INTERNATIONAL STANDARD

ISO/IEC 18181-1

First edition
2022-03

Information technology — JPEG XL image coding system —

Part 1: Core coding system

Technologies de l'information — Système de codage d'images JPEG XL —

Partie 1: Système de codage de noyau

Reference number
ISO/IEC 18181-1:2022(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all parts in the ISO/IEC 18181 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — JPEG XL image coding system —

Part 1: Core coding system

1 Scope

This document defines a set of compression methods for coding one or more images of bi-level, continuous-tone greyscale, or continuous-tone colour, or multichannel digital samples.

This document:

— specifies decoding processes for converting compressed image data to reconstructed image data;

— specifies a codestream syntax containing information for interpreting the compressed image data;

— provides guidance on encoding processes for converting source image data to compressed image data.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 15076-1:2010, *Image technology colour management — Architecture, profile format and data structure — Part 1: Based on ICC.1:2010*

ISO/IEC 60559, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

IEC 61966-2-1, *Multimedia systems and equipment — Colour measurement and management — Part 2-1: Colour management — Default RGB colour space — sRGB*

Rec. ITU-R BT.2100-2, *Image parameter values for high dynamic range television for use in production and international programme exchange*

Rec. ITU-R BT.709-6, *Parameter values for the HDTV standards for production and international programme exchange*

SMPTE ST 428-1, *D-Cinema distribution master — Image characteristics*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1 Data storage

3.1.1
byte
8 consecutive bits encoding a value between 0 and 255

3.1.2
big endian
value representation with bytes in most to least-significant order

3.1.3
bitstream
sequence of bytes from which bits are read starting from the least-significant bit of the first byte

3.1.4
codestream
bitstream representing compressed image data

3.1.5
bundle
structured data consisting of one or more fields

3.1.6
field
numerical value or bundle, or an array of either

3.1.7
histogram
array of unsigned integers representing a probability distribution, used for entropy coding

3.1.8
set
unordered collection of elements

3.2 Inputs

3.2.1
pixel
vector of dimension corresponding to the number of channels, consisting of samples

3.2.2
sample
integer or real value, of which there is one per channel per pixel

3.2.3
greyscale
image representation in which each pixel is defined by a single sample representing intensity (either luminance or luma depending on the ICC profile)

3.2.4
continuous-tone image
image having samples consisting of more than one bit

3.2.5
opsin
photosensitive pigments in the human retina, having dynamics approximated by the XYB colour space

3.2.6
burst
sequences of images typically captured with identical settings

3.2.7
animation
series of pictures and timing delays to display as a video medium

3.2.8
composite
series of images that are superimposed

3.2.9
frame
single image (possibly part of a burst or animation or composite)

3.2.10
preview
lower-fidelity rendition of one of the frames (e.g. lower resolution), or a frame that represents the entire content of all frames

3.3 Processes

3.3.1
decoding process
process which takes as its input a codestream and outputs a continuous-tone image

3.3.2
decoder
embodiment of a decoding process

3.3.3
encoding process
process which takes as its input continuous-tone image(s) and outputs compressed image data in the form of a codestream

3.3.4
encoder
embodiment of an encoding process

3.3.5
lossless
descriptive term for encoding and decoding processes in which the output of a decoding procedure is identical to the input to the encoding procedure

3.3.6
lossy
descriptive term for encoding and decoding processes which are not lossless

3.3.7
upsampling
procedure by which the (nominal) spatial resolution of a channel is increased

3.3.8
downsampling
procedure by which the spatial resolution of a channel is reduced

3.3.9
entropy encoding
lossless procedure designed to convert a sequence of input symbols into a sequence of bits such that the average number of bits per symbol approaches the entropy of the input symbols

3.3.10
entropy encoder
embodiment of an entropy encoding procedure

3.3.11
entropy decoding
lossless procedure which recovers the sequence of symbols from the sequence of bits produced by the entropy encoder

3.3.12
entropy decoder
embodiment of an entropy decoding procedure

3.3.13
Gabor-like transform
convolution with default or signalled 3x3 kernel for deblocking

3.3.14
tick
unit of time such that animation frame durations are integer multiples of the tick duration

3.4 Image organization

3.4.1
grid
2-dimensional array; `a [x, y]` means addressing an element of grid `a` at row `y` and column `x`. Where so specified, addressing elements with coordinates outside of bounding rectangle (`x < 0`, or `y < 0`, or `x >= width`, or `y >= height`) is allowed

3.4.2
sample grid
common coordinate system for all samples of an image, with top-left coordinates (0, 0), the first coordinate increasing towards the right, and the second increasing towards the bottom

3.4.3
channel
component
rectangular array of samples having the same designation, regularly aligned along a sample grid

3.4.4
rectangle
rectangular area within a channel or grid

3.4.5
width
width in samples of a sample grid or a rectangle

3.4.6
height
height in samples of a sample grid or a rectangle

3.4.7
raster order
access pattern from left to right in the top row, then in the row below and so on

3.4.8
naturally aligned
positioning of a power-of-two sized rectangle such that its top and left coordinates are divisible by its width and height, respectively

3.4.9
block
naturally aligned square rectangle covering up to 8 × 8 input pixels

3.4.10
group
naturally aligned square rectangle covering up to $2^n \times 2^n$ (with n between 7 and 10, inclusive) input pixels

3.4.11
table of contents
data structure that enables seeking to a group or the next frame within a codestream

3.4.12
section
part of the codestream with an offset and length that are stored in a frame's table of contents

3.5 DCT

3.5.1
coefficient
input value to the inverse DCT

3.5.2
quantization
method of reducing the precision of individual coefficients

3.5.3
varblock
variable-size rectangle of input pixels

3.5.4
dct_block
an array with 64 elements corresponding to DCT coefficients of a (8 × 8) block

3.5.5
var-DCT
lossy encoding of a frame that applies DCT to varblocks

3.5.6
LF coefficient
lowest frequency DCT coefficient, containing the average value of a block or the lowest-frequency coefficient within the 8 × 8 rectangle of a varblock of size greater than 8 × 8

3.5.7
HF coefficients
all DCT coefficients apart from the LF coefficients, i.e. the high frequency coefficients

3.5.8
pass
data enabling decoding of successively higher resolutions

3.5.9
LF group
$2^n \times 2^n$ LF values from a naturally aligned rectangle covering up to $2^{n+3} \times 2^{n+3}$ input pixels

3.5.10
quantization weight
factor that a quantized coefficient is multiplied by prior to application of the inverse DCT in the decoding process

3.5.11
channel decorrelation
method of reducing total encoded entropy by removing correlations between channels

3.5.12
channel correlation factor
factor by which a channel should be multiplied by before adding it to another channel to undo the channel decorrelation process

4 Abbreviated terms

DCT: discrete cosine transform (DCT-II as specified in I.2)

IDCT: inverse discrete cosine transform (DCT-III as specified in I.2)

LF: N / 8 × M / 8 square of lowest frequency coefficients of N × M DCT coefficients

RGB: additive colour model with red, green, blue channels

LMS: absolute colour space representing the response of cone cells in the human eye

XYB: absolute colour space based on gamma-corrected LMS, in which X is derived from the difference between L and M, Y is an average of L and M (behaves similarly to luminance), and B is derived from the S ("blue") channel

5 Conventions

5.1 Mathematical symbols

`[a, b], (c, d), [e, f)`	closed or open or half-open intervals containing all integers or real numbers x (depending on context) such that `a ≤ x ≤ b, c < x < d, e ≤ x < f.`
`{a, b, c}`	ordered sequence of elements
`π`	the smallest positive zero of the sine function

5.2 Functions

`sqrt(x)`	square root, such that `(sqrt(x))`^{2} `== x` and `sqrt(x) >= 0`. Undefined for x < 0.
`cbrt(x)`	cube root, such that `(cbrt(x))`^{3} `== x`.
`cos(r)`	cosine of the angle r (in radians)
`erf(x)`	Gauss error function: `erf(x)` $= \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
`log(x)`	natural logarithm of x. Undefined for x <= 0.
`log2(x)`	base-two logarithm of x. Undefined for x <= 0.
`floor(x)`	the largest integer that is less than or equal to x
`ceil(x)`	the smallest integer that is greater than or equal to x
`abs(x)`	absolute value of x: equal to -x if x < 0, otherwise x
`sign(x)`	sign of x, 0 if x is 0, +1 if x is positive, -1 if x is negative
`UnpackSigned(u)`	equivalent to u / 2 if u is even, and -(u + 1) / 2 if u is odd

`clamp(x, lo, hi)`	equivalent to `min({max({lo, x}), hi})`
`InterpretAsF16(u)`	the real number resulting from interpreting the unsigned 16-bit integer `u` as a binary16 floating-point number representation (cf. ISO/IEC 60559)
`InterpretAsF32(u)`	the real number resulting from interpreting the unsigned 32-bit integer `u` as a binary32 floating-point number representation (cf. ISO/IEC 60559)
`len(a)`	length (number of elements) of array a
`sum(a)`	sum of all elements of the array/tuple/sequence a
`max(a)`	maximal element of the array/tuple/sequence a
`min(a)`	smallest element of the array/tuple/sequence a

5.3 Operators

This document uses the operators defined by the C++ programming language [2], with the following differences:

`×`	multiplication
`×=`	`a ×= b` is equivalent to `a = a × b`
`/`	division of real numbers without truncation or rounding. Division by zero is undefined.
x^y	exponentiation, x to the power of y
`<<`	left shift: `x << s` is defined as $x \times 2^s$
`>>`	right shift: `x >> s` is defined as `floor(x / 2`^{s}`)`
`Umod`	a `Umod` d is the unique integer r in [0, d) for which `a == r + q×d` for a suitable integer q
`Idiv`	a `Idiv` b is equivalent to `a / b`, rounded towards zero to an integer value

The order of precedence for these operators is listed below in descending order. If several operators appear in the same line, they have equal precedence. When several operators of equal precedence appear at the same level in an expression, evaluation proceeds according to the associativity of the operator (either from right to left or from left to right).

Operators	**Type of operation**	**Associativity**
`++x, --x`	prefix increment/decrement	right to left
x^y	exponentiation	right to left
`!, ~`	logical/bitwise NOT	right to left
`×, /, Idiv, Umod`	multiplication, division, integer division, remainder	left to right
`+, -`	addition and subtraction	left to right
`<<, >>`	left shift and right shift	left to right
`< , >, <=, >=`	relational	left to right
`=`	assignment	right to left
`+=, -=, ×=`	compound assignment	right to left

5.4 Pseudocode

This document describes functionality using pseudocode formatted as follows:

```
// Informative comment
var = u(8);  // Defined in 9.2.1
if (var == 1) return;  // Stop executing this code snippet
[[Normative specification: var != 0]]
(out1, out2) = Function(var, kConstant);
```

Variables such as `var` are typically referenced by text outside the source code.

The semantics of this pseudocode are those of the C++ programming language [2], with the following exceptions:

— Symbols from 5.1 and functions from 5.2 are allowed;

— Multiplication, division, remainder and exponentiation are expressed as specified in 5.3;

— Functions can return tuples which unpack to variables as in the above example;

— `[[ ]]` enclose normative directives specified using prose;

— All integers are stored using two's complement;

— Expressions and variables of which types are omitted, are understood as real numbers.

Where unsigned integer wraparound and truncated division are required, `Umod` and `Idiv` (see 5.3) are used for those purposes.

Numbers with a 0x prefix are in base 16 (hexadecimal), and apostrophe (') characters inside them are understood to have no effect.

EXAMPLE 0x0001'0000 == 65536.

6 Functional concepts

6.1 Image organization

A channel is defined as a rectangular array of (integer or real) samples regularly aligned along a sample grid of `width` sample positions horizontally and `height` sample positions vertically. The number of channels may be 1 to 4099 (see `num_extra_channels` in A.6).

A pixel is defined as a vector of dimension corresponding to the number of channels, consisting of samples with a position matching that of the pixel. The index of a sample is numbered from 0 to number of channels - 1.

An image is defined as the two-dimensional array of pixels, and its width is `width` and height is `height`. Unless otherwise mentioned, channels are accessed in the following "raster order": left to right column within the topmost row, then left to right column within the row below the top, and so on until the rightmost column of the bottom row.

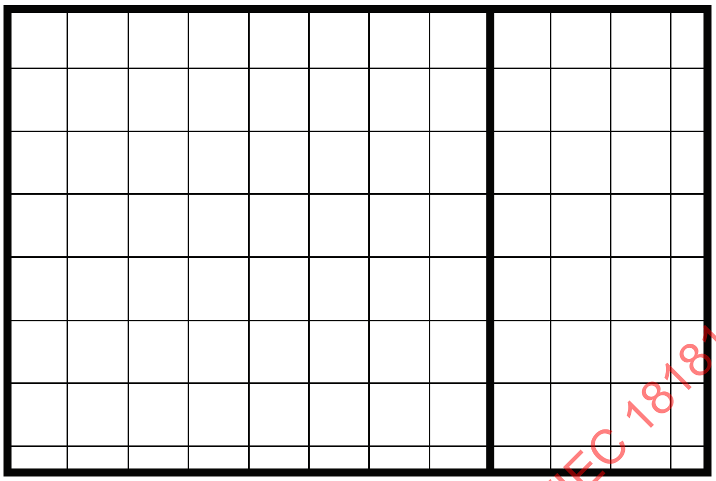
6.2 Group splitting

Channels are logically partitioned into naturally-aligned groups of `kGroupDim` × `kGroupDim` samples. The effective dimension of a group (i.e. how many pixels to read) can be smaller than `kGroupDim` for groups on the right or bottom of the image. The decoder ensures the decoded image has the dimensions specified in `SizeHeader` by cropping at the right and bottom as necessary. Unless otherwise specified, `kGroupDim` is 256.

LF groups likewise consist of `kGroupDim` × `kGroupDim` LF samples, with the possibility of a smaller effective size on the right and bottom of the image.

Groups can be decoded independently. A 'table of contents' stores the size (in bytes) of each group to allow seeking to any group. An optional permutation allows groups to be arranged in arbitrary order within the codestream.

EXAMPLE Figure 1 shows an example of the HF groups and LF groups of an image.



Frame: 2970×1868 pixels

HF groups:
11×7 groups of 256×256 pixels,
1×7 groups of 154×256 pixels,
7×1 groups of 256×76 pixels,
1 group of 154×76 pixels

LF groups:
1 group of 256×233 LF coefficients
(covering 2048×1868 pixels),
1 group of 116×233 LF coefficients
(covering 922×1868 pixels)

Figure 1 — Group splitting example

6.3 Codestream and bitstream

A bitstream is a finite sequence of bytes. A codestream is a bitstream that represent compressed image data and metadata. N bytes can also be viewed as 8 × N bits. The first 8 bits are the bits constituting the first byte, in least to most significant order, the next eight bits (again in least to most significant order) constitute the second byte, and so on. Unless otherwise specified, bits are read from the codestream as specified in 9.2.1.

NOTE Ordering bits from least to most significant allows using special CPU instructions to isolate the least-significant bits.

Subsequent Annexes or subclauses indicate some elements of the codestream are byte-aligned. For such elements, the decoder takes actions before and after reading the element as follows. Immediately before encountering the element, the decoder invokes ZeroPadToByte() (9.2.9). After finishing reading the element, the decoder invokes ZeroPadToByte() (9.2.9).

ZeroPadToByte specifies that the padding bits, if any, are zero for the codestream to be valid. This can serve as an additional indicator of codestream integrity.

6.4 Multiple frames

A codestream may contain multiple frames. These can constitute an animation, a burst (arbitrary images with identical dimensions), or a composite still image with one or more frames rendered on top of the first frame.

NOTE The frame that is being decoded is referred to as the current frame.

6.5 Mirroring

Some operations access samples with coordinates `cx, cy` that are outside the image bounds. The decoder redirects such accesses to a valid sample at the coordinates `Mirror1D(cx, width), Mirror1D(cy, height)`, defined in the following code:

```
Mirror1D(coord, size) {
  if (coord < 0) return Mirror1D(-coord - 1, size);
  else if (coord >= size) return Mirror1D(2 × size - 1 - coord, size);
  else return coord;
}
```

7 Encoder requirements

An encoder is an embodiment of the encoding process. This document does not specify an encoding process, and any encoding process is acceptable as long as the codestream conforms to the codestream syntax specified in this document. Annex M provides an informative description of such an encoding process.

8 Decoder requirements

A decoder is an embodiment of the decoding process. The decoder reconstructs sample values (arranged on a rectangular sampling grid) from a codestream as specified in this document. Annexes A to L are normative in the sense that they are defining an output that alternative implementations shall duplicate.

9 Codestream

9.1 Syntax

The codestream is organized into "bundles" consisting of one or more conceptually related "fields". A field can be a bundle, or an array/value of a type from 9.2. The graph of contained-within relationships between types of bundles is acyclic. This document specifies the structure of each bundle using tables structured as in Table 1, with one row per field (in top to bottom order).

Table 1 — Structure of a table describing a bundle

condition	type	default	name

If `condition` is blank or evaluates to true, the field is read from the codestream (9.1.1). Otherwise, the field is instead initialized to `default` (9.1.2).

A `condition` of the form `for(i = 0; condition; ++i)` is equivalent to replacing this row with a sequence of rows obtained from the current one by removing its condition and replacing the value of `i` in each column with the consecutive values assumed by the loop-variable i until condition is false. If condition is initially false, the current row has no effect.

If name ends with [n] then the field is a fixed-length array with n entries. If condition is blank or evaluates to true, each array element is read from the codestream (9.1.1) in order of increasing index. Otherwise, each element is initialized to default (9.1.2). The name is potentially referenced in the condition or type of a subsequent row, or the condition of the same row.

9.1.1 Reading a field

If a field is to be read from the codestream, the type entry determines how to do so. If it is a basic field type, it is read as described in 9.2. Otherwise type is a (nested) bundle denoted Nested, residing within a parent bundle Parent. Nested is read as if the rows of the table defining Nested were inserted into the table defining Parent in place of the row that defined Nested. This principle is applied recursively, corresponding to a depth-first traversal of all fields.

9.1.2 Initializing a field

If a field is to be initialized to default, the type entry determines how to do so. If it is a bundle, then default is blank and each field of the bundle is (recursively) initialized to the default specified within the bundle's table. Otherwise, if default is not blank, the field is set to default, which is a valid value of the same type.

9.2 Field types

9.2.1 u(n)

u(0) evaluates to the value zero without reading any bits. For n > 0, u(n) reads n bits from the codestream, advances the position accordingly, and returns the value in the range $[0, 2^n)$ represented by the bits. The decoder first reads the least-significant bit of the value, from the least-significant not yet consumed bit in the first not yet fully consumed byte of the codestream. The next bit of the value (in increasing order of significance) is read from the next (in increasing order of significance) bit of the same byte unless all its bits have already been consumed, in which case the decoder reads from the least-significant bit of the next byte, and so on.

9.2.2 U32(d0, d1, d2, d3)

In this subclause, 'distribution' refers to one of the following three encodings of a range of values: Val(u), Bits(n), or BitsOffset(n, offset). The d0, d1, d2, d3 parameters represent distributions.

U32(d0, d1, d2, d3) reads an unsigned 32-bit value in $[0, 2^{32})$ as follows. The decoder first reads a u(2) from the codestream indicating which distribution to use (0 selects d0, 2 selects d2 etc.). Let d denote this distribution, which determines how to decode value.

If d is Val(u), value is the integer u. If d is Bits(n), value is read as u(n). If d is BitsOffset(n, offset), the decoder reads v = u(n). The resulting value is (offset + v) Umod 2^{32}.

NOTE The value of u is implicitly defined by Val(u) and not stored in the codestream.

EXAMPLE For a field of type U32(Val(8), Val(16), Val(32), Bits(7)), the bits 10 result in value = 32. For a U32(Bits(2), Bits(4), Bits(6), Bits(8)) field, the bits 010111 result in value = 7.

9.2.3 U64()

U64() reads an unsigned 64-bit value in $[0, 2^{64})$ using a single variable-length encoding. The decoder first reads a u(2) selector s. If s == 0, value is 0. If s == 1, value is BitsOffset(4, 1) (9.2.2). If s == 2, value is BitsOffset(8, 17). Otherwise s == 3 and value is read from a 12-bit part, zero or more 8-bit parts, and zero or one 4-bit part as specified by the following code:

```
value = u(12); shift = 12;
while (u(1) == 1) {
  if (shift == 60) {
    value += u(4) << shift; // only 4, we already read 60
    break;
  }
  value += u(8) << shift; shift += 8;
}
```

EXAMPLE the largest possible value (2^{64} - 1) is encoded as 73 consecutive 1-bits.

9.2.4 Varint()

Varint() reads an unsigned integer `value` of up to 63 bits as specified by the following code:

```
value = 0; shift = 0;
while (1) {
  b = u(8);
  value += (b & 127) << shift;
  if (b <= 127) break;
  shift += 7;   [[ shift < 63 ]];
}
```

9.2.5 U8()

U8() reads an integer `value` in the range [0, 256) using [1, 12) bits as specified by the following code:

```
if (u(1) == 0) value = 0;
else { n = u(3); value = u(n) + (1 << n); }
```

EXAMPLE The bit 0 results in value 0, bits 1000 result in value 1, bits 10011 in value 3.

9.2.6 F16()

F16() reads a binary16 representation (as specified in ISO/IEC 60559) of a real `value` in [-65504, 65504]. `value` is as specified by the following code:

```
bits16 = u(16);
biased_exp = (bits16 >> 10) & 0x1F;
value = InterpretAsF16(bits16);
```

The value of `biased_exp` is not 31.

NOTE This rules out NaN and infinities.

9.2.7 Bool()

Bool() reads a boolean value as u(1) ? true : false.

9.2.8 Enum(`EnumTable`)

Enum(`EnumTable`) reads v = U32(Val(0), Val(1), BitsOffset(4, 2), BitsOffset(6, 18)). The value v does not exceed 63 and is a `value` defined by the table in the subclause titled `EnumTable`. Such tables are structured according to Table 2, with one row per unique `value`.

Table 2 — Structure of a table describing an enumerated type

name	value	meaning

An enumerated type is interpreted as having the `meaning` of the row where `value` is v. `name` (or where ambiguous, `EnumTable.name`) is an identifier for purposes of referring to the same `meaning` elsewhere in this document. `name` begins with a k prefix, e.g. `kRGB`.

9.2.9 ZeroPadToByte()

The decoder reads a u(n), where `n` is zero if `P`, the 0-based index of the next unread bit in the codestream, is a multiple of 8, otherwise `n` = 8 - (`P` Umod 8). The result is equal to zero.

NOTE The effect of ZeroPadToByte() is skip to the next byte boundary if not already at a byte boundary, and require all skipped bits to have value 0.

9.3 Structure

The codestream consists of headers (Annexes A, H) and an ICC profile, if present (Annex B), followed by one or more frames (Annex C), as shown in Table 3. This table and those it references, together with the syntax description (9.1), specify how a decoder reads the headers and frame(s).

Table 3 — Codestream structure

condition	type	name	Annex
	Headers	`headers`	A, H
`headers.metadata.want_icc`		`icc`	B
`headers.metadata.have_preview`	Frame	`preview_frame`	C
	Frame	`frames`[0]	C
`for (i = 1; !frames[i - 1].frame_header.is_last; ++i)`	Frame	`frames`[`i`]	C

The `preview_frame` is a self-contained frame whose FrameHeader (C.2) specifies `lf_level`=0, `frame_type=kRegularFrame` and `save_as_reference`=0.

10 Decoding process

After reading the header and frame data (9.3 and Annexes A, B and C), the decoder can be viewed as a pipeline with multiple stages, as shown in Figure 2.

Some Annexes or subclauses begin with a condition; the decoder only applies the processing steps described in such an Annex or subclause if its condition holds true.

Annex D specifies how the decoder reads entropy-coded data.

Annex E specifies how the decoder transforms channels decoded as residuals of a self-correcting weighted predictor into reconstructed samples.

The decoder converts adaptively quantized integers to DCT coefficients as specified in Annex F.

Annex G specifies how the decoder recorrelates channels stored as differences w.r.t. a linear function of another channel for the purpose of decorrelation ('chroma from luma').

The decoder performs or skips inverse integral transforms as specified in Annex I.

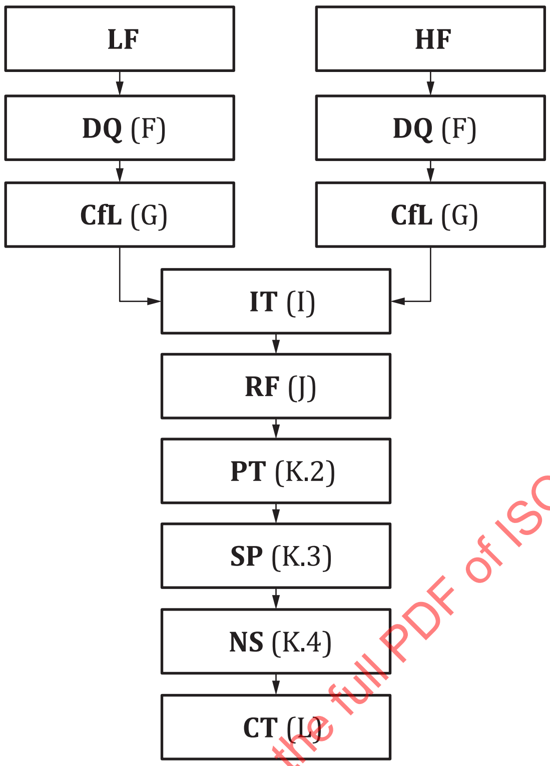
The decoder applies zero, one or two or restoration filters as specified in Annex J.

The presence/absence of additional image features (patches, splines and noise) is indicated in the frame header. The decoder draws these as specified in Annex K. Image features (if present) are rendered after restoration filters (if enabled), in the listed order.

Finally, the decoder performs zero, one or more colour-space transforms as specified in Annex L.

NOTE For an introduction to the coding tools and additional background, refer to the Joint Photographic Experts Group (JPEG) committee JPEG XL website [5].

EXAMPLE The lossless mode does not involve adaptive dequantization.



Key

DQ	dequantization
CfL	chroma from luma
IT	integral transform
RF	restoration filter
PT	patches
SP	splines
NS	noise
CT	colour transform

Figure 2 — Decoder block diagram after LF/HF coefficients of a frame have been read

Annex A
(normative)

Headers

A.1 General

The decoder reads the Headers bundle (Table A.1) as specified in 9.1.

Table A.1 — Headers bundle

condition	type	name	subclause
	Signature	`signature`	A.2
	SizeHeader	`size`	A.3
	ImageMetadata	`metadata`	A.6

A.2 Signature

Table A.2 specifies the Signature bundle.

Table A.2 — Signature bundle

condition	type	default	name
	u(8)		`ff`
	u(8)		`type`

`ff` is 255. `type` is 10.

A.3 Image dimensions

Table A.3 specifies the SizeHeader bundle.

Table A.3 — SizeHeader bundle

condition	type	default	name
	Bool()	false	`small`
`small`	u(5)	0	`height_div8_minus_1`
`!small`	U32(Bits(9), Bits(13), Bits(18), Bits(30))	0	`height_minus_1`
	u(3)	0	`ratio`
`small` && `ratio` == 0	u(5)	0	`width_div8_minus_1`
`!small` && `ratio` == 0	U32(Bits(9), Bits(13), Bits(18), Bits(30))	0	`width_minus_1`

`height` is defined as `small` ? (`height_div8_minus_1` + 1) × 8 : `height_minus_1` + 1. `width` is defined as follows. If `ratio` == 0, then `width` = `small` ? (`width_div8_minus_1` + 1) × 8 : `width_minus_1` + 1. Otherwise, `ratio` is in the range [1, 8) and `width` is computed as defined in Table A.4.

NOTE 1 These encodings cannot represent a zero-width or zero-height image.

Table A.4 — AspectRatio

ratio	Meaning
0	width coded separately
1	width = height
2	width = floor(height × 12 / 10)
3	width = floor(height × 4 / 3)
4	width = floor(height × 3 / 2)
5	width = floor(height × 16 / 9)
6	width = floor(height × 5 / 4)
7	width = floor(height × 2 / 1)

If the `ratio` field of `SizeHeader` or `PreviewHeader` != 0, their `width` field are computed from `height` and `ratio` as defined in Table A.4.

NOTE 2 Unlike other bundles, SizeHeader provides a guarantee on when the decoder can access its fields. In a streaming use case, it can be helpful for the decoder to know the image dimensions before having received the entire codestream. Once at least 94 bits of the codestream are accessible, or the entire codestream is known to have been transmitted, the decoder can decode SizeHeader and access its fields.

Table A.5 specifies the PreviewHeader bundle.

Table A.5 — PreviewHeader bundle

condition	type	default	name
	Bool()		`div8`
`div8`	U32(Val(16), Val(32), BitsOffset(5, 1), BitsOffset(9, 33))	1	`height_div8`
`!div8`	U32(BitsOffset(6, 1), BitsOffset(8, 65), BitsOffset(10, 321), BitsOffset(12, 1345))	8 × `height_div8`	`height`
	u(3)		`ratio`
`div8` && `ratio` == 0	U32(Val(16), Val(32), BitsOffset(5,1), BitsOffset(9, 33))	1	`width_div8`
`!div8` && `ratio` == 0	U32(BitsOffset(6, 1), BitsOffset(8, 65), BitsOffset(10, 321), BitsOffset(12, 1345))	1	`w_explicit`

NOTE 3 The presence of PreviewHeader is signalled by `metadata.have_preview`.

`width` is defined as follows. If `ratio` == 0, then `width` = (`div8` ? `width_div8` × 8 : `w_explicit`). Otherwise, `ratio` is in the range [1, 8) and `width` is computed as defined in Table A.4. The values of `width` and `height` do not exceed 4096.

NOTE 4 This encoding cannot represent a zero-width or zero-height preview.

A.4 ColourEncoding

The ColourEncoding bundle is specified in Table A.13. It references Table A.6 (Customxy), Table A.7 (ColourSpace), Table A.8 (WhitePoint), Table A.9 (Primaries), Table A.10 (TransferFunction), Table A.11 (CustomTransferFunction), and Table A.12 (RenderingIntent).

Table A.6 — Customxy bundle

condition	type	default	name
	U32(Bits(19), BitsOffset(19, 524288), BitsOffset(20, 1048576), BitsOffset(21, 2097152))	0	`ux`
	U32(Bits(19), BitsOffset(19, 524288), BitsOffset(20, 1048576), BitsOffset(21, 2097152))	0	`uy`

`x = UnpackSigned(ux)` and `y = UnpackSigned(uy)` are the coordinates of a point on the CIE xy chromaticity diagram, scaled by 10^6. The unscaled coordinates may be outside [0, 1] for imaginary primaries.

Table A.7 — ColourSpace

name	value	meaning
`kRGB`	0	Tristimulus RGB, with various white points and primaries
`kGrey`	1	Luminance, with various white points; `colour_encoding.primaries` are ignored
`kXYB`	2	XYB (opsin); `colour_encoding.white_point` is `kD65`, `colour_encoding.primaries` is ignored, `colour_encoding.have_gamma` is true, and `colour_encoding.gamma` is 3333333
`kUnknown`	3	None of the other table entries describe the colour space appropriately

Table A.8 — WhitePoint

name	value	meaning
`kD65`	1	CIE Standard Illuminant D65: 0.3127, 0.3290
`kCustom`	2	Custom white point stored in `colour_encoding.white`
`kE`	10	CIE Standard Illuminant E (equal-energy): 1/3, 1/3
`kDCI`	11	DCI-P3 from SMPTE RP 431-2: 0.314, 0.351

The `meaning` column is interpreted as CIE xy chromaticity coordinates.

NOTE 1 The values are a selection of the values defined in [2].

Table A.9 — Primaries

name	value	meaning
`kSRGB`	1	0.639998686, 0.330010138; 0.300003784, 0.600003357; 0.150002046, 0.059997204
`kCustom`	2	Custom red/green/blue primaries stored in `colour_encoding.red/green/blue`
`k2100`	9	0.708, 0.292; 0.170, 0.797; 0.131, 0.046 (As specified in Rec. ITU-R BT.2100-2)
`kP3`	11	0.680, 0.320; 0.265, 0.690; 0.150, 0.060 (As specified in SMPTE RP 431-2)

The values in the `meaning` column are interpreted as CIE xy chromaticity coordinates: red; green; blue, respectively.

NOTE 2 The values are a selection of the values defined in Rec. ITU-T H.273 | ISO/IEC 23091-2:2019 [3] The xy coordinates of `kSRGB` are quantized and match the values that would be stored in an ICC profile.

Table A.10 — TransferFunction

name	value	meaning
`k709`	1	As specified in Rec. ITU-R BT.709-6
`kUnknown`	2	None of the other table entries describe the transfer function
`kLinear`	8	The gamma exponent is 1
`kSRGB`	13	As specified in IEC 61966-2-1 sRGB

Table A.10 *(continued)*

name	value	meaning
kPQ	16	As specified in Rec. ITU-R BT.2100-2 (PQ)
kDCI	17	As specified in SMPTE ST 428-1
kHLG	18	As specified in Rec. ITU-R BT.2100-2 (HLG)

NOTE 3 The values are a selection of the values defined in Rec. ITU-T H.273 | ISO/IEC 23091-2:2019 [3].

Table A.11 — CustomTransferFunction bundle

condition	type	default	name
	Bool()	false	have_gamma
have_gamma	u(24)	10 000 000	gamma
!have_gamma	Enum(TransferFunction)	TransferFunction.kSRGB	transfer_function

have_gamma == false indicates the transfer function is defined by transfer_function. Otherwise, the opto-electrical transfer function is characterized by the exponent gamma / 10^7. This exponent is in (0, 1].

Table A.12 — RenderingIntent

name	value	meaning
kPerceptual	0	As specified in ISO 15076-1:2010 (vendor-specific)
kRelative	1	As specified in ISO 15076-1:2010 (media-relative)
kSaturation	2	As specified in ISO 15076-1:2010 (vendor-specific)
kAbsolute	3	As specified in ISO 15076-1:2010 (ICC-absolute)

NOTE 4 The values are defined by ISO 15076-1:2010.

Table A.13 — ColourEncoding bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	Bool()	false	want_icc
!all_default	Enum(ColourSpace)	kRGB	colour_space
use_desc && not_xyb	Enum(WhitePoint)	kD65	white_point
white_point == WP.kCustom	Customxy		white
use_desc && not_xyb && colour_space != kGrey	Enum(Primaries)	PR.kSRGB	primaries
primaries == PR.kCustom	Customxy		red
primaries == PR.kCustom	Customxy		green
primaries == PR.kCustom	Customxy		blue
use_desc	CustomTransferFunction		tf
use_desc	Enum(RenderingIntent)	kRelative	rendering_intent

In Table A.13, CS denotes ColourSpace, WP denotes WhitePoint, PR denotes Primaries, use_desc expands to !all_default && !want_icc, and not_xyb expands to colour_space != kXYB.

want_icc is true if and only if the codestream stores an ICC profile. If so, it is decoded as specified in Annex B, and describes the colour space. Also, colour_space is kGrey if and only if the ICC profile is greyscale. Otherwise, there is no ICC profile and the other fields describe the colour space.

The applicability of ColourEncoding is specified in ImageMetadata (A.6) and Annex B.

A.5 Extensions

This bundle, specified in Table A.14, is a field in bundles (ImageMetadata, FrameHeader, RestorationFilter) which can be extended (cf. Annex H).

Table A.14 — Extensions bundle

condition	type	default	name
	U64()	0	extensions
extensions != 0	U64()	0	extension_bits[NumExt]

An extension consists of zero or more bits whose interpretation is established by Annex H. Each extension is identified by an ext_i in the range [0, 63), assigned in increasing sequential order. extensions is a bit array where the i-th bit (least-significant == 0) indicates whether the extension with ext_i = i is present. NumExt denotes the number of extensions present, i.e. number of 1-bits in extensions. Extensions that are present are stored in ascending order of their ext_i. extension_bits[i] indicates the number of bits stored for the extension whose ext_i = i, starting after extension_bits has been read. The decoder reads all these bits for all extensions which are present.

A.6 ImageMetadata

Table A.15 specifies the BitDepth bundle.

Table A.15 — BitDepth bundle

condition	type	default	name
	Bool()	false	float_sample
!float_sample	U32(Val(8), Val(10), Val(12), BitsOffset(6,1))	8	bits_per_sample
float_sample	U32(Val(32), Val(16), Val(24), BitsOffset(6,1))	8	bits_per_sample
float_sample	u(4)		exp_bits_minus_one

float_sample is the data type of sample values (true for floating point, false for integers).

bits_per_sample is the number of bits per channel of the original image. The encoding of this value depends on floating_point_sample: for integers the value is in range [1, 31], for floating point restrictions depend on the value exp_bits_minus_one.

exp_bits_minus_one is one less than the amount of bits used for the exponent of floating point values and is only present if float_sample is true. Let exp_bits = exp_bits_minus_one + 1 be the amount of exponent bits and let mantissa_bits be bits_per_sample - exp_bits - 1. The samples of the original image are interpreted as floating point values with 1 sign bit, the indicated amount of exponent bits and mantissa bits and otherwise following the principles of the ISO/IEC 60559 standard with an exponent bias of $2^{\text{exp\_bits} - 1} - 1$. The value of exp_bits is in the range [2, 8], and the value of mantissa_bits is in the range [2, 23].

If a sample's exponent has the highest representable value, the decoded value of this sample is unspecified and platform-dependent.

NOTE 1 On some CPUs and floating-point formats, infinity/Not A Number are disallowed and the largest possible exponent is treated in the same way as the next smaller one.

The ImageMetadata bundle, defined in Table A.16, holds information that applies to all Frames (including the preview); decoders use it to interpret pixel values and restore the original image.

Table A.16 — ImageMetadata bundle

condition	type	default	name
	Bool()	true	`all_default`
`!all_default`	Bool()	false	`extra_fields`
`extra_fields`	u(3)	0	`orientation_minus_1`
`extra_fields`	Bool()	false	`have_intr_size`
`have_intr_size`	SizeHeader		`intrinsic_size`
`extra_fields`	Bool()	false	`have_preview`
`have_preview`	PreviewHeader		`preview`
`extra_fields`	Bool()	false	`have_animation`
`have_animation`	AnimationHeader		`animation`
`!all_default`	BitDepth		`bit_depth`
`!all_default`	Bool()	true	`modular_16bit_buffers`
`!all_default`	U32(Val(0), Val(1), BitsOffset(4, 2), BitsOffset(12, 1))	0	`num_extra_channels`
`!all_default`	ExtraChannelInfo		`extra_channel_info [num_extra_channels]`
`!all_default`	Bool()	true	`xyb_encoded`
`!all_default`	ColourEncoding		`colour_encoding`
`extra_fields`	ToneMapping		`tone_mapping`
`!all_default`	Extensions		`extensions`
	Bool()		`default_transform`
`default_transform && xyb_encoded`	OpsinInverseMatrix		`opsin_inverse_matrix`
`default_transform`	u(3)	0	`cw_mask`
`cw_mask & 1`	F16()	`d_up2`	`up2_weight[15]`
`cw_mask & 2`	F16()	`d_up4`	`up4_weight[55]`
`cw_mask & 4`	F16()	`d_up8`	`up8_weight[210]`

The `orientation` = 1 + `orientation_minus_1` indicates which orientation transform the decoder applies after decoding the image (Table A.17), including the preview frame if present.

Table A.17 — Orientation

orientation	side in first row	side in first column	transform to apply
1	top	left	none
2	top	right	flip horizontally
3	bottom	right	rotate 180°
4	bottom	left	flip vertically
5	left	top	transpose (rotate 90° clockwise then flip horizontally)
6	right	top	rotate 90° clockwise
7	right	bottom	flip horizontally then rotate 90° clockwise
8	left	bottom	rotate 90° counterclockwise

NOTE 2 `orientation` matches the values used by JEITA CP-3451C (Exif version 2.3).

If `have_intr_size`, then `intrinsic_size` denotes the recommended dimensions for displaying the image, i.e. applications are advised to resample the decoded image to the dimensions indicated in `intrinsic_size`.

`have_preview` indicates whether the codestream includes a `preview` and `preview_frame`.

have_animation indicates whether an AnimationHeader is included (animation) and whether FrameHeader (C.2) includes timing information.

modular_16bit_buffers indicates whether signed 16-bit integers have a large enough range to store and apply inverse transforms to all the decoded samples in modular image sub-bitstreams (see C.9). If false, 32-bit integers have to be used. In any case, signed 32-bit integers suffice to store decoded samples in modular image sub-bitstreams, as well as the results of inverse modular transforms. For some of the intermediate arithmetic (e.g. predictor computations), 64-bit arithmetic is needed.

num_extra_channels is the number of additional image channels. If it is nonzero, then extra_channel_info signals the semantics of each extra channel.

xyb_encoded is true if the stored image is in the XYB colour space. In L.2 it is specified how to convert XYB to RGB.

In the rest of this document, the three main colour components are referred to as X, Y, B, even in the case where the XYB colour space is not used for the stored image; in that case the actual components are either R, G, B or Cb, Y, Cr or C, M, Y.

colour_encoding indicates the colour image encoding of the original image. This may differ from the encoding of the stored image if xyb_encoded is true.

tone_mapping (Tables A.18) has information for mapping HDR images to lower dynamic range displays.

Table A.18 — ToneMapping bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	F16()	255	intensity_target
!all_default	F16()	0	min_nits
!all_default	Bool()	false	relative_to_max_display
!all_default	F16()	0	linear_below

intensity_target is > 0. This is an upper bound on the intensity level present in the image in nits, and represents the intensity corresponding to the value "1.0" (but the image need not contain a pixel this bright).

min_nits is > 0 and <= intensity_target. This is a lower bound on the intensity level present in the image in nits, but the image need not contain a pixel this dark.

linear_below represents a value below which tone mapping leaves the values unchanged. If relative_to_max_display is true, linear_below is a ratio in [0, 1] of the maximum display brightness in nits. Otherwise, it is an absolute brightness in nits >= 0.

For the upsampling filters, custom weights are signalled depending on cw_mask. The default weights are defined as follows: `d_up2` = { -0.01716200, -0.03452303, -0.04022174, -0.02921014, -0.00624645, 0.14111091, 0.28896755, 0.00278718, -0.01610267, 0.56661550, 0.03777607, -0.01986694, -0.03144731, -0.01185068, -0.00213539 }, `d_up4` = { -0.02419067, -0.03491987, -0.03693351, -0.03094285, -0.00529785, -0.01663432, -0.03556863, -0.03888905, -0.03516850, -0.00989469, 0.23651958, 0.33392945, -0.01073543, -0.01313181, -0.03556694, 0.13048175, 0.40103025, 0.03951150, -0.02077584, 0.46914198, -0.00209270, -0.01484589, -0.04064806, 0.18942530, 0.56279892, 0.06674400, -0.02335494, -0.03551682, -0.00754830, -0.02267919, -0.02363578, 0.00315804, -0.03399098, -0.01359519, -0.00091653, -0.00335467, -0.01163294, -0.01610294, -0.00974088, -0.00191622, -0.01095446, -0.03198464, -0.04455121, -0.02799790, -0.00645912, 0.06390599, 0.22963888, 0.00630981, -0.01897349, 0.67537268, 0.08483369, -0.02534994, -0.02205197, -0.01667999, -0.00384443 }, and `d_up8` = { -0.02928613, -0.03706353, -0.03783812, -0.03324558, -0.00447632, -0.02519406, -0.03752601, -0.03901508, -0.03663285, -0.00646649, -0.02066407, -0.03838633, -0.04002101, -0.03900035, -0.00901973, -0.01626393, -0.03954148, -0.04046620, -0.03979621, -0.01224485, 0.29895328, 0.35757708, -0.02447552, -0.01081748, -0.04314594, 0.23903219, 0.41119301, -0.00573046, -0.01450239, -0.04246845, 0.17567618, 0.45220643, 0.02287757, -0.01936783, -0.03583255, 0.11572472, 0.47416733, 0.06284440, -0.02685066, 0.42720050, -0.02248939, -0.01155273, -0.04562755, 0.28689496, 0.49093869, -0.00007891, -0.01545926, -0.04562659, 0.21238920, 0.53980934, 0.03369474, -0.02070211, -0.03866988, 0.14229550, 0.56593398, 0.08045181, -0.02888298, -0.03680918, -0.00542229, -0.02920477, -0.02788574, -0.02118180, -0.03942402, -0.00775547, -0.02433614, -0.03193943, -0.02030828, -0.04044014, -0.01074016, -0.01930822, -0.03620399, -0.01974125, -0.03919545, -0.01456093, -0.00045072, -0.00360110, -0.01020207, -0.01231907, -0.00638988, -0.00071592, -0.00279122, -0.00957115, -0.01288327, -0.00730937, -0.00107783, -0.00210156, -0.00890705, -0.01317668, -0.00813895, -0.00153491, -0.02128481, -0.04173044, -0.04831487, -0.03293190, -0.00525260, -0.01720322, -0.04052736, -0.05045706, -0.03607317, -0.00738030, -0.01341764, -0.03965629, -0.05151616, -0.03814886, -0.01005819, 0.18968273, 0.33063684, -0.01300105, -0.01372950, -0.04017465, 0.13727832, 0.36402234, 0.01027890, -0.01832107, -0.03365072, 0.08734506, 0.38194295, 0.04338228, -0.02525993, 0.56408126, 0.00458352, -0.01648227, -0.04887868, 0.24585519, 0.62026135, 0.04314807, -0.02213737, -0.04158014, 0.16637289, 0.65027023, 0.09621636, -0.03101388, -0.04082742, -0.00904519, -0.02790922, -0.02117818, 0.00798662, -0.03995711, -0.01243427, -0.02231705, -0.02946266, 0.00992055, -0.03600283, -0.01684920, -0.00111684, -0.00411204, -0.01297130, -0.01723725, -0.01022545, -0.00165306, -0.00313110, -0.01218016, -0.01763266, -0.01125620, -0.00231663, -0.01374149, -0.03797620, -0.05142937, -0.03117307, -0.00581914, -0.01064003, -0.03608089, -0.05272168, -0.03375670, -0.00795586, 0.09628104, 0.27129991, -0.00353779, -0.01734151, -0.03153981, 0.05686230, 0.28500998, 0.02230594, -0.02374955, 0.68214326, 0.05018048, -0.02320852, -0.04383616, 0.18459474, 0.71517975, 0.10805613, -0.03263677, -0.03637639, -0.01394373, -0.02511203, -0.01728636, 0.05407331, -0.02867568, -0.01893131, -0.00240854, -0.00446511, -0.01636187, -0.02377053, -0.01522848, -0.00333334, -0.00819975, -0.02964169, -0.04499287, -0.02745350, -0.00612408, 0.02727416, 0.19446600, 0.00159832, -0.02232473, 0.74982506, 0.11452620, -0.03348048, -0.01605681, -0.02070339, -0.00458223}.

A.7 AnimationHeader

AnimationHeader, defined in Table A.19, holds meta-information about an animation or image sequence that is present if `metadata.have_animation`.

Table A.19 — AnimationHeader bundle

condition	type	default	name
	U32(Val(100), Val(1000), BitsOffset(10, 1), BitsOffset(30, 1))	0	`tps_numerator`
	U32(Val(1), Val(1001), BitsOffset(8, 1), BitsOffset(10, 1))	0	`tps_denominator`
	U32(Val(0), Bits(3), Bits(16), Bits(32))	0	`num_loops`
	Bool()	false	`have_timecodes`

NOTE This document defines the interval between presenting the current and next frame in units of ticks. In the case where `metadata.have_animation` is false, the frames do not represent an animation, but layers that are overlaid to obtain a composite image.

`tps_numerator` / `tps_denominator` indicates the number of ticks per second.
`num_loops` is the number of times to repeat the animation (0 is interpreted as infinity).
`have_timecodes` indicates whether time codes are signalled in the frame headers.

A.8 OpsinInverseMatrix

OpsinInverseMatrix, defined in Table A.20, specifies parameters related to the inverse XYB colour transform.

Table A.20 — OpsinInverseMatrix bundle

condition	type	default	name
	Bool()	true	`all_default`
`!all_default`	F16()	11.031566901960783	`inv_mat00`
`!all_default`	F16()	-9.866943921568629	`inv_mat01`
`!all_default`	F16()	-0.16462299647058826	`inv_mat02`
`!all_default`	F16()	-3.254147380392157	`inv_mat10`
`!all_default`	F16()	4.418770392156863	`inv_mat11`
`!all_default`	F16()	-0.16462299647058826	`inv_mat12`
`!all_default`	F16()	-3.6588512862745097	`inv_mat20`
`!all_default`	F16()	2.7129230470588235	`inv_mat21`
`!all_default`	F16()	1.9459282392156863	`inv_mat22`
`!all_default`	F16()	-0.0037930732552754493	`opsin_bias0`
`!all_default`	F16()	-0.0037930732552754493	`opsin_bias1`
`!all_default`	F16()	-0.0037930732552754493	`opsin_bias2`
`!all_default`	F16()	1-0.05465007330715401	`quant_bias0`
`!all_default`	F16()	1-0.07005449891748593	`quant_bias1`
`!all_default`	F16()	1-0.049935103337343655	`quant_bias2`
`!all_default`	F16()	0.145	`quant_bias_numerator`

A.9 ExtraChannelInfo

Table A.21 specifies the meaning of the ExtraChannelType values.

Table A.21 — ExtraChannelType

name	value	meaning
`kAlpha`	0	Alpha transparency, where 0 means fully transparent
`kDepth`	1	Depth map
`kSpotColour`	2	Spot colour channel; `red`, `green`, `blue` indicate its colour and `solidity` in [0, 1] indicates the overall blending factor, with 0 corresponding to fully translucent (invisible) and 1 corresponding to fully opaque.
`kSelectionMask`	3	Selection mask, which indicates a (fuzzy) region of interest, for example for image manipulation purposes. Pixels with value zero do not belong to the selection, pixels with the maximum value do belong to the selection.
`kBlack`	4	The K channel of a CMYK image. If present, a CMYK ICC profile is also present, and the RGB samples are to be interpreted as CMY, where 0 denotes full ink.
`kCFA`	5	Channel used to represent Colour Filter Array data (Bayer mosaic)
`kThermal`	6	Infrared thermography image. Sample values are in units of Kelvin.
`kNonOptional`	15	The decoder indicates it cannot safely interpret the semantics of this extra channel.

Table A.21 *(continued)*

name	value	meaning
kOptional	16	Extra channel that can be safely ignored.

Table A.22 specifies the ExtraChannelInfo bundle.

Table A.22 — ExtraChannelInfo bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	ExtraChannelType	kAlpha	type
!all_default	BitDepth		bit_depth
!all_default	U32(Val(0), Val(3), Val(4), BitsOffset(3, 1))	0	dim_shift
!all_default	U32(Val(0), Bits(4), BitsOffset(5,16), BitsOffset(10,48))	0	name_len
	u(8)	0	name[name_len]
!all_default && type == kAlpha	Bool()	false	alpha_associated
type == kSpotColour	F16()	0	red
type == kSpotColour	F16()	0	green
type == kSpotColour	F16()	0	blue
type == kSpotColour	F16()	0	solidity
type == kCFA	U32(Val(1), Bits(2), BitsOffset(4, 3), BitsOffset(8, 19))	1	cfa_channel

Extra channels are interpreted according to their type and are rendered as specified in L.6.

dim_shift is the base-2 logarithm of the downsampling factor of the dimensions of the extra channel with respect to the main image dimensions defined in size. The dimensions of the extra channel are rounded up. The value 1 << dim_shift does not exceed the kGroupDim of any frame (C.2).

EXAMPLE dim_shift == 3 implies 8 × 8 downsampling, e.g. a 3 × 3 extra channel for a 20 × 20 main image.

If name_len > 0, then the extra channel has a name name, interpreted as a UTF-8 encoded string.

Annex B
(normative)

ICC profile

B.1 General

The interpretation of RGB pixel data is governed by a colour space described by a colour profile, for example sRGB is a colour space. Capture devices, display devices, and quantized (integer) pixel data can all have their own colour space. Converting quantized pixels in one colour space to quantized pixels in a different colour space is a lossy operation.

Pixel data can be encoded in two ways: either in the absolute XYB colour space, or as RGB triplets, YCbCr triplets or grayscale values which are to be interpreted according to the colour space described by a given colour profile.

NOTE 1 Both ways are useful in different scenarios:

— XYB is designed to match the human visual system and is excellent for lossy compression.

— When performing lossless compression, integer data has to be kept in its original colour space, since converting it and re-quantizing it in another colour space is lossy.

A colour space is always signalled in the metadata, and it has a different meaning in each scenario:

— In the XYB scenario, the signalled colour space does not give any information about how to interpret the encoded XYB pixels, since they are already in an absolute colour space. It is merely intended to indicate what colour space the original pixel data had, which can be useful when converting the decoded image to integer data. To display the image data on a display device with a display profile, this is irrelevant since it suffices to convert from XYB to the display colour space. The signalled colour space can be unused in that case. Only in case there are multiple frames that need to be blended by the decoder, the signalled colour space is relevant (since the blending may have to be done in that colour space and not in XYB, see C.2).

— In the other scenario, the signalled colour space has direct meaning: it is the colour space of the RGB, YCbCr or grayscale pixel data. To display the image on a display device, the pixel data is to be converted from the signalled colour space to the display colour space.

The `metadata.xyb_encoded` flag (A.6) indicates which of the two scenarios is followed. In the case where `metadata.xyb_encoded` is true, `metadata.colour_encoding` and the ICC profile described in this Annex (if present) are merely suggestions as to a colour encoding to use after the intermediate conversion to the linear sRGB colour space specified in L.2 (unless the frame is saved as a reference frame and `save_before_ct` is true, see C.2). If `xyb_encoded` is false, the decoded samples are to be interpreted in the following way:

— If `!metadata.colour_encoding.want_icc`, the decoded samples are indicated to be interpreted according to the `colour_space`, `white_point`, `primaries`, `transfer_function` and `rendering_intent` of `metadata.colour_encoding`, as specified in A.4.

— Otherwise, the decoded samples are indicated to be interpreted according to an ICC profile decoded as specified in the remainder of this Annex.

EXAMPLE An original image could be in the DCI-P3 colour space. For lossless compression, an encoder could signal the DCI-P3 colour space using `metadata.colour_encoding`, set `metadata.xyb_encoded` to false, and encode the pixel values as they are. For lossy compression, an encoder could also signal the DCI-P3 colour space using `metadata.colour_encoding`, set `metadata.xyb_encoded` to true, and convert the pixel values to the XYB colour space before encoding them. In this case, a decoder would decode the image in the XYB colour space, and apply L.2 to convert it to linear sRGB, which is an intermediate representation where some of the sample values could be negative or higher than the nominal maximum value since they are outside the sRGB gamut. To display the image on an sRGB display device, it would convert this intermediate representation to the sRGB colour space (i.e. apply clamping and adjust the transfer curve). To save the image in another (integer-based) image format, it could convert the intermediate representation to the suggested DCI-P3 colour space that was signalled, quantize the sample values to integers at the bit depth that was signalled, and save the resulting image.

The rest of this annex specifies how to decode an ICC profile.

NOTE 2 ICC profiles are standard display ICC profiles as defined by the International Colour Consortium. For example, an sRGB profile indicates the RGB values are to be interpreted as values in the sRGB colour space.

B.2 Data stream

The bitstream is byte aligned as specified in 6.3. The decoder reads `enc_size` as U64(). The decoder reads 41 clustered distributions as specified in D.3. The decoder then reads `enc_size` integers as specified in D.3.6 to obtain decompressed bytes, using `D[IccContext(index, prev_byte, prev_prev_byte)]` where `index` is the current byte index, `prev_byte` and `prev_prev_byte` are respectively the previous and second-previous bytes or 0 if they do not exist yet, and the `IccContext()` function is defined in the code below. The resulting decompressed byte-based data is the encoded ICC stream. The decoder reconstructs the ICC profile from this encoded ICC stream as described in the subsequent subclauses of this annex.

```
IccContext(i, b1, b2) {
  if (i <= 128) return 0;

  if (b1 >= 'a' && b1 <= 'z') p1 = 0;
  else if (b1 >= 'A' && b1 <= 'Z') p1 = 0;
  else if (b1 >= '0' && b1 <= '9') p1 = 1;
  else if (b1 == '.' || b1 == ',') p1 = 1;
  else if (b1 <= 1) p1 = 2 + b1;
  else if (b1 > 1 && b1 < 16) p1 = 4;
  else if (b1 > 240 && b1 < 255) p1 = 5;
  else if (b1 == 255) p1 = 6;
  else p1 = 7;

  if (b2 >= 'a' && b2 <= 'z') p2 = 0;
   else if (b2 >= 'A' && b2 <= 'Z') p2 = 0;
  else if (b2 >= '0' && b2 <= '9') p2 = 1;
  else if (b2 == '.' || b2 == ',') p2 = 1;
  else if (b2 < 16) p2 = 2;
  else if (b2 > 240) p2 = 3;
  else p2 = 4;

  return 1 + p1 + p2 * 8;
}
```

B.3 Encoded ICC stream

The encoded ICC stream is structured as shown in Table B.1. In this table, Varint() and u() are read from the encoded ICC stream, not from the JPEG XL codestream.

Table B.1 — ICC stream

type	name	comment
Varint()	`output_size`	the size of the decoded ICC profile, in bytes
Varint()	`commands_size`	the size of the command stream, in bytes.
u(8 × `commands_size`)	command stream	sub-stream of the encoded ICC stream which contains command bytes
u(8 × [[remaining bytes]])	data stream	sub-stream of the encoded ICC stream which contains data bytes, starting directly after the command stream and ending at the end of the encoded ICC stream

The command stream is structured as shown in Table B.2. In this table, u() is read from the command stream in the encoded ICC stream, not from the JPEG XL codestream.

Table B.2 — command stream

type	comment
u(8 × [[variable amount]])	commands for decoding tag list
u(8 × [[remaining bytes]])	commands for decoding main content

The data stream is structured as shown in Table B.3. In this table, u() is read from the encoded data stream in the encoded ICC stream, not from the JPEG XL codestream.

Table B.3 — data stream

type	comment
u(8 × [[variable amount]])	data for decoding ICC header
u(8 × [[variable amount]])	data for decoding ICC tag list
u(8 × [[remaining bytes]])	data for decoding main content

The resulting ICC profile, output by the decoder, is structured as shown in Table B.4. In this table, u() is read from the resulting ICC profile, not the JPEG XL codestream.

Table B.4 — ICC profile

type	name	comment
u(8 × [[variable amount]])	ICC header	up to 128 bytes long. Decoding procedure specified in B.4.
u(8 × [[variable amount]])	ICC tag list	Decoding procedure specified in B.5.
u(8 × [[remaining bytes]])	main content	Decoding procedure specified in B.6.

After decoding the `output_size` and `commands_size`, the encoded ICC stream is considered to be divided into two input streams: the command stream, which starts at the current position and runs for commands_size bytes, and the data stream, which starts directly after the command stream and contains all remaining bytes. The decoder maintains the current positions within the command stream and the data stream, initially at the beginning of each respective stream, and increment each when a byte from the corresponding stream is read as indicated in the next subclauses. A stream position does not go beyond the last byte of that stream, and the commands stream does not extend beyond the end of the encoded ICC stream.

As described in the next subclauses, the decoder decodes an ICC header (B.4), ICC tag list (B.5) and main content (B.6) from those two streams. The decoded ICC profile is the concatenation of these three parts in that order, which is indicated in the next subclauses as appending bytes to the result or output. The ICC tag list and main content can be empty if the decoder finishes earlier.

If at any time in subclauses B.4, B.5, or B.6 the ICC decoder is finished, then it returns the result as the ICC profile. The decoder reads all bytes from the data stream as well as all bytes from the command stream (that is, it reads exactly `commands_size` bytes from the command stream and reads exactly all remaining bytes from the data stream). The size of the resulting ICC profile is exactly `output_size` bytes.

B.4 ICC header

The header is the first of three concatenated parts that the decoder outputs.

The `header_size` in bytes is `min(128, output_size)`. The decoder reads `header_size` bytes from the data stream. For each byte `e`, it computes a prediction `p` as indicated below, then computes an output header byte as `(p + e) & 255` and appends it to the ICC result.

Each predicted value `p` is computed as specified by the following code, where `i` is the position in the header of the current byte, starting from 0. `header[j]` refers to the earlier output header byte corresponding to `i == j`. Single characters, and characters in strings, point to ASCII values.

```
if (i == 0 || i == 1 || i == 2 || i == 3)
  // 'output_size[i]' means byte i of output_size encoded as an
  // unsigned 32-bit integer in big endian order
  p = output_size[i];
else if (i == 8) p = 4;
else if (i >= 12 && i <= 23) {
  s = "mntrRGB XYZ ";  // one space after "RGB" and one after "XYZ"
  p = s[i - 12]; }
else if (i >= 36 && i <= 39) { s = "acsp"; p = s[i - 36]; }
else if ((i == 41 || i == 42) && header[40] == 'A') p = 'P';
else if (i == 43 && header[40] == 'A') p = 'L';
else if (i == 41 && header[40] == 'M') p = 'S';
else if (i == 42 && header[40] == 'M') p = 'F';
else if (i == 43 && header[40] == 'M') p = 'T';
else if (i == 42 && header[40] == 'S' && header[41] == 'G') p = 'I';
else if (i == 43 && header[40] == 'S' && header[41] == 'G') p = 32;
else if (i == 42 && header[40] == 'S' && header[41] == 'U') p = 'N';
else if (i == 43 && header[40] == 'S' && header[41] == 'U') p = 'W';
else if (i == 70) p = 246;
else if (i == 71) p = 214;
else if (i == 73) p = 1;
else if (i == 78) p = 211;
else if (i == 79) p = 45;
else if (i >= 80 && i < 84) p = header[4 + i - 80];
else p = 0;
```

If `output_size` is smaller than or equal to 128, then the above procedure has produced the full output, the ICC decoder is finished and the remaining subclauses are skipped.

B.5 ICC tag list

The ICC tag list is the second of three concatenated output parts that the decoder outputs to the resulting ICC profile. The decoder keeps reading from the same command stream and data stream as before, continuing at the positions reached at the end of the previous subclause.

To decode the tag list, the decoder reads the number of tags as specified by the following code. If the end of the command stream is reached, the decoder is finished, the full ICC profile is decoded, the decoder ends this procedure and skips the next subclause.

```
v = Varint() from command stream;
num_tags = v - 1;
if (num_tags == -1) {
  [[output nothing, stop reading the tag list, and proceed to B.6]]
}
[[Append num_tags to the output as a big endian unsigned 32-bit integer (4 output
bytes)]]
previous_tagstart = num_tags × 12 + 128;
previous_tagsize = 0;
```

Then, the decoder repeatedly reads a tag as specified by the following code until a tag with `tagcode` equal to 0 is read or until the end of the command stream is reached.

```
command = u(8) from command stream;
tagcode = command & 63;
if (tagcode == 0) {
  [[decoding the tag list is done, proceed to B.6]]
}
tag = "";  // 4-byte string with tag name
if (tagcode == 1) {
  // the tag string is set to 4 custom bytes read from the data
  tag = u(4 × 8) from data stream;
} else if (tagcode == 2) {
  tag = "rRTC";
} else if (tagcode == 3) {
  tag = "rXYZ";
} else if (tagcode >= 4 && tagcode < 21) {
  // the tag string is set to one of the predefined values
  strings = {
    "cprt", "wtpt", "bkpt", "rXYZ", "gXYZ", "bXYZ", "kXYZ", "rTRC", "gTRC",
    "bTRC", "kTRC", "chad", "desc", "chrm", "dmnd", "dmdd", "lumi"
  };
  tag = strings[tagcode - 4];
} else {
  [[ this branch is not reached ]]
}
tagstart = previous_tagstart + previous_tagsize;
if ((command & 64) != 0) tagstart = Varint() from command stream;
tagsize = previous_tagsize;
if (tag == "rXYZ" || tag == "gXYZ" || tag == "bXYZ" || tag == "kXYZ" ||
    tag == "wtpt" || tag == "bkpt" || tag == "lumi") {
  tagsize = 20;
}
```

```
if ((command & 128) != 0)  tagsize = Varint() from command stream;
previous_tagstart = tagstart;
previous_tagsize = tagsize;
```

The procedure appends the computed ICC tag(s) to the output as specified by the following code. Any integer values such as `tagsize`, `tagstart` and any operations on them are always appended as big endian 32-bit integers (4 bytes), when appended to the output.

```
[[append the following 3 groups of 4 bytes to the output:
  tag, tagstart, tagsize]]
if (tagcode == 2) {
  [[append the following additional 6 groups of 4 bytes to the output:
    "gTRC", tagstart, tagsize,
    "bTRC", tagstart, tagsize]]
} else if (tagcode == 3) {
  [[append the following additional 6 groups of 4 bytes to the output:
    "gXYZ", (tagstart + tagsize), tagsize,
    "bXYZ", (tagstart + 2 × tagsize), tagsize]]
}
```

B.6 Main content

The main content is the third of three concatenated output parts that the decoder outputs to the resulting ICC profile. The decoder keeps reading from the same command stream and data stream as before, continuing at the positions reached at the end of the previous subclause.

In this subclause, `Shuffle(bytes, width)` denotes the following operation: Bytes are inserted in raster order (row by row from left to right, starting with the top row) into a matrix with `width` rows, and as many columns as needed. The last column can have missing elements at the bottom if `len(bytes)` is not a multiple of `width`. Those elements are skipped and no byte is taken from the input for these missing elements: the input is the concatenation of all rows excluding the possibly missing last element of each row. Then the matrix is transposed, and `bytes` are overwritten with elements of the transposed matrix read in raster order, not including the missing elements which are now at the end of the last row.

EXAMPLE shuffling the list (1, 2, 3, 4, 5, 6, 7) with width 2 results in the list (1, 5, 2, 6, 3, 7, 4).

To decode the main content, the decoder reads from the command stream as specified by the following code, until the end of the command stream is reached (which can be immediately).

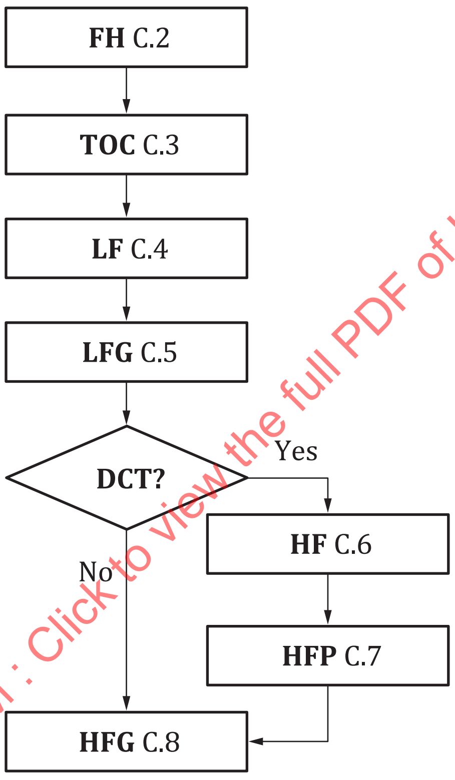
```
command = u(8) from command stream;
if (command == 1) {
  num = Varint() from command stream;
  bytes = u(num × 8) from data stream;
  [[append bytes to output_stream]]
} else if (command == 2 || command == 3) {
  num = Varint() from command stream;
  bytes = u(num × 8) from data stream;
  width = (command == 2) ? 2 : 4;
  Shuffle(bytes, width);
  [[append bytes to output_stream]]
} else if (command == 4) {
  flags = u(8) from command stream;
  width = (flags & 3) + 1;
  [[ width != 3)]];
  order = (flags & 12) >> 2;
  [[ order != 3 ]];
  stride = width;
  if ((flags & 16) != 0) {
    stride = Varint() from command stream;
  }
  [[ stride × 4 < number of bytes already output to ICC profile ]]
  [[ stride >= width ]];
  num = Varint() from command stream;
  bytes = u(num × 8) from data stream;
  if (width == 2 || width == 4) {
    Shuffle(bytes, width);
  }
  // Run an Nth-order predictor on num bytes as follows, with the bytes
  // representing unsigned integers of the given width (but num is not
  // required to be a multiple of width).
  for (i = 0; i < num; i += width) {
    N = order + 1;
    prev = [[N-element array of unsigned integers of width bytes each]]
    for (j = 0; j < N; ++j) {
      prev[j] = [[read u(width × 8) from the output ICC profile starting
      from (stride × (j + 1)) bytes before the current output size,
      interpreted as a big-endian unsigned integer of width bytes]];
    }
    // Compute predicted number p, where p and elements of prev are
    // unsigned integers of width bytes each.
    if (order == 0) p = prev[0];
    else if (order == 1) p = 2 × prev[0] - prev[1];
    else if (order == 2) p = 3 × prev[0] - 3 × prev[1] + prev[2];
    for (j = 0; j < width && i + j < num; ++j) {
      val = (bytes[i + j] + (p >> (width - 1 - j))) & 255;
      [[append val as 1 byte to the ICC profile output]];
    }
  }
} else if (command == 10) {
  [[append "XYZ " (4 ASCII characters) to the output]];
  [[append 4 bytes with value 0 to the output]];
  bytes = u(12 × 8) from data stream;
  [[append bytes to the output]];
} else if (command >= 16 && command < 24) {
  strings = {"XYZ ", "desc", "text", "mluc", "para", "curv", "sf32",
   "gbd "};
  [[append 4 bytes from strings[command - 16] to the output]];
  [[append 4 bytes with value 0 to the output]];
} else {
  [[this branch is not reached]];
}
```

Annex C
(normative)

Frames

C.1 General

Subsequent to headers (Annex A) and ICC profile, if present (Annex B), the codestream consists of one or more frames. Frames have the structure shown in Figure C.1. Each Frame is byte-aligned as specified in 6.3 and is read according to Table C.1.



Key

FH	frame header
TOC	table of contents
LF	LF global
LFG	LF groups
DCT	kVarDCT encoding?
HF	HF global
HFP	HF pass data
HFG	HF groups

Figure C.1 — Per-frame data

Let `num_groups` denote ceil(`width / kGroupDim`) × ceil(`height / kGroupDim`), where `kGroupDim` is the width and height of a group (6.2). `num_lf_groups` denotes ceil(`width` / (`kGroupDim` × 8)) × ceil(`height` / (`kGroupDim` × 8)). Note that `encoding` and `num_passes` are obtained from FrameHeader (C.2).

`width` and `height` are interpreted as the dimensions of the sample grid, not its rotated/mirrored interpretation (which is indicated by `metadata.orientation`).

NOTE The LF coefficients (C.5.3) always correspond to a 1:8 downsampled image, regardless of the sizes of the var-DCT blocks that were used. In the case of DCT8x8, these are the DC coefficients. In the case of bigger block sizes, these also include information derived from the low-frequency AC coefficients (which are recovered using the procedures described in I.2.5), and these low-frequency AC coefficients are skipped in the HF encoding (C.8.3). In the case of smaller block sizes like DCT4x4, the 'LF coefficient' is effectively the average of the smaller-block DC coefficients, and the HF encoding effectively contains the remaining information to restore the smaller-block DC.

Table C.1 — Frame bundle

condition	type	name
	FrameHeader	`frame_header`
	TOC	`toc`
	LfGlobal	`lf_global`
	LfGroup	`lf_group[num_lf_groups]`
`encoding == kVarDCT`	HfGlobal	`hf_global`
`encoding == kVarDCT`	HfPass	`hf_pass[num_passes]`
	PassGroup	`group_pass[num_groups × num_passes]`

The dimensions of a frame in pixels (`width` and `height`) are `size.width` and `size.height` if `!frame_header.have_crop`, otherwise they are given by `frame_header.width` and `frame_header.height`.

C.2 FrameHeader

The decoder reads FrameHeader as specified in Table C.2.

Table C.2 — FrameHeader bundle

condition	type	default	name
	Bool()	true	`all_default`
`!all_default`	FrameType	kRegularFrame	`frame_type`
`!all_default`	u(1)	0	`encoding`
`!all_default`	U64()	0	`flags`
`!all_default && !metadata.xyb_encoded`	Bool()	false	`do_YCbCr`
`do_YCbCr && !flags.kUseLfFrame`	u(2)	0	`jpeg_upsampling[3]`
`!all_default && !flags.kUseLfFrame`	U32(Val(1), Val(2), Val(4), Val(8))	1	`upsampling`
`!all_default && !flags.kUseLfFrame`	U32(Val(1), Val(2), Val(4), Val(8))	1	`ec_upsampling [num_extra_channels]`
`encoding == kModular`	u(2)	1	`group_size_shift`
`!all_default && encoding == kVarDCT && metadata.xyb_encoded`	u(3)	3	`x_qm_scale`
`!all_default && encoding == kVarDCT && metadata.xyb_encoded`	u(3)	2	`b_qm_scale`
`!all_default && frame_type != kReferenceOnly`	Passes		`passes`

Table C.2 *(continued)*

condition	type	default	name
`frame_type == kLFFrame`	U32(Val(1), Val(2), Val(3), Val(4))	0	`lf_level`
`!all_default && frame_type != kLFFrame`	Bool()	false	`have_crop`
`have_crop && frame_type != kReferenceOnly`	U32(Bits(8), BitsOffset(11, 256), BitsOffset(14, 2304), BitsOffset(30, 18688))	0	`x0`
`have_crop && frame_type != kReferenceOnly`	U32(Bits(8), BitsOffset(11, 256), BitsOffset(14, 2304), BitsOffset(30, 18688))	0	`y0`
`have_crop`	U32(Bits(8), BitsOffset(11, 256), BitsOffset(14, 2304), BitsOffset(30, 18688))	0	`width`
`have_crop`	U32(Bits(8), BitsOffset(11, 256), BitsOffset(14, 2304), BitsOffset(30, 18688))	0	`height`
normal_frame	BlendingInfo		`blending_info`
normal_frame	BlendingInfo		`ec_blending_info [num_extra_channels]`
normal_frame && `metadata.have_animation`	U32(Val(0), Val(1), Bits(8), Bits(32))	0	`duration`
normal_frame && `metadata.animation.have_timecode`	u(32)	0	`timecode`
normal_frame	Bool()	`!frame_type`	`is_last`
`!all_default && frame_type != kLFFrame && !is_last`	u(2)	0	`save_as_reference`
`!all_default && frame_type != kLFFrame`	Bool()	`d_sbct`	`save_before_ct`
`!all_default`	U32(Val(0), Bits(4), BitsOffset(5,16), BitsOffset(10,48))	0	`name_len`
	u(8)	0	`name[name_len]`
`!all_default`	RestorationFilter		`restoration_filter`
`!all_default`	Extensions		`extensions`

In the condition column, the abbreviation "normal_frame" denotes the condition `!all_default && (frame_type == kRegularFrame || frame_type == kSkipProgressive)`.

FrameType is defined in Table C.3.

Table C.3 — FrameType

name	value	meaning
`kRegularFrame`	0	A 'regular' frame, which is part of the decoded sequence of frames.
`kLFFrame`	1	Represents the LF of a future frame. Is not itself part of the decoded sequence of frames.
`kReferenceOnly`	2	Frame will only be used as a source for Patches. Is not itself part of the decoded sequence of frames.
`kSkipProgressive`	3	Same as `kRegularFrame` in terms of the (final) decoded sequence of frames, but decoders do not progressively render previews of frames of this type.

`encoding` is defined in Table C.4.

Table C.4 — FrameHeader.encoding

name	value	meaning
kVarDCT	0	Var-DCT mode: the decoder will perform IDCT on varblocks to obtain the decoded image. NOTE This mode is suitable for lossy encodings and for lossless JPEG recompression.
kModular	1	Modular mode: the decoder will perform a signalled chain of zero or more inverse (typically reversible) transforms. NOTE This mode is suitable for lossless encodings.

Both encodings split each frame into groups. This document specifies the meaning of the encodings (e.g. `kVarDCT`) where they are referenced.

`flags` is defined in Table C.5.

Table C.5 — FrameHeader.flags

name	value	meaning
kNoise	xxxx xxx1	Enable noise feature stage
kNoise	xxxx xxx0	Disable noise feature stage
kPatches	xxxx xx1x	Enable patch feature stage
kPatches	xxxx xx0x	Disable patch feature stage
kSplines	xxx1 xxxx	Enable spline feature stage
kSplines	xxx0 xxxx	Disable spline feature stage
kUseLfFrame	xx1x xxxx	Enable use of special LF frames
kUseLfFrame	xx0x xxxx	Disable use of special LF frames
kSkipAdaptiveLFSmoothing	0xxx xxxx	Enable adaptive LF smoothing
kSkipAdaptiveLFSmoothing	1xxx xxxx	Disable adaptive LF smoothing

The bits of `frame_header.flags` encode whether certain features are enabled or disabled in the current frame, as specified in references to this subclause.

EXAMPLE if `flags` == 18, the decoder enables spline rendering and patch rendering.

When a flag name is used, its value is understood as the value of the corresponding bit (0 or 1).

EXAMPLE if `frame_header.flags` == 3, `kPatches` is considered to be 1.

If `do_YCbCr`, then the pixels are stored in YCbCr, and are converted to RGB as specified in L.3.

NOTE In addition, one or more reversible colour transforms can be applied in Modular mode (see C.9 and L.4, L.5). These transforms are not signalled in the FrameHeader, but in the ModularHeader (C.9.2), and can be applied either to the whole frame or on a per-group basis. These are always undone, regardless of the value of `save_before_ct`.

`jpeg_upsampling` indicates the degree of subsampling for each YCbCr channel. 0 denotes {horizontal, vertical} subsampling factors of {1, 1} for the given channel; 1 denotes {2, 2}, 2 denotes {2, 1} and 3 denotes {1, 2}. The horizontal and vertical size, in 8x8 blocks, of each channel is divided (rounding up) by the maximum subsampling factor across all channels, and then multiplied by the subsampling factor for the current channel.

`upsampling` and `ec_upsampling` also indicate subsampling, by a factor of 1, 2, 4 or 8 (both horizontally and vertically), for respectively the colour image and the extra channels. In the case of extra channels this subsampling is cumulative with the subsampling implied by `dim_shift`.

If `upsampling` > 1, then for all extra channels, `extra_channel_info[i].dim_shift` × `ec_upsampling[i]` >= `upsampling`.

The division in groups is not affected by jpeg_upsampling and ec_upsampling: even if a channel is subsampled, group splitting proceeds according to the location of samples and blocks in the image, after subsampling by a factor of upsampling. For example, if jpeg_upsampling is equal to {0, 1, 1}, i.e. 4:2:0 chroma subsampling, and there is an alpha channel for which ec_upsampling is 4, then the groups consist of 256×256 Y samples, 128×128 Cb and Cr samples, and 64×64 alpha samples, which all correspond to the same image region.

After decoding subsampled channels, they are upsampled as specified in L.7 (in the case of jpeg_upsampling) and L.8 (in the case of upsampling and ec_upsampling).

The value of kGroupDim is set to 128 << group_size_shift.

Passes are defined in Table C.6.

Table C.6 — Passes bundle

condition	type	default	name
	U32(Val(1), Val(2), Val(3), BitsOffset(3, 4))	1	num_passes
num_passes != 1	U32(Val(0), Val(1), Val(2), BitsOffset(1, 3))	0	num_ds
num_passes != 1	u(2)	0	shift[num_passes-1]
num_passes != 1	U32(Val(1), Val(2), Val(4), Val(8))	1	downsample[num_ds]
num_passes != 1	U32(Val(0), Val(1), Val(2), Bits(3))	0	last_pass[num_ds]

num_passes indicates the number of passes into which the frame is partitioned.

num_ds indicates the number of (downsample, last_pass) pairs between 0 and 4, inclusive. It is strictly smaller than num_passes.

shift[i] indicates the amount by which the HF coefficients of the pass with index i in the range [0, num_passes) are left-shifted immediately after entropy decoding. The last pass behaves as if it had a shift value of 0.

downsample[i] (i in the range [0, num_ds)) indicates a downsampling factor in x and y direction. last_pass[i] (i in the range [0, num_ds)) indicates the zero-based index of the final pass that is decoded in order to achieve a downsampling factor of least downsample[i]. This index is at most num_passes - 1.

In addition to the (downsample, last_pass) pairs that are explicitly encoded in the codestream, the decoder behaves as if a final pair equal to (1, num_passes - 1) were present.

If lf_level != 0, the samples of the frame (before any colour transform is applied) are recorded as LFFrame[lf_level-1] and may be referenced by subsequent frames. Moreover, in this case, the decoder considers the current frame to have dimensions ceil(width / (1 << (3×lf_level))) × ceil(height / (1 << (3×lf_level))).

If either save_as_reference != 0 or duration == 0, and !is_last && frame_type != kLFFrame, then the samples of the decoded frame are recorded as Reference[save_as_reference] and may be referenced by subsequent frames. The decoded samples are recorded before any colour transform (XYB or YCbCr) if save_before_ct is true, and after colour transform otherwise (in which case they are converted to the RGB color space signalled in the image header). Blending is performed before recording the reference frame.

If have_crop, the decoder considers the current frame to have dimensions width × height, and updates the rectangle of the previous frame with top-left corner x0, y0 with the current frame using the given blend_mode. The dimension values respect the equations x0 + width <= size.width and y0 + height <= size.height. If !have_crop, the frame has the same dimensions as the image. Let full_frame be true if and only if have_crop is false or if the frame area given by width and height and offsets x0 and y0 completely covers the image area.

Let multi_extra be true if and only if and the number of extra channels is at least two. blendinginfo and ec_blendinginfo are defined in Table C.7.

Table C.7 — BlendingInfo bundle

condition	type	default	name
	U32(Val(0), Val(1), Val(2), BitsOffset(2, 3))	0	`mode`
`multi_extra && (mode == kBlend || mode == kAlphaWeightedAdd)`	U32(Val(0), Val(1), Val(2), BitsOffset(3, 3))	0	`alpha_channel`
`multi_extra && (mode == kBlend || mode == kAlphaWeightedAdd || mode == kMul)`	Bool()	false	`clamp`
`mode != kReplace || !full_frame`	U32(Val(0), Val(1), Val(2), Val(3))	0	`source`

The `mode` values are defined in Table C.8. All blending operations consider as "previous sample" the sample at the corresponding coordinates in the source frame, which is the frame that was previously stored in Reference[`source`] – if no frame was previously stored, the source frame is assumed to have all sample values set to zeroes. The `blend_info` affects the three colour channels. The extra channels are blended according to `ec_blend_info` instead. The blending is done in the colour space *after* inverse colour transforms from Annex L have been applied (except for L.6). For blend modes `kBlend` and `kAlphaWeightedAdd`, "the alpha channel" refers to the extra channel with index `alpha_channel`, "sample" is the rgb value or extra channel value after blending, "new_sample" and "old_sample" are the corresponding values from the current and source frame respectively, and "alpha", "new_alpha" and "old_alpha" are the corresponding values for the alpha channel. If `clamp` is true, alpha values are clamped to the interval [0, 1] before blending.

Table C.8 — BlendMode (BlendingInfo.mode)

name	value	description
`kReplace`	0	Each sample is overwritten with the corresponding new sample. sample = new_sample
`kAdd`	1	Each new sample is added to the corresponding previous sample. sample = old_sample + new_sample
`kBlend`	2	Each new sample is alpha-blended over the corresponding previous sample. If the alpha channel has premultiplied semantics (`alpha_associated` == true), then sample = new_sample + old_sample × (1 - new_alpha). Otherwise sample = (new_alpha × new_sample + old_alpha × old_sample × (1 - new_alpha)) / alpha The blending on the alpha channel itself always uses the following formula instead: alpha = old_alpha + new_alpha × (1 - old_alpha)
`kAlphaWeightedAdd`	3	Each new sample is multiplied with alpha and added to the corresponding previous sample. sample = old_sample + alpha × new_sample The blending on the alpha channel itself uses the following formula instead: alpha = old_alpha + new_alpha × (1 - old_alpha)
`kMul`	4	Each new sample is multiplied with the previous sample. sample = old_sample × new_sample

`duration` (in units of ticks, see AnimationHeader) is the intended period of time between presenting the current frame and the next one. If `duration` is zero and `!is_last`, the decoder does not present the current frame, but the frame may be composed together with the next frames, for example through blending. In particular, in the case that `metadata.have_animation` is false, the decoder returns a single image consisting of the composition of all the zero-duration frames.

`timecode` indicates the SMPTE timecode of the current frame, or 0. The decoder interprets groups of 8 bits from most-significant to least-significant as hour, minute, second, and frame. If `timecode` is nonzero, it is strictly larger than that of a previous frame with nonzero `duration`.

The decoder ceases decoding after the current frame if `is_last`.

The default value for `save_before_ct` is `d_sbct = !(full_frame && (frame_type == kRegularFrame || frame_type == kSkipProgressive) && blending_info.mode == kReplace && (duration == 0 || save_as_reference != 0) && !is_last)`.

If `name_len` > 0, then the frame has a name `name`, interpreted as a UTF-8 encoded string.

Finally, `restoration_filter` is defined in Table C.9.

Table C.9 — RestorationFilter bundle

condition	**type**	**default**	**name**
	Bool()	true	`gab`
`gab`	Bool()	false	`gab_custom`
`gab_custom`	F16()	0.115169525	`gab_x_weight1`
`gab_custom`	F16()	0.061248592	`gab_x_weight2`
`gab_custom`	F16()	0.115169525	`gab_y_weight1`
`gab_custom`	F16()	0.061248592	`gab_y_weight2`
`gab_custom`	F16()	0.115169525	`gab_b_weight1`
`gab_custom`	F16()	0.061248592	`gab_b_weight2`
	u(2)	2	`epf_iters`
`epf_iters && encoding = kVarDCT`	Bool()	false	`epf_sharp_custom`
`epf_sharp_custom`	F16()	{0, 1/7, 2/7, 3/7, 4/7, 5/7, 6/7, 1}	`epf_sharp_lut[8]`
`epf_iters`	Bool()	false	`epf_weight_custom`
`epf_weight_custom`	F16()	{40.0, 5.0, 3.5}	`epf_channel_scale[3]`
`epf_weight_custom`	F16()	0.45	`epf_pass1_zeroflush`
`epf_weight_custom`	F16()	0.6	`epf_pass2_zeroflush`
`epf_iters`	Bool()	false	`epf_sigma_custom`
`epf_sigma_custom && encoding == kVarDCT`	F16()	0.46	`epf_quant_mul`
`epf_sigma_custom`	F16()	0.9	`epf_pass0_sigma_scale`
`epf_sigma_custom`	F16()	6.5	`epf_pass2_sigma_scale`
`epf_sigma_custom`	F16()	2 / 3	`epf_border_sad_mul`
`epf_iters && encoding == kModular`	F16()	1.0	`epf_sigma_for_modular`
	Extensions		`extensions`

`gab` and `epf_iters` determine whether the Gabor-like transform (J.2) and the edge-preserving filter (J.3) are applied. The other fields parameterize any enabled filter(s).

C.3 TOC

C.3.1 General

The codestream consists of parts called sections. The TOC (Table of Contents) is an array of numbers. Each TOC number indicates the size in bytes of a section. Each section is aligned to a byte boundary, as defined in 6.3. If `num_groups == 1` and `num_passes == 1`, there is a single TOC entry and section containing all frame data structures. Otherwise, there is one entry for each of the following sections, in the order they are listed: LfGlobal, one per LfGroup in raster order, one for HfGlobal followed by HfPass data for all the passes, and num_groups × frame_header.passes.num_passes PassGroup. The first PassGroup are the groups (in raster order) of the first pass, followed by all groups (in raster order) of the second pass (if present), and so on.

The decoder first reads permuted_toc = u(1). If and only if its value is 1, the decoder reads permutation from a single entropy coded stream with 8 clustered distributions, as specified in D.3, with size equal to the number of TOC entries and skip = 0.

C.3.2 Decoding permutations

The decoder computes a sequence lehmer containing size elements as follows. Let GetContext(x) denote min(8, ceil(log2(x + 1))).

The decoder first decodes an integer end, as specified in D.3.6, using distribution D[GetContext(size)]. Then, (end - skip) elements of the lehmer sequence are produced as follows. For each element, an integer lehmer[skip + i] is read as specified in D.3.6 using the distribution D[prev_elem]. prev_elem is lehmer[skip + i - 1] if i > 0, or 0 otherwise. All other elements of the sequence lehmer are 0. The decoder maintains a sequence of elements temp, initially containing the numbers [0, size) in increasing order, and a sequence of elements permutation, initially empty. Then, for each integer i in the range [0, size), the decoder appends to permutation element temp[lehmer[i]], then removes it from temp, leaving the relative order of other elements unchanged. Finally, permutation is the decoded permutation.

C.3.3 Decoding TOC

The sequence of TOC entries are byte-aligned (6.3). The decoder reads each TOC entry in order of increasing index via U32(Bits(10), BitsOffset(14, 1024), BitsOffset(22, 17408), BitsOffset(30, 4211712)). None of the TOC entries are zero.

The decoder then computes an array group_offsets, which has 0 as its first element and subsequent group_offsets[i] are the sum of all TOC entries [0, i).

If permuted_toc, the decoder permutes group_offsets according to permutation, such that group_offsets[i] is what was previously group_offsets[permutation[i]].

Let P be the index of the byte containing the next unread bit after decoding the TOC. When decoding a group with index i, the decoder reads from the codestream starting at the byte with index P + group_offsets[i].

C.4 LfGlobal

C.4.1 General

The decoder reads the bundles listed in Table C.10.

Table C.10 — LfGlobal bundle

condition	name	subclause
`(frame_header.flags & kPatches) != 0`	Patches	C.4.5
`(frame_header.flags & kSplines) != 0`	Splines	C.4.6
`(frame_header.flags & kNoise) != 0`	NoiseParameters	C.4.7
	LfChannelDequantization	C.4.2
`frame_header.encoding == kVarDCT`	Quantizer	C.4.3
`frame_header.encoding == kVarDCT`	HF Block Context	C.8.4
`frame_header.encoding == kVarDCT`	LfChannelCorrelation	C.4.4
	GlobalModular	C.4.8

C.4.2 LF channel dequantization weights

The decoder reads the bundles listed in Table C.11.

Table C.11 — LfChannelDequantization bundle

condition	type	default	name
	u(1)	true	`all_default`
`!all_default`	F16()	4096	`m_x_lf_unscaled`
`!all_default`	F16()	512	`m_y_lf_unscaled`
`!all_default`	F16()	256	`m_b_lf_unscaled`

C.4.3 Quantizer

Table C.12 specifies the Quantizer bundle.

Table C.12 — Quantizer bundle

condition	type	default	name
	U32(BitsOffset(11,1), BitsOffset(11, 2049), BitsOffset(12, 4097), BitsOffset(16, 8193))	1	`global_scale`
	U32(Val(16), BitsOffset(5,1), BitsOffset(8,1), BitsOffset(16,1))	1	`quant_lf`

The LF dequantization factors `mXDC`, `mYDC` and `mBDC` are computed as specified by the following code:

```
mXDC = m_x_lf_unscaled / (global_scale × quant_lf);
mYDC = m_y_lf_unscaled / (global_scale × quant_lf);
mBDC = m_b_lf_unscaled / (global_scale × quant_lf);
```

C.4.4 Default and LF channel correlation factors

Table C.13 specifies the LfChannelCorrelation bundle.

Table C.13 — LfChannelCorrelation bundle

condition	type	default	name
	u(1)	true	`all_default`
`!all_default`	U32(Val(84),Val(256), BitsOffset(8, 2), BitsOffset(16, 258))	`84`	`colour_factor`
`!all_default`	F16()	`0.0`	`base_correlation_x`
`!all_default`	F16()	`1.0`	`base_correlation_b`

Table C.13 *(continued)*

condition	type	default	name
!all_default	u(8)	127	x_factor_lf
!all_default	u(8)	127	b_factor_lf

C.4.5 Patches

If the kPatches flag in frame_header is not set, this subclause is skipped for that frame.

Otherwise, the patch dictionary contains a set of small image patches, to be added to the decoded image in specified locations as specified in K.2.

The decoder reads a set of 10 clustered distributions D, according to D.3, and then reads the following values from a single stream as specified in D.3.6.

In this subclause, ReadHybridVarLenUint(x) denotes reading an integer from the stream using the distribution D[x] as defined in D.3.6.

Each patch type has the following attributes:

— width, height: dimensions of the image patch.

— ref: index of the reference frame the patch is taken from

— x0, y0: coordinates of the top-left corner of the patch in the reference image

— sample(x, y, c): the sample from channel c at the position (x, y) within the patch, corresponding to the sample from channel c at the position (x0 + x, y0 + y) in Reference[ref]. All referenced samples are within the bounds of the reference frame.

— count: the number of distinct positions

— blending: arrays of count blend mode information structures, which consists of arrays of mode, alpha_channel and clamp

— x, y: arrays of count positions where this patch is added to the image

The decoder first sets num_patches = ReadHybridVarLenUint(0). For i in the range [0, num_patches) in ascending order, the decoder decodes patch[i] as specified by the following code:

```
patch[i].ref = ReadHybridVarLenUint(1);
patch[i].x0 = ReadHybridVarLenUint(3);
patch[i].y0 = ReadHybridVarLenUint(3);
patch[i].width = ReadHybridVarLenUint(2) + 1;
patch[i].height = ReadHybridVarLenUint(2) + 1;
patch[i].count = ReadHybridVarLenUint(7) + 1;
for (j = 0; j < patch[i].count; j++) {
 if (j == 0) {
   patch[i].x[j] = ReadHybridVarLenUint(4);
   patch[i].y[j] = ReadHybridVarLenUint(4);
 } else {
   patch[i].x[j] = UnpackSigned(ReadHybridVarLenUint(5)) +
     patch[i].x[j - 1];
   patch[i].y[j] = UnpackSigned(ReadHybridVarLenUint(5)) +
     patch[i].y[j - 1];
 }
 [[the width × height rectangle with top-left coordinates (x, y) is
 fully contained within the frame]];
 for (k = 0; k < num_extra_channels+1; k++) {
   mode = ReadHybridVarLenUint(5);
   [[ mode < 8 ]]
   patch[i].blending[j].mode[k] = mode;
   if ((mode == kBlendAbove || mode == kBlendBelow ||
        mode == kAlphaWeightedAddAbove || mode == kAlphaWeightedAddBelow)
       && [[ there is more than 1 alpha channel ]]) {
     patch[i].blending[j].alpha_channel[k] = ReadHybridVarLenUint(8);
     [[ this is a valid index of an extra channel ]]
   }
   if (mode == kBlendAbove || mode == kBlendBelow ||
       mode == kAlphaWeightedAddAbove || mode == kAlphaWeightedAddBelow
       || mode == kMul) {
     patch[i].blending[j].clamp[k] = ReadHybridVarLenUint(9);
   }
 }
}
```

C.4.6 Splines

Spline data is present in the codestream if and only if the `kSplines` flag in `frame_header` (C.2) is set. If so, it consists of an ANS stream (D.3). The decoder first reads six clustered distributions as described in D.3, and then reads coordinates as specified by the following code:

```
num_splines = ReadHybridVarLenUint(2) + 1;
quant_adjust = UnpackSigned(ReadHybridVarLenUint(0));
last_x = 0;
last_y = 0;
for (i = 0; i < num_splines; i++) {
  x = ReadHybridVarLenUint(1);
  y = ReadHybridVarLenUint(1);
  if (i != 0) {
    x = UnpackSigned(x) + last_x;
    y = UnpackSigned(y) + last_y;
  }
  (sp_x[i], sp_y[i]) = (x, y);
  last_x = x;
  last_y = y;
}
```

For each of `num_splines` splines, the following steps are applied.The number of control points (including the starting point) is read as `num_control_points = 1 + ReadHybridVarLenUint(3)`. The result of applying double delta encoding to the control point coordinates (separately for `x` and `y`) and then dropping the very first control point is read as interleaved values (`x1`, `y1`, `x2`, `y2`…) via `UnpackSigned(ReadHybridVarLenUint(4))`. That is, for a starting point (`sp_x`, `sp_y`), `DecodeDoubleDelta(sp_x, {x1, x2…})` and `DecodeDoubleDelta(sp_y, {y1, y2})` give the `x` and `y` coordinates of the spline's control points, as specified in the following code:

```
DecodeDoubleDelta(starting_value, delta[n]) {
  [[Append starting_value to the list of decoded values]];
  current_value = starting_value;
  current_delta = 0;
  for (i = 0; i < n; ++i) {
    current_delta += delta[i];
    current_value += current_delta;
    [[Append current_value to the list of decoded values]];
  }
}
```

Then, `UnpackSigned(ReadHybridVarLenUint(5))` is used four times thirty-two times to obtain, in this order, sequences of thirty-two integers representing the following coefficients of the spline along its arc length:

— the quantized DCT32 coefficients of the X channel of the spline

— the quantized DCT32 coefficients of the Y channel of the spline

— the quantized and decorrelated coefficients of the B channel of the spline

— the quantized DCT32 coefficients of the σ parameter of the spline (defining its thickness)

After decoding, the DCT32 coefficients of the X, Y, B and σ values are divided by `quant_adjust >= 0 ? 1 + quant_adjust / 8 : 1 / (1 + quant_adjust / 8)`. The coefficients are then multiplied by `kChannelWeight[channel]`, where `channel` is in the range [0, 4), corresponding respectively to X, Y, B and σ, and `kChannelWeight[4]` is `{0.0042, 0.075, 0.07, 0.3333}`.

Before rendering splines, the decoder adds Y × `base_correlation_x` and Y × `base_correlation_b`, respectively, to the X and B channels (see C.4.4).

C.4.7 Noise synthesis parameters

If the `kNoise` flag in `frame_header` (C.2) is not set, this subclause is skipped for that frame.

The 8 LUT values representing the noise synthesis parameters at different intensity levels are decoded sequentially as specified by the following code:

```
for (i = 0; i < 8; i++) lut[i] = u(10) / (1 << 10);
```

C.4.8 GlobalModular

First, the decoder reads a u(1) to determine whether a global tree is to be decoded. If true, an MA tree is decoded as described in D.7.2.

The decoder then decodes a modular sub-bitstream (C.9), where the number of channels is equal to `(frame_header.encoding == kVarDCT ? 0 : (!frame_header.do_YCbCr && !metadata.xyb_encoded && metadata.colour_encoding.colour_space == kGrey ? 1 : 3)) + num_extra_channels` and the dimensions correspond to `width × height`, except for extra channels with `dim_shift` > 0 (as specified in C.9) which have dimensions `ceil(width / ` $2^{\text{dim\_shift}}$`)` × `ceil(height / ` $2^{\text{dim\_shift}}$`)`.

The channel order is:

— If `frame_header.encoding == kModular`: first Grey or (Red, Green, Blue) or (Y', X', B') or (Y, Cb, Cr)

— Then the extra channels (if any), including alpha channel, in ascending order of index.

However, the decoder only decodes the first `nb_meta_channels` channels and any further channels that have a width and height that are both at most `kGroupDim`. At that point, it stops decoding. No inverse transforms are applied yet.

NOTE The remaining channels are split into groups and are decoded later (see C.5.2 and C.8.2). If the image is smaller than `kGroupDim` × `kGroupDim`, the entire image is encoded in the GlobalModular section. Otherwise, the GlobalModular section only encodes global palettes and colour transforms, and all of the actual image data is encoded in modular LF groups and modular groups, unless the Squeeze transform is used. If the Squeeze transform is used (a Haar-like transform that results in a pyramid representation, see I.3) then the GlobalModular section does contain partial image data corresponding to (very) downscaled versions of the image, though for large images, the bulk of the pixel data (i.e. Squeeze residuals) is still encoded in groups.

C.5 LfGroup

C.5.1 General

In this subclause, `width` and `height` refers to the size of the current LF group, and all coordinates are relative to the top-left corner of the LF group. Table C.14 specifies LfGroup.

Table C.14 — LfGroup

condition	name	type or subclause
	ModularLfGroup	C.5.2
`frame_header.encoding == kVarDCT`	LF coefficients	C.5.3
`frame_header.encoding == kVarDCT`	HF metadata	C.5.4

C.5.2 ModularLfGroup

In the partial image decoded from the GlobalModular section, the pixels in the remaining channel data that correspond to the LF group are decoded as another modular image sub-bitstream (C.9), where the number of channels and their dimensions are derived as follows: for every remaining channel (i.e. not already decoded) in the partially decoded GlobalModular image, if that channel has `hshift` and `vshift` both at least 3, then a channel corresponding to the LF group rectangle is added, with the same `hshift` and `vshift` as in the GlobalModular image, where the group dimensions and the `x`,`y` offsets are right-shifted by `hshift` (for `x` and `width`) and `vshift` (for `y` and `height`).

The decoded modular LF group data is then copied into the partially decoded GlobalModular image in the corresponding positions.

NOTE Since the LF group dimension is (8 × `kGroupDim`) × (8 × `kGroupDim`) and `hshift` and `vshift` are >= 3, the maximum channel dimensions are `kGroupDim` × `kGroupDim`.

C.5.3 LF coefficients

If the `kUseLfFrame` flag in `frame_header` is set, this subclause is skipped and the samples from LFFrame[`frame_header.lf_frame`] are used instead of the values that would be computed by this subclause and F.2.

The decoder first reads `extra_precision` as a u(2). Next, the decoder reads a Modular sub-bitstream as described in C.9, to obtain the quantized LF coefficients `LfQuant`., which consists of three channels with `ceil(height / 8)` rows and `ceil(width / 8)` columns, where the number of rows and columns is optionally right-shifted by one according to `frame_header.jpeg_upsampling`.

Finally the LF is dequantized as specified in F.2.

C.5.4 HF metadata

The decoder reads `nb_blocks = u(ceil(log2(ceil(width/8) × ceil(height/8))`.

Then, the decoder reads a Modular sub-bitstream as described in C.9, for an image with four channels: the first two channels have with `ceil(height/64)` rows and `ceil(width/64)` columns, and are denoted as `XFromY` and `BFromY` (these are the HF colour correlation factors); the third channel has two rows and `nb_blocks` columns and is denoted as `BlockInfo`, and the fourth channel has `ceil(height / 8)` rows and `ceil(width / 8)` columns and is denoted as `Sharpness` (it contains parameters for the restoration filter).

The `DctSelect` and `HfMul` fields are derived from the first and second rows of `BlockInfo`. These two fields have `ceil(height / 8)` rows and `ceil(width / 8)` columns. They are reconstructed by iterating over the columns of `BlockInfo` to obtain a varblock transform type `type` (the sample at the first row) and a quantization multiplier `mul`(the sample at the second row). The `type` corresponds to a valid varblock type and covers a rectangle that does not cross group boundaries; this is the `DctSelect` sample and it is stored at the coordinates of the top-left 8 × 8 rectangle of the varblock, which is positioned as much towards the top and towards the left as possible without overlapping already-positioned varblocks . The `HfMul` sample is stored at the same position and gets the value 1 + `mul`.

The transform types (defined in I.2) are associated with the numerical values in Table C.15.

Table C.15 — Transform type values

Transform type	**Numerical value**	**Dimensions in `DctSelect`**
`DCT8×8`	0	1×1
`Hornuss`	1	1×1
`DCT2×2`	2	1×1
`DCT4×4`	3	1×1
`DCT16×16`	4	2×2
`DCT32×32`	5	4×4
`DCT16×8`	6	2×1
`DCT8×16`	7	1×2
`DCT32×8`	8	4×1
`DCT8×32`	9	1×4
`DCT32×16`	10	4×2
`DCT16×32`	11	2×4
`DCT4×8`	12	1×1
`DCT8×4`	13	1×1
`AFV0 – AFV3`	14 – 17	1×1
`DCT64×64`	18	8×8
`DCT64×32`	19	8×4
`DCT32×64`	20	4×8
`DCT128×128`	21	16×16
`DCT128×64`	22	16×8
`DCT64×128`	23	8×16
`DCT256×256`	24	32×32
`DCT256×128`	25	32×16
`DCT128×256`	26	16×32

C.6 HfGlobal

C.6.1 General

The decoder reads the structures in Table C.16.

Table C.16 — HfGlobal bundle

condition	name	subclause
`frame_header.encoding == kVarDCT`	Dequantization matrices	C.6.2
`frame_header.encoding == kVarDCT`	Number of HF decoding presets	C.6.4

C.6.2 Dequantization matrices

The dequantization matrices are used as multipliers for the HF coefficients, as specified in F.3. They are defined by the channel, the transform type and the index of the coefficient inside the varblock. The parameters that define the dequantization matrices are read from the stream as follows. First, the decoder reads a u(1). If this is 1, all matrices have their default encoding as specified in C.6.3. Otherwise, the decoder reads 11 sets of parameters from the codestream, in ascending order. Each set of parameters is used for the values of the `DctSelect` field specified in Table C.17.

Table C.17 — DctSelect values for dequantization matrices

Parameters index	DctSelect values	Matrix size (rows × columns)
0	`DCT`	8×8
1	`Hornuss`	8×8
2	`DCT2×2`	8×8
3	`DCT4×4`	8×8
4	`DCT16×16`	16×16
5	`DCT32×32`	32×32
6	`DCT16×8, DCT8×16`	8×16
7	`DCT32×8, DCT8×32`	8×32
8	`DCT16×32, DCT32×16`	16×32
9	`DCT4×8, DCT8×4`	8×8
10	`AFV0, AFV1, AFV2, AFV3`	8×8
11	`DCT64×64`	64×64
12	`DCT32×64, DCT64×32`	32×64
13	`DCT128×128`	128×128
14	`DCT64×128, DCT128×64`	64×128
15	`DCT256×256`	256×256
16	`DCT128×256, DCT256×128`	128×256

Each parameter in this subclause is read using F16() (9.2.6) unless otherwise specified.
For each matrix, the `encoding_mode` is read as a u(3) and interpreted per Table C.18.

Table C.18 — EncodingMode and valid indices

encoding_mode	Name	Valid index
0	`Library`	all
1	`Hornuss`	0,1,2,3,9,10
2	`DCT2`	0,1,2,3,9,10
3	`DCT4`	0,1,2,3,9,10
4	`DCT4x8`	0,1,2,3,9,10
5	`AFV`	0,1,2,3,9,10
6	`DCT`	all
7	`RAW`	all

The `encoding_mode` read for a given parameter index is such that the given index is specified as a valid index for that encoding in Table C.18. After reading the `encoding_mode`, the corresponding parameters are read as specified by the following code:

```
ReadDctParams() {
  num_params = u(4) + 1;
  vals = [[read 3 × num_params matrix in raster order]];
  for (i = 0; i < 3; i++) vals(i, 0) ×= 64;
  return vals;
}
if (encoding_mode == Library) {
  params = [[default parameters from C.6.3]];
  dct_params = [[default parameters from C.6.3]];
  dct4x4_params = [[default parameters from C.6.3]];
  [[replace encoding_mode with the default mode from C.6.3]];
} else if (encoding_mode == Hornuss) {
  params = [[read 3x3 matrix in raster order, multiply elements by 64]];
} else if (encoding_mode == DCT2) {
  params = [[read 3x6 matrix in raster order, multiply elements by 64]];
} else if (encoding_mode == DCT4) {
  params = [[read 3x2 matrix in raster order, multiply elements by 64]];
  dct_params = ReadDctParams();
} else if (encoding_mode == DCT) {
  dct_params = ReadDctParams();
} else if (encoding_mode == RAW) {
  params.denominator = F16();
  ZeroPadToByte();
  params = [[read a 3-channel image from a modular sub-bitstream (C.9) of
    the same shape as the required quant matrix]];
} else if (encoding_mode == DCT4x8) {
  parameters = [[read 3x1 matrix in raster order]];
  dct_params = ReadDctParams();
} else if (encoding_mode == AFV) {
  params = [[read 3x9 matrix in raster order]];
  for (i = 0; i < 3; i++) for (j = 0; j < 6; j++) vals(i, j) ×= 64;
  dct_params = ReadDctParams();
  dct4x4_params = ReadDctParams();
}
```

The dequantization matrices are computed for each possible value of the `DctSelect` field as the inverse of the `weights` matrix, that only depends on the corresponding parameters as specified in Table C.17 and is computed per channel.

A `weights` matrix of dimensions `X×Y` is computed as specified by `GetDCTQuantWeights()` in the following code, given an array of parameters `params` of length `len`.

```
Interpolate(pos, max, bands, len) {
  if (len == 1) return bands[0];
  scaled_pos = pos × (len - 1) / max;
  scaled_index = floor(scaled_pos);
  frac_index = scaled_pos - scaled_index;
  A = bands[scaled_index];
  B = bands[scaled_index + 1];
  interpolated_value = A × (B / A)^frac_index;
  return interpolated_value;
}
Mult(v) {
  if (v > 0) return 1 + v;
  return 1 / (1 - v);
}
GetDCTQuantWeights(params) {
  bands(0) = params[0];
  for (i = 1; i < len; i++) {
    bands(i) = bands(i - 1) × Mult(params[i]);
    [[ bands[i] >= 0 ]];
    for (y = 0; y < Y; y++)
      for (x = 0; x < X ; x++) {
        dx = x / (X - 1);
        dy = y / (Y - 1);
        distance = sqrt(dx^2 + dy^2);
        weight = Interpolate(distance, sqrt(2) + 1e-6, bands, num_bands);
        weights(x, y) = weight;
      }
  }
}
```

For encoding mode `DCT`, the `weights` matrix for channel `c` is the matrix of the correct size computed using `GetDctQuantWeights`, using row `c` of `dct_params` as an input.

For encoding mode `DCT4`, the `weights` matrix for channel `c` are obtained by copying into position `(x, y)` the value in position `(x Idiv 2, y Idiv 2)` in the `4 × 4` matrix computed by `GetDctQuantWeights` using as an input row `c` of `dct_params`; coefficients `(0,1)` and `(1,0)` are divided by `params(c, 0)`, and the `(1,1)` coefficient is divided by `params(c, 1)`.

For encoding mode `DCT2`, `params(c, i)` are copied in position `(x, y)` as follows, where rectangles are defined by their top right and bottom left corners, and *symmetric* refers to the rectangle defined by the same points but with swapped x and y coordinates:

— `i == 0`: positions `(0,1)`, `(1, 0)`

— `i == 1`: position `(1,1)`

— `i == 2`: all positions in the rectangle `((2,0), (4,2))`, and symmetric

— `i == 3`: all positions in the rectangle `((2,2), (4,4))`

— `i == 4`: all positions in the rectangle `((4,0), (8,4))`, and symmetric

— `i == 5`: all positions in the rectangle `((4,4), (8,8))`

For encoding mode `Hornuss`, coefficient `(1,1)` is equal to `params(c, 2)`. Coefficients `(0,1)` and `(1,0)` are equal to `params(c, 1)`, and all other coefficients to `params(c, 0)`. Coefficient `(0,0)` is `1`.

For encoding mode `DCT4x8`, the `weights` matrix is obtained by copying into position `(x, y)` the value in position `(x, y Idiv 2)` in the `8 × 4` matrix computed by `GetDctQuantWeights` using as an input row `c` of `dct_params`; coefficient `(0,1)` is then divided by `params(c, 0)`.

For encoding mode `AFV` and channel `c`, the decoder obtains `weights` as specified by the following code:

```
freqs = {0, 0, 0.8517778890324296, 5.37778436506804, 0, 0, 4.734747904497923,
5.449245381693219, 1.6598270267479331, 4, 7.275749096817861, 10.423227632456525,
2.662932286148962,  7.630657783650829, 8.962388608184032, 12.97166202570235};
weights4x8 = [[4x8 matrix produced by GetDctQuantWeights
  using row c of dct_params]];
weights4x4 = [[4x4 matrix produced by GetDctQuantWeights
  using row c of dct4x4_params]];
lo = 0.8517778890324296;
hi = 12.97166202570235;
bands(0) = params(c, 5);
[[ bands[0] >= 0) ]];
for (i = 1; i < 4; i++) {
  bands[i] = bands[i - 1] × Mult(params(c, i + 5));
  [[ bands[i] >= 0 ]];
}
weights(0, 0) = 1;
weights(0, 1) = params(c, 0);
weights(1, 0) = params(c, 1);
weights(0, 2) = params(c, 2);
weights(2, 0) = params(c, 3);
weights(2, 2) = params(c, 4);
for (y = 0; y < 4; y++)
  for (x = 0; x < 4; x++) {
     if (x < 2 && y < 2) continue;
     val = Interpolate(freqs[y × 4 + x] - lo, hi - lo, bands, 4);
     weight(2 × y, 2 × x) = val;
   }
for (y = 0; y < 4; y++)
  for (x = 0; x < 8; x++) {
    if (x == 0 && y == 0) continue;
    weights(x, 2 × y + 1) = weights4x8(x, y);
  }
for (y = 0; y < 4; y++)
  for (x = 0; x < 4; x++) {
    if (x == 0 && y == 0) continue;
    weights(2 × x + 1, 2 × y) = weights4x4(x, y);
  }
```

For encoding mode `RAW`, the `weights` are equal to the element-by-element inverse of the `params` matrix, divided by `params.denominator`.

Finally, the dequantization matrices for channel `c` are the element-wise inverses of the weights matrix computed for channel `c`. None of the resulting values are non-positive or infinity.

C.6.3 Default values for each dequantization matrix

The dequantization matrix encodings defined in Table C.19 are used as the default encodings for each kind of dequantization matrix.

Table C.19 — Default matrices for each DctSelect

DctSelect	mode	dct_params, params
DCT8×8	DCT	{{2560.0, 0.0, -0.4, -0.4, -0.4, -2.0}, {563.2, 0.0, -0.3, -0.3, -0.3, -0.3}, {512.0, -3.0, 0.0, 0.0, -1.0, -2.0}}, {}

Table C.19 *(continued)*

DctSelect	mode	dct_params, params
Hornuss	Hornuss	{}, {{280.0, 3160.0, 3160.0}, {60.0, 864.0, 864.0}, {18.0, 200.0, 200.0}}
DCT2×2	DCT2	{}, {{3840.0, 2560.0, 1280.0, 640.0, 480.0, 300.0}, {960.0, 640.0, 320.0, 180.0, 140.0, 120.0}, {640.0, 320.0, 128.0, 64.0, 32.0, 16.0}}
DCT4×4	DCT4	{{843.649426659137152, 0.0, 0.0, 0.0}, {289.6948005482115584, 0.0, 0.0, 0.0}, {137.04727932185712576, -0.25, -0.25, -0.5}}, {{1.0, 1.0}, {1.0, 1.0}, {1.0, 1.0}}
DCT16×16	DCT	{{8996.8725711814115328, -1.3000777393353804, -0.49424529824571225, -0.439093774457103443, -0.6350101832695744, -0.90177264050827612, -1.6162099239887414}, {3191.48366296844234752, -0.67424582104194355, -0.80745813428471001, -0.44925837484843441, -0.35865440981033403, -0.31322389111877305, -0.37615025315725483}, {1157.50408145487200256, -2.0531423165804414, -1.4, -0.50687130033378396, -0.42708730624733904, -1.4856834539296244, -4.9209142884401604}}, {}
DCT32×32	DCT	{{15718.40830982518931456, -1.025, -0.98, -0.9012, -0.4, -0.48819395464, -0.421064, -0.27}, {7305.7636810695983104, -0.8041958212306401, -0.7633036457487539, -0.5566037999011464, -0.49785304658857626, -0.43699592683512467, -0.40180866526242109, -0.27321683125358037}, {3803.53173721215041536, -3.060733579805728, -2.0413270132490346, -2.0235650159727417, -0.5495389509954993, -0.4, -0.4, -0.3}}, {}
DCT16×8, DCT8×16	DCT	{{7240.7734393502, -0.7, -0.7, -0.2, -0.2, -0.2, -0.5}, {1448.15468787004, -0.5, -0.5, -0.5, -0.2, -0.2, -0.2}, {506.854140754517, -1.4, -0.2, -0.5, -0.5, -1.5, -3.6}}, {}
DCT32×8, DCT8×32	DCT	{{16283.2494710648897, -1.7812845336559429, -1.6309059012653515, -1.0382179034313539, -0.85, -0.7, -0.9, -1.2360638576849587}, {5089.15750884921511936, -0.320049391452786891, -0.35362849922161446, -0.30340000000000003, -0.61, -0.5, -0.5, -0.6}, {3397.77603275308720128, -0.321327362693153371, -0.345076192231179 97, -0.70340000000000003, -0.9, -1.0, -1.0, -1.17546055762 65209}}, {}
DCT16×32, DCT32×16	DCT	{{13844.97076442300573, -0.97113799999999995, -0.658, -0.42026, -0.22712, -0.2206, -0.226, -0.6}, {4798.964084220744293, -0.61125308982767057, -0.83770786552491361, -0.79014862079498627, -0.2692727459704829, -0.38272769465388551, -0.22924222653091453, -0.20719098826199578}, {1807.236946760964614, -1.2, -1.2, -0.7, -0.7, -0.7, -0.4, -0.5}}, {}
DCT4×8, DCT8×4	DCT4x8	{{2198.050556016380522, -0.96269623020744692, -0.76194253026666783, -0.65511406707735 47}, {764.3655248643528689, -0.92630200888366945, -0.9675229603596517, -0.27845290869168118}, {527.107573587542228, -1.4594385811273854, -1.4500820940978715 93, -1.5843722511996204}}, {1.0, 1.0, 1.0}
AFV0, AFV1, AFV2, AFV3	AFV	{{3072, 3072, 256, 256, 256, 414, 0.0, 0.0, 0.0}, {1024, 1024, 50.0, 50.0, 50.0, 58, 0.0, 0.0, 0.0}, {384, 384, 12.0, 12.0, 12.0, 22, -0.25, -0.25, -0.25}}, {{2198.050556016380522, -0.96269623020744692, -0.76194253026666783, -0.6551140670773547}, {764.3655248643528689, -0.92630200888366945, -0.9675229603596517, -0.27845290869168118}, {527.107573587542228, -1.4594385811273854, -1.4500820940978715 93, -1.5843722511996204}}
DCT64×64	DCT	{{23966.1665298448605, SeqA}, {8380.19148390090414, SeqB}, {4493.02378009847706, SeqC}}, {}
DCT32×64, DCT64×32	DCT	{{15358.89804933239925, SeqA}, {5597.360516150652990, SeqB}, {2919.961618960011210, SeqC}}, {}
DCT128×128	DCT	{{47932.3330596897210, SeqA}, {16760.38296780180828, SeqB}, {8986.04756019695412, SeqC}}, {}
DCT64×128, DCT128×64	DCT	{{30717.796098664792, SeqA}, {11194.72103230130598, SeqB}, {5839.92323792002242, SeqC }}, {}
DCT256×256	DCT	{{95864.6661193794420, SeqA}, {33520.76593560361656, SeqB}, {17972.09512039390824, SeqC}}, {}
DCT128×256, DCT256×128	DCT	{{61435.5921973295970, SeqA}, {24209.44206460261196, SeqB}, {12979.84647584004484, SeqC }}, {}

In the above table, the following abbreviations denote a sequence of numbers: SeqA is the sequence -1.025, -0.78, -0.65012, -0.19041574084286472, -0.20819395464, -0.421064, -0.32733845535848671; SeqB is the sequence -0.3041958212306401, 0.3633036457487539, -0.35660379990111464, -0.3443074455424403, -0.33699592683512467, -0.30180866526242109, -0.27321683125358037, and SeqC is the sequence -1.2, -1.2, -0.8, -0.7, -0.7, -0.4, -0.5.

If `mode != AFV`, `dct4x4_params` is empty, otherwise it is {{843.649426659137152, 0.0, 0.0, 0.0}, {289.6948005482115584, 0.0, 0.0, 0.0}, {137.04727932185712576, -0.25, -0.25, -0.5}}.

C.6.4 Number of HF decoding presets

The decoder reads `num_hf_presets_minus_1` as `u(ceil(log2(num_groups)))`. The (per-pass) number of coefficient orders and histograms `num_hf_presets` is equal to `num_hf_presets_minus_1 + 1`.

NOTE The number of bits used to represent this number is not a constant because the set of histograms and orders to be used is decided on a per-group basis.

C.7 HfPass

C.7.1 HF coefficient order

The data described here is read `num_hf_presets` times (as decoded in C.6.4), once for each preset in ascending order. The decoder proceeds as specified by the code below, where `natural_coeff_order` is the natural coefficient order defined by `dcts`, as specified in I.2.5.

```
used_orders = U32(Val(0x5F), Val(0x13), Val(0), Bits(13)); // 13-bit mask
if (used_orders != 0)
  [[read 8 clustered distributions D according to subclause D.3]];
for(b = 0; b < 13; b++)
  if ((used_orders & (1 << b)) != 0) {
    nat_ord_perm = DecodePermutation();
    for [[each i]]
      order[i] = natural_coeff_order[nat_ord_perm[i]];
  } else {
    for [[each i]]
      order[i] = natural_coeff_order[i];
  }
```

`DecodePermutation()` is defined as follows. Let `dcts` be any `DctSelect` value with an order that is the currently decoded one, as defined in I.2.5. The decoder reads a permutation `nat_ord_perm` from a *single* stream (containing permutations for all orders) as specified in C.3.2, where `size` is the number of coefficients covered by `dcts` and `skip = size / 64`.

C.7.2 HF coefficient histograms

Let `nb_block_ctx` be equal to `max(block_ctx_map)+1`. The decoder reads a histogram with 495 × `num_hf_presets` × `nb_block_ctx` clustered distributions D from the codestream as specified in D.3.

C.8 PassGroup

C.8.1 General

In all subclauses of C.8, `width` and `height` refers to the size of the current group (at most `kGroupDim` × `kGroupDim`), and all coordinates are relative to the top-left corner of the group.

Table C.21 — PassGroup bundle

condition	name	subclause
`frame_header.encoding == kVarDCT`	HF coefficients	C.8.3
	Modular group data	C.8.2

C.8.2 Modular group data

In the partial image decoded from the GlobalModular and ModularLfGroup sections, the pixels in the remaining channel data that correspond to the group are decoded as another modular image sub-bitstream (C.9). The values `minshift` and `maxshift` are defined as follows. If this is the first (or only) pass, then `maxshift = 3`, otherwise `maxshift` is equal to the `minshift` of the previous pass. If there is an `n` such that the pass index is equal to `last_pass[n]`, then `minshift = log2(downsample[n])`, otherwise `minshift = maxshift` (i.e. this pass contains no modular data).

The number of channels and their dimensions are derived as follows: for every remaining channel in the partially decoded GlobalModular image (i.e. it is not a meta-channel, the channel dimensions exceed `kGroupDim` × `kGroupDim`, and `hshift` < 3 or `vshift` < 3, and the channel has not been already decoded in a previous pass), if that channel has `hshift` and `vshift` such that `minshift <= min(hshift,vshift) < maxshift`, then a channel corresponding to the group rectangle is added, with the same `hshift` and `vshift` as in the GlobalModular image, where the group dimensions and the `x`,`y` offsets are right-shifted by `hshift` (for `x` and `width`) and `vshift` (for `y` and `height`). The decoded modular group data is then copied into the partially decoded GlobalModular image in the corresponding positions.

When all modular groups are decoded, the inverse transforms are applied to the at that point fully decoded GlobalModular image, as specified in C.9.4.

C.8.3 HF coefficients

The decoder read `hfp = u(ceil(log2(num_hf_presets)))`, which indicates the coefficient order to be used for this group as well as the offset in the histogram, which is given by `offset` = 495 × `nb_block_ctx` × `hfp`. These are chosen from the `num_hf_presets` possibilities of the current *pass*.

To compute context, the decoder uses the procedures and constants in the following code, where `c` is the current channel (with 0=X, 1=Y, 2=B), `s` is the Order ID (see Table I.1) of the `DctSelect` value and `qf` is the `HfMul` value for the current varblock, and `qdc[3]` are the quantized LF values of (the top-left 8x8 block within) the current varblock (taking into account chroma subsampling if needed). The lists of thresholds `qf_thresholds` and `lf_thresholds[3]`, and `block_ctx_map` are as decoded in LfGlobal (C.4.1 and C.8.4).

```
BlockContext() {
  idx = (c < 2 ? c ^ 1 : 2) × 13 + s;
  idx ×= (qf_thresholds.size() + 1);
  for (t : qf_thresholds) if (qf > t) idx++;
  for (i = 0; i < 3; i++) idx ×= (lf_thresholds[i].size() + 1);
  lf_idx = 0;
  for (t : lf_thresholds[0]) if (qdc[0] > t) lf_idx++;
  lf_idx ×= (lf_thresholds[2].size() + 1);
  for (t : lf_thresholds[2]) if (qdc[2] > t) lf_idx++;
  lf_idx ×= (lf_thresholds[0].size() + 1);
  for (t : lf_thresholds[1]) if (qdc[1] > t) lf_idx++;
  return block_ctx_map[idx + lf_idx];
}
NonZerosContext(predicted) {
  if (predicted > 64) predicted = 64;
  if (predicted < 8) return BlockContext() + nb_block_ctx × predicted;
  return BlockContext() + nb_block_ctx × (4 + predicted Idiv 2);
}
CoeffFreqContext[64] = {
    0,   0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,
   15,  15,  16,  16,  17,  17,  18,  18,  19,  19,  20,  20,  21,  21,  22,  22,
   23,  23,  23,  23,  24,  24,  24,  24,  25,  25,  25,  25,  26,  26,  26,  26,
   27,  27,  27,  27,  28,  28,  28,  28,  29,  29,  29,  29,  30,  30,  30,  30};
CoeffNumNonzeroContext[64] = {
    0,      0,  31,  62,  62,  93,  93,  93,  93,  23, 123, 123, 123,
  152, 152, 152, 152, 152, 152, 152, 152, 180, 180, 180, 180, 180,
  180, 180, 180, 180, 180, 180, 180, 206, 206, 206, 206, 206, 206,
  206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206,
  206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206};
CoefficientContext(k, non_zeros, num_blocks, size, prev) {
 non_zeros = (non_zeros + num_blocks - 1) Idiv num_blocks;
 k = k Idiv num_blocks;
 return (CoeffNumNonzeroContext[non_zeros] + CoeffFreqContext[k]) × 2 +
   prev + BlockContext() × 458 + 37 × nb_block_ctx;
}
```

For each block of the group, `PredictedNonZeros(x, y)` corresponds to the following function, where `NonZeros` is defined below:

— if `x == y == 0`, `32`.

— if `x == 0 && y != 0`, `NonZeros(x, y - 1)`.

— if `x != 0 && y == 0`, `NonZeros(x - 1, y)`.

— if `x != 0 && y != 0`, `(NonZeros(x, y-1) + NonZeros(x-1, y) + 1) >> 1`.

After selecting the histogram and coefficient order, the decoder reads symbols from an entropy-coded stream, as specified in D.3.6. The decoder proceeds by decoding varblocks in raster order; for each varblock it reads channels Y, X, then B; if a channel is subsampled, its varblocks are skipped unless the varblock corresponds to the top-left corner of a non-subsampled varblock. For the purposes of this ordering, each varblock corresponds to its top-left block. For each varblock of size `X × Y` in the image, covering `num_blocks = X × Y / 64` blocks, the decoder reads an integer `non_zeros` using D[`NonZerosContext(PredictedNonZeros(x, y)) + offset`]. The decoder then computes the `NonZeros(x, y)` field for each block in the current varblock as follows. Let `i` be the difference between `x` and the x coordinate of the top-left block of the current varblock, and `j` the difference between `y` and its y coordinate. Then let `cur = j × X + i`. `NonZeros(x, y)` is then `(non_zeros + num_blocks - 1) Idiv num_blocks`. Finally, for `k` in the range [`num_blocks`, `size`), the decoder reads an integer `ucoeff` from the codestream, using `D[CoefficientContext(k, non_zeros, num_blocks, size, prev) + offset]`, where `prev` is computed as specified in the following code:

```
if (k == num_blocks) {
  if (non_zeros > size / 16) prev = 1;
  else prev = 0;
} else {
  if ([[decoded coefficient at position (k - 1) is 0]]) prev = 0;
  else prev = 1;
}
```

The decoder then sets the quantized HF coefficient in the position corresponding to index `k` in the coefficient order for the current `DctSelect` value to UnpackSigned(`ucoeff`). If `ucoeff != 0`, the decoder decreases `non_zeros` by `1`. If `non_zeros` reaches `0`, the decoder stops decoding further coefficients for the current block.

If this is not the first pass, the decoder adds decoded HF coefficients to previously-decoded ones.

C.8.4 HF Block Context decoding

The decoder reads a description of the block context model for HF as specified by the following code:

```
if (u(1)) block_ctx_map = { 0, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 6, 6,
                        7, 8, 9, 9, 10, 11, 12, 13, 14, 14, 14, 14, 14,
                        7, 8, 9, 9, 10, 11, 12, 13, 14, 14, 14, 14, 14};
else {
  for (i = 0; i < 3; i++) {
    nb_lf_thr[i] = u(4);
    for (j = 0; j < nb_lf_thr[i]; j++)
      lf_thresholds[i].push_back(UnpackSigned(U32(Bits(4), BitsOffset(8,
                      16), BitsOffset(16, 272), BitsOffset(32, 65808))));
  }
  nb_qf_thr = u(4);
  for (i = 0; i < nb_qf_thr; i++)
      lf_thresholds[i].push_back(UnpackSigned(U32(Bits(4), BitsOffset(8,
                      16), BitsOffset(16, 272), BitsOffset(32, 65808))));
  }
  nb_qf_thr = u(4);
  for (i = 0; i < nb_qf_thr; i++)
    qf_thresholds.push_back(U32(Bits(2), BitsOffset(3, 4),
                             BitsOffset(5, 12), Bitsoffset(8, 44)));
  bsize = 27 × nb_qf_thr × nb_lf_thr[0] × nb_lf_thr[1] × nb_lf_thr[2];
  block_ctx_map = [[ Read a clustering map as in D.3.5; num_distributions
                = bsize <= 1344 and the resulting num_clusters <= 16 ]];
}
```

C.9 Modular image sub-bitstream

C.9.1 General

This subclause describes the modular image sub-bitstream, which encodes all the pixel data of a frame if `frame_header.encoding == kModular`. It is also used to encode additional channels (alpha, depth and extra channels) and auxiliary images in the other frame encoding.

The modular image sub-bitstream encodes an image consisting of an arbitrary number N of channels. The dimensions are implicit (i.e. they are signalled or computed elsewhere). In the trivial case where N is zero, the decoder takes no action. The channels are described as an ordered list: `channel[0]`, `channel[1]`, `...` , `channel[N - 1]`.

Each channel has a width and height (`channel[i].width`, `channel[i].height`); initially, before transformations are taken into account, all channel dimensions are identical (with the exception of dimension-shifted extra channels, if present, and the HF Metadata encoding as described in C.5.4). Transformations can be applied to the image, which can result in changes to the number of channels and their dimensions. The sub-bitstream starts with an encoding of the series of transformations that was applied, so the decoder can anticipate the corresponding changes in the number of channels and their dimensions (so this information does not need to be explicitly signalled) and can afterwards apply the appropriate inverse transformations.

Channels also have a power-of-two horizontal and vertical subsampling factor, denoted by `channel[i].hshift` and `channel[i].vshift`, which are initially set to zero (except for the extra channels, if present, where both are initialized to the corresponding channel shift value), but which can be modified by the transformations. If the channel dimensions correspond to a subsampling of the original image dimensions, this is denoted with positive values for hshift and vshift, where the horizontal subsampling factor is 2^{hshift} and the vertical subsampling factor is 2^{vshift}. If the dimensions are not related to the original dimensions, the hshift and vshift values are set to -1.

channel[i](x, y) denotes the value of the sample in column x and row y of channel i.

The first nb_meta_channels channels are used to store information related to transformations that require extra information (for example a colour palette). Initially, nb_meta_channels is set to zero, but transformations can increment this value.

C.9.2 Image decoding

The decoder reads the fields specified in Table C.22.

Table C.22 — ModularHeader bundle

condition	type	name
	Bool()	use_global_tree
	WPHeader	wp_params
	U32(Val(0), Val(1), BitsOffset(4, 2), BitsOffset(8, 18))	nb_transforms
	TransformInfo	transform[nb_transforms]

The list of channels is initialized according to C.9.1. Their dimensions and subsampling factors are derived from the series of transforms and their parameters (C.9.4).

First, if use_global_tree is false, the decoder reads a MA tree and corresponding clustered distributions as described in D.4.2; otherwise the global MA tree and its clustered distributions are used as decoded from the GlobalModular section (C.4.8). The decoder then starts an ANS stream (D.3) and decodes the data for each channel (in ascending order of index) as specified in C.9.3, skipping any channels having width or height zero. Finally, the inverse transformations are applied (from last to first) as described in C.9.4.

Table C.23 specifies WPHeader and Table C.26 specifies TransformInfo, using TransformId as defined in Table C.24 and SqueezeParams as defined in Table C.25.

Table C.23 — WPHeader bundle

condition	type	default	name
	u(1)	true	default_wp
!default_wp	u(5)	16	wp_p1
!default_wp	u(5)	10	wp_p2
!default_wp	u(5)	7	wp_p3a
!default_wp	u(5)	7	wp_p3b
!default_wp	u(5)	7	wp_p3c
!default_wp	u(5)	0	wp_p3d
!default_wp	u(5)	0	wp_p3e
!default_wp	u(4)	13	wp_w0
!default_wp	u(4)	12	wp_w1
!default_wp	u(4)	12	wp_w2
!default_wp	u(4)	12	wp_w3

Table C.24 — TransformId

name	value	description
kRCT	0	Reversible Colour Transform
kPalette	1	Palette
kSqueeze	2	Haar-like transform

Table C.25 — SqueezeParams bundle

condition	type	name
	Bool()	horizontal
	Bool()	in_place
	U32(Bits(3), BitsOffset(6,8), BitsOffset(10, 72), BitsOffset(13, 1096))	begin_c
	U32(Val(1), Val(2), Val(3), BitsOffset(4, 4))	num_c

Table C.26 — TransformInfo bundle

condition	type	name
	Enum(TransformId)	tr
tr != kSqueeze	U32(Bits(3), BitsOffset(6,8), BitsOffset(10, 72), BitsOffset(13, 1096))	begin_c
tr == kRCT	U32(Val(6), Bits(2), BitsOffset(4, 2), BitsOffset(6, 10))	rct_type
tr == kPalette	U32(Val(1), Val(3), Val(4), BitsOffset(13, 1))	num_c
tr == kPalette	U32(BitsOffset(8, 1), BitsOffset(10, 257), BitsOffset(12, 1281), BitsOffset(16, 5377))	nb_colours
tr == kPalette	U32(Val(0), BitsOffset(8, 1), BitsOffset(10, 257), BitsOffset(16, 1281))	nb_deltas
tr == kPalette	u(4)	d_pred
tr == kSqueeze	U32(Val(0), BitsOffset(4, 1), BitsOffset(6, 9), BitsOffset(8, 41))	num_sq
tr == kSqueeze	SqueezeParams	sp[num_sq]

C.9.3 Channel decoding

The actual channel data is decoded as follows. The following code specifies prediction(x, y, predictor) for a sample at coordinates x, y.

```
left = (x > 0 ? channel[i](x − 1, y) : (y > 0 ? channel[i](x, y − 1) : 0);
top = (y > 0 ? channel[i](x, y − 1) : left);
topleft = (x > 0 && y > 0 ? channel[i](x − 1, y − 1) : left);
topright = (x + 1 < w && y > 0 ? channel[i](x + 1, y − 1) : top);
topright2 = (x + 2 < w && y > 0 ? channel[i](x + 2, y − 1) : topright);
leftleft = (x > 1 ? channel[i](x − 2, y) : left);
grad = top + left − topleft;
if (predictor == 0) return 0;
if (predictor == 1) return left;
if (predictor == 2) return top;
if (predictor == 3) return (left + top) Idiv 2;
if (predictor == 4) return (abs(grad-left) < abs(grad-top) ? left : top);
if (predictor == 5) return median(grad, left, top);
if (predictor == 6) return ([[ 'prediction' of Annex E ]] + 3) >> 3;
if (predictor == 7) return topright;
if (predictor == 8) return topleft;
if (predictor == 9) return leftleft;
if (predictor == 10) return (left + topleft) Idiv 2;
if (predictor == 11) return (topleft + top) Idiv 2;
if (predictor == 12) return (top + topright) Idiv 2;
if (predictor == 13) return (6 × top - 2 × toptop + 7 × left + leftleft +
                             topright2 + 3 × topright + 8) Idiv 16;
```

The decoder uses the MA tree and clustered distributions D as described in C.9.2.

The channel data is then reconstructed as specified by the following code:

```
for (y = 0; y < channel[i].height; y++)
  for (x = 0; x < channel[i].width; x++) {
    [[properties are defined in D.4.1]];
    leaf_node = MA(properties);
    [[diff = (read integer using D[leaf_node.ctx] according to D.3.6,
      with dist_multiplier set to the largest channel width amongst all
      channels that are to be decoded, excluding the meta-channels)]]
    diff = UnpackSigned(diff) × leaf_node.multiplier + leaf_node.offset;
    channel[i](x, y) = diff + prediction(x, y, leaf_node.predictor);
  }
```

C.9.4 Transformations

Table C.27 specifies the transformations.

Table C.27 — Transforms

Name	nb_meta_channels	Channels/dimensions impact	Subclause
kRCT	no change	no change	L.4
kPalette	+1	channels `begin_c + 1` until `begin_c + num_c - 1` are removed; one meta-channel is inserted in the beginning of the channel list	L.5
kSqueeze	no change	depends on parameters, see I.3	I.3

Annex D
(normative)

Entropy decoding

D.1 General

This Annex specifies an entropy decoder, as well as a meta-adaptive context model. Other Annexes and their subclauses indicate how these are used for various elements of the codestream.

D.2 Histogram code

D.2.1 Huffman histogram stream

A Huffman histogram stream stores metadata required for decoding a Huffman symbol stream. See IETF RFC 7932:2016 section 3.5 ("Complex Prefix Codes") for format encoding description and definitions. The alphabet size mentioned in the RFC is explicitly specified as parameter `alphabet_size` when the histogram is being decoded.

D.2.2 Huffman symbol stream

A Huffman symbol stream stores a sequence of unsigned integers (symbols). To read a symbol, the decoder reads a series of bits via u(1), until the bits concatenated in left-to-right order exactly match one of the prefix codes associated with the symbol. See IETF RFC 7932:2016 section 3.2 ("Use of Prefix Coding in the Brotli Format") for a detailed description of how to build the prefix codes from Huffman histogram.

D.3 ANS

D.3.1 General

This subclause describes a symbol-based stream that is decoded to a sequence of up to 8-bit symbols, each of which is taken to belong to a specific probability distribution `D`. `D` is represented as an array of values in `[0, 1 << 12]`, one per symbol, that sum to `1 << 12`. Indexing `D` with an out-of-bounds value results in a `0` value.

To decode the decoding information of sequence of integers encoded with `num_dist` clustered probability distributions, the decoder first reads the LZ77 settings `lz77` as specified in Table D.1.

Table D.1 — LZ77Params bundle

condition	type	default	name
	Bool()	false	`enabled`
`enabled`	U32(Val (224), Val(512), Val(4096), BitsOffset(15, 8))	224	`min_symbol`
`enabled`	U32(Val (3), Val(4), BitsOffset(2, 5), BitsOffset(8, 9))	3	`min_length`

If `lz77.enabled`, the decoder then reads the `HybridUintConfig` for the LZ77 length symbols (`lz_len_conf`) as specified in D.3.7, with `log_alphabet_size` equal to `8`. The decoder then reads the clustering of the `num_dist` distributions (if `lz77.enabled`, `num_dist + 1`) as specified in D.3.5, unless only one distribution is to be decoded, in which case D.3.5 is skipped. The decoder then reads `use_prefix_code` as a u(1). If `use_prefix_code` is `1`, the decoder sets `log_alphabet_size` to 5 + u(2); otherwise, it sets `log_alphabet_size` to 15. The decoder then reads the `HybridUintConfig` for each clustered distribution, in increasing order of index, as specified in D.3.7. If `use_prefix_code` is 0, the decoder reads the clustered distributions as specified in D.3.4. Otherwise, the decoder reads the symbol count for each clustered distribution as follows: if u(1) is 0, the count is 1, otherwise `n` = u(4) is read, and the count is equal to 1 + (1 << `n`) + u(`n`). The symbol count is at most 1<<15 for any distribution. The decoder then proceeds to read the clustered distribution's histograms as specified in D.2.1.

The decoder reads integers from the entropy coded stream as specified in D.3.6. If `use_prefix_code` is `1`, symbols from an entropy coded stream with this set of distributions are read according to D.2.2. Otherwise, they are read according to D.3.3.

D.3.2 Alias mapping

For a given probability distribution `D` of symbols in the range `[0, 1 << log_alphabet_size)`, tables of symbols, offsets and cutoffs are initialized according to the code below.

```
log_bucket_size = 12 - log_alphabet_size;
bucket_size = 1 << log_bucket_size;

if ([[amount of symbols with nonzero probability in D]] == 1) {
  s = [[index of the symbol with nonzero probability]];
  for (i = 0; i < (1 << 12); i++) {
    symbols[i] = s;
    offsets[i] = bucket_size × i;
    cutoffs[i] = 0;
  }
  return;
}

max_symbol = [[highest-index symbol in D with nonzero probability]];
table_size = 1 << log_alphabet_size;

for (i = 0; i < max_symbol; i++) {
  cutoffs[i] = D[i];
  if (cutoffs[i] > bucket_size) overfull[i] = i;
  else underfull[i] = i;
}
for (i = max_symbol; i < table_size; i++) {
  cutoffs[i] = 0; underfull[i] = i;
}
while (overfull.size() > 0) {
  o = overfull.back();
u = underfull.back();
  overfull.pop_back(); underfull.pop_back();
  by = bucket_size − cutoffs[u];
  cutoffs[u] -= by;
  symbols[i] = o;
  offsets[i] = cutoffs[o];
  if (cutoffs[o] < bucket_size) underfull.push_back(o);
  else overfull.push_back(o);
}
for (i = 0; i < table_size; i++) {
  if (cutoffs[i] == bucket_size) {
    symbols[i] = i;
    offsets[i] = 0; cutoffs[i] = 0;
  } else offsets[i] -= cutoffs[i];
}
```

The *alias mapping* of x under D and its corresponding initialized symbols, offsets and cutoffs is the output of the procedure `AliasMapping(x)` specified in the code below, that produces a *symbol* in [0, 128) and an *offset* in [0, 1 << 12).

```
AliasMapping(x) {
  i = x >> log_bucket_size;
  pos = x & (bucket_size - 1);
  if (pos >= cutoffs[i]) symbol = symbols[i];
  else symbol = i;
  offset = offsets[i] + pos;
  return (symbol, offset);
}
```

D.3.3 ANS symbol decoding

The ANS decoder keeps an internal 32-bit `state`. Upon creation of the decoder (immediately before reading the first symbol from a new ANS stream), `state` is initialized with a u(32) from the codestream. After the decoder reads the last symbol in a given stream, `state` is `0x130000`.

The decoder decodes a `symbol` belonging to a given distribution `D` as specified by the code below:

```
index = state & 0xFFF;
(symbol, offset) = AliasMapping(D, index);
state = D[symbol] × (state >> 12) + offset;
if (state < (1 << 16)) state = (state << 16) | u(16);
```

D.3.4 ANS distribution decoding

To decode a distribution `D`, which is a (`1 << log_alphabet_size`)-element array having elements representing symbol probabilities and summing to 1 << 12, the decoder follows the code:

```
table_size = 1 << log_alphabet_size;
[[ initialize D as array with table_size entries with value 0 ]]
if (u(1) == 1) {
  ns = u(1) + 1;
  if (ns == 1) {
    x = U8();
    D[x] = 1 << 12;
  } else {
    v1 = U8(); v2 = U8();
    [[ v1 != v2 ]]
    D[v1] = u(12);
    D[v2] = (1 << 1) - D[v1];
  }
  return;
}
if (u(1) == 1) {
  alphabet_size = U8() + 1;
  i = 0;
  for (; i < ((1 << 12) Umod alphabet_size); i++)
    D[i] = floor((1 << 12) / alphabet_size);
  for (; i < alphabet_size; i++)
    D[i] = floor((1 << 12) / alphabet_size);
  return;
}
len = 0;
while (len < 3) {
  if (u(1)) len++;
  else break;
}
shift = u(len) + (1 << len) - 1;
[[ shift <= (1 << 12) + 1 ]]
alphabet_size = U8() + 3;
kLogCountLut[128][2] =  // this is for prefix decoding
{ {3, 10}, {7, 12}, {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {5, 0},  {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {6, 11}, {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {5, 0},  {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {7, 13}, {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {5, 0},  {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {6, 11}, {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2},
  {3, 10}, {5, 0},  {3, 7}, {4, 3}, {3, 6}, {3, 8}, {3, 9}, {4, 5},
  {3, 10}, {4, 4},  {3, 7}, {4, 1}, {3, 6}, {3, 8}, {3, 9}, {4, 2} };

omit_log = -1; omit_pos = -1;
[[ initialize same as array with alphabet_size entries with value 0 ]]
for (i = 0; i < alphabet_size; i++) {
  h = u(7);
  [[ bitstream is not advanced by 7 when reading these 7 bits ]];
  [[ advance bitstream by kLogCountLut[h][0] bits ]];
  logcounts[i] = kLogCountLut[h][1];
  if (logcounts[i] == 13) {
    rle = U8(); same[i] = rle + 5; i += rle + 3; continue;
  }
  if (logcounts[i] > omit_log) { omit_log = logcounts[i]; omit_pos = i; }
}
[[ omit_pos >= 0 ]]
[[ omit_pos + 1 >= alphabet_size) || (logcounts[omit_pos + 1] != 13) ]]
prev = 0; numsame = 0;
for (i = 0; i < alphabet)size; i++) {
  if (same[i] != 0) {
    numsame = same[i] - 1;
    prev = i > 0 ? D[i - 1] : 0;
  }
  if (numsame > 0) {
    D[i] = prev; numsame--;
  } else {
    code = logcounts[i];
    if (i == omit_pos || code == 0) continue;
    else if (code == 1) D[i] = 1;
    else {
      bitcount = min(max(0, shift - ((12 - code + 1) >> 1)), code - 1);
      D[i] = (1 << (code - 1)) + (u(bitcount) << (code - 1 - bitcount));
    }
  }
  total_count += D[i];
}
D[omit_pos] = (1 << 12) - total_count;
[[ D[omit_pos] >= 0 ]]
```

D.3.5 Distribution clustering

In most cases, the probability distributions that symbols belong to can be clustered together. This subclause specifies how this clustering is read by the decoder. When referring to a distribution to be used for ANS decoding, unless otherwise specified, the distribution to be used is the distribution of the corresponding cluster.

Let `num_distributions` be the number of non-clustered distributions. The output of this procedure is an array with `num_distributions` entries, with values in the range [`0`, `num_clusters`). All integers in `[0, num_clusters)` are present in this array. Position `i` in this array indicates that distribution `i` is merged into the corresponding cluster.

After decoding the clustering map, the decoder decodes the `num_clusters` histograms, one for each distribution, appearing in the bitstream in increasing index order, as specified in D.3.4.

The decoder first reads a u(1) `is_simple` indicating a simple clustering. If `is_simple` is `1`, it then decodes a u(2) representing the number of bits per entry `nbits`. For each non-clustered distribution, the decoder reads a `u(nbits)` value that indicates the cluster that the given distribution belongs to.

Otherwise, if `is_simple` is `0`, the decoder reads a u(1) `use_mtf`. The decoder then initializes a symbol decoder using a single distribution `D` , as specified in D.3.1. For each non-clustered distribution `i`, the decoder reads an integer as specified in D.3.6. Finally, if `use_mtf`, the decoder applies an inverse move-to-front transform to the cluster mapping (see code below).

```
MTF(v[256], index) {
  value = v[index];
  for (i = index; i; --i) v[i] = v[i - 1];
  v[0] = value;
}
InverseMoveToFrontTransform(clusters[num_distributions]) {
  for (i = 0; i < 256; ++i) mtf[i] = i;
  for (i = 0; i < num_distributions; ++i) {
    index = clusters[i]; clusters[i] = mtf[index];
    if (index != 0) MTF(mtf, index);
  }
}
```

D.3.6 Hybrid integer coding

An unsigned integer is read from an entropy coded stream given a clustered context identifier `ctx` as specified by `DecodeHybridVarLenUint` in the following code. `configs[ctx]` is the `HybridUintConfig` read for the given clustered distribution ctx, as specified in D.3.1 and D.3.7. num_to_copy is initialized to 0 when the first symbol is decoded from the stream. dist_multiplier is 0 unless otherwise specified when referencing this subclause.

```
ReadUint(config, token) {
  if (token < config.split) return token;
  n = config.split_exponent + ((token - config.split) >>
                    (config.msb _in_token + config.lsb_in_token));
  lsb = token & ((1 << config.lsb_in_token) - 1);
  token = token >> config.lsb_in_token;
  token &= (1 << config.msb_in_token) - 1;
  token |= (1 << config.msb_in_token);
  return (((token << n) | u(n)) << config.lsb_in_token ) | lsb;
}
kSpecialDistances[120][2] = {
  {0, 1}, {1, 0},  {1, 1}, {-1, 1}, {0, 2}, {2, 0}, {1, 2}, {-1, 2},
  {2, 1}, {-2, 1}, {2, 2}, {-2, 2}, {0, 3}, {3, 0}, {1, 3}, {-1, 3},
  {3, 1}, {-3, 1}, {2, 3}, {-2, 3}, {3, 2}, {-3, 2}, {0, 4}, {4, 0},
  {1, 4}, {-1, 4}, {4, 1}, {-4, 1}, {3, 3}, {-3, 3}, {2, 4}, {-2, 4},
  {4, 2}, {-4, 2}, {0, 5}, {3, 4},  {-3, 4}, {4, 3}, {-4, 3}, {5, 0},
  {1, 5}, {-1, 5}, {5, 1}, {-5, 1}, {2, 5}, {-2, 5}, {5, 2}, {-5, 2},
  {4, 4}, {-4, 4}, {3, 5}, {-3, 5}, {5, 3}, {-5, 3}, {0, 6}, {6, 0},
  {1, 6}, {-1, 6}, {6, 1}, {-6, 1}, {2, 6}, {-2, 6}, {6, 2}, {-6, 2},
  {4, 5}, {-4, 5}, {5, 4}, {-5, 4}, {3, 6}, {-3, 6}, {6, 3}, {-6, 3},
  {0, 7}, {7, 0},  {1, 7}, {-1, 7}, {5, 5}, {-5, 5}, {7, 1}, {-7, 1},
  {4, 6}, {-4, 6}, {6, 4}, {-6, 4}, {2, 7}, {-2, 7}, {7, 2}, {-7, 2},
  {3, 7}, {-3, 7}, {7, 3}, {-7, 3}, {5, 6}, {-5, 6}, {6, 5}, {-6, 5},
  {8, 0}, {4, 7},  {-4, 7}, {7, 4}, {-7, 4}, {8, 1}, {8, 2}, {6, 6},
  {-6, 6}, {8, 3}, {5, 7}, {-5, 7}, {7, 5}, {-7, 5}, {8, 4}, {6, 7},
  {-6, 7}, {7, 6}, {-7, 6}, {8, 5}, {7, 7}, {-7, 7}, {8, 6}, {8, 7}};

DecodeHybridVarLenUint(ctx) {
  if (num_to_copy > 0) {
    r = window[(copy_pos++) & 0xFFFFF]; num_to_copy--;
  } else {
    token = [[ Read symbol from entropy coded stream using D[ctx] ]];
    if (token >= lz77.min_symbol) {
      num_to_copy = ReadUint(lz_len_conf, token - lz77.min_symbol)
                    + lz_ lz77.min_length ;
      token = [[ Read symbol using D[num_dists] ]];
      distance = ReadUint(configs[lz_ctx], token);
      if (dist_multiplier == 0) distance++;
      else if (distance < 120) {
        distance = kSpecialDistances[distance][0];
        distance += dist_multiplier × kSpecialDistances[distance][1];
      } else distance -= 119;
      distance = min(distance, num_decoded, 1 << 20);
      copy_pos = num_decoded - distance;
      return DecodeHybridVarLenUint(ctx);
    }
    r = ReadUint(configs[ctx], token);
  }
  window[(num_decoded++) & 0xFFFFF] = r;
  return r;
}
```

D.3.7 Hybrid integer configuration

The configuration for the hybrid unsigned integer decoder of D.3.6 is read from the bitstream as specified in the following code:

```
ReadUintConfig(log_alphabet_size) {
  split_exponent = u(ceil(log2(log_alpha_size + 1)));
  msb_in_token = 0, lsb_in_token = 0;
  if (split_exponent != log_alpha_size) {
    nbits = ceil(log2(split_exponent + 1)); msb_in_token = u(nbits);
    [[ msb_in_token is <= split_exponent ]]
    nbits = ceil(log2(split_exponent - msb_in_token + 1));
    lsb_in_token = u(nbits);
  }
  [[ lsb_in_token + msb_in_token is <= split_exponent ]]
  split = 1 << split_exponent;
}
```

D.4 Meta-Adaptive (MA) Context Modeling

D.4.1 Meta-Adaptive context model

The Meta-Adaptive context model uses a vector of integer numbers, which are called properties. To determine which context to use, a decision tree is used. The full structure of the tree is known in advance by the decoder; it is explicitly signalled in the codestream (C.9.4 describes where it is signalled and D.7.2 describes how it is signalled) so encoders can tune this tree to obtain a context model that is relevant for the actual image content. The inner nodes (decision nodes) of a MA tree contain a test of the form property[k] > value. If this test evaluates to true, then the left branch is taken, otherwise the right branch is taken. Eventually a leaf node is reached, which corresponds to a context to be used to encode a symbol, a predictor to be used, and a multiplier and offset to apply. These are denoted respectively as leaf_node.ctx, leaf_node.predictor, leaf_node.multiplier, and leaf_node.offset.

The properties used in the MA context model are given in Table D.2. Assuming the value to be decoded corresponds to the sample at column x of row y of channel number c, the neighbouring already-decoded pixels left, top, topleft, topright, and leftleft are defined as in C.9.3 and `toptop = (y > 1 ? channel[i](x, y - 2) : top)`.

Table D.2 — Property definitions

Property	Value
0	c (channel index)

1	stream index:
	for GlobalModular: 0
	for LF coefficients: 1 + LF group index
	for ModularLfGroup: 1 + number of LF groups + LF group index
	for HFMetadata: 1 + 2 × (number of LF groups) + LF group index
	for RAW dequantization tables: 1 + 3 × (number of LF groups) + table index
	for ModularGroup: 1 + 3 × (number of LF groups) + (number of tables) + `num_groups` × pass index + group index

2	y
3	x

Table D.2 *(continued)*

Property	Value
4	abs(top)
5	abs(left)
6	top
7	left
8	if x > 0: left - (the value of property 9 at position (x-1,y)) otherwise: left
9	left+top-topleft
10	left-topleft
11	topleft-top
12	top-topright
13	top-toptop
14	left-leftleft
15	max_error (see E.1)

The following code specifies a variable number of additional ‘previous channel’ properties:

```
k = 16;
for (i = c - 1; i >= 0; i--, k += 4) {
  if (channel[i].width != channel[c].width) continue;
  if (channel[i].height != channel[c].height) continue;
  if (channel[i].hshift != channel[c].hshift) continue;
  if (channel[i].vshift != channel[c].vshift) continue;
  rv = channel[i](x, y);
  rleft = (x > 0 ? channel[i](x - 1, y) : 0);
  rtop = (y > 0 ? channel[i](x, y - 1) : rleft);
  rtopleft = (x > 0 && y > 0 ? channel[i](x - 1, y - 1) : rleft);
  rp = median(rleft + rtop - rtopleft, rleft, rtop);
  property[k + 0] = abs(rv);
  property[k + 1] = rv;
  property[k + 2] = abs(rv - rp);
  property[k + 3] = rv - rp;
}
```

To find the MA leaf node, the MA tree is traversed using the above properties until a leaf node is reached, starting at the root node `tree[0]` and for each decision node `d`, testing if `property[d.property] > d.value`, proceeding to the node `tree[d.left_child]` if the test evaluates to true and to the node `tree[d.right_child]` otherwise, until a leaf node is reached.

D.4.2 MA tree decoding

The MA tree itself is decoded as follows. The decoder reads 6 clustered distributions T from the codestream as specified in D.3, and then decodes the tree as specified by the following code:

```
decode_tree() {
  ctx_id = 0; nodes_left = 1; tree.clear();
  while (nodes_left > 0) {
    [[property = (read using T[1] according to D.3.6) - 1]]
    if (property == -1) {
      leaf_node.ctx = ctx_id++;
      [[leaf_node.predictor = (read using T[2] according to D.3.6)]]
      [[uoffset = (read using T[3] according to D.3.6)]]
      leaf_node.offset = UnpackSigned(uoffset);
      [[mul_log = (read using T[4] according to D.3.6)]]
      [[mul_bits = (read using T[5] according to D.3.6)]]
      leaf_node.multiplier = (mul_bits + 1) << mul_log;
      tree.push_back(leaf_node);
    } else {
      decision_node.property = property;
      [[uvalue = (read using T[0] according to D.3.6)]]
      decision_node.value = UnpackSigned(uvalue);
      decision_node.left_child = tree.size() + nodes_left + 1;
      decision_node.right_child = tree.size() + nodes_left + 2;
      tree.push_back(decision_node); nodes_left += 2;
    }
  }
}
```

None of the values of mul_log are strictly larger than 30, and none of the values of mul_bits are strictly larger than (1 << (31-`mul_log`))-2. At the end of this procedure, `tree.size()` <= (1<<26). MA(`properties`) is defined as the leaf node resulting from traversing the MA tree with given property values `properties`.

Finally, the decoder reads (`tree.size()` + 1) / 2 clustered distributions D as specified in D.3.

Annex E
(normative)

Weighted predictor

E.1 General

This predictor is used in the Modular sub-bitstream (C.9). The decoding of individual samples proceeds sequentially in raster order, with `x` the column index of the current sample (starting from 0) and `y` the row index of the current sample (starting from 0).

For each sample, the predictor is invoked as specified in subclause E.2. It computes a `prediction` (which is used in C.9.3) and a `max_error` (which is used in Table D.1), as well as 4 sub-predictor values `subpred[i]`, with i in `[0, 4)`.

After decoding a sample, the decoder then computes `true_err` and `err[i]` for the current sample and stores them for use in predictions of next samples.

The `true_value` corresponds to the decoded sample value. The `true_err` for that sample is calculated as the difference between the `prediction` and the `true_value` of the sample (left-shifted by 3 bits). The error `err[i]` of sub-predictor `subpred[i]` is computed as follows: `err[i] = abs(((subpred[i] + 3) >> 3) - true_value)`.

E.2 Prediction

The computation of predictors is based on six samples already scanned that are closest to the current sample - the `NW` (NorthWest), `N` (North), `NE` (NorthEast), `W` (West) samples, `NN` (North of North if in third or further row, or same as `N` when in second row) and `WW` (West of West if in third or further column, or same as `W` otherwise), as shown in Figure E.1.

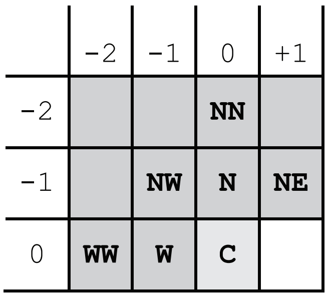


Figure E.1 — — Neighbours used for prediction

The variables `N`, `NW`, `NE`, `W`, `NN`, and `WW` are computed as the true values of their corresponding samples left shifted by 3, and these symbols in subscripts refer to the corresponding variable for the sample at that location (e.g. $\texttt{true\_err}_{W}$).

The variables `x` and `y` are the coordinates of the current sample. When `x` or `y` are near the borders, some neighbour samples might not exist. Unless otherwise noted, the following replacements are done for non-existing samples:

— If no `NE` sample exists, `NE` is set to `N` and subscripts NE to subscripts N, e.g. $\texttt{true\_err}_{NE}$, corresponds to $\texttt{true\_err}_{N}$.

— If no `NN` sample exists, `NN` is set to `N` and subscripts NN to subscripts N, e.g. `true_err`$_{NN}$, corresponds to `true_err`$_{N}$.

— If no `WW` sample exists, `WW` is set to `W` and subscripts WW to subscripts W, e.g. `true_err`$_{WW}$, corresponds to `true_err`$_{W}$.

When computing a sample that is not in the first row or first column of the image, the sub-predictors are computed as specified by the following code:

```
subpred[0] = W + NE - N;
subpred[1] = N - (((true_err_W + true_err_N + true_err_NE) × wp_p1) >> 5);
subpred[2] = W - (((true_err_W + true_err_N + true_err_NW) × wp_p2) >> 5);
subpred[3] = N + (((true_err_NW × wp_p3a + true_err_N × wp_p3b +
     true_err_NE × wp_p3c + (NN - N) × wp_p3d + (NW - W) × wp_p3e) >> 5);
```

The weights `weight[i]` for each of the 4 sub-predictions and `mxe` are computed as specified by the following code, based on the `err` values already computed for sub-predictors of earlier samples. Here `err[i]`$_{LOCATION}$ is taken to have value 0 if the corresponding sample does not exist.

```
error2weight(err_sum, maxweight) {
  shift = floor(log2(err_sum + 1)) - 5;
  if (shift < 0) shift = 0;
  return 4 + (maxweight × ((1 << 24) Idiv ((err_sum >> shift) + 1)));
}
err_sum[i] = err[i]_N + err[i]_W + err[i]_NW + err[i]_WW + err[i]_NE;
weight[i] = error2weight(err_sum[i], wp_wi);
```

The `prediction` is then computed based on the sub-predictions and their `weight` values as specified by the following code:

```
sum_weights = weight[0] + weight[1] + weight[2] + weight[3];
log_weight = floor(log2(sum_weights)) + 1;
for (i = 0; i < 4; i++) weight[i] = weight[i] >> (log_weight - 5);
sum_weights = weight[0] + weight[1] + weight[2] + weight[3];
s = sum_weights >> 1;
for (i = 0; i < 4; i++) s += subpred[i] × weight[i];
prediction = s × ((1 << 24) Idiv sum_weights) >> 24;
// if true_err_N, true_err_W and true_err_NW have the same sign
if (((true_err_N ^ true_err_W) | (true_err_N ^ true_err_NW)) <= 0) {
  prediction = clamp(prediction, min(W, N, NE), max(W, N, NE));
}
```

NOTE The integer divisions above can be implemented efficiently using a look-up table because the numerator is always the same (1 << 24) and the denominator is an integer between 1 and 64.

The value of `max_error` is computed as specified by the following code:

```
max_error = true_errW;
if (abs(true_errN) > abs(max_error)) max_error = true_errN;
if (abs(true_errNW) > abs(max_error)) max_error = true_errNW;
if (abs(true_errNE) > abs(max_error)) max_error = true_errNE;
```

Annex F
(normative)

Adaptive quantization

F.1 General

In var-DCT mode, quantized LF and HF coefficients qX, qY and qB are converted into values dX, dY and dB as specified in F.2 and F.3, respectively.

F.2 LF dequantization

If the kUseLfFrame flag in frame_header is set, this subclause is skipped for that frame. Let mXDC, mYDC and mBDC be the three per-channel LF dequantization multipliers (see C.4.2).

For each LF group, the dequantization process is influenced by the number extra_precision, as described in C.5.2. Dequantized values are computed as specified by the following code:

```
dX = mXDC × qX / (1 << extra_precision);
dY = mYDC × qY / (1 << extra_precision);
dB = mBDC × qB / (1 << extra_precision);
```

After dequantizing LF coefficients, the decoder applies the following adaptive smoothing algorithm, unless the kSkipAdaptiveLFSmoothing flag is set in frame_header. If this adaptive smoothing procedure is applied, no channel is subsampled.

For each LF sample of the image that is not in the first or last row or column, the decoder computes a weighted average wa of the 9 samples that are in neighbouring rows and columns, using a weight of 0.05226273532324128 for the current sample s, 0.20345139757231578 for horizontally/vertically adjacent samples, and 0.0334829185968739 for diagonally adjacent samples. Let gap = max(0.5, abs(wa_X-s_X)/mXDC, abs(wa_Y-s_Y)/mYDC, abs(wa_B-s_B)/mBDC). The smoothed value is (s − wa) × max(0, 3 − 4 × gap) + wa.

After applying this smoothing, the decoder uses the process described in I.2 to compute the top-left X / 8 × Y / 8 coefficients of each varblock of size X × Y, using the corresponding X / 8 × Y / 8 samples from the dequantized LF image.

F.3 HF dequantization

Every quantized HF coefficient quant is first bias-adjusted as specified by the following code, depending on its channel (0 for X, 1 for Y or 2 for B).

```
oim = metadata.opsin_inverse_matrix;
if (abs(quant) <= 1) quant ×= oim.quant_bias[[channel]];
else quant -= oim.quant_bias_numerator / quant;
```

The resulting quant is then multiplied by a per-block multiplier, the value of HfMul at the coordinates of the 8 × 8 rectangle containing the current sample, and, for the X and B channels, by $0.8^{\text{frame\_header.x\_qm\_scale} - 2}$ and $0.8^{\text{frame\_header.b\_qm\_scale} - 2}$, respectively.

The final dequantized value is obtained by multiplying the result by a multiplier defined by the channel, the transform type and the coefficient index inside the varblock, as specified in C.6.2.

Annex G
(normative)

Chroma from luma

This Annex only applies to var-DCT mode. This annex is skipped if any channel is subsampled.

Each X, Y and B sample is reconstructed from the dequantized samples `dX`, `dY` and `dB` using a linear chroma from luma model. The reconstruction uses the colour correlation coefficient multipliers `kX` and `kB` (computed from constants defined in C.4.4) to restore the correlation between the X/B and the Y channel, as specified by the following code:

```
kX = base_correlation_x + x_factor / colour_factor;
kB = base_correlation_b + b_factor / colour_factor;
Y = dY;
X = dX + kX × Y;
B = dB + kB × Y;
```

For LF coefficients, `x_factor` and `b_factor` correspond to `x_factor_lf - 127` and `b_factor_lf - 127`, respectively (C.4.4). For HF coefficients, `x_factor` and `b_factor` are values from `XFromY` and `BFromY` (C.5.4), respectively, at the coordinates of the `64 × 64` rectangle containing the current sample.

Annex H
(normative)

Extensions

The `extensions` field in some header bundles enables backward- and forward-compatible codestream extensions as described in A.5. Extensions for ImageMetadata, FrameHeader, and RestorationFilter are not currently defined, so the decoder reads and ignores any extension bits specified there.

Annex I
(normative)

Integral transforms

I.1 General

In var-DCT mode, the decoder applies forward and inverse Discrete Cosine Transforms as specified in I.2. In I.2, a matrix of size `RxC` refers to a matrix with `R` rows and `C` columns.

In modular mode, the decoder applies Squeeze as specified in I.3.

I.2 Variable DCT

I.2.1 One-dimensional DCT and IDCT

One-dimensional discrete cosine transforms and their inverses (DCT) are computed as follows. The input is a vector of size `s`, where `s` is a power of two.

Let in and out denote the inputs and outputs. Then the forward DCT transform is defined as:

$$\text{out}_k = \frac{1}{s}\left(k == 0\ ?\ 1\ :\ \sqrt{2}\right)\sum_{n=0}^{s-1} \text{in}_n \cdot \cos\left(\frac{\pi k}{s}\left(n+\frac{1}{2}\right)\right)$$

The inverse DCT transform (IDCT) is defined as follows:

$$\text{in}_k = \text{out}_0 + \sum_{n=1}^{s-1} \sqrt{2} \cdot \text{out}_n \cdot \cos\left(\frac{\pi n}{s}\left(k+\frac{1}{2}\right)\right)$$

I.2.2 Two-dimensional DCT and IDCT

In this subclause, `Transpose` is the matrix transpose and `ColumnIDCT/ColumnDCT` performs a one dimensional IDCT/DCT on each column vector of the given matrix as defined in I.2.1. The DCT transform of a `RxC` matrix `DCT_2D(samples)` is specified by the following code:

```
dct1 = ColumnDCT(samples);
dct1_t = Transpose(dct1);
dct2 = ColumnDCT(dct1_t);
if (C > R) result = Transpose(dct2);
else result = dct2;
```

NOTE The final `Transpose` ensures the DCT result has the same shape as the original varblock if the original varblock has more columns than rows, and has the shape of its transpose otherwise.

The inverse DCT (IDCT) to obtain a `RxC` matrix of samples `IDCT_2D(coefficients)` is specified by the following code:

```
if (C > R) dct2 = Transpose(coefficients);
else dct2 = coefficients;
dct1_t = ColumnIDCT(dct2);
dct1 = Transpose(dct1_t);
varblock = ColumnIDCT(dct1);
```

The remainder of this subclause uses the convention that a `4 × 4` matrix is stored in an array such that entry `(x, y)` corresponds to index `4 × y + x`.

The AFV transform uses the orthonormal basis specified by the following code:

```
AFVBasis[16][16] = {{0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25,
0.25, 0.25, 0.25, 0.25, 0.25},
{0.876902929799142, 0.2206518106944235, -0.10140050393753763, -0.1014005039375375,
0.2206518106944236, -0.10140050393753777, -0.10140050393753772, -0.10140050393753763,
-0.10140050393753758, -0.10140050393753769, -0.1014005039375375, -0.10140050393753768,
-0.10140050393753768, -0.10140050393753759, -0.10140050393753763, -0.10140050393753741},
{0.0, 0.0, 0.40670075830260755, 0.44444816619734445, 0.0, 0.0, 0.19574399372042936,
0.2929100136981264, -0.40670075830260716, -0.19574399372042872, 0.0, 0.11379074460448091,
-0.44444816619734384, -0.29291001369812636, -0.1137907446044814, 0.0},
{0.0, 0.0, -0.21255748058288748, 0.3085497062849767, 0.0, 0.4706702258572536,
-0.16212051957222993, 0.0, -0.21255748058287047, -0.16212051957228327,
-0.47067022585725277, -0.1464291867126764, 0.3085497062849487, 0.0, -0.14642918671266536,
0.4251149611657548,},
{0.0, -0.7071067811865474, 0.0, 0.0, 0.7071067811865476, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0},
{-0.4105377591765233, 0.6235485373547691, -0.06435071657946274, -0.06435071657946266,
0.6235485373547694, -0.06435071657946284, -0.0643507165794628, -0.06435071657946274,
-0.06435071657946272, -0.06435071657946279, -0.06435071657946266, -0.06435071657946277,
-0.06435071657946277, -0.06435071657946273, -0.06435071657946274, -0.0643507165794626},
{0.0, 0.0, -0.4517556589999482, 0.15854503551840063, 0.0, -0.04038515160822202,
0.0074182263792423875, 0.39351034269210167, -0.45175565899994635, 0.007418226379244351,
0.1107416575309343, 0.08298163094882051, 0.15854503551839705, 0.3935103426921022,
0.0829816309488214, -0.45175565899994796},
{0.0, 0.0, -0.304684750724869, 0.5112616136591823, 0.0, 0.0, -0.290480129728998,
-0.06578701549142804, 0.304684750724884, 0.2904801297290076, 0.0, -0.23889773523344604,
-0.5112616136592012, 0.06578701549142545, 0.23889773523345467, 0.0},
{0.0, 0.0, 0.3017929516615495, 0.25792362796341184, 0.0, 0.16272340142866204,
0.09520022653475037, 0.0, 0.3017929516615503, 0.09520022653475055, -0.16272340142866173,
-0.35312385449816297, 0.25792362796341295, 0.0, -0.3531238544981624,
-0.6035859033230976,},
{0.0, 0.0, 0.40824829046386274, 0.0, 0.0, 0.0, 0.0, -0.4082482904638628,
-0.4082482904638635, 0.0, 0.0, -0.40824829046386296, 0.0, 0.4082482904638634,
0.408248290463863, 0.0},
{0.0, 0.0, 0.1747866975480809, 0.0812611176717539, 0.0, 0.0, -0.3675398009862027,
-0.307882213957909, -0.17478669754808135, 0.3675398009862011, 0.0, 0.4826689115059883,
-0.08126111767175039, 0.30788221395790305, -0.48266891150598584, 0.0},
{0.0, 0.0, -0.21105601049335784, 0.18567180916109802, 0.0, 0.0, 0.49215859013738733,
-0.38525013709251915, 0.21105601049335806, -0.49215859013738905, 0.0,
0.17419412659916217, -0.18567180916109904, 0.3852501370925211, -0.1741941265991621, 0.0},
{0.0, 0.0, -0.14266084808807264, -0.3416446842253372, 0.0, 0.7367497537172237,
0.24627107722075148, -0.08574019035519306, -0.14266084808807344, 0.24627107722075137,
0.14883399227113567, -0.04768680350229251, -0.3416446842253373, -0.08574019035519267,
-0.047686803502292804, -0.14266084808807242},
{0.0, 0.0, -0.13813540350758585, 0.3302282550303788, 0.0, 0.08755115000587084,
-0.07946706605909573, -0.4613374887461511, -0.13813540350758294, -0.07946706605910261,
0.49724647109535086, 0.12538059448563663, 0.3302282550303805, -0.4613374887461554,
0.12538059448564315, -0.13813540350758452},
{0.0, 0.0, -0.17437602599651067, 0.0702790691196284, 0.0, -0.2921026642334881,
0.3623817333531167, 0.0, -0.1743760259965108, 0.36238173335311646, 0.29210266423348785,
-0.4326608024727445, 0.07027906911962818, 0.0, -0.4326608024727457,
0.34875205199302267,},
{0.0, 0.0, 0.11354987314994337, -0.07417504595810355, 0.0, 0.19402893032594343,
-0.435190496523228, 0.21918684838857466, 0.11354987314994257, -0.4351904965232251,
0.5550443808910661, -0.25468277124066463, -0.07417504595810233, 0.2191868483885728,
-0.25468277124066413, 0.1135498731499429}};
```

The AFV IDCT transform `AFV_IDCT` of a square `4×4` varblock from `coefficients` to `samples` is computed as specified by the following code:

```
for (i = 0; i < 16; ++i) {
  sample = 0;
  for (j = 0; j < 16; ++j) sample += coefficients[j] × AFVBasis[j][i];
  samples[i] = sample;
}
```

I.2.3 Coefficients to samples

I.2.3.1 General

Each of the subsequent subclauses in I.2.3 specifies how the decoder converts the coefficients to pixels where `DctSelect` matches one of the types listed in the heading of the subclause.

I.2.3.2 DCT8×8, DCT8×16, DCT16×8, DCT16×16, DCT8×32, DCT32×8, DCT16×32, DCT32×16, DCT32×32, DCT32×64, DCT64×32, DCT64×64, DCT64×128, DCT128×64, DCT128×128, DCT128×256, DCT256×128, DCT256×256

The `RxC` matrix of samples are obtained using `samples = IDCT_2D(coefficients)` for a `DctSelect` value of DCTR×C.

I.2.3.3 DCT2×2

The samples of a DCT2×2 varblock are computed from an 8×8 block of coefficients. `AuxIDCT2x2(block, S)` is specified by the following code, in which `block` is a matrix of size `S×S`, and `S` is a power of two:

```
AuxIDCT2x2(block, S) {
  num_2x2 = S / 2;
  for (y = 0; y < num_2x2; y++)
    for (x = 0; x < num_2x2; x++) {
      c00 = block(x, y);
      c01 = block(num_2x2 + x, y);
      c10 = block(x, y + num_2x2);
      c11 = block(num_2x2 + x, y + num_2x2);
      r00 = c00 + c01 + c10 + c11;
      r01 = c00 + c01 - c10 - c11;
      r10 = c00 - c01 + c10 - c11;
      r11 = c00 - c01 - c10 + c11;
      result(x × 2, y × 2) = r00;
      result(x × 2 + 1, y × 2) = r01;
      result(x × 2, y × 2 + 1) = r10;
      result(x × 2 + 1, y × 2 + 1) = r11;
    }
}
```

When `block` is larger than `S×S`, `AuxIDCT2x2` modifies only the top `S×S` cells. In that case, the decoder copies the rest of the cells from `block` to `result`. The inverse DCT2×2 is specified by the following code:

```
block = AuxIDCT2x2(block, 2);
block = AuxIDCT2x2(block, 4);
samples = AuxIDCT2x2(block, 8);
```

I.2.3.4 DCT4×4

The samples are computed from the 8×8 coefficient matrix as specified by the following code, in which `block` is a 4×4 matrix, and `sample` is the answer represented here as a 2×2 matrix whose elements are 4×4 matrices.

```
dcs = AuxIDCT2x2(coefficients, 2);
for (y = 0; y < 2; y++)
  for (x = 0; x < 2; x++) {
    for (iy = 0; iy < 4; iy++)
      for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
        block(ix, iy) = coefficients(x + ix × 2, y + iy × 2);
    block(0, 0) = dcs(x, y);
    sample(x, y) = IDCT_2D(block);
  }
```

The decoder re-shapes the `sample` array to an 8×8 matrix using `result(4 × i + k, 4 × j + L) = sample(i, j, k, L)`.

I.2.3.5 Hornuss

The 8×8 input `coefficients` are transformed to 8×8 `samples` as specified by the following code:

```
dcs = AuxIDCT2x2(coefficients, 2);
for (y = 0; y < 2; y++)
  for (x = 0; x < 2; x++) {
    block_lf = dcs(x, y);
    residual_sum = 0;
    for (iy = 0; iy < 4; iy++)
      for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
        residual_sum += coefficients(x + ix × 2, y + iy × 2);
    sample(4 × x + 1, 4 × y + 1) = block_lf - residual_sum / 16.0;
    for (iy = 0; iy < 4; iy++)
      for (ix = 0; ix < 4; ix++) {
        if (ix == 1 && iy == 1) continue;
        sample(x × 4 + ix, y × 4 + iy) =
            coefficients(x + ix × 2, y + iy × 2) +
            sample(4 × x + 1, 4 × y + 1);
      }
    sample(4 × x, 4 × y) =
        coefficients(x + 2, y + 2) + sample(4 × x + 1, 4 × y + 1);
  }
```

I.2.3.6 DCT8×4

The 8×8 `samples` are reconstructed from the 8×8 `coefficients` by dividing them into two 8×4 vertical blocks `samples_8x4`, as specified by the following code, in which `coeffs_4x8` denotes a temporary 4 × 8 matrix and `IDCT_2D` is defined in I.2.2.

```
coef0 = coefficients(0, 0); coef1 = coefficients(0, 1);
dcs = {coef0 + coef1, coef0 - coef1};
for (x = 0; x < 2; x++) {
  coeffs_8x4(0, 0) = dcs[x];
  for (iy = 0; iy < 4; iy++)
    for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++) {
      coeffs_4x8(ix, iy) = coefficients(ix, x + iy × 2);
    }
  samples_8x4[x] = IDCT_2D(coeffs_4x8);
}
```

I.2.3.7 DCT4×8

The 8×8 `samples` are reconstructed from the 8×8 `coefficients` by dividing them into two 4×8 horizontal blocks `samples_4x8`, as specified by the following code, in which `coeffs_4x8` denotes a temporary 4 × 8 matrix and `IDCT_2D` is defined in I.2.2.

```
coef0 = coefficients(0, 0); coef1 = coefficients(0, 1);
dcs = {coef0 + coef1, coef0 - coef1};
for (y = 0; y < 2; y++) {
  coeffs_4x8(0, 0) = dcs[y];
  for (iy = 0; iy < 4; iy++)
    for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++)
      coeffs_4x8(ix, iy) = coefficients(ix, y + iy × 2);
  samples_4x8[y] = IDCT_2D(coeffs_4x8);
}
```

I.2.3.8 AFV0, AFV1, AFV2, AFV3

The 8×8 `samples` are reconstructed from the 8×8 `coefficients` by dividing them into two 4×4 square blocks and one horizontal 4×8, as specified by the following code. Four temporary 4×4 matrices `coeffs_afv`, `samples_afv`, `coeffs_4x4` and `samples_4x4`, as well as two temporary 4×8 matrices `coeffs_4x8`, and `samples_4x8` are used. `AFV_IDCT` and `IDCT_2D` are defined in I.2.2. The values of `flip_x` and `flip_y` are defined as follows: for AFV*n*, `flip_x` is equal to *n* `& 1` and `flip_y` is equal to *n* `Idiv 2`.

```
coeffs_afv(0, 0) =
  (coefficients(0, 0) + coefficients(0, 1) + coefficients(1, 0)) × 4.0;
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
    coeff_afv(ix, iy) = coefficients(ix × 2, iy × 2);
samples_afv = AFV_IDCT(coeff_afv);
for (iy = 0; iy < 4; iy++)
```

```
  for (ix = 0; ix < 4; ix++)
    samples(flip_x × 4 + ix, flip_y × 4  + iy) =
      samples_afv(flip_x == 1 ? 3 - ix : ix, flip_y == 1 ? 3 - iy : iy);
coeffs_4×4(0, 0) = coefficients(0, 0) - coefficients(0, 1)
                   + coefficients(1, 0);
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
    coeffs_4x4(ix, iy) = coefficients(ix × 2 + 1, iy × 2);
samples_4×4 = IDCT_2D(coeffs_4×4);
for (iy = 0; iy < 4; iy++)
  for (ix = 0; ix < 4; ix++)
    samples((flip_x == 1 ? 0 : 4) + ix, flip_y × 4  + iy) =
      samples_4×4(ix, iy);
coeffs_4×8(0, 0) = coefficients(0, 0) - coefficients(1, 0);
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++)
    coeffs_4×8(ix, iy) = coefficients(ix, 1 + iy × 2);
samples_4×8 = IDCT_2D(coeffs_4×8);
for (iy = 0; iy < 4; iy++)
  for (ix = 0; ix < 8; ix++)
    samples(ix, (flip_y == 1 ? 0 : 4) + iy) = samples_4×4(ix, iy);
```

I.2.4 Natural ordering of the DCT coefficients

For every `DctSelect` value (Hornuss, DCT2×2, etc), the natural order of the coefficients is computed as follows. The varblock size `(bwidth, bheight)` for a `DctSelect` value with name "DCTN×M" is `bwidth = max(8, max(N, M))` and `bheight = max(8, min(N, M))`, respectively. The varblock size for all other transforms is `bwidth = bheight = 8`. The natural ordering of the DCT coefficients is defined as a vector order of cell positions `(x, y)` between `(0, 0)` and `(bwidth, bheight)`, described below. The number of elements in the vector `order` is therefore `bwidth × bheight`, and the vector is defined as the elements of `LLF` in their original order followed by the elements of `HF` also in their original order. `LLF` is a vector of lower frequency coefficients, containing cells `(x, y)` with `(x < (bwidth / 8)) && (y < (bheight / 8))`. The cells `(x, y)` that do not satisfy this condition belong to the higher frequencies vector `HF`.

The rest of this subclause specifies how to order the elements within each of the arrays `LLF` and `HF`. The pairs `(x, y)` in the `LLF` vector is sorted in ascending order according to the value `y × bwidth + x`. For the pairs `(x, y)` in the `HF` vector, the decoder first computes the value of the variables `key1` and `key2` as specified by the following code:

```
cx = bwidth / 8; cy = bheight / 8;
scaled_x = x × max(cx, cy) / cx;
scaled_y = y × max(cx, cy) / cy;
key1 = scaled_x + scaled_y;
key2 = scaled_x - scaled_y;
if (key1 Umod 2 == 1) key2 = -key2;
```

The decoder sorts the `(x, y)` pairs on the vector `HF` in ascending order according to the value `key1`. In case of a tie, the decoder also sorts in ascending order according to the value `key2`.

The *order ID* is defined based on the `DctSelect` as defined in Table I.1.

Table I.1 — Order ID for DctSelect values

Order ID	DctSelect
0	DCT8×8
1	Hornuss, DCT2×2, DCT4×4, DCT4×8, DCT8×4, AFV0, AFV1, AFV2, AFV3
2	DCT16×16
3	DCT32×32
4	DCT16×8, DCT8×16
5	DCT32×8, DCT8×32
6	DCT32×16, DCT16×32
7	DCT64×64
8	DCT32×64, DCT64×32
9	DCT128×128
10	DCT64×128, DCT128×64
11	DCT256×256
12	DCT128×256, DCT256×128

I.2.5 LLF coefficients from downsampled image

This subclause specifies how to obtain the low frequency DCT coefficients LLF from an 8 times downsampled image (LF), for each DctSelect value.

The input is a matrix with `bwidth / 8` columns and `bheight / 8` rows containing the LF coefficients, or equivalently, the downsampled image. The possible values for `bwidth` and `bheight` are 8, 16, 32, and 64.

The output is a matrix with `max(bwidth, bheight) / 8` columns and `min(bwidth, bheight) / 8` rows. If the caller requires a matrix of a larger size, the decoder zero-initializes the other elements. The following code specifies the function `ScaleF`.

```
I8(N, u) {
  eps = (u == 0) ? sqrt(0.5) : 1;
  return sqrt(2.0 / N) × eps × cos(u × π / (2.0 × N));
}
D8(N, u) { return 1 / (N × I8(N, u)); }
I(N, u) { return (N == 8) ? I8(N, u) : D8(N, u); }
D(N, u) { return (N == 8) ? D8(N, u) : I8(N, u); }
C(N, n, x) {
  if (n > N) return 1 / C(n, N, x);
  if (n == N) return 1;
  else return cos(x × π / (2 × N)) × C(N / 2, n, x);
}
ScaleF(N, n, x) { return sqrt(n × N) × D(N, x) × I(n, x) × C(N, n, x); }
```

For DctSelect types DCT8×8, DCT8×16, DCT8×32, DCT16×8, DCT16×16, DCT32×8, DCT32×16, DCT32×32, DCT32×64, DCT64×32, DCT64×64, DCT64×128, DCT128×64, DCT128×128, DCT128×256, DCT256×128, and DCT256×256 the output is computed as specified by the following code:

```
cx = bwidth / 8;
cy = bheight / 8;
dc = DCT_2D(input);
for (y = 0; y < cy; y++)
```

```
  for (x = 0; x < cx; x++)
    output(x, y) = dc(x, y) × ScaleF(cy, bheight, y) ×
                               ScaleF(cx, bwidth, x);
```

For DctSelect types IDENTIFY, DCT2×2, DCT4×4, DCT8×4, DCT4×8, AFV0, AFV1, AFV2, AFV3, the output is equal to the input.

I.3 Squeeze

I.3.1 Parameters

The squeeze transform consists of a series of horizontal and vertical squeeze steps. The sequence of squeeze steps to be applied is defined by `sp`, which is an array of SqueezeParams (see Table C.25); if the array is empty, default parameters are used which are derived from the image dimensions and number of channels.

For every squeeze step `sp[i]`, the `sp[i].num_c` input channels starting at position `sp[i].begin_c` are replaced by the squeezed channels, and the residual channels are inserted either right after the squeezed channels (if `sp[i].in_place == true`), or at the end of the channel list (otherwise). To determine the new list of channels (i.e. when interpreting the transformation description, before channel data decoding starts), the steps are applied in the order in which they are specified. After the channel data has been decoded, to apply the inverse transformation, the steps are applied in reverse order.

Let `begin = sp[i].begin_c`, `end = begin + sp[i].num_c - 1`, and `offset = (sp[i].in_place ? end + 1 : channel.size());`

The channel list is modified as specified by the following code:

```
for (i = 0; i < sp.size() - 1; i++) {
  r = sp[i].in_place ? end + 1 : channel.size();
  for (c = begin; c <= end; c++) {
    if (sp[i].horizontal) {
      w = channel[c].width;
      channel[c].width = (w + 1) Idiv 2;
      channel[c].hshift++;
      residu = channel[c].copy();
      residu.width = w Idiv 2;
    } else {
      h = channel[c].height;
      channel[c].height = (h + 1) Idiv 2;
      channel[c].vshift++;
      residu = channel[c].copy();
      residu.height = h Idiv 2;
    }
    [[Insert residu into channel at index r + c - begin]]
  }
}
```