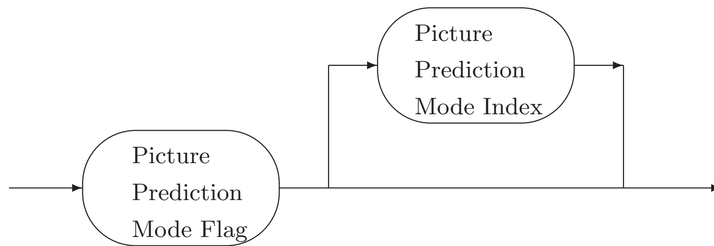

Figure 43: Picture Prediction Mode

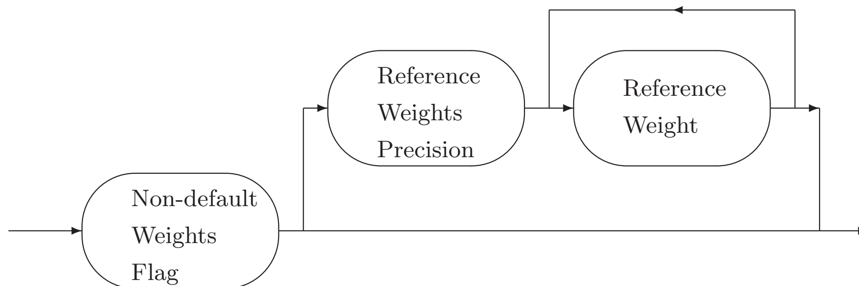

Figure 44: Reference Picture Weights

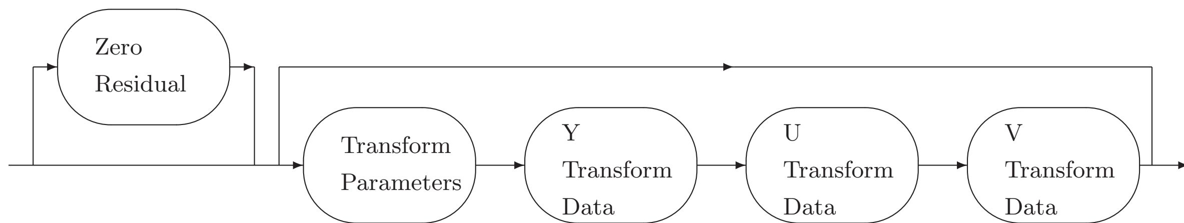

Figure 45: Wavelet Transform

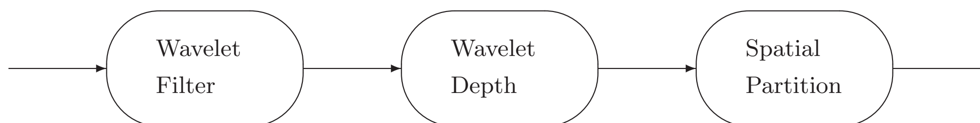

Figure 46: Transform Parameters

Wavelet Index

Non-default Wavelet Flag

Figure 47: Wavelet Filter

Transform Depth

Non-default Wavelet Depth Flag

Figure 48: Wavelet Depth

Number of Codeblocks

Non-default Partition Flag

Codeblock Mode

Spatial Partition Flag

Figure 49: Spatial Partition

B Video systems model and source parameters

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

[Include discussion of YCgCo here. Different matrixing requirements]

B.1 Colour

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

The R, G and B are tristimulus values (e.g. candelas/meter2). 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 ER, EG, EB values, are related to the RGB values by non-linear transfer functions labelled f() and g() in the diagram. Normally, these values also fall in the range [0,1], but in the case of extended gamut, negative values may be allowed also. The transfer function f() is typically performed in the camera and is specified in the Transfer Characteristic part of the Colour Specification. For aesthetic and psychovisual reasons the transfer function g() is not quite the inverse of f(). In fact the combined effect of f() and g() is such that

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

The non-linear ER, EG, EB values are subject to a matrix operation (known as non-constant luminance coding), which transforms them into luma (EY) and chroma (normally ECb and ECr, but sometimes ECg and ECo). EY is normally limited to the range [0,1] and the chroma values to the range [-0.5, 0.5]. This is YUV coding and sometimes the chroma components are subsampled, either horizontally or both horizontally and vertically. UV sampling is specified by the CHROMA_FORMAT value.

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

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

B.2 Frame rate

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

order specified by the TOP_FIELD_FIRST flag.

B.3 Aspect ratios and clean area

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

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

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

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

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

Given two separate sequences, with identical IMAGE_ASPECT_RATIO, if the top left corner and bottom left corners of their clean apertures are coincident when displayed, then the images as a whole should be exactly coincident. This is regardless of the actual pixel dimensions of the images or their clean areas. This allows sequences to be combined together appropriately if they are appropriately scaled.

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

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

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

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 you can define your own ratio. You should be aware that many display devices may ignore your decision and may use different and unsuitable values.

B.4 Signal range

The offset and excursion values should be used to convert the integer-valued decoded luma and chroma data Y, Cb, Cr to intermediate values EY, ECr, and ECb by the recipe

EY, is normally clipped to the range [0,1], and ECr, and ECb to the range [-0.5,0.5]. This effectively clips Y to

[LUMA_OFFSET, LUMA_OFFSET+LUMA_EXCURSION]

and Cb, Cr to

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

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

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

B.5 Colour primaries

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

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

B.6 Colour matrix

Luma and chroma values EY, ECb, ECr should be used to derive ER, EG, EB values by the following equations.

This follows by inverting the equations

In the case of YCgCo coding, ER, EG, EB should be directly computed from the integer Y, Cg and Co values by the following recipe, whereby integer RGB IR, IG, IB values are decoded by

Y-=LUMA_OFFSET

Cg-=CHROMA_OFFSET

Co-=CHROMA_OFFSET

TEMP=Y-(Cg¿¿1)

IG=TEMP+Cg

IB=TEMP-(Co¿¿1)

IR=IB+Co

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

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

B.7 Transfer charactaristics

TV transfer characteristic

Denoting R or G or B as L (light) and ER, EG, EB as E then E=f(L) such is that;

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

Extended Colour Gamut

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

Denoting R or G or B as L (light) and ER, EG, EB as E then E=f(L) such that;

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

Linear

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

B.8 Source parameter presets

Source parameters are signalled using a range of preset indices into the following tables, as specified in Section 5.5.

index	**default_state**[frame_rate_numer]	**default_state**[frame_rate_denom]
1	24000	1001
2	24	1
3	25	1
4	30000	1001
5	30	1
6	50	1
7	60000	1001
8	60	1

Table 15: Available preset frame rate values

index	**default_state**[aspect_ratio_numer]	**default_state**[aspect_ratio_denom]
1 (Square Pixels)	1	1
2 (525-line systems)	10	11
3 (625-line systems)	12	11

Table 16: Available preset aspect ratio values

index	**default_state**[luma_offset]	**default_state**[luma_excursion]
1 (8 Bit Full Range)	0	255
2 (8 Bit Video)	16	235
3 (10 Bit Video)	64	876

Table 17: Luma signal range available presets

index	**default_state**[chroma_offset]	**default_state**[chroma_excursion]
1 (8 Bit Full Range)	128	255
2 (8 Bit Video)	128	224
3 (10 Bit Video)	512	896

Table 18: Chroma signal range available presets

Informative: The only presets available cover a full 8-bit range or 8- or 10-bit SDI video ranges. If other video depths have been selected, custom signal range parameters should be signalled, or the resulting video may have an unintended appearance that affects video quality adversely on a display device.

index	**Description**	**Primaries**	**Matrix**	**Transfer function**
0	Custom, HDTV, PC & Internet	0 - ITU709 & sRGB	0 - HDTV/PC	0 - TV
1	NTSC	1 - SMPTE C	1 - SDTV	0 - TV
2	PAL	2 - EBU Tech 3213	1 - SDTV	0 - TV
3	D-Cinema	3 - CIE XYZ	2 - YCgCo	3 - DCI

Table 19: Colour specification presets

index	**Primaries**
0	ITU709 & sRGB
1	SMPTE C (as used for NTSC)
2	EBU Tech 3213, as used for PAL
3	CIE XYZ

Table 20: Colour primaries presets

index	**Matrix**
0	HDTV, PC & Internet: $K_R = 0.2126$, $K_B = 0.0722$
1	SDTV: $K_R = 0.299$, $K_B = 0.114$
2	Reversible: YCgCo

Table 21: Colour matrix presets

index	**Transfer function**
0	TV
1	Extended Gamut
2	Linear
3	DCI Gamma

Table 22: Transfer function presets

[Must explain all these presets]

C Video format defaults

This Section specifies the default values for decoder state variables that are determined by the value of **default_state**[video_format]. These defaults reduce overhead by allowing a large number of parameters to be set without explicit signalling. The defaults are applied to **default_state** variables immediately on parsing the video_format variable value.

In the case **default_state**[video_format] == 0 (Custom), these tables specify only the initial value of **default_state** parameters: sequence and source parameter values may be overridden within the Access Unit header.

	Video Formats						
	0 – Custom	1 – QSIF	2 – QCIF	3 – SIF	4 – CIF	5 – 4SIF	6 – 4CIF
Sequence parameters							
luma_width	640	176	176	352	352	704	704
luma_height	480	120	144	240	288	480	576
chroma_format	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
video_depth	8	8	8	8	8	8	8
Source parameters							
interlaced	**False**	**False**	**False**	**False**	**False**	**False**	**False**
top_field_first	**True**	**True**	**True**	**True**	**True**	**True**	**True**
sequential_fields	**False**	**False**	**False**	**False**	**False**	**False**	**False**
frame_rate_numer frame_rate_denom	30 1	15000 1001	25 2	15000 1001	25 2	15000 1001	25 2
aspect_ratio_numer aspect_ratio_denom	1 1	10 11	12 11	10 11	12 11	10 11	12 11
clean_width	640	176	176	352	352	704	704
clean_height	480	120	144	240	288	480	576
left_offset	0	0	0	0	0	0	0
top_offset	0	0	0	0	0	0	0
luma_offset	0	0	0	0	0	0	0
luma_excursion	255	255	255	255	255	255	255
chroma_offset	128	128	128	128	128	128	128
chroma_excursion	254	254	254	254	254	254	254
colour_spec	0	1	2	1	2	1	2
colour_primaries	ITU709	SMPTE C	EBU3213	SMPTE C	EBU3213	SMPTE C	EBU3213
K_R K_B	0.2126 0.0722	0.299 0.144	0.299 0.144	0.299 0.144	0.299 0.144	0.299 0.144	0.299 0.144
transfer_fn	TV	TV	TV	TV	TV	TV	TV
Decoding parameters							
wavelet_depth	4	4	4	4	4	4	4
wavelet_index INTRA INTER	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)
luma_xbsep	8	4	4	8	8	8	8
luma_xblen	12	8	8	12	12	12	12
luma_ybsep	8	4	4	8	8	8	8
luma_yblen	12	8	8	12	12	12	12
mv_precision	2	2	2	2	2	2	2
picture_weight_ref1 picture_weight_ref2 picture_weight_bits	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1
codeblocks($h \times v$) INTRA – levels 0-2 – level > 2 INTER – levels 0-1 – level 2 – level > 2	 1,1 4 ,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8

Table 23: Default video parameters for video formats 0–6.

	Video Formats					
	7 – SD480	8 – SD576	9 – HD720	10 – HD1080	11 – 2KCinema	12 – 4KCinema
Sequence parameters						
luma_width	720	720	1280	1920	2048	4096
luma_height	480	576	720	1080	1556	3112
chroma_format	4:2:0	4:2:0	4:2:0	4:2:0	4:4:4	4:4:4
video_depth	8	8	8	8	16	16
Source parameters						
interlaced	**False**	**False**	**False**	**False**	**False**	**False**
top_field_first	**True**	**True**	**True**	**True**	**True**	**True**
sequential_fields	**False**	**False**	**False**	**False**	**False**	
frame_rate_numer frame_rate_denom	24000 1001	25 1	24 1	24 1	24 1	24 1
aspect_ratio_numer aspect_ratio_denom	10 11	12 11	1 1	1 1	1 1	1 1
clean_width	720	720	1280	1920	2048	4096
clean_height	480	576	720	1080	1536	3072
left_offset	0	0	0	0	0	0
top_offset	0	0	0	0	0	0
luma_offset	16	16	16	16	0	0
luma_excursion	235	235	235	235	65535	65535
chroma_offset	128	128	128	128	32768	32768
chroma_excursion	244	244	244	244	65534	65534
colour_spec	1	2	0	0	3	3
colour_primaries	SMPTE C	EBU3213	ITU709	ITU709	Not defined	Not defined
K_R K_B	0.299 0.144	0.299 0.144	0.2126 0.0722	0.2126 0.0722	0.25 0.25	0.25 0.25
transfer_fn	TV	TV	TV	TV	Linear	Linear
Decoding parameters						
wavelet_depth	4	4	4	4	4	4
wavelet_index INTRA INTER	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 0 – DD(9,3) 1 – (5,3)	 6 – Fidelity 1 – DD(9,3)	 6 – Fidelity 0 – DD(9,3)
luma_xbsep	8	8	12	16	16	16
luma_xblen	12	12	16	24	24	24
luma_xbsep	8	8	12	16	16	16
luma_xblen	12	12	16	24	24	24
mv_precision	2	2	2	2	2	2
picture_weight_ref11 picture_weight_ref21 picture_weight_bits1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1
codeblocks($h \times v$) INTRA – levels 0-2 – level > 2 INTER – levels 0-1 – level 2 – level > 2	 1,1 4 ,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8	 1,1 4,3 1,1 8,6 12,8

Table 24: Default video parameters for video formats 7–12

D Profiles and levels

Index

aspect_ratio_denom, 32, 95, 98, 99
aspect_ratio_numer, 32, 95, 98, 99
au_picture_number, 28

bits_left, 18, 19
block_data, 43–50, 71, 74, 75, 77
blocks_x, 44, 75, 76
blocks_y, 44, 75, 76

chroma_excursion, 33, 96, 98, 99
chroma_format, 98, 99
chroma_format_index, 29, 30
chroma_height, 30, 40, 61, 70, 71
chroma_offset, 33, 96, 98, 99
chroma_width, 30, 40, 61, 70, 71
chroma_xblen, 36, 37, 71
chroma_xbsep, 36, 37, 71
chroma_yblen, 36, 37, 71
chroma_ybsep, 36, 37, 71
clean_height, 32, 33, 98, 99
clean_width, 32, 33, 98, 99
code, 18–20
codeblock_mode, 41, 42, 56
codeblocks, 41, 55, 56, 98, 99
coefficient_count, 55, 60
coefficient_reset, 55, 60
colour_primaries, 98, 99
colour_spec, 98, 99
component_height, 40, 52, 53
component_width, 40, 52, 53
contexts, 18–21, 44, 60
current_byte, 15
current_picture, 61–63

frame_rate_denom, 32, 95, 98, 99
frame_rate_numer, 32, 95, 98, 99

global_params, 37, 71, 77, 78

high, 18–20

interlaced, 31, 98, 99

left_offset, 32, 33, 98, 99
level, 28
low, 18–20
luma_excursion, 33, 95, 98, 99
luma_height, 29, 30, 33, 40, 44, 61, 70, 71, 98, 99
luma_offset, 33, 95, 98, 99
luma_width, 29, 30, 33, 40, 44, 61, 70, 71, 98, 99
luma_xblen, 36, 37, 71, 98, 99
luma_xbsep, 36, 37, 44, 71, 98, 99
luma_yblen, 36, 37, 71, 98
luma_ybsep, 36, 37, 44, 71, 98

mv_precision, 37, 77–80, 98, 99

next_bit, 15
next_parse_offset, 25
num_refs, 46

parse_code, 25, 26
parse_info_prefix, 25
picture_number, 35, 61, 62
picture_prediction_mode, 39
picture_weight_bits, 98, 99
picture_weight_ref1, 98, 99
picture_weight_ref2, 98, 99
previous_parse_offset, 25
profile, 28

ref1_picture_number, 35, 62
ref1_weight, 39, 74, 75
ref2_picture_number, 35, 62
ref2_weight, 39, 74, 75
ref_buffer, 61–63
refs_weight_precision, 39, 73–76
retired_picture_list, 35, 63

sb_split, 45, 48
sequential_fields, 31, 32, 98, 99
superblocks_x, 44, 45
superblocks_y, 44, 45

top_field_first, 31, 98, 99
top_offset, 32, 33, 98, 99
transfer_fn, 98, 99
transform_depth, 52–54

u_transform, 39, 40, 62
using_global, 37, 45, 46

v_transform, 39, 40, 62
version_major, 28
version_minor, 28
video_depth, 30, 33, 49, 50, 63, 73–75, 79, 98, 99
video_format, 29, 97

wavelet_depth, 41, 64, 70, 98, 99
wavelet_index, 40, 41, 65, 67–69, 98, 99

y_transform, 39, 40, 62

zero_residual, 40, 62