SMPTE ST 2042-1:2022
Revision of SMPTE ST 2042-1:2017

SMPTE STANDARD

VC-2 Video Compression

Table of Contents

Approved April 15, 2022

Foreword

SMPTE (the Society of Motion Picture and Television Engineers) is an internationally recognized standards developing organization. Headquartered and incorporated in the United States of America, SMPTE has members in over 80 countries on six continents. SMPTE's Engineering Documents, including Standards, Recommended Practices, and Engineering Guidelines, are prepared by SMPTE's Technology Committees. Participation in these Committees is open to all with a bona fide interest in their work. SMPTE cooperates closely with other standards-developing organizations, including ISO, IEC and ITU.

SMPTE Engineering Documents are drafted in accordance with the rules given in its Standards Operations Manual. This SMPTE Engineering Document was prepared by Technology Committee 10E Essence.

Intellectual Property

At the time of publication, no notice had been received by SMPTE claiming patent rights essential to the implementation of this Standard. However, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. SMPTE shall not be held responsible for identifying any or all such patent rights.

Introduction

This clause is entirely informative and does not form an integral part of this document.

The VC-2 standard specifies the compressed stream syntax and reference decoder operations for a video compression system. VC-2 is an intra-frame video compression system aimed at professional applications that provides efficient coding at many resolutions including various flavors of CIF, SDTV and HDTV. VC-2 utilizes wavelet transforms that decompose the video signal into frequency bands. The codec is designed to be simple and flexible, yet able to operate across a wide range of resolutions and application domains.

The system provides the following capabilities:

- **Multi-resolution transforms**. Data is encoded using the wavelet transform and packed into the bitstream subband by subband. High compression ratios result in a gradual loss of resolution. Lower resolution output pictures can be obtained by extracting only the lower resolution data.
- **Frame and field coding**. Both frames and fields can be individually coded.
- **CBR and VBR operation**. VC-2 permits both constant bit rate and variable bit rate operation. For low delay pictures, the bit rate will be constant for each area (VC-2 slice) in a picture to ensure constant latency.
- **Variable bit depths**. 8-, 10-, 12- and 16-bit formats and beyond are supported.
- **Multiple color difference sampling formats**. 444, 422 and 420 video are all supported.
- **Lossless and RGB coding**. A common toolset is used for both lossy and lossless coding. RGB coding is supported either via the YC_GC_O integer color transform for maximum compression efficiency, or by directly compressing RGB signals.
- **Wavelet filters**. A range of wavelet filters can be used to trade off performance against complexity. The Daubechies (9,7) filter is supported for compatibility with JPEG2000. A Fidelity filter is provided for improved resolution scalability.

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

- **Low Delay Syntax**. The syntax of the picture encoding is designed to minimize delay both for encoding and decoding. This can be as low as a few lines of input or output video.

- **Multiple Profiles**. VC-2 provides multiple profiles to address the specific requirements of particular applications. Different profiles include or omit particular coding tools in order to best match the requirements of their intended applications. The Low Delay profile includes DC prediction which provides superior compression at higher compression ratios. The High Quality profile provides lighter compression and supports variable bit rate and lossless coding.

1 Scope

This standard defines the VC-2 video compression system through the stream syntax, entropy coding, coefficient unpacking process and picture decoding process. The decoder operations are defined by means of a mixture of pseudo-code and mathematical operations.

VC-2 is an intra-frame video codec that uses wavelet transforms together with entropy coding that can be readily implemented in hardware or software at very high bit rates. Additional standards and recommended practices can define specific constraints on the encoding for particular applications.

2 Conformance Notation

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

All text in this document is, by default, normative, except: the Introduction, any clause explicitly labeled as "Informative" or individual paragraphs that start with "NOTE”.

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

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

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

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

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

Unless otherwise specified the order of precedence of the types of normative information in this document shall be as follows. Normative prose shall be the authoritative definition. Tables shall be next, followed by formal languages, then figures, and then any other language forms.

3 Normative References

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

Recommendation ITU-R BT.601-7 (03/2011), Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios

Recommendation ITU-R BT.709-6 (06/2015), Parameter Values for the HDTV Standards for Production and International Programme Exchange

Recommendation ITU-R BT.1361 (02/1998), Worldwide Unified Colorimetry and Related Characteristics of Future Television and Imaging Systems

Recommendation ITU-R BT.2020-2 (10/2015), Parameter Values for Ultra-high Definition Television Systems for Production and International Programme Exchange

Recommendation ITU-R BT.2100-2 (07/2018), Image Parameter Values for High Dynamic Range Television for Use in Production and International Programme Exchange

Recommendation ITU-T H.264 (06/2019) | ISO/IEC 14496-10:2008, Advanced Video Coding (AVC)

SMPTE ST 428-1:2019, D-Cinema Distribution Master (DCDM) — Image Characteristics

SMPTE ST 2036-1:2014, Ultra High Definition Television — Image Parameter Values for Program Production

4 Definition of Acronyms and Terms

This clause defines the acronyms and terms used in the VC-2 specification.

4.1 Acronyms

4CIF: Four times CIF

4SIF: Four times SIF

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

CIF: Common Image Format a.k.a. SIF 625

CSF: Contrast Sensitivity Function

DC: Direct Current

DWT: Discrete Wavelet Transform

FIR: Finite Impulse Response

HD(TV): High Definition (Television)

HH: High-High (of a subband)

HL: High-Low (of a subband)

IDWT: Inverse Discrete Wavelet Transform

ITU: International Telecommunications Union

LH: Low-High (of a subband)

LL: Low-Low (of a subband)

LS(B): Least Significant (Bit)

LUT: Look Up Table

MS(B): Most Significant (Bit)

NTSC: National Television Systems Committee

QCIF: Quarter Common Image Format

QSIF: Quarter Source Image Format

SD(TV): Standard Definition (Television)

SIF: Source Input Format

VC: Video Codec

VLC: Variable Length Code

4.2 Terms

4.2.1
AC (sub)band

signal band that is not the DC subband

4.2.2
codec

coder and decoder

4.2.3
color difference

weighted difference between an RGB component of color video and the luma, either gamma-corrected or non-gamma-corrected

Note 1 to entry: In some cases, the term ”difference” has been truncated to ”diff” to save space (e.g., “color_difference” to ”color_diff”).

4.2.4
DC prediction

prediction of coefficients within the DC subband from neighboring coefficients

4.2.5
DC (sub)band

signal band that represents data composed from the lowest frequency band of a wavelet transform (0-LL)

4.2.6
entropy coding

mathematical processing that is intended to reduce the number of bits needed to encode data in a lossless manner

4.2.7
exp-Golomb

exponential-Golomb

Note 1 to entry: Exponential-Golomb coding is a form of variable-length entropy coding. The VC-2 specification uses an interleaved variant of exp-Golomb coding, see A.4 for more information.

4.2.8
inverse quantization

mapping a quantized value to a representative value for the subrange indicated by the quantized value

4.2.9
lifting

technique for implementing a DWT in a computationally efficient and reversible manner

Note 1 to entry: See Bibliography item “Ripples in Mathematics”, chapter 3, for more information.

4.2.10
low delay

VC-2 mode that can be used to compress video with a delay of less than one frame duration

4.2.11
luma

weighted sum of RGB components of color video, either gamma-corrected or non-gamma-corrected

Note 1 to entry: This term is used to prevent confusion with the term ”luminance” that is created only from linear light levels as used in color science.

4.2.12
parsing

process by which text strings and numerical values within data are recognized and used to provide syntactic meaning

4.2.13
picture

single frame or field of video

4.2.14
quantization

division of the range of coefficient values into a number of subranges, each of which is represented by a quantized value

4.2.15
raster scan

left-to-right, then top-to-bottom scan of a two-dimensional array of samples

4.2.16
subband

signal band representing data composed from a single frequency band of a wavelet transform

5 VC-2 Conventions

5.1 General

This clause defines the VC-2 conventions including the naming conventions, the decoder state representation, number formats, data types, arithmetic operations, variables, logical operations, order of operator precedence and pseudocode conventions.

5.2 Naming Conventions

Syntax names are expressed as single text strings in a monotype font (such as Courier) with any word spacing using the underscore character. For all syntax names:

`plain_text` is used for variables,

`bold text` is used for keywords and,

`italicized_text` is used for labels.

5.3 State Representation

This standard uses a state model to express parsing and decoding operations. The state of the decoder/parser shall be stored in the variable `state`. Individual elements of the decoder state (state variables) can be accessed by means of labels, e.g., `state[`*`a_specific_label`*`]`, (i.e., `state` is a map, as defined in 5.4.2).

The decoder state shall be globally accessible within the decoder. Other variables, not declared as inputs to a process, shall be local to that process.

The parsing and decoding operations are defined in terms of modifying the decoder state. The state variables need not directly correspond to elements of the stream, but can be calculated from them taking into account the decoder state as a whole. For example, a state variable value can be differentially encoded with respect to another value, with the difference, not the variable itself, encoded in the stream.

The VC-2 stream syntax structure is illustrated with informative parse diagrams that complement the normative stream syntax definitions. The parsing process is defined by means of pseudocode and/or mathematical formulae.

5.4 Number Formats

Numbers without a prefix shall be interpreted as decimal numbers.

Numbers with the prefix ‘0b’ indicate that the following value shall be interpreted as a binary natural number (non-negative integer).

Example: The value 0b1110100 is equal to the decimal value 116.

Numbers with the prefix ‘0x’ indicate the following value shall be interpreted as a hexadecimal (base 16) natural number.

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

5.5 Data Types

5.5.1 Elementary Data Types

Three basic types are used in the pseudocode:

- **Boolean** – A Boolean variable shall have only two possible values: **True** and **False**.
- **Integer** – A positive or negative whole number or zero.
- **Label** – A unique immutable value used in control structures and to access maps (see 5.5.2). Labels are not given values in this standard. The value assigned to each label is implementation dependent but shall be unique within the implementation.

At various places in this specification informative notes are included to indicate the type or range of variables.

5.5.2 Compound Data Types

There shall be two compound data types:

- **Map** – A collection (set) of data types whose elements are accessed by their corresponding label. For example, `p[Y], p[C1], p[C2]` might be the values of the different video components of a pixel.
- **Array** – A collection of data types accessed by a non-negative integer index. This compound data type is typically used to represent an array of variables. Elements of a 1-dimensional array `a` are accessed by `a[n]` for `n` in the range 0 to `length(a) - 1`.

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

Elements are added to a map or array by assignment using the appropriate index (label or integer). For example, `a[7]=2`, assigns the value 2 to element 7 of the array `a`.

5.6 Functions and Operators

5.6.1 General

5.6 defines the functions and operators used in the pseudocode in this specification. Functions and operators are similar, though functions use the syntax, function(argument1, argument2,…), whereas operators are simply placed before or between operands, e.g., a + b. The difference is purely syntactic and corresponds with conventional mathematical notation.

5.6.2 Assignment

The assignment operation (=) applies to all variable types. After performing the assignment operation:

```
a=b
```

the value of `a` shall become equal to that of `b`, and the value of `b` shall remain unchanged.

For a map, this means the elements of `a[]` shall match in number, label and value every element of `b[]`. For an array, this means the elements of `a[]` shall match in number, index, and value, every element of `b[]`.

5.6.3 Boolean Functions and Operators

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

not (`not a`) shall return **True** for a Boolean value `a` if and only if `a` is **False**

and `(a and b)` shall return **True** if and only if `a` and `b` are both **True**. Operator "and" is used in pseudocode conditions to denote the logical AND between Boolean values, for example:

```
if((condition1 == True) and (condition2 == True)):
    ... etc.
```

or `(a or b)` shall return **True** if either `a` or `b` are **True**, else it returns **False.** Operator "or" is used in pseudocode conditions to denote the logical OR between Boolean values, for example:

```
if((condition1 == True) or (condition2 == True)):
 ... etc.
```

5.6.4 Integer Functions and Operators

The following functions and operators are defined for one or more numerical arguments.

Absolute value	`|a| = a` if `(a >= 0)` else `|a| = -a`
Sign	`sign(a) = 1` if `a > 0`, `0` if `a == 0`, and `-1` if `a < 0`
Addition	The sum of `a` and `b` shall be represented by `a + b`
Subtraction	`a` minus `b` shall be represented by `a - b`.
Multiplication	`a` multiplied by `b` shall be represented by `a * b`.
Integer division	Integer division is defined for integer values `a` and `b`, `b > 0` where: `n = a // b` shall be defined to be the largest number such that `n * b ≤ a` i.e., numbers are rounded towards −infinity. NOTE 1 This differs from C/C++ convention of rounding towards 0.
Remainder	For integers `a, b`, with `b > 0`, the remainder `a % b` shall be defined by: `a % b = a - (a // b) * b` where `a % b` shall always satisfy `0 ≤ (a % b) < b`
Exponentiation	For integer values `a, b > 0`, exponentiation shall be denoted by `a ** b`. and is equal to `a * a * … * a * a` (`b` times). `a ** 0` is defined to be 1 for all `a`.
Maximum	`max(a,b)` shall return the largest value of `a` and `b`
Minimum	`min(a,b)` shall return the smallest of values `a` and `b`

Clip — `clip(a,b,t)` shall clip the value a to the range defined by `b` and `t` such that: `clip(a,b,t) = min(max(a,b),t)`

Shift down (>>) — For integers `a`, `b`, with `b >= 0`, `a >> b` shall be defined as `a // 2^b`.

Shift up (<<) — For integers `a`, `b`, with `b >= 0`, `a << b` shall be defined as `a * 2^b`.

Integer logarithm — `m = intlog2(n)`, for `n > 1`, `m` shall be the integer such that $2^{m-1} < n \leq 2^m$ e.g., `intlog2(25) = intlog2(32) = 5`.

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

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

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

& — Logical *and* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g., 13&6 is 0b1101 & 0b110, which equals 0b100, or 4.

| — Logical *or* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g., 13 | 6 is 0b1101 | 0b110, which equals 0b1111, or 15.

^ — Logical *xor* shall be applied between the corresponding bits in the 2's complement binary representation of two numbers, e.g., 13 ^ 6 is 0b1101 ^ 0b110, which equals 0b1011, or 11.

&= — `a &= b` is equivalent to `a = (a & b)`.

^= — `a ^= b` is equivalent to `a = (a ^ b)`.

|= — `a |= b` is equivalent to `a = (a | b)`.

NOTE 2 Bitwise-not is not defined for integers to avoid ambiguity concerning leading zeroes.

The following logical operators are defined for integer arguments:

== 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

<= Less than or equal to

> Greater than

>= Greater than or equal to

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

The following combined assignment operators are defined for integer arguments:

Operators +, -, *, //, **, %, >>, <<, &, |, ^ combined with the assignment operator (as for the Boolean operators &, |, and ^ above)

For example:

+= `a += b` is equivalent to `a = (a + b)`

−= `a -= b` is equivalent to `a = (a - b)`

5.6.5 Array and Map Functions and Operators

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

Array creation	Arrays shall be created by the `new_array` function. A 1-dimensional array of length `n` and undefined initial contents shall be created by `new_array(n)`. A 2-dimensional array with width `w` and height `h` shall be created by `new_array(h, w)`.
Map creation	An empty map shall be created using the syntax `{}`.
Indexing	For an array or map `a`, `a[index]` shall return an element of `a`. If `a` is a map the index shall be a label, else if `a` is an array the index shall be an integer.
Insertion	`a[index] = b` shall insert a copy of `b` into set an if the element does not already exist.
Length	for a one dimensional array `a`, `length(a)` shall return the number of elements in the array.
Width	for a two dimensional array `a`, `width(a)` shall return the width the array. The width shall be the number of scalar elements corresponding to the right most array index.
Height	for a two dimensional array `a`, `height(a)` shall return the height the array. The height shall be the number of one dimensional arrays in the two dimensional array and the height dimension shall correspond to the left most array index.

5.6.6 Precedence and Associativity of Operators

To avoid any confusion over the order of operator precedence, every equation makes extensive use of the expression operators “(“ and “)”. All operations shall recursively execute the innermost expression(s) first until the calculation has been completed. In cases where the expression operators do not make clear the order of precedence, the order of operator precedence and the associativity of each operator shall be as defined in Table 1 with the topmost operator having greatest precedence.

Table 1 — Operator precedence and associativity.

Operator Precedence	Associativity
`()`	
`**`	right to left
`+x -x` (unary operators)	right to left
`* // %`	left to right
`+ -`	left to right
`<< >>`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`< <= > >= == !=`	left to right
`not`	right to left
`and`	left to right
`or`	left to right
`= += -= *= //= &= ^= |= <<= >>=`	right to left

5.7 Pseudocode

5.7.1 General

Most of the normative specification is defined by means of pseudocode. The syntax is intended to be both precise and descriptive. The pseudocode is not intended to form the basis for the implementation of a VC-2 decoder.

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

NOTE Care needs to be taken when selecting integer sizes. Values can grow in size substantially during processing in a manner which is strongly dependent on the pictures being processed. Values for ‘typical’ pictures might fit into integers several bits smaller than the worst-case and near-worst case. Pictures which yield large values might appear visually similar to more typical examples and not otherwise represent a challenging case.

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

Pseudocode in this standard is sometimes accompanied by a figure illustrating the content of the code.

5.7.2 Processes and Functions

Decoding and parsing operations are specified by means of processes—a series of operations acting on input data and the state variables. A process can also be a function, which means it returns a value, but it need not do so.

A process taking arguments `in1` and `in2` looks like:

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

While a function process looks like:

```
bar(in1, in2):
    op1(in1)
    op2(in2)
    ...
    return out1
```

5.7.3 Variables

Primitive types (numbers, Booleans and labels) are passed *by value* whilst compound types (arrays and maps) are passed *by reference* in this specification, as common to many programming languages such as C, C++ or Python.

For example, if we define the processes *foo* and *bar* as follows:

```
foo():
    num = 100
    map = {}
    map[a] = 200
    bar(num, map)
```

```
bar(number, mapping):
    number += 1
    mapping[a] += 2
    mapping[b] = 300
```

At the end of *foo*, `num` will be 100, `map` will contain 202 in element `a` and 300 in element `b`.

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

```
foo():
    code
    ...
```

[text]

```
...
    more code
...
```

[text]

```
...
    even_more_code
```

The intervening text is a normative part of the specification of the process.

5.7.4 Control Flow

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

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

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

`if:`

The **`if`** control evaluates a Boolean or a Boolean function, and if **True**, passes the flow to the 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 that catches remaining cases.

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

The **if** and **else if** conditions are evaluated in the order in which they are presented.

For example, if `control1` or `control2` is **True** in the preceding example, `block3` will not be executed even if `control3` is **True**; neither will `block4`.

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 in order. For example,

```
for each c in Y, C1, C2:
    foo(c)
```

calls *foo*(Y), then *foo*(C1), then *foo*(C2).

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
```

6 Overall VC-2 Specification

The VC-2 specification comprises the parts listed in Table 2:

Table 2 — Clauses of the VC-2 specification.

Clause #	Title	Description
7	Video Formats	A specification of the video formats supported by VC-2.
8	Encoding Overview (Informative)	An informative overview of the processes used by a VC-2 encoder to compress the video. The sequence of functional elements in the encoder, and the methods used for realizing them, are implementation dependent. The purpose of the encoder is to compress a video input to produce a bitstream that complies with the normative VC-2 stream syntax.
9	Decoding Overview	An overview of VC-2 decoder operations, providing the starting point for subsequent clauses.
10	VC-2 Stream	A specification for the top-level parsing and decoding of data units in a VC-2 stream.
11	Sequence Header	A definition of the metadata contained in the sequence header, necessary for a compliant decoder to parse the remainder of the stream and access picture data.
12	Picture Syntax	A definition of the picture data unit syntax and semantics, including all metadata necessary for configuring wavelet transforms and coefficient decoding.
13	Transform Data Syntax	A definition of the processes needed to unpack and dequantize the wavelet coefficient data from the VC-2 stream, ready for handing over to the picture decoder.
14	Fragment Syntax	A definition of the processes needed to reassemble fragment data from the VC-2 stream ready for handing over to the picture decoder.
15	Picture Decoding	The specification of the picture decoder that takes decoded wavelet transform data, decoding it to produce viewable images.

The VC-2 specification is supported by a number of normative annexes, listed in Table 3, that provide core information needed for the system definition.

Table 3 — Annexes of the VC-2 specification.

Annex A	VC-2 data coding definitions	Defines the VC-2 data coding definitions for wavelet coefficients and header elements.
Annex B	Predefined video formats	Defines the parameters for a range of base video formats. The listing in this annex does not prevent other video format specifications from being used, but provides a short-cut method for identifying commonly used video formats.
Annex C	Profiles and levels	Defines the VC-2 Profiles and describes the use of VC-2 Levels and how they relate to Profiles.
Annex D	Quantization matrices	Defines the Quantization Matrices.

The remaining annexes are informative and provide additional information that can aid implementers but do not provide any further normative provisions of this standard.

NOTE Earlier versions of this specification included two syntaxes called the core syntax and the low delay syntax. The core syntax (along with the main and simple profiles which used it) is no longer part of the specification. The current specified syntax is the one which was formerly called low delay.

7 Video Formats

7.1 General

Clause 7 defines the video formats supported by this standard.

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

This list is not exhaustive, however, and VC-2 is a general-purpose video compression system. A much larger range of video formats is supported by means of syntax to modify the various elements of the predefined base formats. The sequence parameters of the bitstream (Clause 11) are used for signaling both the base video format and any modifications to it.

7.2 Color Model

VC-2 supports any video format that codes the raw image colors in a luma (grey-level) component with two associated color difference components. These components are referred to as Y, C_1 and C_2.

In ITU defined systems (including ITU-R BT.601, ITU-R BT.709, ITU-R BT.2020 and ITU-R BT.2100), the Y, C_1, C_2 values relate to the E'_Y, E'_{CB}, E'_{CR} or Y', C'_B, C'_R video components respectively. These video components are also widely referred to as Y, U, V and Y, C_B, C_R.

In the ITU-T H.264 reversible color transform, the Y, C_1 and C_2 values correspond to the video components Y, C_G, C_O.

In SMPTE ST 428-1 D-Cinema formats, the Y, C_1 and C_2 values correspond to the CV_Y, CV_Z and CV_X components respectively.

VC-2 also supports the RGB video format. The Y, C_1 and C_2 values relate to the G, B and R components respectively.

NOTE Coding using Y, C_G, C_O. provides a simple reversible conversion to and from R-G-B components by using lossless integer transforms. The use of Y, C_G, C_O supports lossless coding of RGB video and allows VC-2 to be treated as an RGB compression system for applications that require this feature.

The color model is signaled in the bitstream using the specifications in 11.4.10.

7.3 Interlace

VC-2 supports both interlace and progressive video formats.

VC-2 codes pictures where a picture can be a frame or a field. A frame contains lines of spatial information of a video signal.

For progressive video, lines contain samples starting from one time instant and continuing through successive lines to the bottom of the frame.

For interlaced video, a frame consists of two fields, a top field and a bottom field. Each field is the assembly of alternate lines of a frame. One of these fields will commence one field period later than the other (i.e., the fields may be coded as either top field first or bottom field first). Each line of a top field is spatially located immediately above the corresponding line of the bottom field. Conversely, each line of a bottom field is spatially located immediately below the corresponding line of the top field.

A pair of fields constituting a frame may correspond to distinct time intervals (true interlace scanning) or to the same time interval (progressive segmented frames).

7.4 Component Sampling

Color difference components C_1 and C_2 shall be coded either with the same dimensions as the Y component (4:4:4) sampling, with half-width (4:2:2) or with half-dimension (4:2:0) sampling.

Y, C_1 and C_2 picture components shall be sampled at the same temporal instant.

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

7.5 Bit Resolution and Signal Ranges

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

Video is represented internally within the decoder specification as a bipolar (signed integer) signal. Video is presented at the video interface as an unsigned integer value by addition of an offset to these values (15.5). Metadata concerning black level and white level is transmitted within the data stream (11.4.9) but is not enforced at the decoder video interface. Output video can undershoot or overshoot these values.

7.6 Video Frame Size and Rate

The frame size and frame rate are, in principle, unrestricted. Application-specific codecs may restrict both the supported frame size and frame rate to a single value or a limited range of values. See C.3.

8 Encoding Overview (Informative)

8.1 General

Clause 8 identifies the modules of the VC-2 picture encoder and describes the function of each module. It provides guidance for implementers through a generalized description of the encoder. It does not normatively define the construction of a VC-2 encoder. The encoder has to create a VC-2 stream that complies with the provisions of the normative decoder specification (Clauses 9-15). Clause 8 is wholly informative and contains no normative provisions.

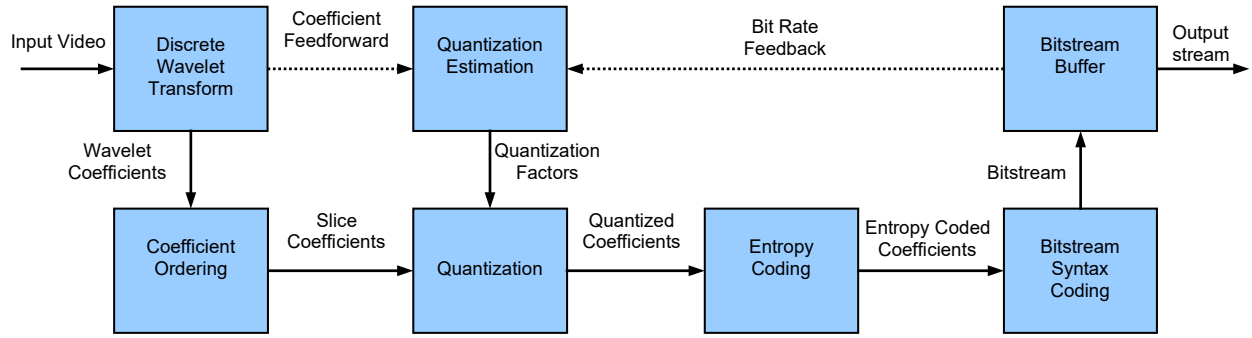

Figure 1 — Sample encoder functional block diagram.

The video picture data is transformed in both the horizontal and vertical axes by a Discrete Wavelet Transform into an array of transform coefficients. These coefficients are then reordered into subbands or slices and quantized.

An implementation-dependent quantization estimation process determines the quantizers based on application requirements. No quantizer estimation processes are defined by this standard. Figure 1 illustrates two possible quantizer estimation sources (“coefficient feedforward” and “bit rate feedback”).

The quantized samples are entropy encoded using exp-Golomb coding to reduce the output bit rate.

The entropy coded coefficients are encapsulated and formatted with VC-2 signaling information to create a stream that complies with the VC-2 syntax defined in Clauses 10-13.

For some applications and for some methods of deriving the quantization factors, a buffer can be used to achieve a constant bit rate stream. The buffer, the rate control scheme and the method for deriving the quantization factors are beyond the scope of this standard.

8.2 Picture Input Processing

The encoder operates independently on input video pictures, which can be frames or fields depending on the configuration of the encoder, and so constitutes a separate processing path.

Depending on the picture source format, the color difference components can be subsampled relative to the luma component.

8.3 Wavelet Transform

The encoder uses a Discrete Wavelet Transform (DWT) to separate the frequency bands of each video component.

The wavelet transform can be thought of as being applied to a picture component as a whole, conceived of as a two-dimensional array of values.

In order to calculate the wavelet coefficients corresponding to an area of the picture only a few lines from the vicinity of that area need be known. Hence the coefficients for slices at the top of the picture can be processed and output before input pixels for the bottom of the picture are available to the encoder. In this way encoding delays as low as a few lines can be achieved.

A single DWT stage, in one dimension, is a filter bank that decomposes a component signal into pair of signals that represent the lower and upper frequencies of the source signal. The transform process can be applied in successive stages, where each stage is applied to the low frequency band of the previous transform, resulting in a series of transforms that represent the signal components in octave frequency bands. This same process can be applied in the two dimensions of an image source by applying both a vertical and horizontal filtering step in each stage, resulting in data that is decomposed in both the horizontal and vertical dimensions to produce four spatial frequency bands. The “low-low” frequency band of the previous transform stage can be further decomposed, analogously with the one-dimensional case. In some cases the final low-low band can be decomposed further with a number of levels of a one-dimensional transform. The process of the DWT and the resulting transformed image is illustrated in Annex F.

As mentioned in 15.4.2 and 15.4.3, in some wavelet filters, accuracy bits are added at each transform step to mitigate aliasing through non-linear rounding effects. This bit-shift can, however, lead to increasing losses during multi-generation encoding/decoding. For this reason, filters without a shift, such as Haar with no shift, are suggested for multi-generation applications.

8.4 Division into Subbands

The two-dimensional DWT creates a series of subbands for each picture, representing spatial frequency bands.

In a single transform stage, illustrated in Figure 2, a picture component is first filtered horizontally by low-pass and high-pass filters to produce low and high frequency subbands. These subbands are subsampled by a factor of two so that there is no increase in the number of samples. Then both these subbands are themselves filtered vertically and subsampled in the same way, to produce four subbands labeled LL, HL, LH and HH and carrying respectively: low-horizontal, low-vertical; high-horizontal, low-vertical; low-horizontal, high-vertical; and high-horizontal, high-vertical frequencies.

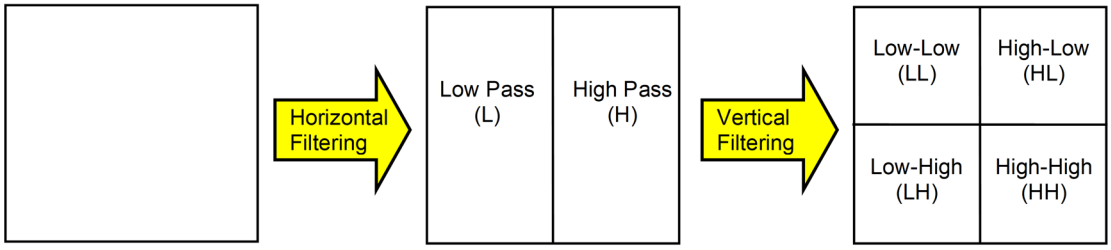

Figure 2 — A single DWT stage.

In a multi-level transform, the LL band at each stage is filtered and subsampled in the same way, resulting in a hierarchy of subbands. In the case of a 4-level transform, the results are as illustrated in Figure 3.

In some cases, a further series of one-dimensional transforms are applied to the low-low subband after all two-dimensional transform levels have been performed. In this case the transform is performed horizontally only.

The resulting subbands are identified by level and orientation, as shown in Figure 3 and Figure 4. In the case where there is no one-dimensional transform portion the 0-LL subband is the lowest frequency subband, the only subband that carries a DC component, and thus appears as a small, but viewable, picture. The HL, LH and HH bands occur at all other wavelet levels. As a result of the iterative filtering process, successive levels carry increasingly higher frequencies and bandwidth. In the case where a one-dimensional transform is applied the lowest frequency subband is the 0-L subband, and the H-subbands carry higher horizontal frequencies.

In principle, there is no limit to the depth of either part of a wavelet transform, subject to the coding level constraints of Annex C, and to the requirement that there is an integral number of coefficients in the 0-LL or 0-L subband for each slice.

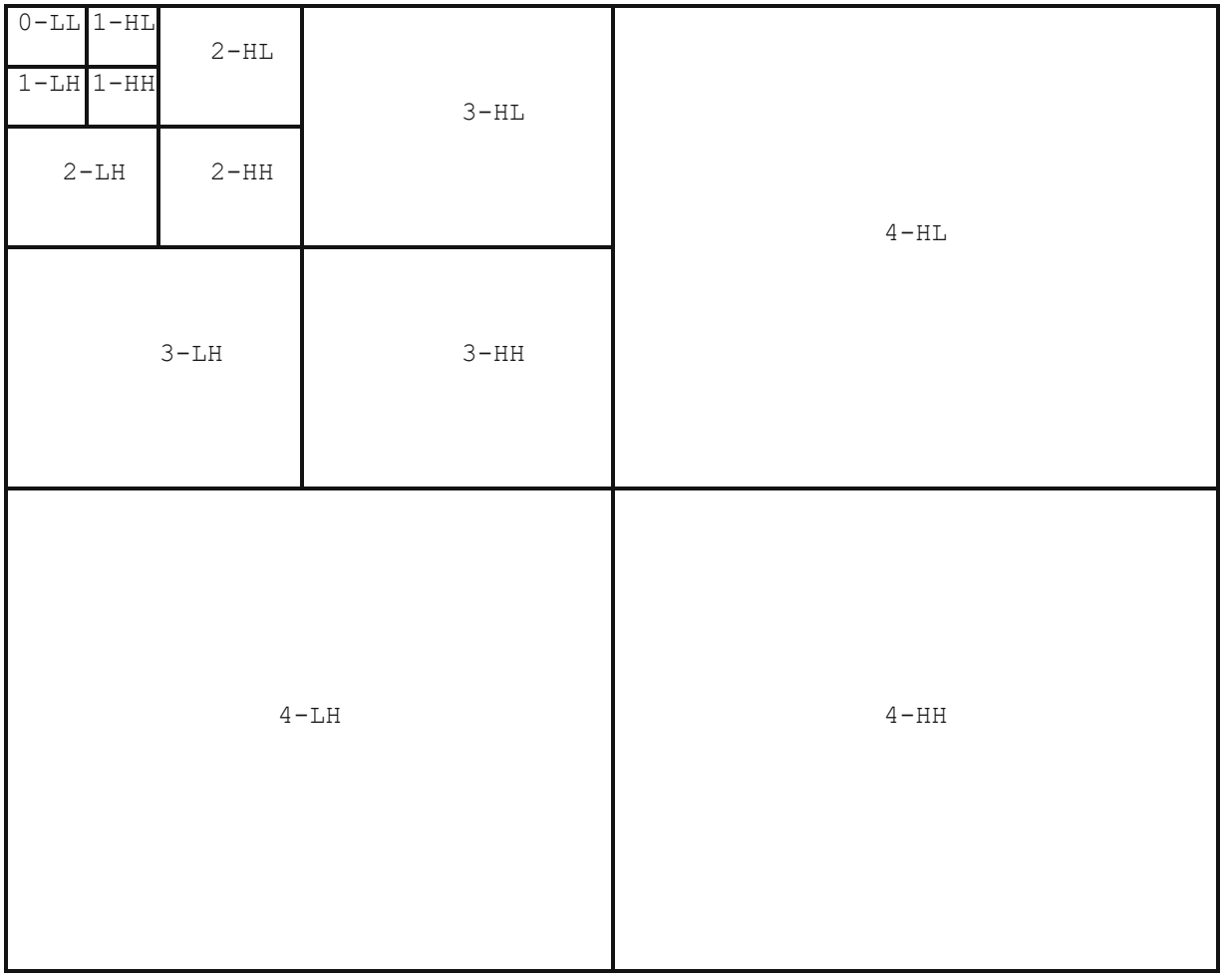

Figure 3 — Identification of the subbands of a 4-level 2D wavelet transform.

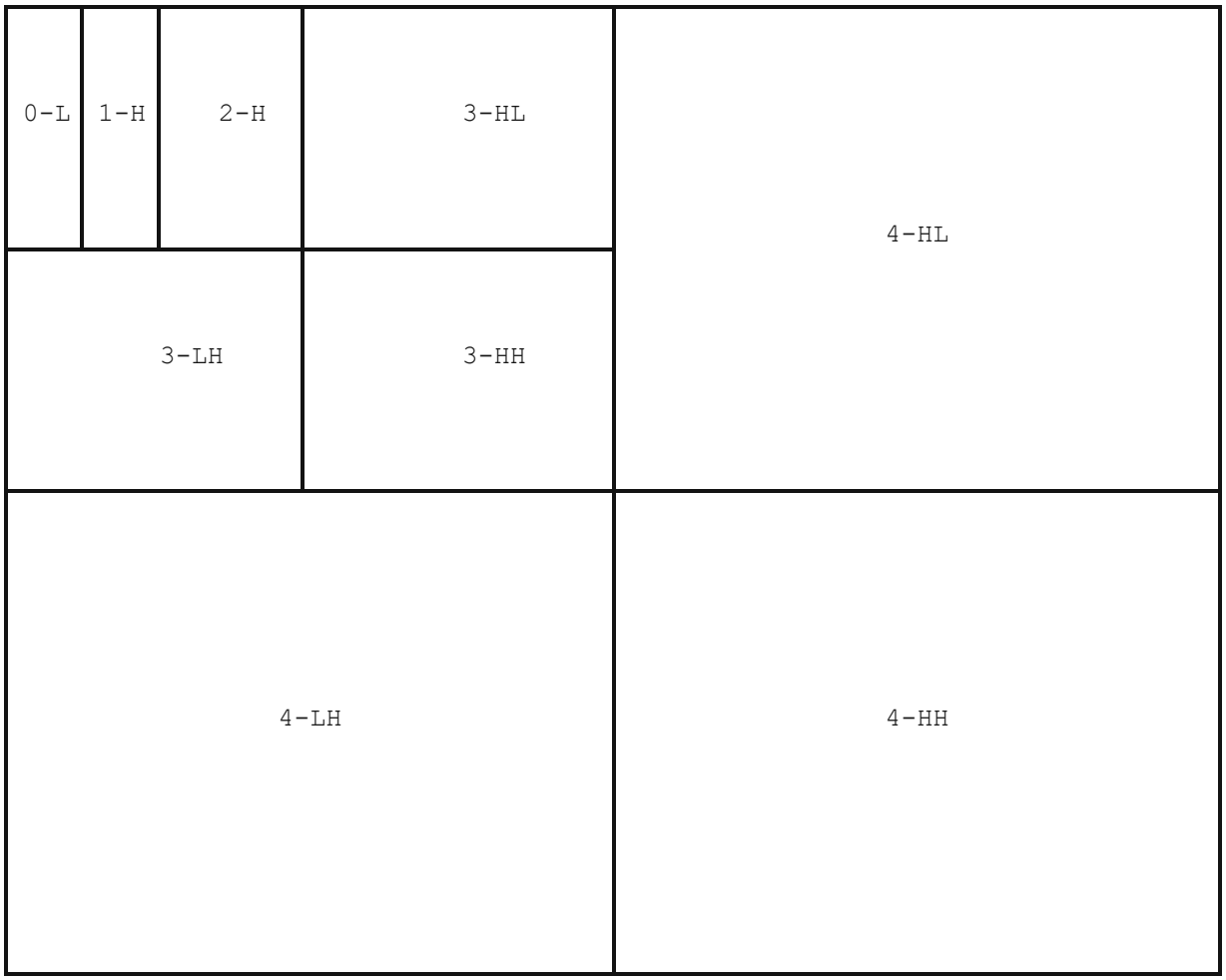

Figure 4 — Identification of the subbands of a 2-level 2D plus 2-level 1D wavelet transform.

8.5 DC Subband Prediction

Spatial prediction can be applied to the 0-LL or 0-L (DC) subband. For each pixel in the 0-LL or 0-L subband, it provides a 3-pixel prediction to improve the coding efficiency. The prediction process is defined in 13.4. Some profiles do not use this coding tool in order to minimize complexity and avoid predictive dependencies in the coded data. Avoiding predictive dependencies increases the robustness of the coded signal with respect to bit errors, which is important in some applications such as archive storage.

8.6 Coefficient Scanning

The coefficients for each subband are reordered into an array of ‘local’ wavelet transforms. Each local transform corresponds approximately to a frequency decomposition of a region of the picture. Each slice contains data from all three video components. This use of slices permits decoding of small areas of the picture thus reducing the coding and decoding delay. Figure 5 and Figure 6 illustrate an example of forming and arranging slices, for a two-level transform.

Each slice is scanned and coded in turn, in raster order, with each slice subband for each video component coded before the next slice.

It is the transform coefficients that are partitioned into slices, not the input pixels. This is in contrast to block transform compression systems, which partition the input pixels prior to performing the transform. As a result, slice coefficients can depend on picture samples adjacent to the nominal slice boundaries as well as within it. Thus slices correspond to a series of overlapping transforms on the picture data.

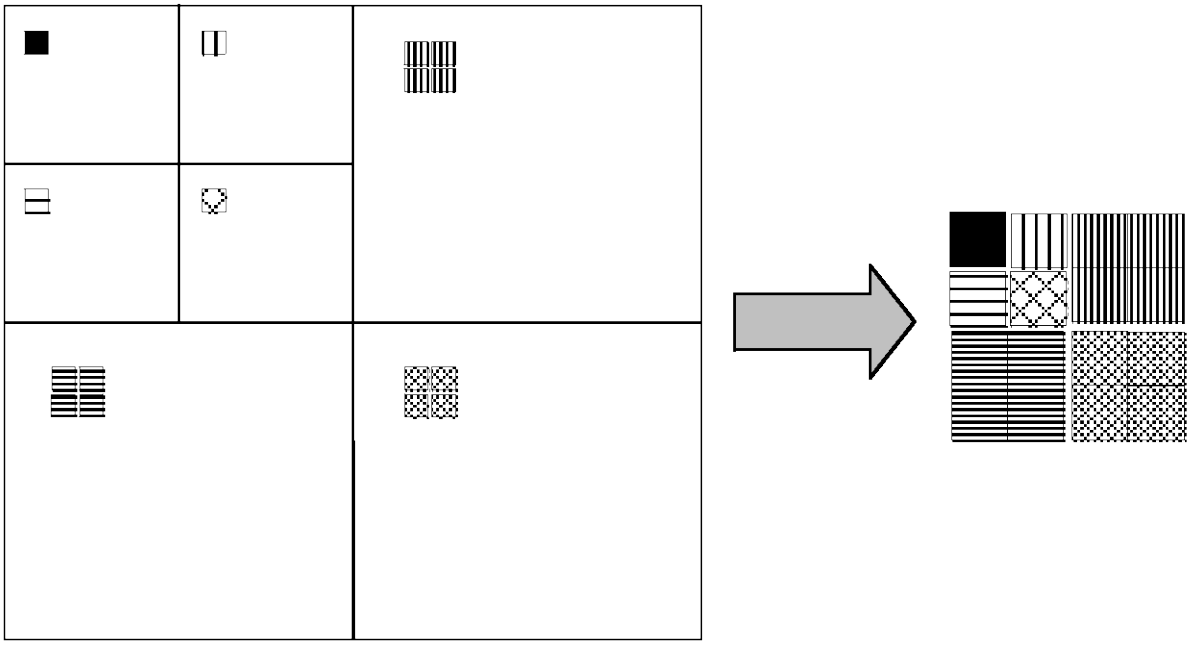

Figure 5 — Formation of a single 4x4 slice from 2-level 2D transform coefficients.

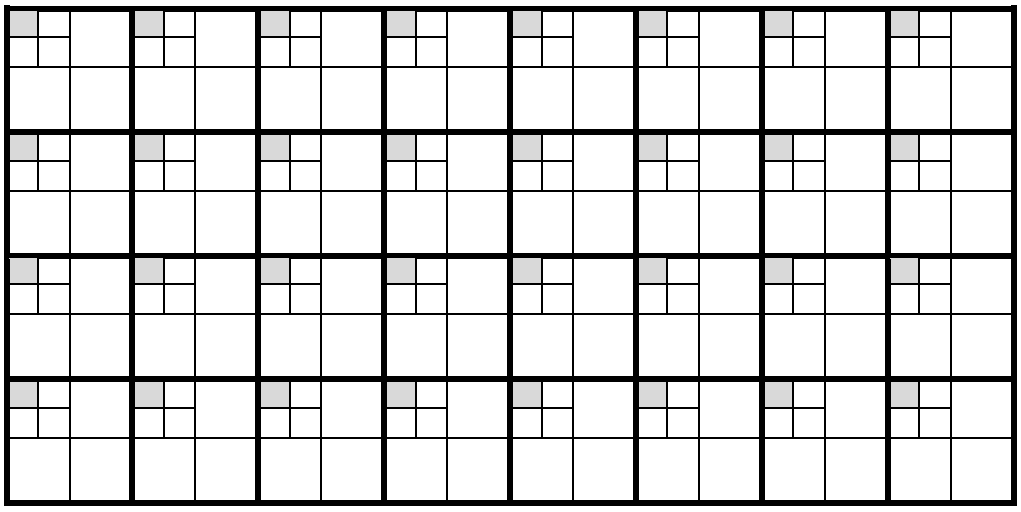

Figure 6 — An 8x4 array of slices for a 2-level 2D wavelet transform.

In the case of the Haar wavelet transform, the order in which the transform and the slice partitioning are performed can be interchanged. That is, for the Haar wavelet transform only, VC-2 can be implemented as a block transform system.

8.7 Quantization and Quantizer Estimation

This specification only defines the quantization process. Inverse quantization is defined in 13.3. Conceptually, forward quantization can be regarded as the integer division of the transform coefficients by a quantization factor.

Quantization factors, determined by the encoder, are encoded into the VC-2 stream to allow inverse quantization at the decoder. The process of determining quantization factors by the encoder is implementation specific and is beyond the scope of this specification.

The quantization factors are coded as quantization indices. A quantization index of zero indicates no quantization. Each increment of the quantization index indicates an increase in quantization by one quarter bit. For example, a quantization index of 4 means one bit of quantization or a (rounded) division by two.

A VC-2 encoder provides a quantizer index for each slice. Quantization indexes for each subband within a slice are derived by means of a quantization matrix.

8.8 Quantization of the DC Band

In the DC subband, the encoded coefficients can be prediction residues after spatial prediction. Quantization is applied to these prediction residues or directly to the transform coefficients if prediction is not used. To eliminate drift between encoder and decoder, when prediction is used, an encoder has to use fully reconstructed (inverse-quantized and inverse-predicted) coefficients for predicting any subsequent coefficients.

8.9 Data Encoding

A variety of methods are employed for encoding data within the stream. These employ two basic data coding methods: fixed-length codes and variable-length codes.

The variable-length codes used are the interleaved form of exp-Golomb coding (see A.4).

Metadata needed to describe the video and specify coding parameters is encoded using fixed-length and variable length codes, defined in A.3 and A.4. Using variable length codes allows VC-2 to support a very wide range of video formats and coding parameters without an excessive bit rate overhead. Using VLCs will also allow the size of tables to be extended without a change of syntax, in any future revisions or extensions of this standard.

Wavelet coefficients are coded using variable-length codes. The variable length codes used for coefficient coding are the same as those used for metadata, except that they are modified so as to be used in data blocks of a known size. This modification inserts bits of value 1 when data within a block has been exhausted. This has the effect of allowing a run of zero coefficients at the end of a data block to be decoded without signaling any further bits.

8.10 Fragmentation

Encoded picture data can optionally be broken into smaller fragments which will be independently encapsulated within the data stream. The intention of this capability is that it can be used to assist with down-stream packet-based transport formats such as network transmission of the VC-2 data. The actual coded data is the same regardless of whether pictures or picture fragments are used as the basic encapsulation units of the stream.

8.11 Stream Syntax

The VC-2 stream syntax (Clause 10) defines the structure of the VC-2 stream and the data types of all data within the stream.

9 Decoding Overview

9.1 General

Clause 9 presents an overview of the VC-2 decoding process. It defines the modules of the decoder and describes the functional definitions of each module.

9.2 Decoding Functions

9.2.1 Functional Units

Figure 7 illustrates the main functional units in a VC-2 decoder.

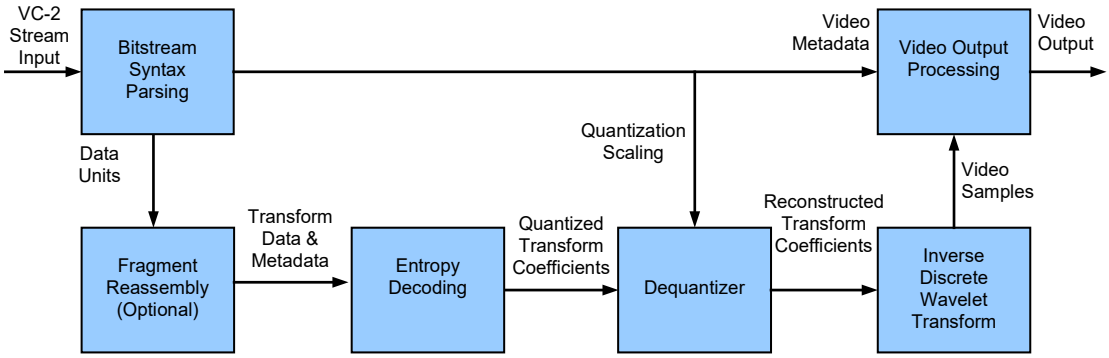

Figure 7 — Functional VC-2 decoder block diagram.

The VC-2 stream syntax is defined in Clause 10 (and subsequent Clauses 11-14), allowing a decoder to extract (unpack) and reassemble the subband coefficients and the associated metadata necessary for successful decoding.

The data decoding processes shall be as defined in 9.2.3.

The inverse discrete wavelet transform decoding and video output processing processes shall be as defined in Clause 15.

9.2.2 Functional Description

The following list defines the core functions of a VC-2 decoder.

- The VC-2 stream at the input to a VC-2 decoder shall comply with the stream syntax defined in Clauses 10-14.
- The entropy encoded data within the VC-2 stream shall be decoded to create the coefficients for each subband together with all other data including quantization values and video metadata.
- The dequantizer process shall be used to recreate transform coefficient samples. The transform samples shall then be input to the inverse DWT for conversion back to picture samples.

9.2.3 Data Decoding

The VC-2 stream shall comprise the sequence of data coded according to the VC-2 stream syntax and the transform data as defined in Clause 10. Different data coding methods are employed in different parts of the VC-2 stream.

Each element of the VC-2 stream shall be decoded according to one of the provisions defined in Annex A. These include:

- Bit-packing and byte alignment elements,
- A Boolean element (values of **True**[=1] and **False**[=0]),
- N-bit literal (a defined number of bits as an unsigned integer, msb first),
- N-byte literal (a defined number of bytes including single byte literals),
- Unsigned interleaved exp-Golomb codes,
- Signed interleaved exp-Golomb codes

9.2.4 VC-2 Data Codings

Stream syntax header data and wavelet coefficients shall use fixed-length and variable-length codes as defined in A.3 and A.4. The variable-length code (VLC) used shall be an interleaved form of exp-Golomb coding.

All syntax elements within the VC-2 stream shall be decoded as follows:

- Fixed length codes shall be decoded using:
  - *read_bool()*: read a single Boolean value
  - *read_boolb()*: read a single Boolean value from a bounded block
  - *read_nbits(n)*: read n bits
  - *read_uint_lit(n)*: read n bytes

- Variable length codes (for header data) shall be decoded using:
  - *read_uint*(): read an unsigned integer
  - *read_sint*(): read a signed integer

- Variable length codes (for coefficient data) shall be decoded using:
  - *read_uintb*(): read an unsigned integer from a bounded block
  - *read_sintb*(): read a signed integer from a bounded block

9.2.5 VC-2 Syntax Decoding

The VC-2 syntax elements, defined in Clause 11, constitute the metadata that describes the characteristics of the source pictures and the compression coding and data formatting parameters. The syntax elements that are needed to decode the compressed data shall be stored in a set of state variables within the decoder and these state variables shall be used by the decoder to extract and decode the transform data to form the output picture.

9.2.6 Fragment Reassembly

If some pictures in a VC-2 stream have been fragmented into smaller data units, those data units shall be reassembled into a picture as described in Clause 14 before further decoding can proceed.

9.2.7 Subband Decoding

Figure 3 and Figure 4 in Clause 8 illustrate the subbands and their frequency characteristics.

With the exception of the DC subband (0-LL or 0-L), the coefficient decoding process for each subband is dependent on data from the subbands of the next lower depth level.

A VC-2 decoder shall decode each subband within a slice and then each slice within a picture as defined in 13.5.3 and 13.5.4.

NOTE: In the specific case of the Haar filter, the inverse transform can be performed on each slice in isolation. If other wavelet filters are used, then the decoding process for a slice will need access to adjacent unpacked slice data to reconstruct the picture data correctly.

9.2.8 DC Band Prediction

Spatial prediction may be applied to the 0-LL or 0-L (DC) subband, as specified for the profile being used. Spatial prediction shall not be applied to other subbands. If spatial prediction is applied then for each coefficient in the DC subband, the prediction shall be defined as follows:

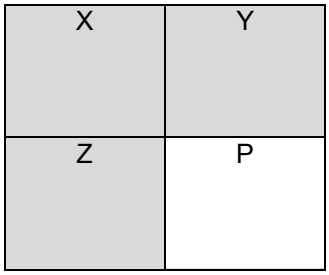

Figure 8 — Prediction aperture for DC bands.

The prediction of the value of coefficient P, as shown in Figure 8, shall be calculated as follows:

- Where coefficients X, Y and Z are available, coefficient P shall be predicted to be the mean value of surrounding pixels X, Y and Z.
- Where P is on the first line, it shall be predicted as the value of Z alone.
- Where P is the first sample in the row, it shall be predicted as the value of Y alone.
- Where P is the first sample in the row and on the first line only, the predicted value shall be zero.

Coefficient P shall be calculated as the sum of the prediction plus the value reconstructed from the stream.

The VC-2 syntax for DC subband prediction shall be as defined in 13.4.

9.2.9 Inverse Quantization

The inverse VC-2 quantization process shall be as defined in 13.3.

A quantizer shall be provided for each slice, from which a quantizer for each slice subband shall be derived by means of a quantization matrix.

9.2.10 Quantizer Factor and Offset (Informative)

The quantizer factor can be considered to be the divisor in the quantization process in the encoder. The value of the quantizer factor is an integer approximation to:

```
4*(2**(qi/4))
```

where `qi` is the quantizer index. The premultiplication by 4 avoids large fractional changes in the quantization factor for small changes of the quantization index. The value of the quantizer factor is defined by integer operations only.

For a quantizer index of q, the quantized value of x would be 4x//quantizer_factor, which is approximately:

```
x/(2**(q/4))
```

The precise details of the quantization process performed by an encoder are outside the scope of this specification and are left to the encoder implementer.

In the decoder, the quantizer offset value is added to the quantized value before multiplication by the quantizer factor to provide a more accurate approximation to the original value.

The quantizer offset is usually the quantizer factor divided by two, except for low values of the quantizer index: the offsets having been selected so as to make inverse quantization and requantization by the same quantization factor transparent i.e., in this case the following two signal chains are equivalent:

1. Quantize -> Inverse Quantize -> (Re-)Quantize
2. Quantize

This property allows for coding with no multi-generational loss, provided that the same quantization index can be selected by each encoding stage. This can be done by minimizing quantization error in subsequent stages.

9.2.11 Coefficient Coding Order

Wavelet coefficients shall be ordered in subband order in the stream.

- The first subband shall be the DC subband (0-LL or 0-L).
- If the transform featured any 1D transform levels, the next subbands shall be subband x-H where x runs from 1 to max1D, and max1D is the number of 1D transform levels.
- These shall be followed by subbands for each successive level from x = (max1D+1) to max in the order x-HL, x-LH, x-HH, where ‘max’ is the level of wavelet decimation.

The following rules shall apply:

- Wavelet coefficients from all subbands shall be spatially partitioned into slices. Coding within a slice shall be in subband order and in raster scan order within a subband.
- A single quantization index shall be coded per slice.

9.2.12 Inverse Discrete Wavelet Transform

The decoder shall use the Inverse Discrete Wavelet Transform (IDWT) to recombine the frequency bands of each video component.

Figure 9 illustrates the progressive nature of the IWDT process on a 2-stage 2D wavelet coded picture.

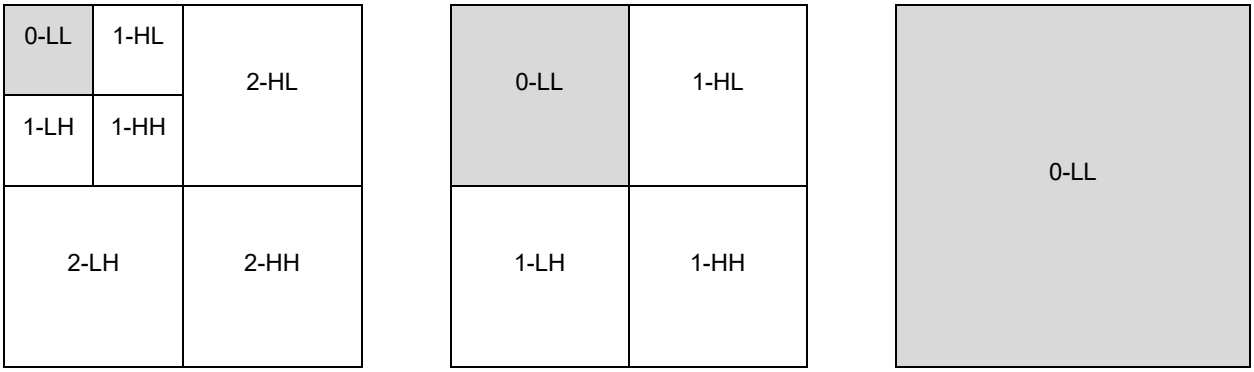

Figure 9 — Wavelet decoding steps (2D).

The leftmost picture represents a wavelet coded picture with all the wavelet coefficients grouped together (as wavelet bands 0-LL, 1-HL/LH/HH and 2-HL/LH/HH).

The first stage of reconstruction shall rebuild a new 0-LL band in two steps as follows:

1. Combine 0-LL and 1-LH to form an intermediate picture 0-L, and combine 1-HL and l-HH to form an intermediate picture 0-H.

2. Combine intermediate pictures 0-L and 0-H to reconstruct a new 0-LL picture.

The result is a new 0-LL picture of twice the original 0-LL picture size as illustrated in the center part of Figure 9.

NOTE The numbers of horizontal and vertical filtering steps need to be the same.

The second reconstruction stage shall be a repetition of the first stage, though the new 0-LL picture is now the same size as the 1-HL, 1-LH and 1-HH pictures in the center part of the figure. The result is thus the reconstructed picture, denoted 0-LL, shown on the rightmost part of Figure 9.

The decoding order is important because small rounding errors can accumulate if the decoding order is not correct. The vertical reconstruction shall always occur before horizontal reconstruction, i.e., all vertical lifting stages are applied first at each level. This process is defined in detail in 15.4.

Figure 10 shows the nature of the IDWT process for the 1D component of a picture with that option. It is very similar.

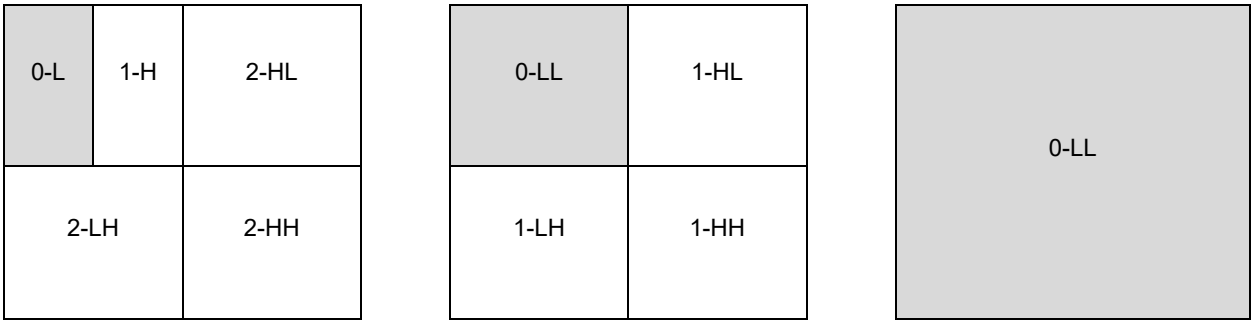

Figure 10 — Wavelet decoding steps (combined 1D and 2D).

See also informative Annex F for an illustration of the Wavelet decimation and reconstruction processes.

9.2.13 Wavelet Filter Support

The list of supported filters for use in the IDWT shall be as defined by the list in Table 15.

The inverse wavelet transform shall be implemented using a lifting process and integer arithmetic.

With the sole exception of the Haar transform, the wavelet filters use video samples that extend beyond each edge of the picture at both the horizontal and vertical axes. Edge effects are overcome by using the edge extension procedure that is part of each lifting process. In the case where no quantization is applied, this process ensures that the decoded picture is perfectly reproduced even at picture edges.

NOTE 1 The edge extension procedures are part of the lifting process definitions in 15.4.4.1.

NOTE 2 An accessible reference to the lifting process is explained in “Ripples in Mathematics, The Discrete Wavelet Transform” as listed in the bibliography.

9.2.14 Clipping

The compression process can introduce signal errors in the video components that might result in signal overshoots and undershoots. Clipping of the output pictures (as defined in 15.5) shall ensure that the signals lie within the range that can be supported by the output interface. Additional clipping may be applied to ensure that the signal falls between the black and white levels defined for a specific video format or application.

10 VC-2 Stream

10.1 General

Clause 10 defines the structure of VC-2 streams and the overall processes for parsing and decoding.

10.2 Pseudocode

The parsing process is normatively defined using pseudocode and/or mathematical formulae. The definitions of stream syntax operations and pseudocode shall be as defined in Clause 5. The stream parsing specifications are augmented by informative parse diagrams throughout Clause 10, each of which illustrates that part of the stream structure in graphical form.

The VC-2 stream syntax uses a state model to express the stream in a way that can be parsed and used for decoding operations (see 5.3). The parsing and decoding operations are defined in terms of modifying the decoder state according to the data extracted from the VC-2 stream. The state of the decoder is stored in the variable `state`. This is a map (see 5.5.2) and individual elements are accessed by means of labels, e.g., `state[a_specific_label]`. The state variables comprise the parameters that shall be used in parsing and decoding a picture.

Some parameters are encoded in the stream as indices to tables of values. The indices shall be coded as variable length integers. This allows the tables to be extended to contain new entries, in future versions of this specification, without changing the syntax.

10.3 VC-2 Stream Syntax

A VC-2 Stream shall be a concatenation of one or more VC-2 Sequences.

The process for parsing or decoding a VC-2 Stream shall be as follows:

parse_stream`(state):`	**Ref**
`    while (not is_end_of_stream(state)):`	A.2.5
`        parse_sequence(state)`	10.4.1

Stream parsing shall proceed as a single call to ***parse_stream*** with a `state` map with `state[current_byte]` and `state[next_bit]` initialized as defined in A.2.1.

NOTE Although each VC-2 Sequence is, in effect, a newly decodable entity, care needs to be taken to ensure that any changes in encoded video parameters can be supported by downstream equipment. For example, changes to parameters such as frame rate and picture size might not be supported.

10.4 VC-2 Sequence Syntax

10.4.1 Sequence Structure

A VC-2 Sequence shall comprise an alternating sequence of parse info headers and data units, as shown in Figure 11. The first data unit shall be a sequence header, and further sequence headers may be inserted at any data unit point in the sequence.

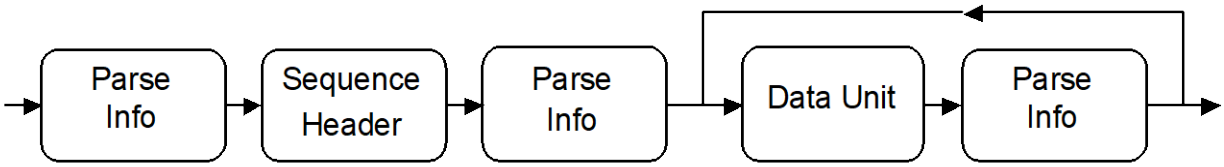

Figure 11 — Overview of VC-2 sequence structure.

Data units shall be one of the following (see 10.4.3):

- a sequence header,
- a picture,
- a picture fragment,
- auxiliary data or
- padding data.

The process for parsing a VC-2 sequence shall be defined as follows:

parse_sequence(state):	Ref
reset_state(state)	10.4.1
parse_info(state)	10.5.1
while (**not** is_end_of_sequence(state)):	Table 5
if (is_seq_header(state)):	Table 5
state[*video_parameters*] = sequence_header(state)	11.1
else if (is_picture(state)):	Table 5
picture_parse(state)	12.1
picture_decode(state)	15.2
else if (is_fragment(state)):	Table 5
fragment_parse(state)	14.1
if (state[*fragmented_picture_done*]):	14.4
picture_decode(state)	15.2
else if (is_auxiliary_data(state)):	Table 5
auxiliary_data(state)	10.4.4
else if (is_padding_data(state)):	Table 5
padding(state)	10.4.5
parse_info(state)	10.5.1

Each VC-2 Sequence shall start and end with a parse info header.

The `reset_state(state)` function shall clear all entries from the `state` map except `state[`*`current_byte`*`]` and `state[`*`next_bit`*`]` as defined in A.2.1.

10.4.2 Parse Info Headers

Parse info headers shall contain a 32-bit code so that the decoder can be synchronized with the stream. They are defined in 10.5.

The parse info headers support navigating through the stream without the need to decode any data units. Each parse info header contains pointers to the location of the next and previous parse info headers within the stream. The stream can thus be thought of as a doubly linked list of data units.

Each parse info header shall contain a code that identifies the type of data held in the following data unit. This is the only information contained within the parse info headers that is needed to decode the sequence.

10.4.3 Data Units

This clause describes the five types of data unit used within a sequence.

A sequence header shall contain metadata describing the coded sequence and metadata needed to decode the stream. The sequence header is defined in Clause 11. The first data unit in a sequence shall be a sequence header. To support reverse-parsing applications, the last data unit in a sequence should also be a sequence header.

NOTE The number of sequence headers in a VC-2 Sequence is application dependent. In a streaming application it could be appropriate to have one sequence header for each picture. When a VC-2 stream is stored in a file on a computer system it could be appropriate to have only a single sequence header per sequence.

A picture data unit shall contain sufficient data to decode a single picture (frame or field of video), subject to having parsed a sequence header within the sequence. Each picture, whether a frame or field, shall be coded with no dependency on adjacent pictures. The picture data unit is defined in Clause 12.

A picture fragment data unit shall contain a portion of the data sufficient to decode a single picture (frame or field of video), subject to having parsed a sequence header within the sequence. Multiple picture fragments can be assembled into a picture which is otherwise the same as carried in a picture data unit. Each fragment, whether from a frame or field, shall be coded with no dependency on fragments from adjacent pictures. The picture fragment data unit is defined in Clause 14.

Pictures within a sequence shall be either all frames or all fields. Where pictures are fields, a sequence shall comprise a whole number of frames (i.e., an even number of fields) and shall begin and end with a whole frame/field-pair.

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

The sequence decoding and parsing processes shall start by extracting and decoding the information contained in a sequence header. Picture data units contain the additional information needed to decode a single output picture.

Auxiliary and padding data units comprise undefined data for the purposes of this standard and do not contribute to the decoding process. These data units (together with the correct preceding parse info header) may be interposed at any point in the stream. A decoder may skip these data units or it may read the data and use it for an application-specific purpose, out of scope of this standard. Parsing of auxiliary and padding data units is defined in 10.4.4 and 10.4.5 respectively.

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

For the purposes of subsequent parts of this standard, the potential presence of auxiliary and padding data is ignored.

10.4.4 Auxiliary Data

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

auxiliary_data(state):	Ref
for I = 1 **to** state[*next_parse_offset*]-- 13:	10.5.1
read_uint_lit(state, 1)	

10.4.5 Padding

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

padding(state):	Ref
for i = 1 **to** state[*next_parse_offset*]-- 13:	10.5.1
read_uint_lit(state, 1)	

10.5 Parse Info Header Syntax

10.5.1 Parse Info Header

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

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

- at the beginning of a sequence,
- at the end of a sequence,
- before each data unit (whether sequence header, picture, padding or auxiliary data).

The parse info header shall consist of 13 bytes and shall be byte-aligned within the sequence. Thus, it ensures that succeeding data elements are byte aligned.

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

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

parse_info(state):	Comment
byte_align(state)	
read_uint_lit(state, 4)	Parse info prefix
state[*parse_code*] = read_uint_lit(state, 1)	
state[*next_parse_offset*] = read_uint_lit(state, 4)	
state[*previous_parse_offset*] = read_uint_lit(state, 4)	

The syntax is illustrated in Figure 12.

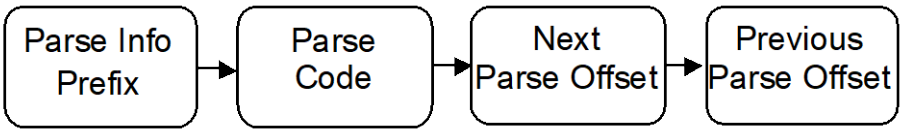

Figure 12 — Parse info header syntax.

The parse info parameters shall satisfy the following constraints:

- The parse info prefix shall be 0x42 0x42 0x43 0x44.
- The parse code shall be one of the supported values listed in in Table 4.
- The 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 no following parse info header in the sequence, the next parse offset shall be zero. If the parse info header is part of a picture or fragmented picture data unit, this value may be zero to indicate an unknown length.
- The 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 preceding parse info header, it shall be zero.

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

NOTE 1 The parse info prefix is the character string: “BBCD” as expressed by ISO/IEC 646.

NOTE 2 The parse info prefix, next parse offset and previous parse offset values are provided to support navigation. See 10.6.

NOTE 3 The parse offset values will normally be non-zero. However, at the beginning and end of a stream there are no preceding or following parse info headers respectively. In these circumstances the value of the offset is zero. Typically, a zero value for a previous or next parse offset can only occur in the sequence header and end of sequence data unit at the beginning and end of a stream respectively. However, real-time, variable bit rate encoders that are unable to determine the position of the next parse info header without adding significant encoding delay can set the next parse offset value to zero if the parse info header precedes a picture or fragmented picture.

10.5.2 Parse Codes

The values of parse codes allowed within the VC-2 syntax shall be as shown in Table 4.

Table 4 — Parse Code Values.

Parse Code (hex)	Binary Code (Informative)	Description
VC-2 Syntax:		
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
Pictures:		
0xC8	1100 1000	Low Delay Picture
0xE8	1110 1000	High Quality Picture
Picture Fragments:		
0xCC	1100 1100	Low Delay Picture Fragment
0xEC	1110 1100	High Quality Picture Fragment

The two least significant bits (bits 0, 1) in the parse code shall always be zero. Other permutations of bits 0 and 1 shall be reserved for future expansion.

The parse codes shall be associated with a group of functions, listed in Table 5, which shall determine the type of subsequent data and the parsing and decoding processes which shall be used. All functions shall return a Boolean value.

Table 5 — Parse Code Functions.

Function	Decoding Operation
is_seq_header(state):	**return** state[*parse_code*] == 0x00
is_end_of_sequence(state):	**return** state[*parse_code*] == 0x10
is_auxiliary_data(state):	**return** (state[parse_code] & 0xF8) == 0x20
is_padding_data(state):	**return** state[*parse_code*] == 0x30
is_ld(state):	**return** (state[parse_code] & 0xF8) == 0xC8
is_hq(state):	**return** (state[parse_code] & 0xF8) == 0xE8
is_picture(state):	**return** (state[parse_code] & 0x8C) == 0x88
is_fragment(state):	**return** (state[parse_code] & 0x0C) == 0x0C
using_dc_prediction(state):	**return** (state[parse_code] & 0x28) == 0x08

Previous versions of this specification included two additional parse codes, decoders complying with this version of the specification shall either parse data units that immediately follow parse info blocks containing the codes 0x08 and 0x48 in conformance with version 2 of this specification or discard such data units. It is possible that future versions of this specification will introduce new parse codes. In order that decoders complying with this version of the specification can decode future versions of the coded stream, the decoder shall discard data units that immediately follow parse info blocks containing parse codes not defined in this version of the specification or any earlier version.

10.5.3 Parse Code Values (Informative)

The rationale for the parse code values defined in Table 4 is as follows:

- The MS two bits (bits 7 and 6) indicate the use of features not present in the current version of this specification. On pictures (when bit 3 is set) they are all set, in all other cases they are all unset. All other permutations are reserved.

- The next three MS bits (bits 5, 4 and 3) indicate the type of data unit following the parse info unit. Bit 3 indicates whether it contains picture or non-picture data. Bits 5 and 4 indicate the 4 other parse codes.

- The next MSB bit (bit 2) indicates whether a picture data unit is a complete picture or a fragment of a picture.

10.6 Non-Sequential Picture Decoding (Informative)

The ability to decode pictures in a non-sequential manner is important for many applications, such as video editing. Non-sequential access means decoding a stream in any manner other than decoding pictures sequentially from the beginning of the stream to the end. Non-sequential picture access is outside the scope of this specification. Nevertheless the VC-2 stream has been designed to support this feature. This clause provides informative notes on this aspect of the VC-2 stream specification.

Stream navigation, including non-sequential access is supported by the information in the parse info headers in the stream. Details of parse info headers are defined in 10.5.

In order to start decoding, other than at the start of a stream, the decoder first needs to synchronize to the stream. The parse info prefix is present to support such synchronization. A decoder would first search for the parse info prefix to locate the start of a parse info header. The parse info prefix is not guaranteed to occur uniquely within parse info headers (the entropy coding used in VC-2 precludes this), the parse info prefix can, by chance, occur within a data unit. The decoder could read the next or previous parse info offsets to confirm that an occurrence of the parse info prefix corresponds to a parse info header.

When the decoder finds a parse info prefix it can skip backwards by the value of the previous parse offset and check whether the next four bytes are also equal to the 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 can search for the next prefix value in the stream 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. The test for two independent parse info prefixes, correctly separated is, therefore, more than adequate in practice.

Having synchronized with the stream the decoder now needs to locate a sequence header in order to find parameters needed to decode pictures. This can be done by moving forward (or backward) through the stream, between successive parse info headers, using next (or previous) parse offsets, until a parse info header is found containing the parse code for a sequence header. Decoding can now start, as if from the beginning of the stream.

The VC-2 stream also supports seeking to a particular picture number. The first four bytes of the picture header contain a 4-byte picture number. Picture numbers provide a continuously incrementing identifier for each picture with a sequence. So, to find a particular picture, the decoder can move forward or backward in the stream, using the offsets in the parse info headers, until the correct picture number is reached. A picture can be decoded once the parameters within a sequence header, in the same sequence, have been read.

Fragments, when present, add slight extra complexity to seeking within streams. To begin decoding a fragmented picture the decoder needs to seek to a fragment which contains the necessary transform parameters to decode the picture, before decoding the slices. This fragment can be identified because the first four bytes will contain the picture number, the next two bytes will contain the length of the data in the fragment and the following two bytes will be zero. If these two bytes are not zero, then the fragment contains coded slice data rather than transform parameters.

11 Sequence Header

11.1 General

Clause 11 defines the structure of the sequence header syntax. The sequence header is illustrated in Figure 13.

Parsing the sequence header consists of reading the sequence parameters (parse parameters, base video format, source parameters and picture coding mode) and initializing the decoder parameters. The decoder parameters shall be initialized by the *set_coding_parameters* process, as defined in 11.6.1.

The sequence header shall remain byte identical throughout a sequence.

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

sequence_header(state):	**Ref**
parse_parameters(state)	11.2
base_video_format = read_uint(state)	11.3
video_parameters = source_parameters(state, base_video_format)	11.4
state[*picture_coding_mode*] = read_uint(state)	11.5
set_coding_parameters(state, video_parameters)	11.6
return video_parameters	

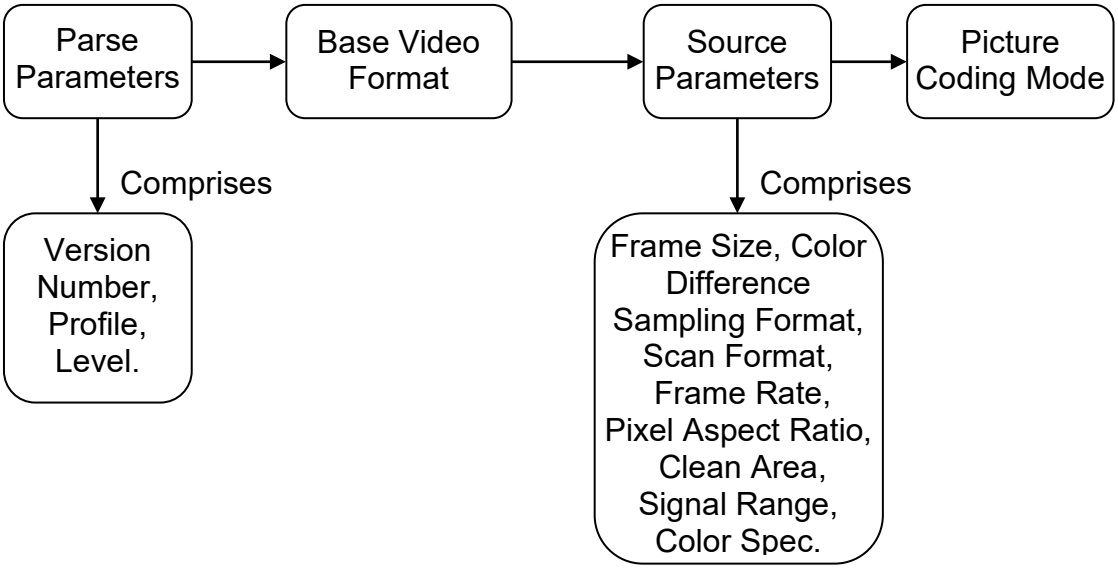

Figure 13 — Sequence header.

Parse parameters contain information that a decoder can use to determine whether it is able to parse or decode the stream.

The base video format is a numerical index denoting a default set of parameters that describe the video source. The numerical index values shall be as defined in Annex B.

Source parameters may override some or all of the parameters indicated by the base video format. However, for many common applications, these predefined formats will be sufficient without the need for further metadata to be present in the stream.

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

Picture coding mode indicates how the video frames have been coded (e.g., as a sequence of frames or fields).

11.2 Parse Parameters

11.2.1 General

The process for reading parse parameters shall be as follows:

parse_parameters(state):	Ref
state[*major_version*] = read_uint(state)	11.2.2
state[*minor_version*] = read_uint(state)	11.2.2
state[*profile*] = read_uint(state)	11.2.3
state[*level*] = read_uint(state)	11.2.3

The syntax is illustrated in Figure 14.

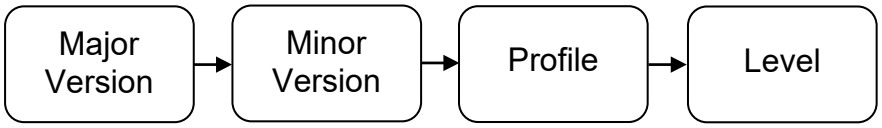

Figure 14 — Parse parameters.

The parse parameters data values shall be constant (byte-for-byte identical) for all instances of the sequence header within a VC-2 sequence. For VC-2 Stream interchange, the VC-2 Profile and Level values should also be identical for all VC-2 sequences within a VC-2 stream.

11.2.2 Version Number

The version number of the VC-2 syntax specification shall be used by the decoder to determine whether it can decode the sequence.

The major version number shall define the version of the syntax with which the stream complies. A decoder complies with a major version number if it can parse all bitstreams that comply with that version number. Decoders that comply with a major version of the specification need not be able to parse the bitstream corresponding to a different specification.

Depending on the profile and level defined, it is possible that a decoder compliant with a given major version number will not be able to decode fully all parts of a stream.

All minor versions of the specification will be functionally compatible with earlier minor versions with the same major version number. It is possible that later minor versions will contain corrections, clarifications, and removal of ambiguities. Later minor version numbers will not contain new features or new normative provisions.

Functional compatibility shall imply that a decoder with the same major version number and later minor version number as a stream shall be capable of decoding the stream and producing decoded pictures that are substantially equivalent to that of a decoder with the same version number as the stream. This shall be assessed by the appropriate technology committee when the minor version number is incremented.

There are three different major versions for this specification depending on which features are used. If any of the following is true, then the major version of the stream shall be 3:

- The `custom_frame_rate_flag` in the sequence header is set to **`True`** and the index encoded after it in the stream (see 11.4.6) is greater than 11.
- The `custom_signal_range_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.9) is greater than 4.
- The `custom_color_spec_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.1) is greater than 4.
- The `custom_color_spec_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.1) is 0 and the `custom_color_primaries_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.2) is greater than 3.

- The `custom_color_spec_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.1) is 0 and the `custom_color_matrix_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.3) is greater than 3.

- The `custom_color_spec_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.1) is 0 and the `custom_transfer_function_flag` is set to **`True`** and the index encoded after it in the stream (see 11.4.10.4) is greater than 3.

- If asymmetric filtering is being used, i.e., **`state`**`[dwt_depth_ho]` is not equal to 0 or **`state`**`[wavelet_index_ho]` is not equal to **`state`**`[wavelet_index]` (see 12.4).

- If the stream includes Fragments, that is the Parse Info Header (see 10.5) includes the parse codes (0xCC) or (0xEC) for Picture Fragments (Table 4).

Furthermore if none of the above bullets are true, but the stream includes High Quality Pictures, i.e., the Parse Info Header (see 10.5) includes the parse code (0xE8) for High Quality Pictures (Table 4), then the major version number of a stream compliant with this version of the VC-2 standard shall be 2.

Finally in all other cases the major version number of a stream compliant with this version of the VC-2 standard shall be 1.

The minor version number of a stream compliant with this version of the VC-2 standard shall be 0.

NOTE The purpose of the major version number is to indicate to a decoder whether it can decode a stream. If a stream does not contain Fragments or High Quality Pictures, does not use asymmetric transforms and does not make use of any of the extra values for the enumerations in the sequence header, then it can be decoded by the 2012 version of this specification, i.e., the stream is backward compatible (indeed identical) to the 2012 specification. If it contains High Quality pictures but otherwise is the same as above then it can be decoded by the first amended version of this specification, i.e., the stream is backward compatible (indeed identical) to the first amendment of the specification. Therefore, the major version number of the stream only has the value 3 if the stream includes fragments or asymmetric transforms or makes use of the extended ranges on some enumerated values.

11.2.3 Profiles and Levels

The profile shall define the toolset that is sufficient to decode a VC-2 sequence.

The level shall define decoder resources (picture and data buffers; computational resources) sufficient to decode a sequence.

Profiles are defined in C.2. Levels are described in C.3.

NOTE It is possible that a decoder in compliance with this version of the specification will not be able to decode a stream which conforms to a profile not detailed in this version of this specification. Support for profiles from earlier versions of this specification is optional.

11.3 Base Video Format

The value of `base_video_format` decoded in parsing the sequence header shall be an index into one of a set of predefined video formats defined in Table 6. For each predefined video format entry in the table, the base video parameters shall be as defined in Annex B.

The selection of a base format represents an initial approximation of the video format, which can then be refined to capture all the video format characteristics accurately by overriding parameters as necessary. In particular, the predefined video formats listed in Table 6 do not represent all the video formats supported by VC-2; any video format parameters can be defined and supported by a VC-2 sequence.

These base parameters may be modified by subsequent metadata present in the stream, with the exception of the `top_field_first` parameter which can only be set by the base video format (see 11.4.5).

Table 6 — VC-2 base video formats.

base_video_format	Video Format Name (Informative)
0	Custom Format
1	QSIF525
2	QCIF
3	SIF525
4	CIF
5	4SIF525
6	4CIF
7	SD 480i60
8	SD 576i50
9	HD 720p60
10	HD 720p50
11	HD 1080i60
12	HD 1080i50
13	HD 1080p60
14	HD 1080p50
15	DC 2K-24
16	DC 4K-24
17	UHDTV 4K-60
18	UHDTV 4K-50
19	UHDTV 8K-60
20	UHDTV 8K-50
21	HD 1080p24
22	SD Pro486

NOTE 1 The custom format is intended for use when no other suitable base video format is available from the table. Video format default values will still be set as per Annex B, but these are token values that will be almost wholly overridden by the subsequent source parameter values.

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

NOTE 3 The video format name is purely informative and does not imply adherence to any video interface.

11.4 Source Parameters

11.4.1 General

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

The source parameters shall comprise frame size, sampling format, scan format, frame rate, aspect ratio, clean area, signal range and color specification, as illustrated in Figure 15. The frame size, sampling format, scanning format and the signal range are required to decode the video. Display and downstream processing falls outside the scope of this specification, hence the interpretation of the other parameters (not required to decode the video) is not normatively defined, with the exception of frame rate (see 11.4.6). The frame rate imposes requirements on compliant decoders for a given level and profile (Annex C).

Source parameter data shall remain constant throughout a VC-2 sequence.

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

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

source_parameters(state, base_video_format):	**Ref**
video_parameters = set_source_defaults(base_video_format)	11.4.2
frame_size(state, video_parameters)	11.4.3
color_diff_sampling_format(state, video_parameters)	11.4.4
scan_format(state, video_parameters)	11.4.5
frame_rate(state, video_parameters)	11.4.6
pixel_aspect_ratio(state, video_parameters)	11.4.7
clean_area(state, video_parameters)	11.4.8
signal_range(state, video_parameters)	11.4.9
color_spec(state, video_parameters)	11.4.10
return video_parameters	

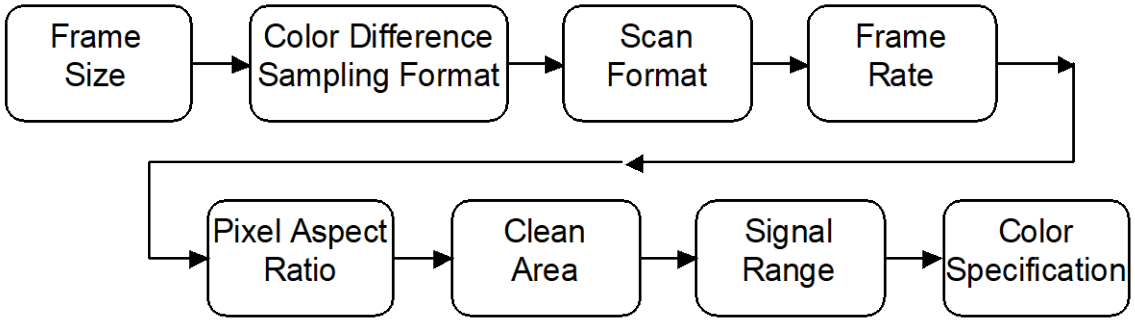

Figure 15 — Source parameters.

11.4.2 Setting Source Defaults

The function that sets the default values of the source video parameters shall take the base video format index as an argument. That is, the signature of this function is:

```
set_source_defaults(base_video_format)
```

where `base_video_format` is an unsigned integer. The function returns a map of source video parameters.

The source video parameters shall be set, based on the base video format index, as defined in Annex B. The parameters set by this function shall be: frame size, sampling format (4:4:4, 4:2:2 or 4:2:0), scan format (progressive or interlace), frame rate, pixel aspect ratio, clean area, signal range, color specification. The labels used to access the map returned by the function shall be as defined in 11.4.3 through 11.4.10 that specify how to override the default video source parameters. The labels are also listed in Annex B.

NOTE 11.4.3 through 11.4.10 each specify a Boolean flag that determines if custom video parameters are required. If the custom parameters are required, then the custom parameters are present in the stream else the custom parameters are omitted from the stream. For example, if `video_format` == 4, CIF defaults are set, with picture size equal to 352x288 with a 4:2:0 color difference format, amongst other parameters. A frame width of 360 pixels can be encoded by setting the Boolean flag to True and overriding the default CIF frame dimensions.

11.4.3 Frame Size

The frame size decoding process shall be defined as follows:

```
frame_size(state, video_parameters):
    custom_dimensions_flag = read_bool(state)
    if (custom_dimensions_flag):
        video_parameters[frame_width] = read_uint(state)
        video_parameters[frame_height] = read_uint(state)
```

The syntax is illustrated in Figure 16.

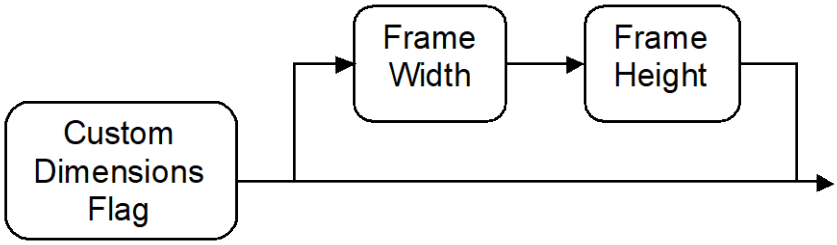

Figure 16 — Frame size.

Thus, if `custom_dimensions_flag` is set to **`True`**, the frame size defined by the default values is overridden by the new values.

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

The frame width and frame height shall be at least 1.

11.4.4 Color Difference Sampling Format

The color difference sampling format decoding process shall be defined as follows:

```
color_diff_sampling_format(state, video_parameters):
    custom_color_diff_format_flag = read_bool(state)
    if (custom_color_diff_format_flag):
        video_parameters[color_diff_format_index] = read_uint(state)
```

The syntax is illustrated in Figure 17.

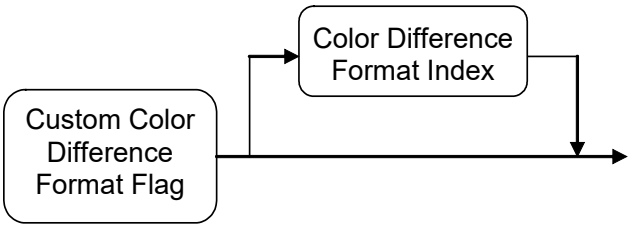

Figure 17 — Sampling format.

Thus if the `custom_color_diff_format_flag` is set to **`True`**, the color difference sampling format defined by the default value shall be overridden by the new values defined by the index value.

The decoded value of `video_params[color_diff_format_index]` shall lie in the range 0 to 2 with values defined as in Table 7.

Table 7 — Color difference sampling formats.

Index	Color difference format
0	4:4:4
1	4:2:2
2	4:2:0

The color difference sampling format shall be used to determine the width and height of the color difference components of the coded video, as described in 11.6.2.

11.4.5 Scan Format

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

The scan format decoding process shall be defined as follows:

```
scan_format(state, video_parameters):
    custom_scan_format_flag = read_bool(state)
    if (custom_scan_format_flag):
        video_parameters[source_sampling] = read_uint(state)
```

The syntax is illustrated in Figure 18.

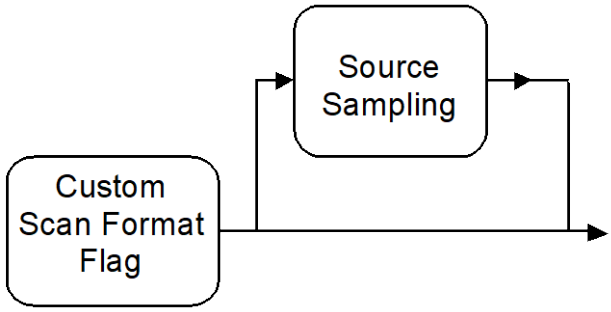

Figure 18 — Scan format.

If the `custom_scan_format_flag` is set to **`True`**, the scan format parameters defined by the default values shall be overridden by the new values.

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

The parameter `video_parameters[top_field_first]` shall be **`True`** if the top line of the frame is in the earlier field, else `video_parameters[top_field_first]` shall be **`False`**. This shall be set only by the base video format and cannot be overridden in the source parameters.

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

11.4.6 Frame Rate

The frame rate value (in frames per second) shall be defined by the ratio of the parameters:

`video_parameters[`*`frame_rate_numer`*`]` divided by `video_parameters[`*`frame_rate_denom`*`]`

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

```
frame_rate(state, video_parameters):
    custom_frame_rate_flag = read_bool(state)
    if (custom_frame_rate_flag):
        index = read_uint()
        if (index == 0):
            video_parameters[frame_rate_numer] = read_uint(state)
            video_parameters[frame_rate_denom] = read_uint(state)
        else:
            preset_frame_rate(video_parameters, index)
```

The syntax is illustrated in Figure 19.

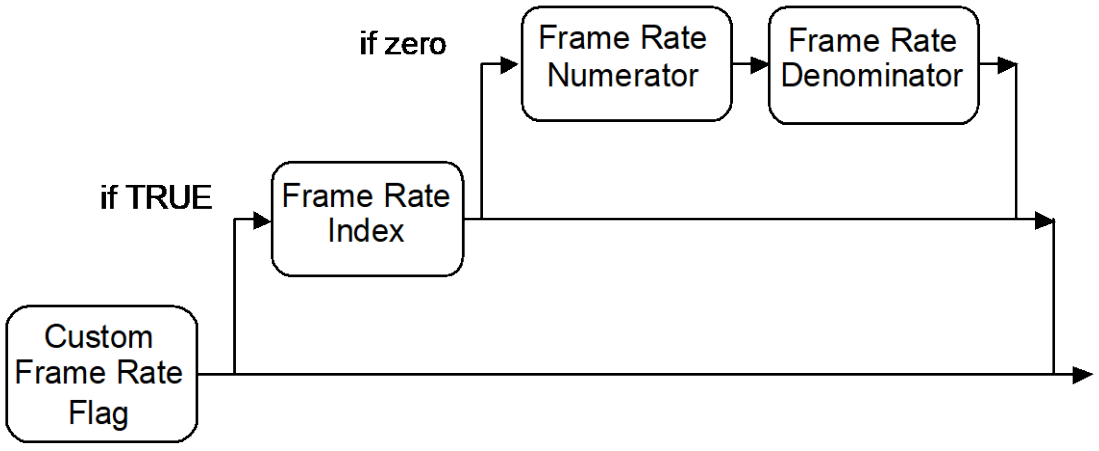

Figure 19 — Frame rate.

If `custom_frame_rate_flag` is set to **`True`**, the frame rate defined by the default values shall be overridden by the new values defined by the index value.

NOTE What is encoded is the frame rate and not picture rate. If the source video is interlaced, and is encoded as fields rather than frames, then the picture rate is twice the encoded frame rate.

The value of `index` shall lie in the range from 0 to the maximum value defined by Table 8.

If the value of `index` is 0, then the frame rate numerator and denominator shall be individually defined by unsigned integer values. When given, the frame rate numerator and denominator values shall both be at least 1.

For values >0, the process `preset_frame_rate(video_parameters, index)` shall set the frame rate elements in `video_parameters` according to Table 8.

Table 8 — Preset frame rate values.

Frame Rate Index	Frame Rate Numerator	Frame Rate Denominator	Frame Rate (Hz)
1	24000	1001	24/1.001
2	24	1	24
3	25	1	25
4	30000	1001	30/1.001
5	30	1	30
6	50	1	50
7	60000	1001	60/1.001
8	60	1	60
9	15000	1001	15/1.001
10	25	2	25/2
11	48	1	48
12	48000	1001	48/1.001
13	96	1	96
14	100	1	100
15	120000	1001	120/1.001
16	120	1	120

11.4.7 Pixel Aspect Ratio

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

`video_parameters[`*`pixel_aspect_ratio_numer`*`]:video_parameters[`*`pixel_aspect_ratio_denom`*`]`

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

```
pixel_aspect_ratio(state, video_parameters):
    custom_pixel_aspect_ratio_flag = read_bool(state)
    if (custom_pixel_aspect_ratio_flag):
        index = read_uint(state)
        if (index == 0):
            video_parameters[pixel_aspect_ratio_numer] = read_uint(state)
            video_parameters[pixel_aspect_ratio_denom] = read_uint(state)
        else:
            preset_aspect_ratio(video_parameters, index)
```

The syntax is illustrated in Figure 20.

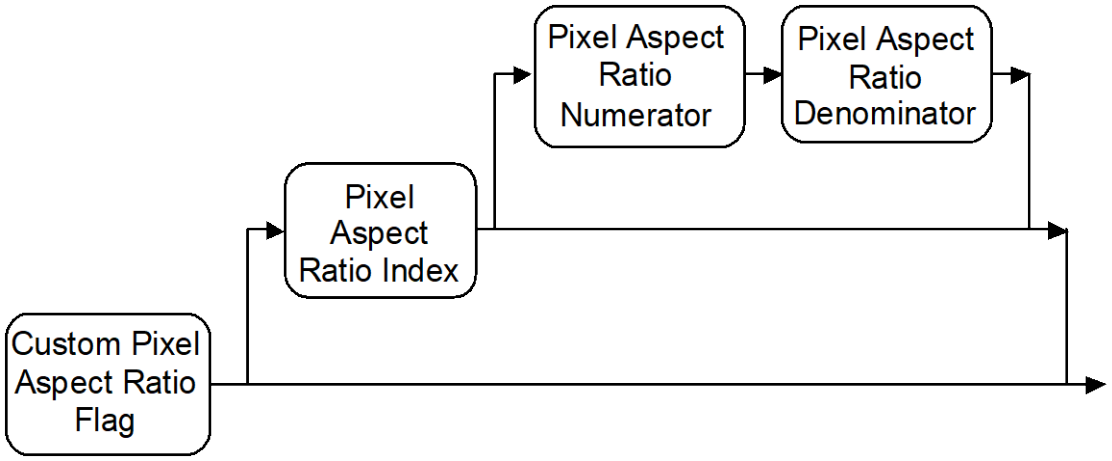

Figure 20 — Pixel aspect ratio.

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

The value of `index` shall lie in the range 0 to 6.

If the value is 0, then the pixel aspect ratio numerator and denominator shall be individually defined by unsigned integer values. When given, the pixel aspect ratio numerator and denominator values shall both be at least 1.

For values >0, the process *`preset_aspect_ratio`*`(video_parameters, index)` shall set the pixel aspect ratio elements in `video_parameters` according to Table 9.

Table 9 — Preset pixel aspect ratio values.

Aspect Ratio Index	Pixel Aspect Ratio
1	1:1
2	10:11 (4:3 525 line systems)
3	12:11 (4:3 625 line systems)
4	40:33 (16:9 525 line systems)
5	16:11 (16:9 625 line systems)
6	4:3 (reduced horizontal resolution systems)

NOTE 1 The pixel aspect ratio value defines the intended ratio of the pixel sampling such that the viewed picture has no geometric distortion. The pixel aspect ratio of an image is the ratio of the spacing of horizontal samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios (PARs) are fundamental properties of sampled images because they determine the displayed shape of objects in the image. Failure to use the right PAR will result in distorted images, for example circles will be displayed as ellipses, etc.

NOTE 2 The pixel aspect ratios shown in Table 9 assume a 704 x 480 active picture for the 525-line systems and a 704 x 576 active picture for the 625-line systems. See also E.4.1.2.

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

11.4.8 Clean Area

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

```
clean_area(state, video_parameters):
    custom_clean_area_flag = read_bool(state)
    if (custom_clean_area_flag):
        video_parameters[clean_width] = read_uint(state)
        video_parameters[clean_height] = read_uint(state)
        video_parameters[left_offset] = read_uint(state)
        video_parameters[left_offset] = read_uint()
        video_parameters[top_offset] = read_uint(state)
        video_parameters[top_offset] = read_uint()
```

The syntax is illustrated in Figure 21.

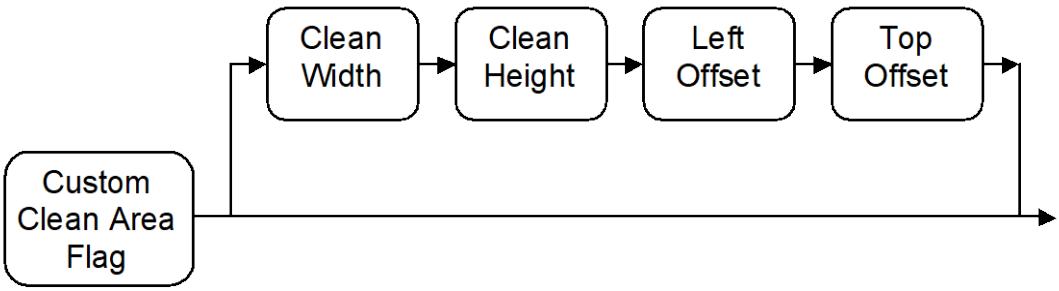

Figure 21 — Clean area.

If `custom_clean_area_flag` is set to **`True`**, the clean area parameters defined by the default values shall be overridden by the new values.

Regardless of how the clean area is defined, the following restrictions shall apply:

$$clean\_width + left\_offset \leq video\_parameters[frame\_width]$$

$$clean\_height + top\_offset \leq video\_parameters[frame\_height]$$

NOTE The meaning and use of clean area is application defined: it might correspond to that part of the picture which is to be displayed, or define a “container” within a picture of larger size.

11.4.9 Signal Range

The signal range parameters indicate how the signal range of the picture component data, decoded by the VC-2 decoder, should be adjusted prior to the color matrixing operations (described in informative Annex E).

The signal range parameters shall also be used to determine the luma depth and color difference depth parameters (see 11.6.3) and the resulting clipping levels applied to the decoded video (see 15.5).

The process for decoding the signal range parameters shall be as follows:

```
signal_range(state, video_parameters):
 custom_signal_range_flag = read_bool(state)
 if (custom_signal_range_flag):
    index = read_uint(state)
    if (index == 0):
       video_parameters[luma_offset] = read_uint(state)
       video_parameters[luma_excursion] = read_uint(state)
       video_parameters[color_diff_offset] = read_uint(state)
       video_parameters[color_diff_excursion] = read_uint(state)
    else:
       preset_signal_range(video_parameters, index)
```

The syntax is illustrated in Figure 22.

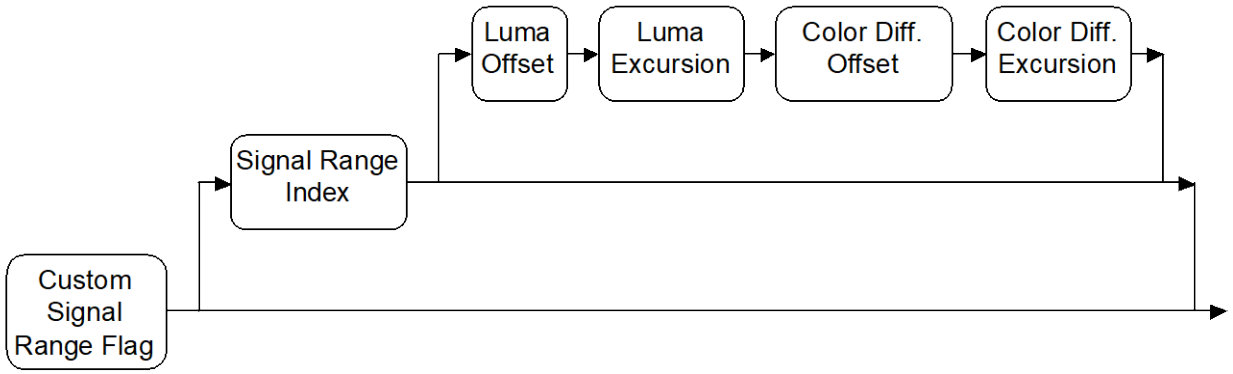

Figure 22 — Signal range.

If `custom_signal_range_flag` is set to **`True`**, the signal ranges defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 8.

If the index value is 0, then the luma and color difference offset and excursion values are individually defined as per the pseudocode in this clause. When given, the luma and color difference excursion values shall both be at least 1.

VC-2 bitstreams that contain a signal range that is not a preset value, should specify a custom signal range (or ranges) within the applicable VC-2 level specification.

For index values >0, the process *preset_signal_range*(video_parameters, index) shall set the signal range elements in video_parameters according to Table 10.

Table 10 — Preset signal ranges.

Index	Description	luma_offset	luma_excursion	color_diff_offset	color_diff_excursion
1	8-bit Full Range	0	255	128	255
2	8-bit Video	16	219	128	224
3	10-bit Video	64	876	512	896
4	12-bit Video	256	3504	2048	3584
5	10-bit Full Range	0	1023	512	1023
6	12-bit Full Range	0	4095	2048	4095
7	16-bit Video	4096	56064	32768	57344
8	16-bit Full Range	0	65535	32768	65535

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

11.4.10 Color Specification

11.4.10.1 General

The color specification shall consist of three component parts:

- Color primaries
- Color matrix
- Transfer function

Defaults are available for all three parts collectively and individually.

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

color_spec(state, video_parameters):	Ref
custom_color_spec_flag = read_bool(state)	
if (custom_color_spec_flag):	
index = read_uint(state)	
preset_color_spec(video_parameters, index)	11.4.10.1
if (index == 0):	
color_primaries(state, video_parameters)	11.4.10.2
color_matrix(state, video_parameters)	11.4.10.3
transfer_function(state, video_parameters)	11.4.10.4

The syntax is illustrated in Figure 23.

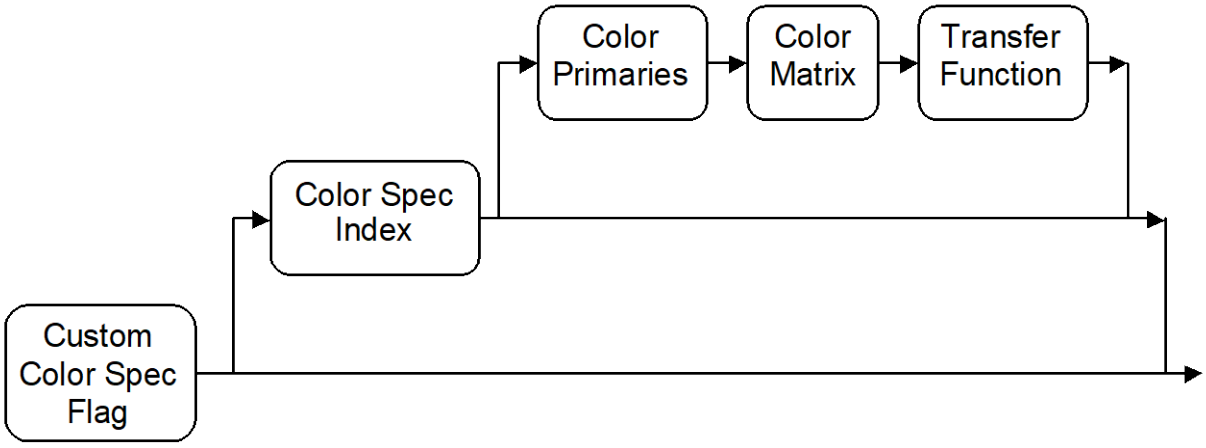

Figure 23 — Color specification.

If `custom_color_spec_flag` is set to **`True`**, the color specification defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 6.

The process *`preset_color_spec`*`(video_parameters, index)` shall set the color specification elements (i.e., color primaries, color matrix and transfer function) in `video_parameters` to be as defined in Table 11.

Table 11 — Preset color specifications.

Index	Description	Color Primaries	Color Matrix	Transfer Function
0	Custom	HDTV	HDTV	TV Gamma
1	SDTV 525	SDTV 525	SDTV	TV Gamma
2	SDTV 625	SDTV 625	SDTV	TV Gamma
3	HDTV	HDTV	HDTV	TV Gamma
4	D-Cinema	D-Cinema	Reversible	D-Cinema
5	UHDTV	UHDTV	UHDTV	TV Gamma
6	HDR-TV PQ	UHDTV	UHDTV	Perceptual Quantization
7	HDR-TV HLG	UHDTV	UHDTV	Hybrid Log Gamma

If the value of `index` is 0, then the color primaries, color matrix and transfer function values may additionally be overridden individually according to 11.4.10.2, 11.4.10.3, and 11.4.10.4.

NOTE The names used for describing the color primaries, color matrix and transfer function values are defined in the tables in 11.4.10.2, 11.4.10.3 and 11.4.10.4.

11.4.10.2 Color Primaries

The color primaries decoding process shall be defined as follows:

```
color_primaries(state, video_parameters):
    custom_color_primaries_flag = read_bool(state)
    if (custom_color_primaries_flag):
        index = read_uint(state)
        preset_color_primaries(video_parameters, index)
```

If `custom_color_primaries_flag` is set to **`True`**, the color primaries defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 4.

The process *`preset_color_primaries`*`(video_parameters, index)` shall set the *`color_primaries_index`* element in `video_parameters`. The meaning of the index shall be as specified in Table 12.

Table 12 — Preset color primaries.

Index	Description	Specification	Comment
0	HDTV	ITU-R BT.709	Also Computer, Web, sRGB
1	SDTV 525	ITU-R BT.601	525 Primaries
2	SDTV 625	ITU-R BT.601	625 Primaries
3	D-Cinema	SMPTE ST 428-1	CIE XYZ
4	UHDTV	ITU-R BT.2020	Used in UHDTV and HDR

11.4.10.3 Color Matrix

The color matrix decoding process shall be defined as follows:

```
color_matrix(state, video_parameters):
    custom_color_matrix_flag = read_bool(state)
    if (custom_color_matrix_flag):
        index = read_uint(state)
        preset_color_matrix(video_parameters, index)
```

If `custom_color_matrix_flag` is set to **`True`**, the color matrix defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 4. The process *`preset_color_matrix`*`(video_parameters, index)` shall set the *`color_matrix_index`* element in `video_parameters`. The meaning of the index shall be as specified in Table 13.

Table 13 — Preset color matrices.

Index	Description	Specification	Color Matrix	Comment (informative)
0	HDTV	ITU-R BT.709	$Y=E'_Y$, $C_1=E'_{CB}$, $C_2=E'_{CR}$	K_R= 0.2126, K_B= 0.0722 Also Computer and Web
1	SDTV	ITU-R BT.601	$Y=E'_Y$, $C_1=E'_{CB}$, $C_2=E'_{CR}$	K_R= 0.299, K_B= 0.114
2	Reversible	ITU-T H.264	$Y=Y$, $C_1=C_G$, $C_2=C_O$	
3	Identity	ITU-R RGB formats SMPTE ST 428-1	$Y=G'$, $C_1=B'$, $C_2=R'$ $Y=CV_Y$, $C_1=CV_Z$, $C_2=CV_X$	
4	UHDTV	ITU-R BT.2020	$Y=Y'$, $C_1=C'_B$, $C_2=C'_R$	K_R=0.2627, K_B=0.0593

11.4.10.4 Transfer Function

The transfer function decoding process shall be defined as follows:

```
transfer_function(state, video_parameters):
    custom_transfer_function_flag = read_bool(state)
    if (custom_transfer_function_flag):
        index = read_uint(state)
        preset_transfer_function(video_parameters, index)
```

If `custom_transfer_function_flag` is set to **`True`**, the transfer function defined by the default values shall be overridden by the new values defined by the index value.

The value of `index` shall lie in the range 0 to 4.

The process *`preset_transfer_function`*`(video_parameters, index)` shall set the *`transfer_function_index`* element in `video_parameters`. The meaning of the index shall be as specified in Table 14.

Table 14 — Preset transfer functions.

Index	Description	Specification
0	TV Gamma	ITU-R BT.2020
1	Extended Gamut (deprecated)	ITU-R BT.1361 (suppressed)
2	Linear	Linear
3	D-Cinema Transfer Function	SMPTE ST 428-1
4	Perceptual Quantization	ITU-R BT.2100
5	Hybrid Log Gamma	ITU-R BT.2100

NOTE The gamma specification in ITU-R BT.2020 for 10-bit signals is equivalent to those in ITU-R BT.601 and ITU-R BT.709.

11.5 Picture Coding Mode

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

If the picture coding mode is 1 then pictures shall correspond to fields; if the picture coding mode is 0, then pictures shall correspond to frames. Values greater than 1 shall be reserved.

If video is coded as fields, then the earliest field in each frame shall have an even picture number (see 12.2). That is, the LSB of the frame number, expressed as a binary number, indicates field parity.

With field coding, each frame shall be split into two fields as indicated by the scan format (see 11.4.5).

An effect of field coding shall be to halve the vertical dimensions of coded pictures. Hence, once the picture coding mode is known, the picture dimensions shall be set and stored as part of the global **`state`** variable.

NOTE It is possible to code progressive video as fields. In this case, the assignment of frame lines to fields will be determined by the value of `top_field_first` in the base video format parameters (Annex B).

According to 11.4.5, this base format default cannot be overridden. Sometimes progressive source video is conveyed as if it were interlaced (for example using interlaced SDI modes), and could be signaled as such. This is known as progressive segmented frames. A VC-2 encoder could detect progressive segmented frames, and signal video as progressive, yet still code the video as fields in order to introduce no additional buffering delay in the signal chain. Or it could take the signaled video format at face value.

11.6 Initializing Coding Parameters

11.6.1 General

The *`set_coding_parameters`*`()` process shall initialize the dimensions of the coded picture and the video depth (the maximum number of bits in a decoded video sample), which are needed to decode pictures. Picture dimensions and video depth shall remain constant throughout a VC-2 sequence and shall be initialized as elements of the decoder **`state`**.

Initialization of the coding parameters shall be defined as follows:

`set_coding_parameters``(state, video_parameters):`	**Ref**
`picture_dimensions(state, video_parameters)`	11.6.2
`video_depth(state, video_parameters)`	11.6.3

11.6.2 Picture Dimensions

The *picture_dimensions()* process, which determines the size of coded pictures, shall be defined by:

```
picture_dimensions(state, video_parameters):
    state[luma_width] = video_parameters[frame_width]
    state[luma_height] = video_parameters[frame_height]
    state[color_diff_width] = state[luma_width]
    state[color_diff_height] = state[luma_height]
    if (video_parameters[color_diff_format_index] == 1):
        state[color_diff_width] //= 2
    if (video_parameters[color_diff_format_index] == 2):
        state[color_diff_width] //= 2
        state[color_diff_height] //= 2
    if (state[picture_coding_mode] == 1):
        state[luma_height] //= 2
        state[color_diff_height] //= 2
```

NOTE The parameter frame_height in video_parameters refers to the height of a frame. The parameter luma_height in **state** refers to the height of a picture. A picture will be either a frame or a field depending on whether it is being coded in an interlaced or progressive mode.

The value of frame_height shall be an integer multiple of color_diff_height and the value of frame_width shall be an integer multiple of color_diff_width.

11.6.3 Video Depth

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

```
video_depth(state, video_parameters):
    state[luma_depth] = intlog2(video_parameters[luma_excursion] + 1)
    state[color_diff_depth]
        = intlog2(video_parameters[color_diff_excursion] + 1)
```

VC-2 bitstreams that contain a signal range that is not a preset value should specify the video depth value within a VC-2 level specification.

NOTE For the YC_GC_O format, the luma and color difference depths will be different.

12 Picture Syntax

12.1 General

Clause 12 defines the syntax of VC-2 picture data units.

Picture data can be parsed after parsing a sequence header within the same VC-2 sequence. The picture parsing process shall be defined as follows:

picture_parse(state):	**Ref**
byte_align(state)	
picture_header(state)	12.2
byte_align(state)	
wavelet_transform(state)	12.3

The syntax is illustrated in Figure 24.

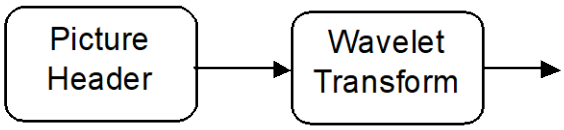

Figure 24 — VC-2 picture.

12.2 Picture Header

The picture header shall immediately follow a parse info header that has a picture parse code (see 10.5.2). The picture header parsing process shall be defined as follows:

picture_header(state):
state[*picture_number*] = read_uint_lit(state, 4)

Picture numbers within a sequence shall increment by one for each successive picture and shall wrap back to zero if the picture number would exceed $2^{32} - 1$. Decoders shall not rely on a continuously incrementing number across sequence boundaries.

If the video is coded as fields, then the earliest field of each frame shall have an even picture number.

NOTE Various operations, such as editing, can result in a discontinuity of picture number values between sequences within a VC-2 stream. Although encoders might reasonably be expected to create a stream with incrementing picture number values, a byte-stream could be edited and spliced, thus breaking up the original sequence of incrementing values. In this case, either a new sequence ought to be started within the stream or picture numbers rationalized.

12.3 Wavelet Transform

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

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

wavelet_transform(state):	**Ref**
transform_parameters(state)	12.4
byte_align(state)	
transform_data(state)	13

The syntax is illustrated in Figure 25.

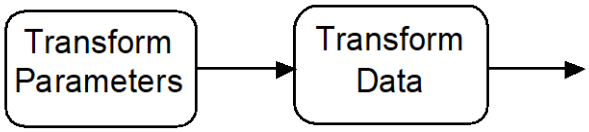

Figure 25 — Wavelet transform data.

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

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

12.4 Transform Parameters

12.4.1 General

The wavelet transform parameters shall define the metadata required to configure the inverse wavelet transform. The *transform_parameters()* process shall be as follows:

transform_parameters(state):	**Ref**
state[*wavelet_index*] = read_uint(state)	12.4.2
state[*dwt_depth*] = read_uint(state)	12.4.3
state[*wavelet_index_ho*] = state[*wavelet_index*]	12.4.4.2
state[*dwt_depth_ho*] = 0	12.4.4.3
if (state[*major_version*] >= 3):	
extended_transform_parameters(state)	12.4.4
slice_parameters(state)	12.4.5.2
quant_matrix(state)	12.4.5.3

The syntax is illustrated in Figure 26.

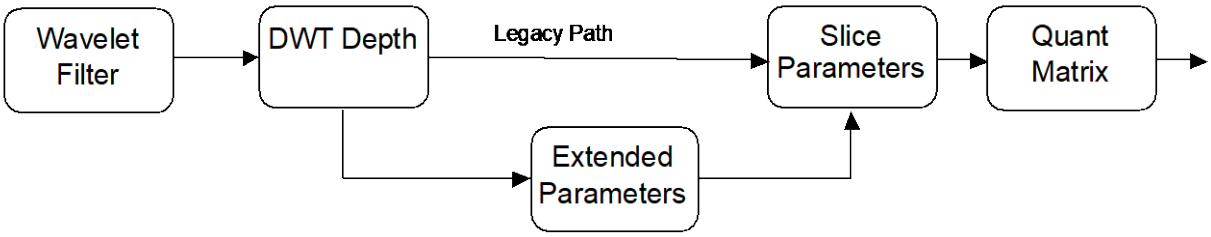

Figure 26 — Transform parameters.

12.4.2 Wavelet Filter

The wavelet filter parameter shall define the wavelet filter used by the VC-2 stream.

The value of **state**[wavelet_index] shall lie in the range 0 to 6 with values as defined in Table 15.

Table 15 — VC-2 wavelet filters.

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

The implementation of the chosen wavelet filter shall be as defined in 15.4.4.

NOTE For consistency, the filter nomenclature (*m, n*) refers to the length of the analysis low-pass and high-pass filters in the conventional pre-filtering (i.e., before subsampling) model of wavelet filtering (see Annex F). They do not reflect the length of lifting filters, which operate in the subsampled domain: see 15.4. Deslauriers-Dubuc filters are normally referred to in terms of the number of vanishing moments of their synthesis filters, so the (9,7) and (13,7) filters are referred to in the literature as (2,2) and (4,2) filters, respectively.

12.4.3 Transform Depth

The transform depth parameter **state**[dwt_depth] shall determine the number of stages in the two-dimensional portion of the wavelet transform.

NOTE A given transform depth and horizontal only transform depth (see 12.4.4.3) value defines the number of subbands and thus the dimensions of the subband data array (see 13.2).

12.4.4 Extended Transform Parameters

12.4.4.1 General

The extended transform parameters shall define additional metadata for features added in version 3 of this specification. Sequences compliant with versions 1 or 2 of this specification do not include these parameters. The use or non-use of these parameters is determined by the major version indicated in the Sequence Headers. The *extended_transform_parameters()* process shall be as follows:

extended_transform_parameters(state):	Ref
asym_transform_index_flag = read_bool(state)	
if (asym_transform_index_flag):	
state[*wavelet_index_ho*] = read_uint(state)	12.4.4.2
asym_transform_flag = read_bool(state)	
if (asym_transform_flag):	
state[*dwt_depth_ho*] = read_uint(state)	12.4.4.3

The syntax is illustrated in Figure 27.

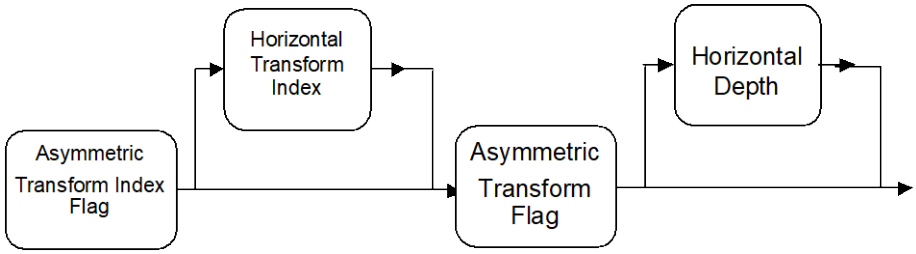

Figure 27 — Extended transform parameters.

12.4.4.2 Horizontal Specific Transform Index

The horizontal transform index parameter **state**[wavelet_index_ho] shall determine the wavelet filter used by the VC-2 stream for the horizontal stages. If not set explicitly then this value defaults to **state**[wavelet_index].

12.4.4.3 Horizontal Only Transform Depth

The horizontal only transform depth parameter **state**[dwt_depth_ho] shall determine the number of additional horizontal levels of transform have been applied after the two-dimensional portion of the discrete wavelet transform. If not set explicitly then this value defaults to 0.

NOTE If **state**[dwt_depth_ho] is 0 and **state**[wavelet_index_ho] is **state**[wavelet_index] then the inverse wavelet transform process (see 13.2) is identical to that defined in earlier versions of this specification.

12.4.5 Slice Coding Parameters

12.4.5.1 General

Slice coding provides for compression of small parts (slices) of a picture in order to reduce delay.

12.4.5.2 and 12.4.5.3 define the number of slices (horizontally and vertically), slice size (low delay pictures only) and quantization matrix for VC-2 slices.

12.4.5.2 Slice Parameters

The *slice_parameters()* process for extracting slice parameter values shall be defined as follows:

```
slice_parameters(state):
    state[slices_x] = read_uint(state)
    state[slices_y] = read_uint(state)
    if (is_ld(state)):
        state[slice_bytes_numerator] = read_uint(state)
        state[slice_bytes_denominator] = read_uint(state)
    if (is_hq(state)):
        state[slice_prefix_bytes] = read_uint(state)
        state[slice_size_scaler] = read_uint(state)
```

The syntax is illustrated in Figure 28.

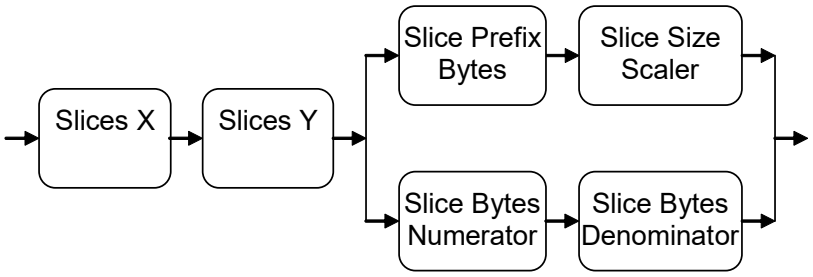

Figure 28 — Slice parameters.

The number of horizontal slices, `state[slices_x]`, and the number of vertical slices, `state[slices_y]`, shall each be at least 1.

For low delay pictures, the slice bytes denominator shall be at least 1 and the slice bytes numerator shall be at least as large as the denominator.

For high quality pictures, the slice size scaler shall be at least 1.

NOTE The sizes of the slice parameters are unbounded. These values will be constrained by other documents that define additional VC-2 levels.

12.4.5.3 Quantization Matrices

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

The *quant_matrix()* process for extracting quantization matrix values shall be defined as follows:

quant_matrix(state):	Ref
custom_quant_matrix = read_bool(state)	
if (custom_quant_matrix):	
state[*quant_matrix*] = new_array(state[*dwt_depth_ho*] + state[*dwt_depth*] + 1)state[*dwt_depth*] + 1)	
if (state[*dwt_depth_ho*] == 0):	
state[*quant_matrix*][0] = {}	
state[*quant_matrix*][0][*LL*] = read_uint(state)	
else:	
state[*quant_matrix*][0] = {}	
state[*quant_matrix*][0][*L*] = read_uint(state)	
for level = 1 **to** state[*dwt_depth_ho*]:	
state[*quant_matrix*][level] = {}	
state[*quant_matrix*][level][*H*] = read_uint(state)	
for level = state[*dwt_depth_ho*] + 1 **to** state[*dwt_depth_ho*] + state[*dwt_depth*]:	
state[*quant_matrix*][level] = {}	
state[*quant_matrix*][level][*HL*] = read_uint(state)	
state[*quant_matrix*][level][*LH*] = read_uint(state)	
state[*quant_matrix*][level][*HH*] = read_uint(state)	
else:	
set_quant_matrix(state)	12.4.5.3, Annex D

In the following cases, *custom_quant_matrix* shall be set to **True**:

- **state**[*dwt_depth*] > 4
- **state**[*dwt_depth_ho*] > 4
- **state**[*dwt_depth_ho*] + **state**[*dwt_depth*] > 5
- **state**[*wavelet_index_ho*] != **state**[*wavelet_index*] and **state**[*wavelet_index_ho*] != 1
- **state**[*wavelet_index_ho*] != **state**[*wavelet_index*] and **state**[*wavelet_index*] != 3

Otherwise *custom_quant_matrix* may be set to **True**, for example to apply a different degree of perceptual weighting (see Annex D).

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

13 Transform Data Syntax

13.1 General

Clause 13 the process for unpacking (parsing, entropy decoding and inverse quantizing) wavelet transform coefficient data from the VC-2 stream. Wavelet coefficients shall be signed integer values and shall be extracted using the integer VLC functions listed in 9.2.4 and defined in Annex A.

The result of this process shall be a set of fully populated wavelet subband data arrays, as defined in 13.2.

Coefficients shall be grouped into slices that represent coefficients pertaining to an area of the picture. Each slice shall contain data for all video components and spatial frequency bands. Unpacking a slice allows an area of picture to be extracted without extracting (or even receiving) the remaining picture data.

The overall process for unpacking transform data is described in 13.5.2.

Unpacked wavelet coefficient data shall be stored in the state variables: **state**[y_transform], **state**[c1_transform] and **state**[c2_transform] for the IDWT process (see 15.3).

13.2 Subband Data Structure

13.2.1 General

Subband data shall be ordered by level (0, 1, 2, 3, etc.) and orientation (L, H, LL, HL, LH and HH).

The total number of levels is equal to **state**[dwt_depth_ho] + **state**[dwt_depth].

If the transform is symmetrical (i.e., **state**[dwt_depth_ho] is 0) then in level 0, only the LL orientation shall be available (known also as the DC band); in the other levels only the HL, LH and HH orientations shall be available.

If the transform is asymmetrical (i.e., **state**[dwt_depth_ho] > 0) then in level 0 only the L orientation shall be available; in levels from 1 to **state**[dwt_depth_ho] only the H orientation shall be available; and in the other levels only the HL, LH and HH orientations shall be available.

The DC subband shall be presented first in each slice. Within each subband depth level with multiple subbands the orientation order shall be x-*HL*, x-*LH*, x-*HH* (where 'x' is the transform depth level).

The subbands shall partition the spatial frequencies by orientation and level so that a 2+2-level subband array is as illustrated in Figure 29.

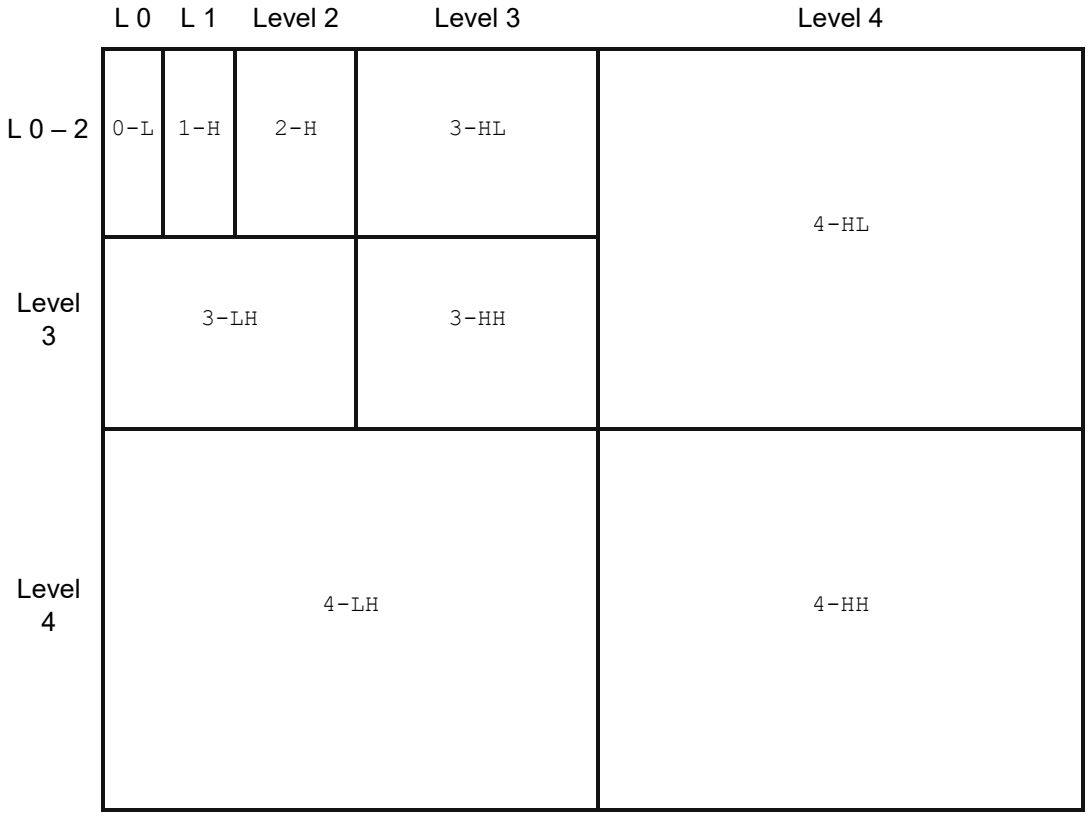

Figure 29 — (2+2)-level subband array.

NOTE: The coefficient data is effectively a four-dimensional array of maps of arrays comprising the two dimensions *level* and *orientation* for a given subband and a further two dimensions for the H and V coefficient array within each subband.

13.2.2 Wavelet Data Initialization

This clause defines the *initialize_wavelet_data(*`state, comp`*)* process, which returns a structure ready to contain the wavelet coefficients for each picture component indicated by `comp`.

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

```
band = coeff_data[level][orient]
```

Valid levels shall be integer values in the range `0` to **`state`**`[dwt_depth_ho] + `**`state`**`[dwt_depth]` inclusive.

If **`state`**`[dwt_depth_ho]` is 0 then level 0 shall consist of a single subband with orientation LL. All other levels shall consist of 3 subbands with orientation HL, LH, and HH in that order within the VC-2 stream.

If **state**[dwt_depth_ho] > 0 then level 0 shall consist of a single subband with orientation L; the subbands from 1 to **state**[dwt_depth_ho] inclusive shall consist of a single subband with orientation H; and all other levels shall consist of 3 subbands with orientation HL, LH, and HH in that order within the VC-2 stream.

Each subband array shall be initialized so that:

width(coeff_data[level][orient]) = *subband_width*(state, level, comp)

height(coeff_data[level][orient]) = *subband_height*(state, level, comp)

as defined in 13.2.3. These dimensions shall correspond to a wavelet transform being performed on a copy of the component data which has been padded (if necessary) so that its dimensions are a multiple of $2^{(\textbf{state}[\text{dwt\_depth\_ho}] + \textbf{state}[\text{dwt\_depth}])}$.

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

c = coeff_data[level][orient][y][x]

for coordinates (x, y) with:

0 <= x < *subband_width*(state, level, comp) and

0 <= y < *subband_height*(state, level, comp).

13.2.3 Wavelet Subband Dimensions

This clause defines the values of the *subband_width*(state, level, comp) and *subband_height*(state, level, comp) coefficients, giving the width and height of subbands at a given level for a given component, and hence the range of subband vertical and horizontal indices.

```
Subband_width(state, level, comp):
    if (comp == Y):
        w = state[luma_width]
    else:
        w = state[color_diff_width]
    scale_w = 2 ** (state[dwt_depth_ho] + state[dwt_depth])
    pw = scale_w * ((w + scale_w - 1) // scale_w)
    if (level == 0):
        return pw // 2 ** (state[dwt_depth_ho] + state[dwt_depth])
    else:
        return pw // 2 ** (state[dwt_depth_ho] + state[dwt_depth] - level + 1)
```

```
subband_height(state, level, comp):
    if (comp == Y):
        h = state[luma_height]
    else:
        h = state[color_diff_height]
    scale_h = 2 ** (state[dwt_depth])
    ph = scale_h * ((h + scale_h - 1) // scale_h)
    if (level <= state[dwt_depth_ho]):
        return ph // 2 ** (state[dwt_depth])
    else:
        return ph // 2 ** (state[dwt_depth_ho] + state[dwt_depth] - level + 1)
```

NOTE In the encoding process, these padded dimensions can be achieved by padding the video component data up to the padded dimensions and applying the forward Discrete Wavelet Transform (the inverse of the operations defined in 15.4). Any values can 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, are obtained by using edge extension. A poor choice of padding can cause visible artifacts near the bottom and right edges at high levels of compression.

13.3 Inverse Quantization

13.3.1 General

13.3 defines the operation of inverse quantization, which scales the magnitude of unpacked wavelet coefficients according to a predetermined factor.

The *inverse_quant*() function shall be as defined as follows:

inverse_quant(quantized_coeff, quant_index):	**Ref**
magnitude = abs(quantized_coeff)	
if (magnitude != 0):	
magnitude *= quant_factor(quant_index)	13.3.2
magnitude += quant_offset(quant_index)	13.3.2
magnitude += 2	
magnitude //= 4	
return sign(quantized_coeff) * magnitude	

NOTE 1 VC-2 quantization is an integer approximation of dead-zone quantization, in which a value `x` is quantized as

`|x|//qf` for `x>=0`, or

`-(|x|//qf)` for `x<0`.

Since this process involves rounding down, the inverse quantization process adds an offset to non-zero values after multiplying up by `qf`. This produces a reconstruction value closer on average to the original value.

NOTE 2 The pseudocode description separates inverse quantization from decoding. However, since dead-zone quantization is used, the `inverse_quant()` function has to compute the magnitude. Hence it is more efficient to first decode the coefficient magnitude, then inverse quantize, and then decode the coefficient sign.

NOTE 3 For low delay pictures the quantization index is limited to 7 bits (i.e., a maximum value of 127), and for high quality pictures it is limited to 8 bits. These limits are sufficient for most practical scenarios.

13.3.2 Quantization Factors and Offsets

This clause defines the operation of the *`quant_factor()`* and *`quant_offset()`* functions for performing inverse quantization.

Quantization factors shall be determined as follows:

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

The quantization offsets shall be determined as follows:

```
quant_offset(index):
   if (index == 0):
      offset = 1
   else if (index == 1):
      offset = 2
   else:
      offset = (quant_factor(index) + 1) // 2
   return offset
```

The value of `index` passed to the *quant_offset()* and *quant_factor()* functions shall be greater than or equal to zero.

13.4 DC Subband Prediction

This clause defines the operation of the *dc_prediction*(band) process for reconstructing values within DC (0-L or 0-LL) bands using spatial prediction. This function shall be applied when DC subband prediction is specified for the profile being used.

NOTE High quality pictures do not use DC subband prediction.

This function can be applied once all coefficients within the DC band have been inverse quantized, or it can be applied progressively to each coefficient as soon as it has been inverse quantized.

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

The DC subband prediction process shall be defined as follows:

```
dc_prediction(band):
   for y = 0 to height(band) - 1:
      for x = 0 to width(band) - 1:
         if (x > 0 and y > 0):
            prediction
               = mean(band[y][x - 1], band[y - 1][x - 1], band[y - 1][x])
         else if (x > 0 and y == 0):
            prediction = band[0][x - 1]
         else if (x == 0 and y > 0):
            prediction = band[y - 1][0]
         else:
            prediction = 0
         band[y][x] += prediction
```

13.5 VC-2 Wavelet Coefficient Unpacking

13.5.1 General

Clause 13.5 defines the stream syntax that is used for VC-2 streams. Clause 13.5 defines the syntax and parsing operations only; picture decoding operations are defined in Clause 15.

The VC-2 syntax shall partition the wavelet coefficients into a number of slices, from all subbands, corresponding to localized areas of the picture.

A slice shall meet the following provisions:

1. A single quantizer, weighted for each subband by a quantization matrix, shall be used for quantization of the coefficients in each slice.
2. All wavelet coefficients shall be entropy-coded using variable-length coding.
3. In an LD Picture, the number of bytes used per slice shall be the same, to within one byte, for each slice in the picture. For an HQ Picture this is not required.
4. Each picture may change the slice parameters within the picture by setting the relevant wavelet transform parameters (see 12.4.5).

NOTE 1 The slice structure implies that in practice incremental picture decoding can be easily achieved without accumulating an entire picture data set, yielding a decoding delay proportional to the height of the slices (the actual achievable delay could be more than one slice height due to vertical filtering delay).

NOTE 2 Using a fixed number of bits per slice does have impact on compression efficiency but simplifies both encoder and decoder hardware, and assists a chain of multiple encoders and decoders using the same slice parameters in producing identical coding decisions and hence no cascading loss. These factors are of great significance in a professional environment.

13.5.2 Overall Process

The transform data process shall be defined as follows:

transform_data(state):	Ref
state[*y_transform*] = initialize_wavelet_data(state, *Y*)	13.2.2
state[*c1_transform*] = initialize_wavelet_data(state, *C1*)	13.2.2
state[*c2_transform*] = initialize_wavelet_data(state, *C2*)	13.2.2
for sy = 0 **to** state[*slices_y*]-- 1:	
for sx = 0 **to** state[*slices_x*]-- 1:	
slice(state, sx, sy)	13.5.2
if (using_dc_prediction(state)):	10.5.2
if (state[*dwt_depth_ho*] == 0):	
dc_prediction(state[*y_transform*][0][*LL*])	13.4
dc_prediction(state[*c1_transform*][0][*LL*])	13.4
dc_prediction(state[*c2_transform*][0][*LL*])	13.4
else:	
dc_prediction(state[*y_transform*][0][*L*])	13.4
dc_prediction(state[*c1_transform*][0][*L*])	13.4
dc_prediction(state[*c2_transform*][0][*L*])	13.4

NOTE 1 Low delay picture slices cannot be decoded in isolation since DC values at the top and left edges of a slice are predicted from DC values in previously decoded slices, which therefore need to be retained. High quality picture slices can be decoded separately.

slice(state, sx, sy):	Ref
if (is_ld(state)):	Table 5
ld_slice(state, sx, sy)	13.5.3
else if (is_hq(state)):	Table 5
hq_slice(state, sx, sy)	13.5.4

NOTE 2 A key difference between low delay and high quality pictures is the way slices are coded in the bitstream. Once the complete, inverse quantized, wavelet transform has been extracted from the stream, decoding is the same for all profiles.

13.5.3 Slice Unpacking for Low Delay Pictures

13.5.3.1 General

This clause defines the operation of the *ld_slice*(state, sx, sy) process for parsing a low delay picture slice with coordinates (sx, sy). Each slice shall contain the relevant coefficients from all subbands and for all components. Luma data shall be unpacked first, and shall be followed by the color difference data, in which the color difference coefficients shall be interleaved. A length code shall allow the luma and color difference coefficients each to be terminated early, with remaining values set to zero by means of bounded read operations.

The overall slice unpacking process shall be defined as follows:

ld_slice(state, sx, sy):	Ref
slice_bits_left = 8 * slice_bytes(state, sx, sy)	13.5.3.2
qindex = read_nbits(state, 7)	
slice_bits_left -= 7	
slice_quantizers(state, qindex)	13.5.5
length_bits = intlog2((8 * slice_bytes(state, sx, sy))-- 7)	
slice_y_length = read_nbits(state, length_bits)	
slice_bits_left -= length_bits	
state[*bits_left*] = slice_y_length	
if (state[*dwt_depth_ho*] == 0):	
slice_band(state, *y_transform*, 0, *LL*, sx, sy)	13.5.6.3
for level = 1 **to** state[*dwt_depth*]:	
for each orient **in** *HL*, *LH*, *HH*:	
slice_band(state, *y_transform*, level, orient, sx, sy)	13.5.6.3
else:	
slice_band(state, *y_transform*, 0, *L*, sx, sy)	13.5.6.3
for level = 1 **to** state[*dwt_depth_ho*]:	
slice_band(state, *y_transform*, level, *H*, sx, sy)	13.5.6.3
for level = state[*dwt_depth_ho*] + 1 **to** state[*dwt_depth_ho*] + state[*dwt_depth*]:	
for each orient **in** *HL*, *LH*, *HH*:	
slice_band(state, *y_transform*, level, orient, sx, sy)	13.5.6.3
flush_inputb(state)	A.4.2
slice_bits_left -= slice_y_length	
state[*bits_left*] = slice_bits_left	
if (state[*dwt_depth_ho*] == 0):	
color_diff_slice_band(state, 0, *LL*, sx, sy)	13.5.6.4
for level = 1 **to** state[*dwt_depth*]:	

`for each orient in HL, LH, HH:`	13.5.6.4
`color_diff_slice_band(state, level, orient, sx, sy)`	
else:	
`color_diff_slice_band(state, 0, L, sx, sy)`	13.5.6.4
`for level = 1 to state[dwt_depth_ho]:`	
`color_diff_slice_band(state, level, H, sx, sy)`	13.5.6.4
`for level = state[dwt_depth_ho] + 1 to state[dwt_depth_ho] + state[dwt_depth]:`	
`for each orient in HL, LH, HH:`	
`color_diff_slice_band(state, level, orient, sx, sy)`	13.5.6.4
`flush_inputb(state)`	A.4.2

`slice_y_length` shall satisfy the following condition:

```
slice_y_length <= 8 * slice_bytes(state, sx, sy)-- 7-- length_bits
```

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

13.5.3.2 Determining the Number of Bytes in a Low Delay Picture Slice

The *slice_bytes*() function shall be defined as follows:

slice_bytes`(state, sx, sy):`
`slice_number = sy * state[slices_x] + sx`
`bytes = ((slice_number + 1) * state[slice_bytes_numerator]) // state[slice_bytes_denominator]`
`bytes -= (slice_number * state[slice_bytes_numerator]) // state[slice_bytes_denominator]`
return bytes

NOTE This function produces an integer value which will, on average, be the ratio of **state**[*slice_bytes_numerator*] to **state**[*slice_bytes_denominator*]. In many applications, this ratio will not be an integer number of bytes, and the number of bytes per slice will vary by 1 byte from time to time. This allows a low delay picture to support any rational compression ratio exactly, so that a picture can be compressed to an arbitrary number of bytes.

13.5.4 Slice Unpacking for High Quality Pictures

This clause defines the operation of the *hq_slice*(state, sx, sy) process for parsing a high quality picture slice with coordinates (sx, sy). Each slice shall contain the relevant coefficients from all subbands and for all components. Each slice shall start with zero or more slice_prefix_bytes (see 12.4.5.2) followed by a single byte specifying the common quantization index. These are followed by the color components, which shall be unpacked separately and in the order Y, C_1, C_2 (or G, B, R). Each component shall commence with a one byte length code which, multiplied by the slice_size_scaler (see 12.4.5.2), specifies the number of following bytes occupied by that component's transform coefficients. The length code shall allow the sequence of component coefficients to be terminated early, with remaining values set to zero by means of bounded read operations.

A decoder may skip over the contents of the slice_prefix_bytes or it may read the data and use it for an application-specific purpose, out of scope of this standard.

NOTE The use of a separate length code for each color component allows the choice of constant or variable bit rate coding and lossless coding.

The overall slice unpacking process shall be defined as follows:

hq_slice(state, sx, sy):	Ref
read_uint_lit(state, state[*slice_prefix_bytes*])	12.4.5.2
qindex = read_uint_lit(state, 1)	
slice_quantizers(state, qindex)	13.5.5
for each transform **in** *y_transform*, *c1_transform*, *c2_transform*:	
length = state[*slice_size_scaler*] * read_uint_lit(state, 1)	
state[*bits_left*] = 8 * length	
if (state[*dwt_depth_ho*] == 0):	
slice_band(state, transform, 0, *LL*, sx, sy)	13.5.6.3
for level = 1 **to** state[*dwt_depth*]:	
for each orient **in** *HL*, *LH*, *HH*:	
slice_band(state, transform, level, orient, sx, sy)	13.5.6.3
else:	
slice_band(state, transform, 0, *L*, sx, sy)	13.5.6.3
for level = 1 **to** state[*dwt_depth_ho*]:	
slice_band(state, transform, level, *H*, sx, sy)	13.5.6.3
for level = state[*dwt_depth_ho*] + 1 **to** state[*dwt_depth_ho*] + state[*dwt_depth*]:	
for each orient **in** *HL*, *LH*, *HH*:	
slice_band(state, transform, level, orient, sx, sy)	13.5.6.3
flush_inputb(state)	A.4.2

13.5.5 Setting Slice Quantizers

This clause defines how quantizers for individual subbands shall be determined from the quantization matrix and the quantization index. The *slice_quantizers*() process shall be defined as follows:

```
slice_quantizers(state, qindex):
    state[quantizer] = new_array(state[dwt_depth_ho] + state[dwt_depth] + 1)
    if (state[dwt_depth_ho] == 0):
        state[quantizer][0] = {}
        state[quantizer][0][LL] = max(qindex-- state[quant_matrix][0][LL], 0)
        for level = 1 to state[dwt_depth]:
            state[quantizer][level] = {}
            for each orient in HL, LH, HH:
                qval = max(qindex-- state[quant_matrix][level][orient], 0)
                state[quantizer][level][orient] = qval
    else:
        state[quantizer][0] = {}
        state[quantizer][0][L] = max(qindex-- state[quant_matrix][0][L], 0)
        for level = 1 to state[dwt_depth_ho]:
            state[quantizer][level] = {}
            qval = max(qindex-- state[quant_matrix][level][H], 0)
            state[quantizer][level][H] = qval
        for level = state[dwt_depth_ho] + 1 to state[dwt_depth_ho] +
          state[dwt_depth]:
            state[quantizer][level] = {}
            for each orient in HL, LH, HH:
                qval = max(qindex-- state[quant_matrix][level][orient], 0)
                state[quantizer][level][orient] = qval
```

NOTE The non-negative quantization matrix values are subtracted from the slice quantizer value, and so a higher value in the quantization matrix represents a lower quantization index and thus a lower degree of quantization. The quantization index value is also clipped to 0 so that it is non-negative. This ensures that as many subbands as possible within the slice can be coded losslessly.

13.5.6 Slice Subbands

13.5.6.1 General

13.5.6 defines the operations for unpacking slices. For low delay pictures, the functions `slice_band(state, transform, level, orient, sx, sy)` and `color_diff_slice_band(state, level, orient, sx, sy)` shall be used for unpacking luma and color difference slice subbands respectively. For high quality pictures, the function `slice_band(state, transform, level, orient, sx, sy)` shall be used for unpacking all component slice subbands.

13.5.6.2 Slice Subband Area

The rectangular set of coefficients covered by a slice component (Y, C_1 and C_2) with index `(sx,sy)` is demarcated by the values *slice_left, slice_right, slice_top, slice_bottom*, defined as fractions of the subband dimensions by the following functions:

```
slice_left(state, sx, c, level):
    return (subband_width(state, level, c) * sx) // state[slices_x]
```

```
slice_right(state, sx, c, level):
    return (subband_width(state, level, c) * (sx + 1)) // state[slices_x]
```

```
slice_top(state, sy, c, level):
    return (subband_height(state, level, c) * sy) // state[slices_y]
```

```
slice_bottom(state, sy, c, level):
    return (subband_height(state, level, c) * (sy + 1)) // state[slices_y]
```

where `c` is Y for luma coefficients, and C1 or C2 for color difference coefficients.

NOTE 1 Slice subbands can change dimension by 1 from one slice to another if **state**[slices_x] or **state**[slices_y] do not divide the horizontal or vertical dimensions exactly.

NOTE 2 Color difference slice subbands might not have exactly the scaled dimensions of the luma slice subband, since the **state**[slices_x] and **state**[slices_y] values could exactly divide luma dimensions but not color difference dimensions; and color difference components might receive different padding, depending on the transform depth.

NOTE 3 These issues can easily be avoided in particular applications by choosing suitable values for the transform depth and **state**[slices_x] and **state**[slices_y].

13.5.6.3 Single Component Slice Subband Data

The process for unpacking slice coefficients for a single color component shall be defined as follows:

slice_band(state, transform, level, orient, sx, sy):	Ref
if (transform == *y_transform*):	
comp = *Y*	
else if (transform == *c1_transform*):	
comp = *C1*	
else if (transform == *c2_transform*):	
comp = *C2*	
qi = state[*quantizer*][level][orient]	

for y = slice_top(state, sy, comp, level) **to** slice_bottom(state, sy, comp, level)-- 1:	
for x = slice_left(state, sx, comp, level) **to** slice_right(state, sx, comp, level)-- 1:	
val = read_sintb(state)	A.4.4
state[transform][level][orient][y][x] = inverse_quant(val, qi)	13.3

NOTE “transform” can be one of y_transform, c1_transform or c2_transform.

13.5.6.4 Color Difference Slice Subband Data

Color difference slice subband coefficients shall follow luma coefficients within each slice. The two color difference components shall be interleaved coefficient by coefficient. The process for unpacking color difference slice coefficients shall be defined as follows:

color_diff_slice_band(state, level, orient, sx, sy):	**Ref**
qi = state[*quantizer*][level][orient]	
for y = slice_top(state, sy, *C1*, level) **to** slice_bottom(state, sy, *C1*, level)-- 1:	
for x = slice_left(state, sx, *C1*, level) **to** slice_right(state, sx, *C1*, level)-- 1:	
val = read_sintb(state)	A.4.4
state[*c1_transform*][level][orient][y][x] = inverse_quant(val, qi)	13.3
val = read_sintb(state)	A.4.4
state[*c2_transform*][level][orient][y][x] = inverse_quant(val, qi)	13.3

14 Fragment Syntax

14.1 General

Clause 14 the syntax and parsing of VC-2 fragment data units.

Fragment data can be parsed after parsing a sequence header within the same VC-2 sequence. The fragment parsing process shall be defined as follows:

fragment_parse(state):	**Ref**
fragment_header(state)	14.2
if (state[*fragment_slice_count*] == 0):	
transform_parameters(state)	12.4
initialize_fragment_state(state)	14.3
else:	
fragment_data(state)	14.4

Fragments come in two types: setup fragments and data fragments. A fragmented picture shall be coded as a setup fragment followed by one or more data fragments. A setup fragment shall contain a fragment slice count of zero and shall contain only transform parameters. A data fragment shall have a non-zero fragment slice count and carry fragment slice count consecutive picture slices. A fragmented picture is complete once all picture slices have been encoded in data fragments.

A setup fragment shall not be followed by any further setup fragments or non-fragmented picture data units until the fragmented picture is complete. A sequence shall not end while a fragmented picture is incomplete.

14.2 Fragment Header

The fragment header shall immediately follow a parse info header that has a fragment parse code (see 10.4.1). The fragment header parsing process shall be defined as follows:

fragment_header(state):
state[*picture_number*] = read_uint_lit(state, 4)
state[*fragment_data_length*] = read_uint_lit(state, 2)
state[*fragment_slice_count*] = read_uint_lit(state, 2)
if (state[*fragment_slice_count*] != 0):
state[*fragment_x_offset*] = read_uint_lit(state, 2)
state[*fragment_y_offset*] = read_uint_lit(state, 2)

Picture numbers within a sequence shall increment by one for each successive setup fragment (i.e., for each picture) and shall wrap back to zero if the picture number would exceed $2^{32} - 1$. Data fragments shall contain the same picture number as their associated setup fragment. Decoders shall not rely on a continuously incrementing number across sequence boundaries.

If the video is coded as fields, then the earliest field of each frame shall have an even picture number.

If the video is coded as fields, then the earliest fragment of the earliest field of each frame shall have an even picture number.

NOTE Various operations, such as editing, can result in a discontinuity of picture number values between sequences within a VC-2 stream. Although encoders might reasonably be expected to create a stream with incrementing picture number values, a byte-stream could be edited and spliced, thus breaking up the original sequence of incrementing values. In this case, either a new sequence ought to be started within the stream or picture numbers rationalized.

The fragment data length field contains undefined data for the purposes of this standard and does not contribute to the decoding process.

Each data fragment shall carry at least one slice and set the fragment x and y offsets to the coordinate of the first slice carried. Slices shall be coded in raster scan order starting with slice (0, 0). Slices shall not be omitted or repeated. Exactly **state**[slices_x] * **state**[slices_y] slices shall be coded for each fragmented picture.

14.3 Initializing Fragment State

The process for initializing the state variables used for fragment reconstruction shall be as follows:

initialize_fragment_state(state):	**Ref**
state[*y_transform*] = initialize_wavelet_data(state, *Y*)	13.2.2
state[*c1_transform*] = initialize_wavelet_data(state, *C1*)	13.2.2
state[*c2_transform*] = initialize_wavelet_data(state, *C2*)	13.2.2
state[*fragment_slices_received*] = 0	
state[*fragmented_picture_done*] = **False**	

14.4 Fragment Data

This clause defines the process for unpacking (parsing, entropy decoding and inverse quantizing) wavelet transform coefficient data from data fragments in the VC-2 stream, making frequent references to Clause 13. The reader is referred to that clause for an informative description of how wavelet data is arranged in a picture. Fragmented pictures contain the same data, in the same order, as non-fragmented pictures but split over several data units.

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

fragment_data(state):	**Ref**
for s = 0 **to** state[*fragment_slice_count*] - 1:	
slice_x = (state[*fragment_y_offset*] * state[*slices_x*] + state[*fragment_x_offset*] + s) % state[*slices_x*]	
slice_y = (state[*fragment_y_offset*] * state[*slices_x*] + state[*fragment_x_offset*] + s) // state[*slices_x*]	
slice(state, slice_x, slice_y)	
state[*fragment_slices_received*] += 1	
if (state[*fragment_slices_received*] == state[*slices_x*] * state[*slices_y*]):	
state[*fragmented_picture_done*] = **True**	
if (using_dc_prediction(state)):	Table 5

if (state[*dwt_depth_ho*] == 0):	
dc_prediction(state[*y_transform*][0][*LL*])	13.4
dc_prediction(state[*c1_transform*][0][*LL*])	13.4
dc_prediction(state[*c2_transform*][0][*LL*])	13.4
else:	
dc_prediction(state[*y_transform*][0][*L*])	13.4
dc_prediction(state[*c1_transform*][0][*L*])	13.4
dc_prediction(state[*c2_transform*][0][*L*])	13.4

NOTE 1 The key difference between fragments and pictures is how the slices are distributed into data units. Once the complete inverse quantized wavelet transform has been extracted from the stream, decoding is the same, regardless of whether fragments or pictures were used in the stream.

NOTE 2 The above ***fragment_data***(state)**:** code is actually independent of the order in which the slice data fragments are decoded, meaning that decoding can be done in any order provided that the setup fragment is decoded before the data fragments for a particular picture.

15 Picture Decoding

15.1 General

Clause 15 defines the processes for decoding a picture from a VC-2 stream. Picture decoding depends upon correctly parsing the VC-2 stream, and decoding operations are dependent on decoding the VC-2 stream and picture syntax (Clauses 11 and 12) and unpacking the coefficient data (Clause 13).

15.2 Overall Picture Decoding Process

This clause defines the processes for decoding a picture from a VC-2 stream.

During the decoding process picture data from the current picture being decoded shall be stored in the **state**[current_picture] state variable. This is a map with an index *pic_num* and data arrays

```
state[current_picture][Y]
state[current_picture][C1]
state[current_picture][C2]
```

representing the video components of the picture.

After decoding, the decoded picture is provided to the decoding application via the implementation-defined output_picture() callback.

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

picture_decode(state):	Ref
state[*current_picture*] = {}	
state[*current_picture*][*pic_num*] = state[*picture_number*]	12.2
inverse_wavelet_transform(state)	15.3
clip_picture(state, state[*current_picture*])	15.5
offset_picture(state, state[*current_picture*])	15.5
output_picture(state[*current_picture*], state[*video_parameters*], state[*picture_coding_mode*])	

15.3 Picture IDWT

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

inverse_wavelet_transform(state):	Ref
state[*current_picture*][*Y*] = idwt(state, state[*y_transform*])	15.4
state[*current_picture*][*C1*] = idwt(state, state[*c1_transform*])	15.4
state[*current_picture*][*C2*] = idwt(state, state[*c2_transform*])	15.4
for each c **in** *Y*, *C1*, *C2*:	
idwt_pad_removal(state, state[*current_picture*][c], c)	15.4.5

15.4 Component IDWT

15.4.1 General

This clause defines the process *idwt*(coeff_data) for reconstructing picture component data from decoded subband data *coeff_data* using the inverse discrete wavelet transform (IDWT). The IDWT shall be invoked in the picture decoding process only after successful unpacking of the wavelet coefficient data as defined in Clause 13.

The IDWT process shall return a pixel array corresponding to a single reconstructed video component.

Since wavelet filtering can operate 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 = row(a,k), then b[r] is a reference to the value of a[k][r].

This means that modifying the value of b[r] for any index r modifies the value of a[k][r].

If b = column(a,k), then b[r] is a reference to the value of a[r][k].

This means that modifying the value of b[r] for any index r modifies the value of a[r][k].

The *idwt* shall be an iterative procedure operating on either two or four subbands (L and H; or LL, HL, LH and HH) at each iteration stage to produce a new DC subband. The procedure shall be as follows:

idwt(state, coeff_data):	**Ref**
if (state[*dwt_depth_ho*] == 0):	
DC_band = coeff_data[0][*LL*]	
else:	
DC_band = coeff_data[0][*L*]	
for n = 1 **to** state[*dwt_depth_ho*]:	
new_DC_band = h_synthesis(state, DC_band, coeff_data[n][*H*])	15.4.2
DC_band = new_DC_band	
for n = state[*dwt_depth_ho*] + 1 **to** state[*dwt_depth_ho*] + state[*dwt_depth*]:	
new_DC_band = vh_synthesis(state, DC_band, coeff_data[n][*HL*], coeff_data[n][*LH*], coeff_data[n][*HH*])	15.4.3
DC_band = new_DC_band	
return DC_band	

NOTE At each stage, the dimensions of the input DC_band will be the same as those of the other input bands, whereas the output width is double that of the input bands and the output height is double that of the input bands for *vh_synthesis* and the same as that of the input bands for *h_synthesis*.

15.4.2 Horizontal Synthesis

This clause defines the operation of the horizontal synthesis process used in the optional pure horizontal portion of a transform:

h_synthesis(state, L_data, H_data):	**Ref**
...	

h_synthesis() shall return an array of twice the width and the same height as each of the input argument arrays.

h_synthesis() is repeatedly invoked by the IDWT synthesis operation (see 15.4.1), and operates on each set of two subband data arrays of identical dimensions to produce a new array synth, which shall be returned as the result of the process.

Step 1. synth is a temporary two-dimensional array that shall be initialized as follows:

...	
synth = new_array(height(L_data), 2 * width(L_data))	
...	

Step 2. The data from the two arrays shall be interleaved in the `synth` array as follows:

`...`	
`for y = 0 to (height(synth)) - 1:`	
`  for x = 0 to (width(synth) // 2) - 1:`	
`  synth[y][2 * x] = L_data[y][x]`	
`  synth[y][(2 * x) + 1] = H_data[y][x]`	
`...`	

NOTE 1 This interleaving enables in-place calculation during the inverse filter process.

Step 3. Data shall be synthesized horizontally by operating on each row of data using a set of one-dimensional filters. The one-dimensional lifting filters used shall be determined by the value of **state**[wavelet_index_ho] according to Table 16 to Table 22. The process shall be as follows:

`...`	
`for y = 0 to height(synt-) - 1:`	
`  oned_synthesis(row(synth, y), state[`*wavelet_index_ho*`])`	15.4.4
`...`	

Step 4. Finally, the synthesized subband data shall implement a bit-shift to remove any accuracy bits. The bit-shift value *filter_bit_shift()* used shall be determined by the value of **state**[wavelet_index_ho] according to Table 16 to Table 22. The process shall be as follows:

`...`	
`shift = filter_bit_shift(state)`	Table 16 to Table 22
`if (shift > 0):`	
`  for y = 0 to height(synt-) - 1:`	
`  for x = 0 to width(synt-) - 1:`	
`  synth[y][x] = (synth[y][x] + (1 << (shift - 1))) >> shift`	
`return synth`	

NOTE 2 Accuracy bits are added in the encoder by shifting up all coefficients in the L band prior to applying any filtering (this includes an initial shift of all values in the component data). Adding a small shift before each decomposition stage is the most efficient way of providing additional resolution mitigating aliasing through non-linear rounding effects.

15.4.3 Vertical and Horizontal Synthesis

This clause defines the operation of the vertical and horizontal synthesis process:

vh_synthesis(state, LL_data, HL_data, LH_data, HH_data):	**Ref**
...	

vh_synthesis() shall return an array of twice the dimensions of each of the input argument arrays.

vh_synthesis() is repeatedly invoked by the IDWT synthesis operation (see 15.4.1), and operates on each set of four subband data arrays of identical dimensions to produce a new array synth, which shall be returned as the result of the process.

Step 1. synth is a temporary two-dimensional array that shall be initialized as follows:

...	
synth = new_array(2 * height(LL_data), 2 * width(LL_data))	
...	

Step 2. The data from the four arrays shall be interleaved in the synth array as follows:

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

NOTE 1 This interleaving enables in-place calculation during the inverse filter process.

Step 3. Data shall be synthesized vertically by operating on each column of data using a one-dimensional filter, and then horizontally by operating on each row of data using a set of one-dimensional filters. The one-dimensional lifting filters used for the vertical synthesis shall be determined by the value of **state**[wavelet_index] according to Table 16 to Table 22. The one-dimensional lifting filters used for the horizontal synthesis shall be determined by the value of **state**[wavelet_index_ho] according to Table 16 to Table 22. The process shall be as follows:

...	
for x = 0 **to** width(synth) - 1:	
oned_synthesis(column(synth, x), state[*wavelet_index*])	15.4.4
for y = 0 **to** height(synth) - 1:	
oned_synthesis(row(synth, y), state[*wavelet_index_ho*])	15.4.4
...	

Step 4. Finally, the synthesized subband data shall implement a bit-shift to remove any accuracy bits. The bit-shift value *filter_bit_shift()* used shall be determined by the value of **state**[wavelet_index_ho] according to Table 16 to Table 22. The process shall be as follows:

...	
shift = filter_bit_shift(state)	Table 16 to Table 22
if (shift > 0):	
for y = 0 **to** height(synth) - 1:	
for x = 0 **to** width(synth) - 1:	
synth[y][x] = (synth[y][x] + (1 << (shift - 1))) >> shift	
return synth	

NOTE 2 Accuracy bits are added in the encoder by shifting up all coefficients in the LL band prior to applying any filtering (this includes an initial shift of all values in the component data). Adding a small shift before each decomposition stage is the most efficient way of providing additional resolution mitigating aliasing through non-linear rounding effects.

15.4.4 One-Dimensional Synthesis

15.4.4.1 General

This clause defines the one-dimensional synthesis process. *oned_synthesis*(A, Index) shall apply to a one-dimensional array of coefficients of even length (A), consisting of either a row or a column of a two-dimensional integral data array.

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

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

1. a filter length value L
2. a filter offset value D
3. an array of taps of length L : taps[0], taps[1], ... taps[L-1]
4. a scale factor S

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

lift1(A, L, D, taps, S),
lift2(A, L, D, taps, S),
lift3(A, L, D, taps, S), and
lift4(A, L, D, taps, S)

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

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

```
lift1(A, L, D, taps, S):
   for n = 0 to (len(A) // 2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2 * (n + i) - 1
         pos = min(pos, len(A) - 1)
         pos = max(pos, 1)
         sum += taps[i - D] * A[pos]
      if (S > 0):
         sum += 1 << (S - 1)
      A[2 * n] += sum >> S
```

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

```
lift2(A, L, D, taps, S):
   for n = 0 to (len(A) // 2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2 * (n + i) - 1
         pos = min(pos, len(A) - 1)
         pos = max(pos, 1)
         sum += taps[i - D] * A[pos]
      if (S > 0):
         sum += 1 << (S - 1)
      A[2 * n] -= sum >> S
```

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

```
lift3(A, L, D, taps, S):
   for n = 0 to (len(A) // 2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2 * (n + i)
         pos = min(pos, len(A) - 2)
         pos = max(pos, 0)
         sum += taps[i - D] * A[pos]
      if (S > 0):
         sum += 1 << (S - 1)
      A[2 * n + 1] += sum >> S
```

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

```
lift4(A, L, D, taps, S):
   for n = 0 to (len(A) // 2) - 1:
      sum = 0
      for i = D to L + D - 1:
         pos = 2 * (n + i)
         pos = min(pos, len(A) - 2)
         pos = max(pos, 0)
         sum += taps[i - D] * A[pos]
      if (S > 0):
         sum += 1 << (S - 1)
      A[2 * n + 1] -= sum >> S
```

oned_synthesis shall apply the sequence of lifting filters specified in 15.4.4.3 corresponding to the value of `Index` which corresponds to either **state**`[wavelet_index]` or **state**`[wavelet_index_ho]`, and shall invoke the corresponding lifting processes with the parameters defined.

15.4.4.2 Mathematical Formulation of Lifting Processes (Informative)

The lifting processes defined in 15.4.4.1 are extremely similar, and careful attention needs to be paid to the detail of their operation in any implementation. The four different variants arise from two factors: the 'phase' (odd or even) of the lifting operation, and their implementation using integer-only operations, which introduces rounding errors and makes addition and subtraction subtly different.

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

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

$$A[2n] \mathrel{+}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)-1] + \left(1 \ll (s-1)\right) \right) \gg s \qquad \text{(Type 1)}$$

$$A[2n] \mathrel{-}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)-1] + \left(1 \ll (s-1)\right) \right) \gg s \qquad \text{(Type 2)}$$

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

$$A[2n+1] \mathrel{+}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)] + \left(1 \ll (s-1)\right) \right) \gg s \qquad \text{(Type 3)}$$

$$A[2n+1] \mathrel{-}= \left(\sum_{i=-N}^{M} t_i A[2(n+i)] + \left(1 \ll (s-1)\right) \right) \gg s \qquad \text{(Type 4)}$$

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

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

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

15.4.4.3 Lifting Filter Parameters

The lifting filters and filter bit-shift operations that apply for each value of `Index` shall be as specified in Table 16 to Table 22.

Table 16 — Index 0: Deslauriers-Dubuc (9,7) lifting stages and shift values.

Lifting steps	*filter_bit_shift*()
Two lifting stages: 1. Type 2, S = 2, L=2, D=0, taps = [1,1] i.e.: $A[2n] \mathrel{-}= (A[2n-1] + A[2n+1] + 2) \gg 2$ 2. Type 3, S = 4, L=4, D=-1, taps=[-1,9,9,-1] i.e.: $A[2n+1] \mathrel{+}= (-A[2n-2] + 9A[2n] + 9A[2n+2] - A[2n+4] + 8) \gg 4$	1

Table 17 — Index 1: LeGall (5,3) lifting stages and shift values.

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 2, L = 2, D = 0, taps = [1,1] i.e.: $A[2n] -= (A[2n-1] + A[2n+1] + 2) \gg 2$ 2. Type 3, S = 1, L = 2, D = 0, taps=[1,1] i.e.: $A[2n+1] += (A[2n] + A[2n+2] + 1) \gg 1$	1

Table 18 — Index 2: Deslauriers-Dubuc (13,7) lifting stages and shift values.

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

Table 19 — Index 3: Haar filter with no shift.

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 1, L = 1, D = 1, taps = [1] i.e.: $A[2n] -= (A[2n+1] + 1) \gg 1$ 2. Type 3, S = 0, L = 1, D = 0, taps = [1] i.e.: $A[2n+1] += A[2n]$	0

Table 20 — Index 4: Haar filter with single shift.

Lifting steps	*filter_bit_shift()*
Two lifting stages: 1. Type 2, S = 1, L = 1, D = 1, taps = [1] i.e.: $A[2n] -= (A[2n+1] + 1) \gg 1$ 2. Type 3, S = 0, L = 1, D = 0, taps = [1] i.e.: $A[2n+1] += A[2n]$	1

Table 21 — Index 5: Fidelity filter.

Lifting steps	*filter_bit_shift*()
Two lifting stages: 1. Type 3, S = 8, L = 8, D = -3, taps = [-2,10,-25,81,81,-25,10,-2] i.e.: $A[2n+1] \mathrel{+}= (-2(A[2n-6]+A[2n+8]) + 10(A[2n-4]+A[2n+6]) - 25(A[2n-2]+A[2n+4]) + 81(A[2n]+A[2n+2]) + 128) \gg 8$ 2. Type 2, S = 8, L = 8, D = -3, Taps = [-8,21,-46,161,161,-46,21,-8] i.e.: $A[2n] \mathrel{-}= (-8(A[2n-7]+A[2n+7]) + 21(A[2n-5]+A[2n+5]) - 46(A[2n-3]+A[2n+3]) + 161(A[2n-1]+A[2n+1]) + 128) \gg 8$	0

NOTE The Fidelity filter was specifically developed by the VC-2 creators using an extensive search process to select a filter set with simple integer filter coefficients limited to 8-bit resolution and offering good alias rejection in both analysis and synthesis operations. The resulting filter is almost mirror symmetric and the good anti-aliasing characteristics provide improved picture quality of the subsampled images. The FIR filter coefficient equivalents of the filters are as follows:

The equivalent analysis low-pass FIR filter is:

(-8, 0, 21, 0, -46, 0, 161, 256, 161, 0, -46, 0, 21, 0, -8) / 256

and the equivalent analysis high-pass FIR filter is:

(-16, 0, 122, 0, -502, 0, 1955, 512, -3491, -2560, 4148, 6400, -4343, -20736, 37122, -20736, -5103, 6400, 7368, 2560, -271, 512, 1035, 0, -82, 0, -38, 0, -16) / 65536

The equivalent synthesis low-pass FIR filter is:

(-16, 0, 122, 0, -502, 0, 1955, -512, -3491, 2560, 4148, -6400, -4343, 20736, 37122, 20736, -5103, -6400, 7368, -2560, -271, -512, 1035, 0, -82, 0, -38, 0, -16) / 65536

and the equivalent synthesis high-pass FIR filter is:

(8, 0, -21, 0, 46, 0, -161, 256, -161, 0, 46, 0, -21, 0, 8) / 256

Table 22 – Index 6: Integer lifting approximation to Daubechies (9,7).

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

15.4.5 Removal of IDWT Pad Values

This clause defines the decoding process *idwt_pad_removal*(state, pic, c) for removing extraneous values after performing the IDWT.

13.2.3 requires that subband coefficient data arrays are padded to ensure that the reconstructed data array *pic* dimensions are multiples of $2^{\text{state[dwt\_depth\_ho] + state[dwt\_depth]}}$.

The functions delete_rows_after(pic, height) and delete_columns_after(pic, width) shall be defined as discarding all rows and columns (respectively) beyond the specified height and width (respectively).

```
idwt_pad_removal(state, pic, c):
    if (c == Y):
        width = state[luma_width]
        height = state[luma_height]
    else if (c == C1 or c == C2):
        width = state[color_diff_width]
        height = state[color_diff_height]
    delete_rows_after(pic, height)
    delete_columns_after(pic, width)
```

15.5 Picture Output Ranges

Picture data shall be clipped to the upper and lower signal extremes prior to being output. The *clip_picture()* process shall be defined as follows:

```
clip_picture(state, current_picture):
    for each c in Y, C1, C2:
        clip_component(state, current_picture[c], c)
```

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

```
clip_component(state, comp_data, c):
    for y = 0 to height(comp_data) - 1:
        for x = 0 to width(comp_data) - 1:
            if (c == Y):
                comp_data[y][x] = clip(comp_data[y][x], -(2 **
                  (state[luma_depth] - 1)), 2 ** (state[luma_depth] - 1) - 1)
            else:
                comp_data[y][x] = clip(comp_data[y][x], -(2 **
                  (state[color_diff_depth] - 1)), 2 ** (state[color_diff_depth]
                  - 1) - 1)
```

Picture data shall be offset so as to be output as unsigned integer values. The *offset_picture()* process shall be defined as follows:

```
offset_picture(state, current_picture):
    for each c in Y, C1, C2:
        offset_component(state, current_picture[c], c)
```

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

```
offset_component(state, comp_data, c):
    for y = 0 to height(comp_data) - 1:
        for x = 0 to width(comp_data) - 1:
            if (c == Y):
                comp_data[y][x] += 2 ** (state[luma_depth] - 1)
            else:
                comp_data[y][x] += 2 ** (state[color_diff_depth] - 1)
```

NOTE Where the picture data is used on video interfaces such as a SMPTE Serial Digital Interface (SDI), there can be further constraints on the minimum and maximum values.

Annex A VC-2 Data Coding Definitions (Normative)

A.1 General

Data shall be represented in the VC-2 stream using one of three basic methods:

1. fixed-length bitwise codings,
2. fixed-length bytewise codings and
3. variable-length codes.

This annex defines how data shall be represented in the VC-2 stream and how sequences of bits shall be extracted as values of various types using the aforementioned fundamental data coding types.

A.2 Bit Packing and Data Input

A.2.1 General

This clause defines the operation of the *read bit()*, *read byte()* and *byte align()* functions used for direct access to fixed length encodings in the VC-2 stream.

The stream data shall be accessed byte by byte, and a decoder is deemed to maintain a copy of the current byte, **state**[current_byte], and an index to the next bit (in the byte) to be read, **state**[next_bit].

state[current_byte] shall be initialized to contain the first byte in the stream.

state[next_bit] shall be an integer from bit 0 (the least-significant bit) to bit 7 (the most-significant bit). Bits within bytes shall be accessed from the msb first to the lsb last. state[next_bit] shall be initialized to 7.

A.2.2 Reading a Byte

The *read_byte()* function shall perform the following steps:

1. Set **state**[next_bit] = 7
2. Set **state**[current_byte] to the next unread byte in the stream if one is available. If no more bytes are available, the contents of **state**[current_byte] shall be undefined and any attempt to use this value should cause a decoder to indicate an error condition.

A.2.3 Reading a Bit

The read_bit() function shall be defined as follows:

```
read_bit(state):
    bit = (state[current_byte] >> state[next_bit]) & 1
    state[next_bit] -= 1
    if (state[next_bit] < 0):
        state[next_bit] = 7
        read_byte(state)
    return bit
```

A.2.4 Byte Alignment

The *byte_align()* function shall be used to discard data in the current byte and begin data access at the next byte, except where the input is already at the beginning of a byte. The *byte_align()* function shall be defined as follows:

```
byte_align(state):
    if (state[next_bit] != 7):
        read_byte(state)
```

This function shall be used to ensure that a whole number of bytes are read before beginning reading a new stream element: for example, each parse info header within the stream is byte aligned.

A.2.5 End of Stream Detection

is_end_of_stream() shall be a function which returns **True** if no more bytes are available in the stream and all bits in state[*current_byte*] have been consumed. It shall return **False** otherwise.

A.3 Fixed Length Data

A.3.1 General

VC-2 uses three fixed length bitstream data encodings which are defined in A.3.2, A.3.3 and A.3.4.

A.3.2 Boolean

The *read_bool()* function shall return **True** if 1 is read from the stream and **False** otherwise. The *read_bool()* function shall be defined as follows:

```
read_bool(state):
    if (read_bit(state) == 1):
        return True
    else:
        return False
```

A.3.3 n-bit Unsigned Integer Literal

An n-bit unsigned integer in literal format shall be decoded by extracting n bits in order, using the *read_bit()* function (see A.2.3) and placing the first bit in the leftmost position, the second bit in the next position, and so on. The resulting value shall be interpreted as an unsigned integer.

The *read_nbits()* function shall be defined as follows:

```
read_nbits(state, n):
    val = 0
    for i = 0 to n - 1:
        val <<= 1
        val += read_bit(state)
    return val
```

A.3.4 n-byte Unsigned Integer Literal

A single byte shall be interpreted as an unsigned integer value in the range 0 to 255. The process defined below reads one or more bytes as needed to create an n-byte value with the most significant byte read first. The *read_uint_lit()* function shall be defined as follows:

```
read_uint_lit(state, n):
    return read_nbits(state, 8 * n)
```

A.4 Variable-length Codes

A.4.1 General

Variable-length codes (VLC) are used in two ways in the VC-2 data stream.

1. The first use is for the representation of header values into the stream.
2. The second use is for the representation of wavelet coefficients.

When used for representing wavelet coefficients, VLCs shall be employed within a data block of known length. It is possible to gain additional compression by early termination; maintaining a count of remaining bits, and returning default values when this length is exceeded. This shall be achieved by the use of the *read_bitb()*, *read_uintb()* and *read_sintb()* functions for reading values from data blocks.

A.4.2 Data Input for Bounded Block Operation

This clause defines the operation of the *read_bitb()* and *read_boolb()* processes for reading bits and Boolean values from a block of known size, and the *flush_inputb()* process for discarding the remainder of a block of data.

These processes shall use **state**[bits_left] to determine the remaining bits to the end of the block.

The *read_bitb()* function shall be defined as follows:

```
read_bitb(state):
    if (state[bits_left] == 0):
        return 1
    else:
        state[bits_left] -= 1
        return read_bit(state)
```

When all bits in the block have been read, then *read_bitb()* shall return 1 by default.

The *read_boolb()* function operates analogously to the *read_bool()* function and shall be defined as follows:

```
read_boolb(state):
    if (read_bitb(state) == 1):
        return True
    else:
        return False
```

It is possible that not all data in a block is exhausted after a sequence of read operations. At the end of a sequence of bounded block read operations, the decoder shall flush the block.

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

```
flush_inputb(state):
    while (state[bits_left] > 0):
        read_bit(state)
        state[bits_left] -= 1
```

A.4.3 Unsigned Interleaved Exp-Golomb Codes

This clause defines the unsigned interleaved exp-Golomb data format and the operation of the *read_uint()* and the *read_uintb()* functions.

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

The K + 1 bits in the even positions (counting from zero) shall be the “follow” bits, and the K bits in the odd positions shall be the “data” bits b_i that are used to construct the decoded value itself. A follow bit value of 0 shall indicate a subsequent data bit, whereas a follow bit value of 1 shall terminate the code, a typical sequence being:

$$0,\ x_{K-1},\ 0,\ x_{K-2},\ \ldots\ 0,\ x_0,\ 1$$

The data bits x_i shall be the binary representation of the first K bits of the (K + 1) bit number (N + 1), where N is the number to be decoded, i.e.,

$$N + 1 = 0\mathrm{b}1x_{K-1}x_{K-2}\ldots x_0 = 2^K + x^{K-1} \times 2^{K-1} + x^{K-2} \times 2^{K-2} + \ \ldots + x_0$$

A list of encodings of the first 10 values is shown in Table A.1.

Table A.1 — Example conversions from unsigned interleaved exp-Golomb-coded values to unsigned integers.

Bit sequence	Decoded value
1	0
001	1
011	2
00001	3
00011	4
01001	5
01011	6
0000001	7
0000011	8
0001001	9

Although apparently complex, the interleaving ensures that the code has a very simple decoding loop. The *read_uint()* function returns a non-negative integer value and shall be defined as follows:

```
read_uint(state):
    value = 1
    while (read_bit(state) == 0):
        value <<= 1
        if (read_bit(state) == 1):
            value += 1
    value -= 1
    return value
```

NOTE 1 Conventional exp-Golomb coding places all follow bits at the beginning as a prefix. This is easier to read, but needs a count of the prefix length to be maintained. Values can only be decoded in two loops – the prefix followed by the data bits. Interleaved exp-Golomb coding allows values to be decoded in a single loop, without the need for a length count.

The *read_uintb()* function is identical to *read_uint()* function except that the block-bounded read operation is employed and shall be defined as follows:

```
read_uintb(state):
   value = 1
   while (read_bitb(state) == 0):
      value <<= 1
      if (read_bitb(state)):
         value += 1
   value -= 1
   return value
```

NOTE 2 When **state**[bits_left] == 0, all subsequent values read by *read_uintb()* will be zero.

A.4.4 Signed Interleaved Exp-Golomb Codes

This clause defines the signed interleaved exp-Golomb data format and the operation of the *read_sint()* and *read_sintb()* functions.

The code for the signed interleaved exp-Golomb data format shall consist of the unsigned interleaved exp-Golomb code for the magnitude, followed by a sign bit for non-zero values as shown in Table A.2.

Table A.2 – Example conversions from signed interleaved exp-Golomb-coded values to signed integers

Bit sequence	Decoded value
000111	-4
000011	-3
0111	-2
0011	-1
1	0
0010	1
0110	2
000010	3
000110	4

The decoding process for *read_sint()* shall be as follows:

```
read_sint(state):
   value = read_uint(state)
   if (value != 0):
      if (read_bit(state) == 1):
         value = -value
   return value
```

The *read_sintb()* function shall be identical to *read_sint()* except that the block-bounded read operation shall be employed and shall be as defined as follows:

```
read_sintb(state):
   value = read_uintb(state)
   if (value != 0):
      if (read_bitb(state) == 1):
         value = -value
   return value
```

NOTE When **state**[bits_left] == 0, all subsequent values read by *read_sintb()* will be zero.

Annex B Predefined Video Formats (Normative)

This annex defines the default values of video parameters that are determined by the value of `base_video_format`. These defaults reduce overhead by allowing a large number of parameters to be set without explicit signaling.

The collection of default values for each value of `base_video_format` constitutes a map, which shall be returned by the *`set_source_defaults`*`(base_video_format)` function and used as a basis for defining the source video format in the sequence header as per 11.4. The default parameter values for each value of `base_video_format` shall be as listed in Table B.1, Table B.2, and Table B.3.

The labels used to access the map are as follows (with reference to the clause where each are first defined):

- *`frame_width`* (11.4.3)
- *`frame_height`* (11.4.3)
- *`color_diff_format_index`* (11.4.4)
- *`source_sampling`* (11.4.5)
- *`top_field_first`* (11.4.5)
- *`frame_rate_numer`* (11.4.6)
- *`frame_rate_denom`* (11.4.6)
- *`pixel_aspect_ratio_numer`* (11.4.7)
- *`pixel_aspect_ratio_denom`* (11.4.7)
- *`clean_width`* (11.4.8)
- *`clean_height`* (11.4.8)
- *`left_offset`* (11.4.8)
- *`top_offset`* (11.4.8)
- *`luma_offset`* (11.4.9)
- *`luma_excursion`* (11.4.9)
- *`color_diff_offset`* (11.4.9)
- *`color_diff_excursion`* (11.4.9)
- *`color_primaries_index`* (11.4.10.2)
- *`color_matrix_index`* (11.4.10.3)
- *`transfer_function_index`* (11.4.10.4)

All source parameters, with the exception of *top_field_first*, for any of the predefined video formats can be overridden in the sequence header. In the specific case of base_video_format == 0 (Custom Format), all the values (with the exception of *top_field_first*) are intended as default values to be overridden as needed to specify the custom format.

NOTE 1 When selecting a predefined video format, keep in mind that the value of *top_field_first* cannot be overridden.

NOTE 2 The predefined format SD 480i-60 has *top_field_first* = False. If lines are selected from the 525-line raster defined in SMPTE ST 125 according to the 480i format defined in SMPTE RP 202, the appropriate setting is likely to be *top_field_first* = True. In this case, an alternative predefined format will need to be used.

NOTE 3 The predefined formats for D-Cinema (base_video_format 15 and 16) are likely to need to be overridden. More suitable values might be: *color_matrix_index* = 3 (Identity), *luma_offset* = *color_diff_offset* = 0, *luma_excursion* = *color_diff_excursion* = 4095.

Table B.1 — Default source parameters for video formats 0 to 6.

base_video_format (Table 6)	0	1	2	3	4	5	6
video format name (informative)	Custom Format	QSIF525	QCIF	SIF525	CIF	4SIF525	4CIF
frame_width	640	176	176	352	352	704	704
frame_height	480	120	144	240	288	480	576
color_diff_format_index (Table 7)	2	2	2	2	2	2	2
color difference format name (informative)	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0	4:2:0
source_sampling (11.4.5)	0	0	0	0	0	0	0
top_field_first	False	False	True	False	True	False	True
frame_rate_numer	24000	15000	25	15000	25	15000	25
frame_rate_denom	1001	1001	2	1001	2	1001	2
pixel_aspect_ratio_numer	1	10	12	10	12	10	12
pixel_aspect_ratio_denom	1	11	11	11	11	11	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
color_diff_offset	128	128	128	128	128	128	128
color_diff_excursion	255	255	255	255	255	255	255
color_primaries_index (Table 12)	0	1	2	1	2	1	2
color primaries name (informative)	HDTV	SDTV 525	SDTV 625	SDTV 525	SDTV 625	SDTV 525	SDTV 625
color_matrix_index (Table 13)	0	1	1	1	1	1	1
color matrix name (informative)	HDTV	SDTV	SDTV	SDTV	SDTV	SDTV	SDTV
transfer_function_index (Table 14)	0	0	0	0	0	0	0
transfer function name (informative)	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma

Table B.2 — Default source parameters for video formats 7 to 14.

base_video_format (Table 6)	7	8	9	10	11	12	13	14
video format name (informative)	SD 480i-60	SD 576i-50	HD 720p-60	HD 720p-50	HD 1080i-60	HD 1080i-50	HD 1080p-60	HD 1080p-50
frame_width	720	720	1280	1280	1920	1920	1920	1920
frame_height	480	576	720	720	1080	1080	1080	1080
color_diff_format_index (Table 7)	1	1	1	1	1	1	1	1
color difference format (informative)	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2
source_sampling (11.4.5)	1	1	0	0	1	1	0	0
top_field_first	False	True	True	True	True	True	True	True
frame_rate_numer	30000	25	60000	50	30000	25	60000	50
frame_rate_denom	1001	1	1001	1	1001	1	1001	1
pixel_aspect_ ratio_numer	10	12	1	1	1	1	1	1
pixel_aspect_ ratio_denom	11	11	1	1	1	1	1	1
clean_width	704	704	1280	1280	1920	1920	1920	1920
clean_height	480	576	720	720	1080	1080	1080	1080
left_offset	8	8	0	0	0	0	0	0
top_offset	0	0	0	0	0	0	0	0
luma_offset	64	64	64	64	64	64	64	64
luma_excursion	876	876	876	876	876	876	876	876
color_diff_offset	512	512	512	512	512	512	512	512
color_diff_excursion	896	896	896	896	896	896	896	896
color_primaries_index (Table 12)	1	2	0	0	0	0	0	0
color primaries name (informative)	SDTV 525	SDTV 625	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV
color_matrix_index (Table 13)	1	1	0	0	0	0	0	0
color matrix name (informative)	SDTV	SDTV	HDTV	HDTV	HDTV	HDTV	HDTV	HDTV
transfer_function_index (Table 14)	0	0	0	0	0	0	0	0
transfer function name (informative)	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma

Table B.3 — Default source parameters for video formats 15 to 22.

base_video_format (Table 6)	15	16	17	18	19	20	21	22
video format name (informative)	DC 2K-24	DC 4K-24	UHDTV 4K-60	UHDTV 4K-50	UHDTV 8K-60	UHDTV 8K-50	HD 1080p-24	SD Pro486
frame_width	2048	4096	3840	3840	7680	7680	1920	720
frame_height	1080	2160	2160	2160	4320	4320	1080	486
color_diff_format_index (Table 7)	0	0	1	1	1	1	1	1
color difference format (informative)	4:4:4	4:4:4	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2	4:2:2
source_sampling (11.4.5)	0	0	0	0	0	0	0	1
top_field_first	True	True	True	True	True	True	True	False
frame_rate_numer	24	24	60000	50	60000	50	24000	30000
frame_rate_denom	1	1	1001	1	1001	1	1001	1001
pixel_aspect_ ratio_numer	1	1	1	1	1	1	1	10
pixel_aspect_ ratio_denom	1	1	1	1	1	1	1	11
clean_width	2048	4096	3840	3840	7680	7680	1920	720
clean_height	1080	2160	2160	2160	4320	4320	1080	486
left_offset	0	0	0	0	0	0	0	0
top_offset	0	0	0	0	0	0	0	0
luma_offset	256	256	64	64	64	64	64	64
luma_excursion	3504	3504	876	876	876	876	876	876
color_diff_offset	2048	2048	512	512	512	512	512	512
color_diff_excursion	3584	3584	896	896	896	896	896	896
color_primaries_index (Table 12)	3	3	4	4	4	4	0	0
color primaries name (informative)	D-Cinema	D-Cinema	UHDTV	UHDTV	UHDTV	UHDTV	HDTV	HDTV
color_matrix_index (Table 13)	2	2	4	4	4	4	0	0
color matrix name (informative)	Reversible	Reversible	UHDTV	UHDTV	UHDTV	UHDTV	HDTV	HDTV
transfer_function_index (Table 14)	3	3	0	0	0	0	0	0
transfer function name (informative)	D-Cinema	D-Cinema	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma	TV Gamma

Annex C Profiles and Levels (Normative)

C.1 General

A VC-2 decoder shall support one or more different profiles and levels. Profiles and levels determine which tools, syntax elements and structures shall be supported, and what decoder resources (computational and memory) are required.

C.2 Profiles

C.2.1 General

A given profile requires that particular syntax/syntax elements shall be used and that decoder variables or functions shall be set to particular values.

This version of the VC-2 specification defines two profiles, low delay and high quality, corresponding to different picture types.

NOTE Earlier versions of this specification also defined two additional profiles, main and simple.

These shall satisfy the following conditions:

- A low delay profile VC-2 sequence shall set **`state`**`[profile]` equal to a value of 0 in the parse parameters (see 10.5) for each sequence header in the sequence.
- A high quality profile sequence shall set **`state`**`[profile]` equal to a value of 3 in the parse parameters for each sequence header in the sequence.
- The values 1 and 2 for **`state`**`[profile]` refer to profiles from earlier versions of this specification which are no longer supported. A decoder may handle these profiles in conformance with an earlier version of this specification, provided the streams have a major version less than 3.
- All other values for **`state`**`[profile]` are reserved for future use.

A VC-2 sequence shall comply with one of the supported profiles.

C.2.2 Low Delay Profile

A low delay profile sequence shall contain only those data units whose parse codes are listed in Table C.1.

Table C.1 — Parse code values for low delay profile sequences.

Parse Code (hex)	Binary code (Informative)	Description
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
0xC8	1100 1000	Low Delay Picture
0xCC	1100 1100	Low Delay Picture Fragment

A low delay profile sequence shall contain only low delay fragmented and non-fragmented pictures.

C.2.3 High Quality Profile

A High Quality profile sequence shall contain only those data units whose parse codes are listed in Table C.2.

Table C.2 — Parse code values for high quality profile sequences.

Parse Code (hex)	Binary code (Informative)	Description
0x00	0000 0000	Sequence header
0x10	0001 0000	End of sequence
0x20	0010 0000	Auxiliary data
0x30	0011 0000	Padding data
0xE8	1110 1000	High Quality Picture
0xEC	1110 1100	High Quality Picture Fragment

A High Quality profile sequence shall contain only high quality fragmented and non-fragmented pictures.

C.3 Levels

A given value of level defines constraints on the decoder resources needed to decode a compliant sequence. The parameter values (or range of values) that are constrained for each level shall be defined in associated SMPTE documents. Level definitions can include specific limits on the coded bit rates defined typically as limits on the number of bits per picture or bits per slice.

NOTE Generalized level values are defined in the companion standard SMPTE ST 2042-2. Specialized level values are defined in other associated SMPTE documents.

Annex D Quantization Matrices (Normative)

D.1 General

This annex defines the default quantization tables to be used and provides an informative description of quantization matrix design principles.

D.2 Default Quantization Matrices

This annex defines default quantization matrices to be used for the quantization of slice coefficients. The quantization matrices shall be as listed in Table D.1 to Table D.8.

Table D.1 to Table D.8 define matrices for two different set of cases.

The first set of cases shall meet the following conditions:

- **state**[wavelet_index_ho] is equal to **state**[wavelet_index]
- **state**[dwt_depth] is smaller than or equal to 4
- **state**[dwt_depth_ho] is smaller than or equal to 4
- the sum of **state**[dwt_depth] and **state**[dwt_depth_ho] is lower than or equal to 5

The second set of cases shall meet the following conditions:

- **state**[wavelet_index_ho] is not equal to **state**[wavelet_index]
- **state**[wavelet_index_ho] is equal to 1 (LeGall)
- **state**[wavelet_index] is equal to 3 (Haar with no shift)
- **state**[dwt_depth] is smaller than or equal to 4
- **state**[dwt_depth_ho] is smaller than or equal to 4
- the sum of **state**[dwt_depth] and **state**[dwt_depth_ho] is lower than or equal to 5

Combined values of **state**[wavelet_index], **state**[wavelet_index_ho], **state**[dwt_depth] and **state**[dwt_depth_ho] not present in the tables in this annex shall require a custom matrix to be encoded, as per 12.4.5.3.

Informative advice for constructing custom quantization matrices based on noise power conservation and perceptual weighting is given in D.3.

Table D.1 — Default quantization matrices for `state[wavelet index]` == 0 and `state[wavelet index_ho]` == 0 (Deslauriers-Dubuc (9,7)).

		state[dwt_depth_ho**]=0**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	5	5	5	5
1	HL, LH , HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
2	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4 ,1
3	HL, LH, HH	-	-	-	5, 5 ,2	5, 5, 2
4	HL, LH, HH	-	-	-	-	6, 6, 3
		state[dwt_depth_ho**]=1**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	3	3
1	H	0	0	0	0	0
2	HL, LH, HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
3	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4, 1
4	HL, LH, HH	-	-	-	5, 5, 2	5, 5, 2
5	HL, LH, HH	-	-	-	-	6, 6, 3
		state[dwt_depth_ho**]=2**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	3	-
1	H	0	0	0	0	-
2	H	3	3	3	3	-
3	HL, LH, HH	-	5, 5, 3	5, 5, 3	5, 5, 3	-
4	HL, LH, HH	-	-	6, 6, 4	6, 6, 4	-
5	HL, LH, HH	-	-	-	7, 7, 5	-
		state[dwt_depth_ho**]=3**				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	-	-
1	H	0	0	0	-	-
2	H	3	3	3	-	-
3	H	5	5	5	-	-
4	HL, LH, HH	-	8, 8, 5	8, 8, 5	-	-
5	HL, LH, HH	-	-	9, 9, 6	-	-
		state[dwt_depth_ho**]=4**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	-	-	-
1	H	0	0	-	-	-
2	H	3	3	-	-	-
3	H	5	5	-	-	-
4	H	8	8	-	-	-
5	HL, LH, HH	-	10, 10, 8	-	-	-

Table D.2 — Default quantization matrices for `state[wavelet index]` == 1 and `state[wavelet index_ho]` == 1 (LeGall (5,3)).

		state[dwt_depth_ho]=0				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	4	4	4	4
1	HL, LH , HH	-	2, 2, 0	2, 2, 0	2, 2, 0	2, 2, 0
2	HL, LH, HH	-	-	4, 4, 2	4, 4, 2	4, 4, 2
3	HL, LH, HH	-	-	-	5, 5, 3	5, 5, 3
4	HL, LH, HH	-	-	-	-	7, 7, 5
		state[dwt_depth_ho]=1				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	2	2	2
1	H	0	0	0	0	0
2	HL, LH, HH	-	3, 3, 1	3, 3, 1	3, 3, 1	3, 3, 1
3	HL, LH, HH	-	-	4, 4, 2	4, 4, 2	4, 4, 2
4	HL, LH, HH	-	-	-	6, 6, 4	6, 6, 4
5	HL, LH, HH	-	-	-	-	8, 8, 6
		state[dwt_depth_ho]=2				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	2	2	-
1	H	0	0	0	0	-
2	H	3	3	3	3	-
3	HL, LH, HH	-	6, 6, 4	6, 6, 4	6, 6, 4	-
4	HL, LH, HH	-	-	7, 7, 5	7, 7, 5	-
5	HL, LH, HH	-	-	-	9, 9, 7	-
		state[dwt_depth_ho]=3				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	2	-	-
1	H	0	0	0	-	-
2	H	3	3	3	-	-
3	H	6	6	6	-	-
4	HL, LH, HH	-	8, 8, 6	8, 8, 6	-	-
5	HL, LH, HH	-	-	10, 10, 8	-	-
		state[dwt_depth_ho]=4				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	-	-	-
1	H	0	0	-	-	-
2	H	3	3	-	-	-
3	H	6	6	-	-	-
4	H	8	8	-	-	-
5	HL, LH, HH	-	11, 11, 9	-	-	-

Table D.3 — Default quantization matrices for `state[wavelet index]` == 2 and `state[wavelet index_ho]` == 2 (Deslauriers-Dubuc (13,7)).

		state[dwt_depth_ho]=0				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	5	5	5	5
1	HL, LH , HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
2	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4, 1
3	HL, LH, HH	-	-	-	5, 5, 2	5, 5, 2
4	HL, LH, HH	-	-	-	-	6, 6, 3
		state[dwt_depth_ho]=1				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	3	3
1	H	0	0	0	0	0
2	HL, LH, HH	-	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0
3	HL, LH, HH	-	-	4, 4, 1	4, 4, 1	4, 4, 1
4	HL, LH, HH	-	-	-	5, 5, 2	5, 5, 2
5	HL, LH, HH	-	-	-	-	6, 6, 3
		state[dwt_depth_ho]=2				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	3	-
1	H	0	0	0	0	-
2	H	3	3	3	3	-
3	HL, LH, HH	-	5, 5, 2	5, 5, 2	5, 5, 2	-
4	HL, LH, HH	-	-	6, 6, 4	6, 6, 4	-
5	HL, LH, HH	-	-	-	7, 7, 5	-
		state[dwt_depth_ho]=3				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	3	-	-
1	H	0	0	0	-	-
2	H	3	3	3	-	-
3	H	5	5	5	-	-
4	HL, LH, HH	-	8, 8, 5	8, 8, 5	-	-
5	HL, LH, HH	-	-	9, 9, 6	-	-
		state[dwt_depth_ho]=4				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	3	3	-	-	-
1	H	0	0	-	-	-
2	H	3	3	-	-	-
3	H	5	5	-	-	-
4	H	8	8	-	-	-
5	HL, LH, HH	-	10, 10, 8	-	-	-

Table D.4 — Default quantization matrices for `state[wavelet index]` == 3 and `state[wavelet index_ho]` == 3 (Haar with no shift).

		`state[dwt_depth_ho]=0`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	8	12	16	20
1	HL, LH , HH	-	4, 4, 0	8, 8, 4	12, 12 ,8	16, 16, 12
2	HL, LH, HH	-	-	4, 4, 0	8, 8, 4	12, 12, 8
3	HL, LH, HH	-	-	-	4, 4, 0	8, 8, 4
4	HL, LH, HH	-	-	-	-	4, 4, 0
		`state[dwt_depth_ho]=1`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	4	10	14	18	22
1	H	0	6	10	14	18
2	HL, LH, HH	-	4, 4, 0	8, 8, 4	12, 12 ,8	16, 16, 12
3	HL, LH, HH	-	-	4, 4, 0	8, 8, 4	12, 12, 8
4	HL, LH, HH	-	-	-	4, 4, 0	8, 8, 4
5	HL, LH, HH	-	-	-	-	4, 4, 0
		`state[dwt_depth_ho]=2`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	6	12	16	20	-
1	H	2	8	12	16	-
2	H	0	6	10	14	-
3	HL, LH, HH	-	4, 4, 0	8, 8, 4	12, 12, 8	-
4	HL, LH, HH	-	-	4, 4, 0	8, 8, 4	-
5	HL, LH, HH	-	-	-	4, 4, 0	-
		`state[dwt_depth_ho]=3`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	8	14	18	-	-
1	H	4	10	14	-	-
2	H	2	8	12	-	-
3	H	0	6	10	-	-
4	HL, LH, HH	-	4, 4, 0	8, 8, 4	-	-
5	HL, LH, HH	-	-	4, 4, 0	-	-
		`state[dwt_depth_ho]=4`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	10	16	-	-	-
1	H	6	12	-	-	-
2	H	4	10	-	-	-
3	H	2	8	-	-	-
4	H	0	6	-	-	-
5	HL, LH, HH	-	4, 4, 0	-	-	-

Table D.5 — Default quantization matrices for `state[wavelet index]` == 4 and `state[wavelet index_ho]` == 4 (Haar with single shift per level).

		state[dwt_depth_ho**]=0**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	8	8	8	8
1	HL, LH , HH	-	4, 4, 0	4, 4, 0	4, 4, 0	4, 4, 0
2	HL, LH, HH	-	-	4, 4, 0	4, 4, 0	4, 4, 0
3	HL, LH, HH	-	-	-	4, 4, 0	4, 4, 0
4	HL, LH, HH	-	-	-	-	4, 4, 0
		state[dwt_depth_ho**]=1**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	4	6	6	6	6
1	H	0	2	2	2	2
2	HL, LH, HH	-	4, 4, 0	4, 4, 0	4, 4, 0	4, 4, 0
3	HL, LH, HH	-	-	4, 4, 0	4, 4, 0	4, 4, 0
4	HL, LH, HH	-	-	-	4, 4, 0	4, 4, 0
5	HL, LH, HH	-	-	-	-	4, 4, 0
		state[dwt_depth_ho**]=2**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	4	4	4	4	-
1	H	0	0	0	0	-
2	H	2	2	2	2	-
3	HL, LH, HH	-	4, 4, 0	4, 4, 0	4, 4, 0	-
4	HL, LH, HH	-	-	4, 4, 0	4, 4, 0	-
5	HL, LH, HH	-	-	-	4, 4, 0	-
		state[dwt_depth_ho**]=3**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	4	4	4	-	-
1	H	0	0	0	-	-
2	H	2	2	2	-	-
3	H	4	4	4	-	-
4	HL, LH, HH	-	6, 6, 2	6, 6, 2	-	-
5	HL, LH, HH	-	-	6, 6, 2	-	-
		state[dwt_depth_ho**]=4**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	4	4	-	-	-
1	H	0	0	-	-	-
2	H	2	2	-	-	-
3	H	4	4	-	-	-
4	H	6	6	-	-	-
5	HL, LH, HH	-	8, 8, 4	-	-	-

Table D.6 — Default quantization matrices for `state[wavelet index]` == 5 and `state[wavelet index_ho]` == 5 (Fidelity).

		state[dwt_depth_ho]=0				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	0	0	0	0
1	HL, LH , HH	-	4, 4, 8	4, 4, 8	4, 4, 8	4, 4, 8
2	HL, LH, HH	-	-	8, 8, 12	8, 8, 12	8, 8, 12
3	HL, LH, HH	-	-	-	13, 13, 17	13, 13, 17
4	HL, LH, HH	-	-	-	-	17, 17, 21
		state[dwt_depth_ho]=1				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	0	0
1	H	4	4	4	4	4
2	HL, LH, HH	-	6, 6, 10	6, 6, 10	6, 6, 10	6, 6, 10
3	HL, LH, HH	-	-	11, 11, 15	11, 11, 15	11, 11, 15
4	HL, LH, HH	-	-	-	15, 15, 19	15, 15, 19
5	HL, LH, HH	-	-	-	-	19, 19, 23
		state[dwt_depth_ho]=2				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	0	-
1	H	4	4	4	4	-
2	H	6	6	6	6	-
3	HL, LH, HH	-	8, 8, 12	8, 8, 12	8, 8, 12	-
4	HL, LH, HH	-	-	13, 13, 17	13, 13, 17	-
5	HL, LH, HH	-	-	-	17, 17, 21	-
		state[dwt_depth_ho]=3				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	-	-
1	H	4	4	4	-	-
2	H	6	6	6	-	-
3	H	8	8	8	-	-
4	HL, LH, HH	-	11, 11, 15	11, 11, 15	-	-
5	HL, LH, HH	-	-	15, 15, 19	-	-
		state[dwt_depth_ho]=4				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	-	-	-
1	H	4	4	-	-	-
2	H	6	6	-	-	-
3	H	8	8	-	-	-
4	H	11	11	-	-	-
5	HL, LH, HH	-	13, 13, 17	-	-	-

NOTE The values given in Table D.6 do not correctly compensate for differential power gain in filter subbands as intended. However, for compatibility with previous versions of this standard, the values in Table D.6 have not been changed. Corrected values are provided in Table D.9 which can be used as custom quantization matrix values which override these defaults.

Table D.7 — Default quantization matrices for `state[wavelet index]` == 6 and `state[wavelet index_ho]` == 6 (Daubechies (9,7)).

		state[dwt_depth_ho]=0				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	3	3	3	3
1	HL, LH , HH	-	1, 1, 0	1, 1, 0	1, 1, 0	1, 1, 0
2	HL, LH, HH	-	-	4, 4, 2	4, 4, 2	4, 4, 2
3	HL, LH, HH	-	-	-	6, 6, 5	6, 6, 5
4	HL, LH, HH	-	-	-	-	9, 9, 7
		state[dwt_depth_ho]=1				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	1	1	1	1	1
1	H	0	0	0	0	0
2	HL, LH, HH	-	3, 3, 2	3, 3, 2	3, 3, 2	3, 3, 2
3	HL, LH, HH	-	-	6, 6, 4	6, 6, 4	6, 6, 4
4	HL, LH, HH	-	-	-	8, 8, 7	8, 8, 7
5	HL, LH, HH	-	-	-	-	11, 11, 9
		state[dwt_depth_ho]=2				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	1	1	1	1	-
1	H	0	0	0	0	-
2	H	3	3	3	3	-
3	HL, LH, HH	-	6, 6, 5	6, 6, 5	6, 6, 5	-
4	HL, LH, HH	-	-	9, 9, 8	9, 9, 8	-
5	HL, LH, HH	-	-	-	11, 11, 10	-
		state[dwt_depth_ho]=3				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	1	1	1	-	-
1	H	0	0	0	-	-
2	H	3	3	3	-	-
3	H	6	6	6	-	-
4	HL, LH, HH	-	10, 10, 8	10, 10, 8	-	-
5	HL, LH, HH	-	-	12, 12, 11	-	-
		state[dwt_depth_ho]=4				
		state[dwt_depth]				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	1	1	-	-	-
1	H	0	0	-	-	-
2	H	3	3	-	-	-
3	H	6	6	-	-	-
4	H	10	10	-	-	-
5	HL, LH, HH	-	13, 13, 12	-	-	-

Table D.8 — Default quantization matrices for `state[wavelet index]` == 3 (Haar0) and `state[wavelet index_ho]` == 1 (LeGall).

		state[dwt_depth_ho**]=0**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	6	6	6	6
1	HL, LH , HH	-	4, 2, 0	4, 2, 0	4, 2, 0	4, 2, 0
2	HL, LH, HH	-	-	5, 3, 1	5, 3, 1	5, 3, 1
3	HL, LH, HH	-	-	-	6, 4, 2	6, 4, 2
4	HL, LH, HH	-	-	-	-	6, 5, 2
		state[dwt_depth_ho**]=1**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	3	3	3	3
1	H	0	1	1	1	1
2	HL, LH, HH	-	4, 2, 0	4, 2, 0	4, 2, 0	4, 2, 0
3	HL, LH, HH	-	-	5, 3, 1	5, 3, 1	5, 3, 1
4	HL, LH, HH	-	-	-	6, 4, 2	6, 4, 2
5	HL, LH, HH	-	-	-	-	6, 5, 2
		state[dwt_depth_ho**]=2**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	2	2	-
1	H	0	0	0	0	-
2	H	3	3	3	3	-
3	HL, LH, HH	-	6, 4, 2	6, 4, 2	6, 4, 2	-
4	HL, LH, HH	-	-	6, 5, 2	6, 5, 2	-
5	HL, LH, HH	-	-	-	7, 5, 3	-
		state[dwt_depth_ho**]=3**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	2	-	-
1	H	0	0	0	-	-
2	H	3	3	3	-	-
3	H	6	6	6	-	-
4	HL, LH, HH	-	8, 7, 4	8, 7, 4	-	-
5	HL, LH, HH	-	-	9, 7, 5	-	-
		state[dwt_depth_ho**]=4**				
		state[dwt_depth**]**				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	2	2	-	-	-
1	H	0	0	-	-	-
2	H	3	3	-	-	-
3	H	6	6	-	-	-
4	H	8	8	-	-	-
5	HL, LH, HH	-	11, 9, 7	-	-	-

Table D.9 provides custom quantization matrices for use with the Fidelity wavelet. Unlike the default quantization matrix values in Table D.6, these correctly compensate for differential filter gain. Encoders can encode these matrices as custom quantization matrices when the fidelity filter is used, rather than using the default quantization matrix.

Table D.9 — Suggested custom quantization matrices for `state[wavelet index]` == 5 and `state[wavelet index_ho]` == 5 (Fidelity) (Informative).

		`state[dwt_depth_ho]=0`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	LL	0	0	0	0	0
1	HL, LH , HH	-	3, 3, 7	3, 3, 7	3, 3, 7	3, 3, 7
2	HL, LH, HH	-	-	7, 7, 10	7, 7, 10	7, 7, 10
3	HL, LH, HH	-	-	-	10, 10, 13	10, 10, 13
4	HL, LH, HH	-	-	-	-	13, 13, 16
		`state[dwt_depth_ho]=1`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	0	0
1	H	3	3	3	3	3
2	HL, LH, HH	-	5, 5, 8	5, 5, 8	5, 5, 8	5, 5, 8
3	HL, LH, HH	-	-	8, 8, 12	8, 8, 12	8, 8, 12
4	HL, LH, HH	-	-	-	11, 11, 15	11, 11, 15
5	HL, LH, HH	-	-	-	-	15, 15, 18
		`state[dwt_depth_ho]=2`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	0	-
1	H	3	3	3	3	-
2	H	5	5	5	5	-
3	HL, LH, HH	-	7, 7, 10	7, 7, 10	7, 7, 10	-
4	HL, LH, HH	-	-	10, 10, 13	10, 10, 13	-
5	HL, LH, HH	-	-	-	13, 13, 16	-
		`state[dwt_depth_ho]=3`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	0	-	-
1	H	3	3	3	-	-
2	H	5	5	5	-	-
3	H	7	7	7	-	-
4	HL, LH, HH	-	8, 8, 12	8, 8, 12	-	-
5	HL, LH, HH	-	-	11, 11, 15	-	-
		`state[dwt_depth_ho]=4`				
		`state[dwt_depth]`				
Level	**Orientation**	**0**	**1**	**2**	**3**	**4**
0	L	0	0	-	-	-
1	H	3	3	-	-	-
2	H	5	5	-	-	-
3	H	7	7	-	-	-
4	H	8	8	-	-	-
5	HL, LH, HH	-	10, 10, 13	-	-	-

D.3 Quantization Matrix Design and Quantizer Selection (Informative)

D.3.1 General

D.3 provides an informative guide to the principles used to design the default quantization matrix.

D.3.2 Noise Power Normalization

The quantization matrices defined in D.2 are designed to counteract the differential power gain of the various wavelet filters, so that quantization noise from each subband is weighted equally in terms of its contribution to noise power when transformed back into the picture domain.

Let α and β represent the RMS noise gain factors of the low-pass and high-pass wavelet filters used in wavelet decomposition (see Annex F for a description of wavelet filtering). In the terminology of Annex F where *B* and *D* are the low-pass and high-pass synthesis filters, the gain factors are set so that:

$$\alpha = \left(\sum_n B(n)^2 \right)^{1/2} \text{ and } \beta = \left(\sum_n D(n)^2 \right)^{1/2}$$

(In this standard, wavelet synthesis filters have been defined in terms of lifting stages, which are filters operating on subsampled data. Wavelet filters are more conventionally represented in terms of an iterated binary filter bank: the relationship between these representations and how the lifting filters determine the filter bank is described in Annex F.)

In a single level of wavelet decomposition, quantization noise in each of the four subbands is therefore weighted by the factors shown in Figure D.1.

LL - α^2	HL - $\alpha\beta$
LH - $\alpha\beta$	HH - β^2

Figure D.1 — Subband weights for a 1-level decomposition.

For higher levels of decomposition, these subband weighting factors iterate in the same manner as the wavelet transform itself. For example, with a 1-level decomposition, the first level LL band, with weight α^2 is further decomposed to give four more bands with weights as for the 1-level decomposition, but multiplied by α^2. This yields the weights shown in Figure D.2.

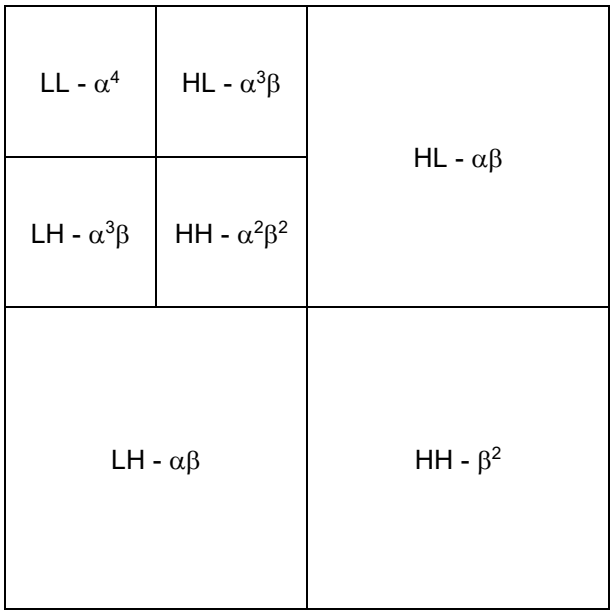

Figure D.2 — Subband weights for a 2-level decomposition.

The quantization offset for a subband is determined from the normalized power gain for that subband. Normalization is performed by dividing by the smallest power gain of all subbands, and ensures that the smallest quantization offset is zero. The actual quantization offset is determined from the normalized subband power gain, w, by computing $4*\log_2(w)$ rounded to the nearest integer. The tables in D.2 were produced using this process.

These factors also need to take into account the shift factors used to add accuracy bits prior to each wavelet decomposition stage. For a filter shift of d, α and β are each multiplied by $2^{-d/2}$.

A software implementation of the procedure described in D.3.2 is available at:
https://github.com/bbc/vc2_quantisation_matrices/

D.3.3 Custom Quantization Matrices

Custom matrices can also be defined that take into account not only noise power normalization but also other factors, for example perceptual weighting based on spatial frequency: additional multiplicative factors are computed for each subband, which produces a matrix of quantization offsets which can then be added to the default unweighted quantization matrices to produce a weighted quantization matrix.

An example perceptual weighting can be constructed from the ITU-R BT.959-2 Contrast Sensitivity Function (CSF). This is a function `csf(s)` which produces a value representing the sensitivity to detail at a given normalized spatial frequency *s*.

For luma, this is defined by:

$$csf(s) = 0.255 \times (1 + 0.2561 \times s^2)^{-0.75}$$

Assuming an isotropic response, we can form a 2-dimensional perceptual weighting function on horizontal and vertical spatial frequencies s_x, s_y by:

$$CSF(s_x, s_y) = csf\left(\left(s_x^2 + s_y^2\right)^{1/2}\right)$$

Each subband in a wavelet decomposition represents a subset of spatial frequencies according to level and orientation, partitioning the spatial frequency domain.

This partitioning is not normalized, since output pictures (and their compression artifacts) can be viewed over a range of distances.

Accordingly we can pick a representative, un-normalized horizontal and vertical spatial frequency ($f_x(b)$, $f_y(b)$), perhaps the middle frequency of the band. For example, an LH band b at level 1 in a 1-level decomposition will have mid frequency at ($pw/4$,$3*ph/4$) where ph and pw are the padded width and height of the picture. This can be turned into a true spatial frequency by normalizing by the number of horizontal and vertical cycles per degree the output pictures will subtend at the target viewing distance and aspect ratio:

$$\left(f_x(b)/cpd_x, f_y(b)/cpd_y\right)$$

and this value fed into the weighting function *CSF* to get a value $c(b)$. The appropriate quantization offset for that subband is then $4*\log_2(c(b))$, which can be used to modify the unweighted quantization matrix.

Annex E Video Systems Model (Informative)

E.1 Color Models

E.1.1 General

All current video systems use a YC_1C_2 form of coding of the RGB source values (Y,C_B,C_R is a commonly used variant of YC_1C_2). Although Y,C_B,C_R, is widely used, VC-2 can support other color systems such as Y,C_G,C_O as defined by ITU-T H.264 AVC Annex E. For this reason, the non-luma components are generalized to the terms C_1 and C_2.

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

The E_R, E_G, E_B values, are related to the linear RGB values by non-linear transfer functions. Normally, these values also fall in the range [0,1], but in the case of extended gamut systems, negative values can occur. The non-linear transfer function is typically performed in the camera and is specified in the transfer characteristic part of the appropriate color specification. For aesthetic and psycho-visual reasons, the encoding transfer function is not always the inverse of the decoding transfer function. In fact, the combined effect of the encoding and decoding transfer functions is such that the rendering intent or end-to-end gamma of the system can vary between about 1.1 and 1.6 depending on viewing conditions. The rationale for this is given in “Digital Video and HDTV” by Charles Poynton, (2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7).

The non-linear E_R, E_G, E_B values are subject to a matrix operation (known as ‘non-constant’ luma coding), which transforms them into luma (E_Y) and color difference (EC_1 and EC_2) values. E_Y is normally limited to the normalized range [0,1] and the color difference values to the normalized range [-0.5, 0.5]. In this standard, the color difference components are not to be confused with the chroma signals used by composite television systems where the color difference signals are significantly reduced in both resolution and signal amplitude. The color difference components used in this standard can be subsampled, either horizontally, vertically, or both horizontally and vertically.

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 luma and color difference values by direct operations subsuming and approximating all the real-number operations described here. Generally, these approximations cause loss through quantization of intermediate values, and the restriction of values to particular ranges also restricts the color gamut.

E.1.2 YC_BC_R Coding

The E_Y, EC_1, EC_2 values are mapped to values for Y, C_B, C_R. Typically they are mapped to an 8-bit unsigned integer [0, 255]. The way this mapping occurs is defined by the signal range parameters. It is these integer values that are actually output from a decoder. In order to display the decoded video, the inverse to the operations described in E.1.1 needs to be performed to convert this data to E_Y, EC_B, EC_R, then to E_R, E_G, E_B values and finally to R, G and B.

E.1.3 YC_GC_O Coding

In the case of Y, C_G, C_O coding, a lossless direct integer transform can be used, so that, together with 4:4:4 sampling and lossless compression, VC-2 can support efficient lossless RGB coding.

E.1.4 Signal Range

The offset and excursion values are used to convert the integer-valued decoded luma and color difference data Y, C_1, C_2 to normalized intermediate values E_Y, EC_1, and EC_2 such that:

- The E_Y value is the Y value scaled so that the range

  `[luma_offset, luma_offset + luma_excursion]`

  is mapped to the range [0,1],

- The EC_1 and EC_2 values are the C_1 and C_2 scaled so that the range

  `[color_diff_offset - color_diff_excursion/2, color_diff_offset + color_diff_excursion/2]`

  is mapped to [−0.5,0.5].

In video systems, values overshooting or undershooting these values are normally allowed.

In the case of Y, C_G, C_O coding, E_Y, EC_G, and EC_O are not calculated. Instead, direct integer conversion to RGB is done and excursion values will be ignored in this integer conversion.

E.1.5 Color Primaries

The color primaries allow device dependent linear RGB color coordinates to be mapped to device independent linear CIE XYZ space. The primaries specified are the CIE (1931) XYZ chromaticity coordinates of the primaries and the white point of the device.

The color primary specification therefore allows exact color reproduction of decoded RGB values on different displays with different display primaries.

E.1.6 Color Matrix

For conventional Y, C_B, C_R coding, luma and color difference values E_Y, EC_1, EC_2 are used to derive E_R, E_G, E_B values by applying a matrix.

In the case of Y, C_G, C_O coding, E_Y, EC_1, EC_2 are directly computed from the integer Y, C_G and C_O values by the following procedure, whereby integer RGB I_R, I_G, I_B values are decoded by:

Y −= `luma_offset`, C_G −= `color_diff_offset`, C_O −= `color_diff_offset`

$temp = Y - (C_G >> 1)$

$I_G = temp + C_G$, $I_B = temp - (C_O >> 1)$, $I_R = I_B + C_O$

These are scaled down by dividing by (255 << accuracy_bits) and clipped to give the E_R, E_G and E_B values.

If the inverse transform has been correctly applied prior to coding and lossless coding employed, then clipping will be unnecessary.

This matrix implies that the color difference range is twice as large as the RGB range (and the luma range), since the color difference 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.

E.2 Transfer Characteristics

E.2.1 TV Transfer Characteristic

Standard and high definition systems for both 50 Hz and 60 Hz based systems use an encoding gamma value of 0.45 with a linear portion at the low end of the scale to avoid the need for infinite gain at the receiver. The gamma values are defined in ITU-R BT.601-6 for standard definition, ITU-R BT.709 for high definition and ITU-R BT.2020 for ultra-high definition. The gamma specifications are defined in Recommendations ITU-R BT.601-6 for standard definition, ITU-R BT.709 for high definition and ITU-R BT.2020 for ultra-high definition. The gamma specifications in these Recommendations are equivalent for 10-bit signals.

E.2.2 Extended Color Gamut

ITU-R BT.1361 (Worldwide Unified Colorimetry of Future TV Systems) defined a color system with an extended color gamut. The document is now suppressed.

ISO/IEC 61966-2 (Extended RGB Color Space) defines another color system with an extended color gamut. Refer to IEC 61966-2-2:2003 for details.

Use of the full range of Y, C_1, C_2 values can create negative R, G or B values in this case. The original color gamut equations were designed around the CRT (cathode ray tube) device. Some flat panel displays are capable of displaying a wider color gamut resulting in the desire to extend the color gamut to maximize the impact of these displays.

E.2.3 Linear

A linear transfer characteristic has f(x)=x; i.e., $E_X = X$.

E.3 Frame Rate

The frame rate value encodes the intended rate at which frames are to be displayed at the output of the decoder. If the scan format value defines that the video is interlaced, then fields are displayed at double the frame rate, in the order specified by the `top_field_first` flag. If `interlaced` is false, then the lines of each frame are displayed directly.

E.4 Aspect Ratios and Clean Area

E.4.1 Pixel Aspect Ratio

E.4.1.1 General

The pixel aspect ratio value of an image is the ratio of the intended spacing of horizontal samples (pixels) to the spacing of vertical samples (picture lines) on the display device. Pixel aspect ratios are fundamental properties of sampled images because they determine the displayed shape of objects in the whole image. Failure to use the correct value of pixel aspect ratio will result in distorted images where circles will be displayed as ellipses.

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

For a number NH of pixels per unit length and NV pixels per unit height, this ratio is 1/NH : 1/NV or NV : NH. For a video standard of WxH pixels displayed at 4:3 picture aspect ratio, NH=W/4 and NV=H/3.

E.4.1.2 Using Non-Square Pixel Aspect Ratios

The defined pixel aspect ratios are designed to give image aspect ratios for standard definition television operating with a standard 4:3 picture aspect ratio.

For 525-line video, defining a 704 x 480 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 10:11 (i.e., 480/3 : 704/4).

For 625-line video defining a 704 x 576 picture with a 4:3 aspect ratio results in a H:V pixel aspect ratio of 12:11 (i.e., 576/3 : 704/4).

If the intended image aspect ratio is 16:9, then the H:V pixel aspect ratios change accordingly to 40:33 for 525-line video and 16:11 for 625-line video.

The values specified above are widely, but not unanimously, agreed to be the correct values. Differences of viewpoint arise from how much of the available horizontal picture size of 720 Y pixels is intended for display.

Implementers are advised to use one of the default pixel aspect ratios. VC-2 does allow non-standard pixel aspect ratios, although many display devices could ignore them and default to using different (and possibly unsuitable) values.

E.4.2 Clean Area

The clean area is intended to define an area within which picture information is subjectively uncontaminated by all edge distortions and possible unintended picture content such as microphones appearing at the top of the picture. It could be appropriate to display the clean area rather than the whole picture, which can contain edge distortions or unintended content.

The top-left corner of the clean area has coordinates (`left_offset`, `top_offset`) and dimensions `clean_width` and `clean_height`.

The clean area and the pixel aspect ratio can be used to determine the displayed image aspect ratio (which is the ratio of the width of the intended display area to the height of the intended display area). Regardless of the size of the clean area for display purposes, the pixel aspect ratio is maintained in order to avoid any geometric distortion.

Given two separate sequences, with identical image aspect ratios, if the top left corner and bottom left corners of their clean apertures are coincident when displayed, then all the images will 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.

Annex F Wavelet decimation and reconstruction processes (informative)

F.1 Overview of Wavelet Processing

Figure F.1 illustrates a single stage of a generalized wavelet decimation followed by reconstruction. The aim is to get perfect (or near-perfect) reconstruction of the output so that it is identical to the original input.

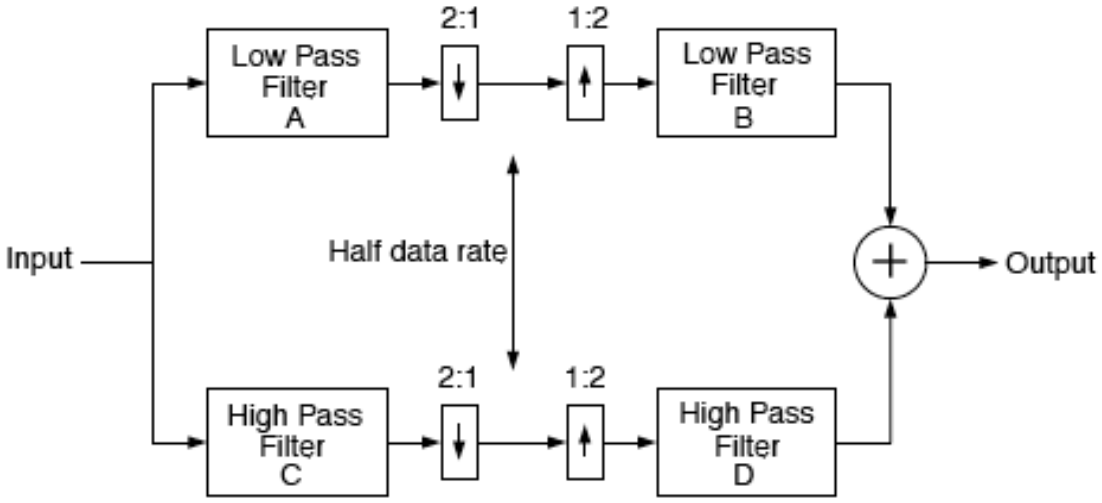

Figure F.1 — Single wavelet processing stage comprising decimation and reconstruction filters.

Figure F.2 illustrates how the frequency components are distributed both during decimation and reconstruction. This figure illustrates how the alias frequencies created during the decimation process are cancelled out during the reconstruction process. This feature of alias cancellation results from the wavelet process and is a specific attribute of wavelet coding. It is important to note that if the decoder receives imperfect signals (caused, for example, by quantization errors) then the imperfections will result in distortion in the reconstructed output.

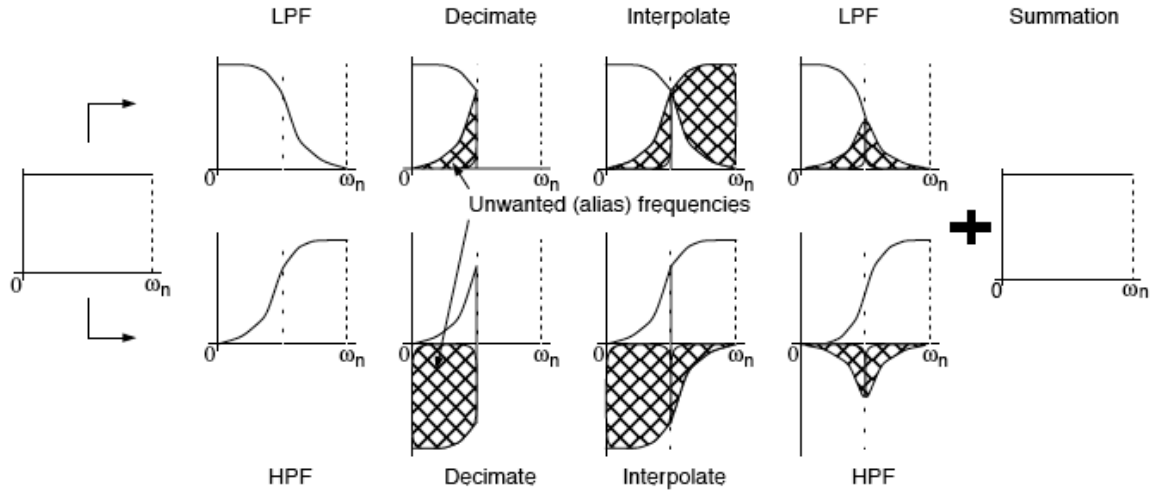

Figure F.2 — Illustration of the alias frequency generation and cancellation in a wavelet filter bank.

A single wavelet stage is insufficient for most video coding applications. Figure F.3 illustrates how only the low-pass path is passed on to the next wavelet decimation step. Because each step of the wavelet decimation is self-contained, the reconstructed output is still identical to the input (barring quantization errors).

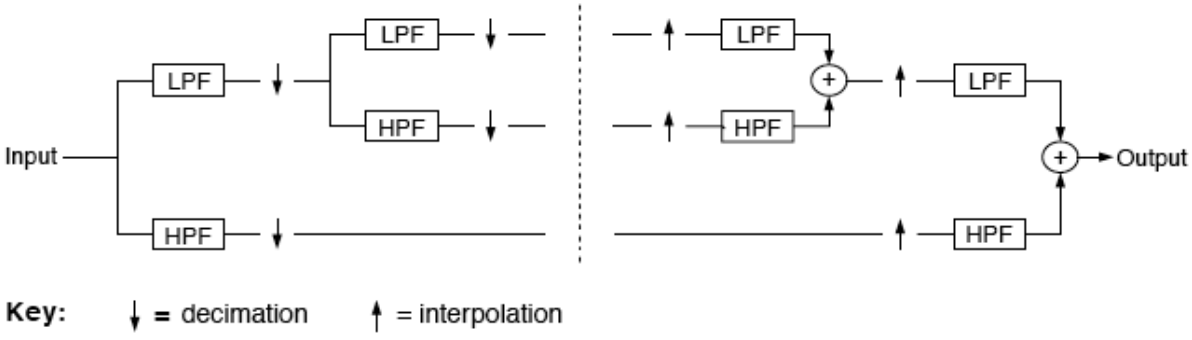

Figure F.3 — Two-step wavelet processing filter bank.

The application of wavelet filter banks in picture coding results in a two-dimensional decimation process as illustrated in Figure F.4.

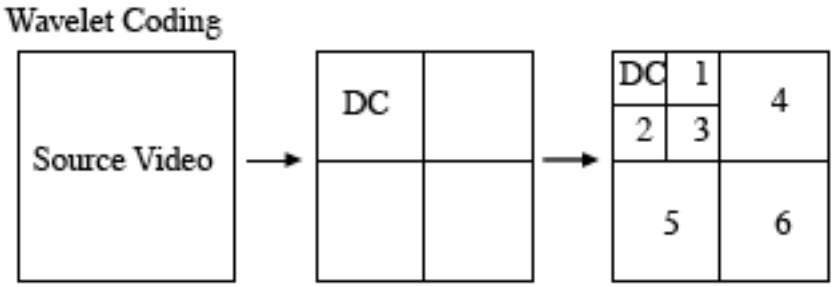

Figure F.4 — Decomposition of a single image into 7 wavelet frequency bands.

Figure F.5 illustrates how a real image is decimated to produce a low-frequency proxy in the top-left corner and a range of increasing frequency band components extending to the right side for increasing horizontal frequencies and downwards for increasing vertical frequencies.

Figure F.5 — Decomposition of the EBU “Boats” picture into 7 wavelet frequency bands.

F.2 The Lifting Process

This clause provides a mathematical definition of how the lifting process can be applied to wavelet filter banks.

For any set of filters, the analysis and synthesis filter banks illustrated in Figure F.1 can be easily re-expressed as polyphase filter banks by means of applying *matrices* of filters in the subsampled domain. This is shown in Figure F.6, where $A(z)$ is the z–transform of the analysis polyphase filter matrix, and $S(z)$ is the z–transform of the synthesis polyphase filter matrix (the entries of both matrices being Laurent polynomials).

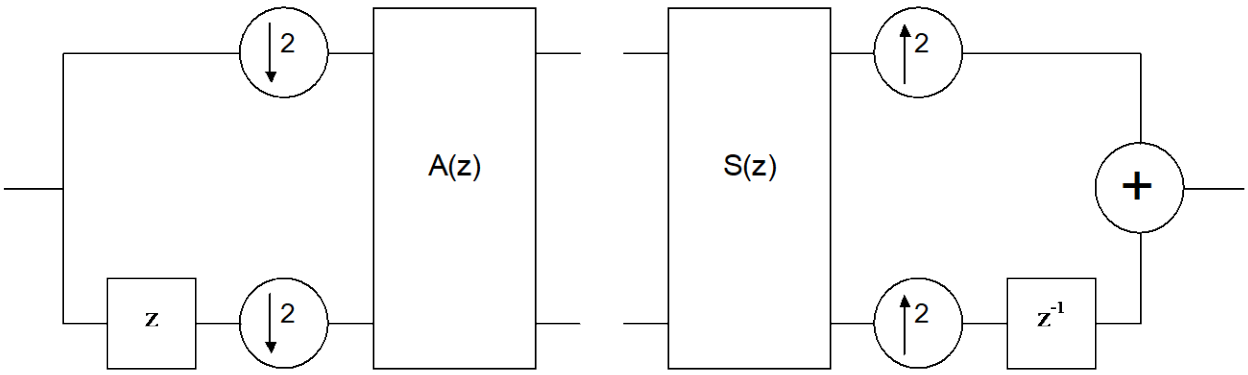

Figure F.6 — Polyphase representation of wavelet filter banks.

In this representation, linear combinations of filters operate on both even and odd samples to produce new even and odd samples:

$$\begin{pmatrix} x_e^{out}(z) \\ x_0^{out}(z) \end{pmatrix} = A(z) \begin{pmatrix} x_e^{in}(z) \\ x_0^{in}(z) \end{pmatrix}$$

Since the filter process is invertible, it can be shown that the analysis and synthesis matrices are related by

$$A(z) = (S(z^{-1})^T)^{-1}$$

Hence, in particular both the analysis and synthesis matrices are invertible. It can also be shown that this means that they are (up to gain factors and delays) factorizable into products of upper and lower triangular matrices as follows:

$$A(z) = \begin{pmatrix} 1 & a_1(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b_1(z) & 1 \end{pmatrix} \begin{pmatrix} 1 & a_2(z) \\ 0 & 1 \end{pmatrix} \ldots$$

Each upper or lower-triangular polyphase matrix represents a so-called *lifting* stage whereby either even coefficients are modified solely by odd coefficients or odd coefficients solely by even coefficients.

For example, if

$$\begin{pmatrix} x_e^{out}(z) \\ x_0^{out}(z) \end{pmatrix} = \begin{pmatrix} 1 & a(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_e^{in}(z) \\ x_0^{in}(z) \end{pmatrix}$$

then

$$x_e^{out}(z) = x_e^{in}(z) + a(z)x_0^{in}(z)$$
$$x_0^{out}(z) = x_0^{in}(z)$$

and the filter $a(z)$ has been applied to the odd coefficients and then used to modify the even coefficients. Not only is this computationally efficient by breaking long filters into a number of shorter filters successively applied, but the factorization into such filter stages allows for computations to be done in place.

F.3 Signal Ranges

Care is needed to ensure sufficient integer precision for all arithmetic operations during wavelet filtering. The wavelet transform and quantization can both cause signal levels to grow significantly above input picture levels.

Bibliography

ISO/IEC 15444-1:2002, JPEG 2000 Image Coding System

ITU-R Report BT.959-2 (1990), Experimental Results Relating Picture Quality to Objective Magnitude of Impairment

ISO/IEC 61966-2-2:2003, Multimedia Systems and Equipment — Colour Measurement and Management — Part 2-2: Colour Management — Extended RGB Colour Space — scRGB

ISO/IEC 646:1991, Information Technology - ISO 7-Bit Coded Character Set for Information Interchange

SMPTE ST 125:2013, SDTV Component Video Signal Coding 4:4:4 and 4:2:2 for 13.5 MHz and 18 MHz Systems

SMPTE ST 2042-2:2017, VC-2 Level Definitions

SMPTE RP 177:1993, Derivation of Basic Television Color Equations

SMPTE RP 202:2008, Video Alignment for Compression Coding

“Digital Video and HDTV”, Charles Poynton 2003, Morgan Kaufmann Publishers, ISBN 1-55860-792-7

“Ripples in Mathematics, The Discrete Wavelet Transform”, A. Jenson and A. la Cour-Harbo, Springer, ISBN 3-540-41662-5