



International
Standard

ISO/IEC 18181-1

**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

**Second edition
2024-07**

Reference number
ISO/IEC 18181-1:2024(en)

Full document details: ISO IEC 18181-1-2024

 **Documents**
© ISO/IEC 2024 Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword	iv
1 Scope	1
2 Normative references	1
3 Terms, definitions and abbreviated terms	1
3.1 Terms and definitions	1
3.2 Inputs	2
3.3 Processes	3
3.4 Image and codestream organization	4
3.5 Abbreviated terms	5
4 Conventions	5
4.1 Mathematical symbols	5
4.2 Functions	6
4.3 Operators	6
4.4 Pseudocode	7
5 Functional concepts	8
5.1 Image organization	8
5.2 Mirroring	8
5.3 Group splitting	8
5.4 Codestream organization	8
6 Encoder requirements	9
7 Decoder requirements	9
Annex A (normative) Codestream overview	10
Annex B (normative) Header syntax	11
Annex C (normative) Entropy decoding	14
Annex D (normative) Image header	20
Annex E (normative) Colour encoding	25
Annex F (normative) Frame header	34
Annex G (normative) Frame data sections	41
Annex H (normative) Modular	45
Annex I (normative) VarDCT	55
Annex J (normative) Restoration filters	70
Annex K (normative) Image features	74
Annex L (normative) Colour transforms	82
Annex M (normative) Profiles and levels	85
Annex N (normative) Extensions	87
Annex O (informative) Encoder overview	88
Bibliography	91

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).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and <https://patents.iec.ch>. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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.

This second edition cancels and replaces the first edition (ISO/IEC 18181-1:2022), which has been technically and editorially revised. It also incorporates the Amendment ISO/IEC 18181-1:2022/Amd 1:2022.

The main changes are as follows:

- technical corrections and clarifications, in particular to correct the values of various constants, correct errors in pseudocode, and clarify ambiguities in order to remove discrepancies between this document and ISO/IEC 18181-3 and ISO/IEC 18181-4;
- a thorough update of the document structure in order to improve clarity of presentation and to obtain a more logical ordering of the material from the point of view of decoder implementation.

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.

Information technology — JPEG XL image coding system —

Part 1: Core coding system

1 Scope

This document specifies 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, *Image technology colour management — Architecture, profile format and data structure — Part 2: Based on ICC.1:2022*

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*

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

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

IETF RFC 7932:2016, *Brotli Compressed Data Format*

SMPTE ST 428-1, *D-Cinema Distribution Master — Image Characteristics*

3 Terms, definitions and abbreviated terms

3.1 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>

3.1.1

bitstream

sequence of bytes

3.1.2

codestream

bitstream representing compressed image data

3.1.3

bundle

structured data consisting of one or more fields

3.1.4

field

numerical value or bundle, or an array of either

3.1.5

histogram

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

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

grid

2-dimensional array

3.2.4

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

3.2.5

channel component

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

3.2.6

rectangle

rectangular area within a channel or grid

3.2.7

width

width in samples (number of columns) of a sample grid or a rectangle

3.2.8

height

height in samples (number of rows) of a sample grid or a rectangle

3.2.9

frame

single image (possibly part of an animation, composite, or multi-page image), i.e. a 2-dimensional array of pixels

3.2.10

greyscale

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

3.2.11

opsin

photosensitive pigments in the human retina, having dynamics approximated by the XYZ colour space

3.2.12

animation

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

3.2.13

tick

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

3.2.14

composite

series of images that are superimposed

3.2.15

multi-page image

sequence of pictures to display in a paged way

3.2.16

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 an image

3.3.2

decoder

embodiment of a decoding process

3.3.3

encoding process

process which takes as its input 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

channel decorrelation

method of reducing total encoded entropy by removing correlations between channels

3.3.14

channel correlation factor

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

3.3.15

VarDCT

lossy encoding of a frame that applies DCT to varblocks

3.3.16

quantization

method of reducing the precision of (DCT) coefficients

3.3.17

quantization weight

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

3.4 Image and codestream organization

3.4.1

raster order

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

3.4.2

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

3.4.3

block

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

3.4.4

varblock

variable-size rectangle of one or more blocks

3.4.5

group

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

3.4.6

table of contents

data structure that enables seeking to a group or the next frame within a codestream

3.4.7

section

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

3.4.8

coefficient

input value to the inverse DCT

3.4.9

pass

data enabling decoding of successively more detail

3.4.10

LF group

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

3.5 Abbreviated terms

DCT	discrete cosine transform (as specified in L.7)
HF	all DCT coefficients apart from the LF coefficients, i.e. the high frequency coefficients
IDCT	inverse discrete cosine transform (as specified in L.7)
LF	lowest frequency coefficients of DCT coefficients, i.e. block averages
RGB	additive colour model with red, green, blue channels
XYB	absolute colour space based on gamma-corrected LMS (a colour space representing the response of cone cells in the human eye), in which X is derived from the difference between L and M (red-green), Y is an average of L and M (behaves similarly to luminance), and B is derived from S (blue)

4 Conventions

4.1 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 $a \leq x \leq b, c < x < d, e \leq x < f$
$\{a, b, c\}$	ordered sequence of elements
π	the smallest positive zero of the sine function (π)

4.2 Functions

<code>pow(x, y)</code>	exponentiation, x to the power of y .
<code>sqrt(x)</code>	square root, such that $\text{pow}(\text{sqrt}(x), 2) == x$ and $\text{sqrt}(x) \geq 0$. Undefined for $x < 0$.
<code>cbrt(x)</code>	cube root, such that $\text{pow}(\text{cbrt}(x), 3) == x$.
<code>cos(r)</code>	cosine of the angle r (in radians).
<code>erf(x)</code>	Gauss error function: $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
<code>log(x)</code>	natural logarithm of x . Undefined for $x \leq 0$.
<code>log2(x)</code>	base-two logarithm of x . Undefined for $x \leq 0$.
<code>floor(x)</code>	the largest integer that is less than or equal to x .
<code>ceil(x)</code>	the smallest integer that is greater than or equal to x .
<code>abs(x)</code>	absolute value of x : equal to $-x$ if $x < 0$, otherwise x .
<code>sign(x)</code>	sign of x , 0 if x is 0, +1 if x is positive, -1 if x is negative.
<code>len(a)</code>	length (number of elements) of array a .
<code>sum(a)</code>	sum of all elements of the array/tuple/sequence a .
<code>max(a)</code>	maximal element of the array/tuple/sequence a .
<code>min(a)</code>	smallest element of the array/tuple/sequence a .
<code>UnpackSigned(u)</code>	equivalent to $u/2$ if u is even, and $-(u + 1)/2$ if u is odd.
<code>clamp(x, lo, hi)</code>	equivalent to $\min(\max(\text{lo}, x), \text{hi})$.
<code>InterpretAsF16(u)</code>	the real number resulting from interpreting the unsigned 16-bit integer u as a binary16 floating-point number representation (cf. ISO/IEC 60559)
<code>InterpretAsF32(u)</code>	the real number resulting from interpreting the unsigned 32-bit integer u as a binary32 floating-point number representation (cf. ISO/IEC 60559)
<code>NarrowToI32(x)</code>	the signed 32-bit integer corresponding to the 32 least significant bits of the two's complement representation of x
<code>BitSet(u, b)</code>	the bit corresponding to b is 1 in u : true if and only if $u \& b$ (bitwise AND)

4.3 Operators

This document uses the operators defined by the C++ programming language (ISO/IEC 14882), with the following differences:

<code>/</code>	division of real numbers without truncation or rounding. Division by zero is undefined.
<code><<</code>	left shift: $x \ll s$ is defined as $x * \text{pow}(2, s)$
<code>>></code>	right shift: $x \gg s$ is defined as $\text{floor}(x / \text{pow}(2, s))$
<code>Umod</code>	remainder: $a \text{ Umod } d$ is the unique integer r in $[0, d)$ for which $a == r + q * d$ for a suitable integer q

<code>Idiv</code>	integer division: <code>a Idiv b</code> is equivalent to <code>a / b</code> , rounded towards zero to an integer value
<code>and</code>	logical AND (<code>&&</code> in C++), true if and only if both operands are true, with short-circuit evaluation
<code>or</code>	logical OR (<code> </code> in C++), false if and only if both operands are false, with short-circuit evaluation

The order of precedence for these operators is listed in [Table 1](#) 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).

Table 1 — Operator precedence

Operators	Type of operation	Associativity
<code>++x, --x</code>	prefix increment/decrement	left to right
<code>x++, x--</code>	postfix increment/decrement	left to right
<code>!, ~</code>	logical/bitwise NOT	right to left
<code>*, /, Idiv, Umod</code>	multiplication, division, integer division, remainder	left to right
<code>+, -</code>	addition and subtraction	left to right
<code><<, >></code>	left shift and right shift	left to right
<code><, >, <=, >=, ==, !=</code>	relational	left to right
<code>&</code>	bitwise AND	left to right
<code>^</code>	bitwise XOR	left to right
<code> </code>	bitwise OR	left to right
<code>and</code>	logical AND	left to right
<code>or</code>	logical OR	left to right
<code>=</code>	assignment	right to left
<code>+=, -=, *=</code>	compound assignment	right to left

4.4 Pseudocode

This document describes functionality using pseudocode formatted as follows:

```
// Informative comment
var = u(8);
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 (ISO/IEC 14882), except that:

- Symbols from [4.1](#) and functions from [4.2](#) are allowed;
- Division, remainder and exponentiation are expressed as specified in [4.3](#);
- 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 [4.3](#)) are used for those purposes.

5 Functional concepts

5.1 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` in [D.3](#)).

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.

An image is defined as a two-dimensional array of pixels, and its width is `width` and height is `height`. Unless otherwise mentioned, 'raster order' is used: 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.

A codestream may contain multiple image frames. These can constitute an animation (a sequence of images) or a composite still image. Frames can have dimensions that differ from the image dimensions, and they can be blended over preceding frames. The frame that is being decoded is referred to as the 'current' frame.

5.2 Mirroring

Some operations access samples with coordinates `cx`, `cy` that are outside the current frame bounds. The decoder redirects such accesses to a valid sample at the coordinates `Mirror(cx, cy)` as 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;
}

Mirror(x, y) {
    return (Mirror1D(x, width), Mirror1D(y, height));
}
```

5.3 Group splitting

Channels are logically partitioned into naturally-aligned groups of `group_dim × group_dim` samples. The effective dimension of a group (i.e. how many pixels to read) can be smaller than `group_dim` for groups on the right or bottom of the image. The decoder ensures the decoded image has the dimensions specified in `SizeHeader` ([D.2](#)) by cropping at the right and bottom as necessary. Unless otherwise specified, `group_dim` is 256.

LF groups likewise consist of `group_dim × group_dim` 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' ([E.3](#)) stores the size (in bytes) of each group to allow seeking to any group. An optional permutation allows arbitrary reordering of groups within the codestream.

5.4 Codestream organization

A bitstream is a sequence of bytes. A codestream is a bitstream that represent compressed image data and metadata. `N` bytes can also be viewed as $8 \times N$ 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 [B.2.1](#).

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

6 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 O](#) provides an informative description of such an encoding process.

7 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 [N](#) define an output that alternative implementations shall duplicate.

Annex A (normative)

Codestream overview

A.1 Codestream structure

The codestream consists of image headers ([Annexes D](#) and [N](#)) and an ICC profile, if present ([E.4](#)), followed by one or more frames ([Annex F](#)), as shown in [Table A.1](#). This table and those it references, together with the syntax description ([B.1](#)), specify how a decoder reads the headers and frame(s).

Table A.1 — Codestream structure

condition	type	name	Annex
	Headers	headers	Annex D , Annex N
<code>headers.metadata.colour_encoding.want_icc</code>		icc	E.4
<code>headers.metadata.have_preview</code>	Frame	preview_frame	Annex F
	Frame	frames[0]	Annex F
<code>for (i = 1; !frames[i - 1].frame_header.is_last; ++i)</code>	Frame	frames[i]	Annex F

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

A.2 Decoding process

[Annex B](#) specifies how the decoder parses header information.

[Annex C](#) specifies how the decoder reads entropy-coded data.

After reading the image header ([Annexes D](#) and [E](#)) and the frame header ([Annex F](#)), the decoder reads the frame data ([Annex G](#)). The reconstruction of the decoded image can be viewed as a pipeline with multiple stages.

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 H](#) specifies the **kModular** frame encoding and Modular sub-bitstreams.

[Annex I](#) specifies the **kVarDCT** frame encoding.

The decoder applies 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 colour transforms as specified in [Annex L](#).

NOTE For an introduction to the coding tools and additional background, please refer to Reference [\[3\]](#).

Annex B (normative)

Header syntax

B.1 General

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 [B.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 B.1](#), with one row per field (in top to bottom order).

Table B.1 — Example structure of a table describing a bundle

condition	type	default	name
	Bool()	false	div8
div8	1 + u(5)	0	h_div8
!div8	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	8 * h_div8	height
	u(3)	0	ratio
div8 and !ratio	1 + u(5)	0	w_div8
!div8 and !ratio	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	d_width	width

If the condition is blank or evaluates to `true`, the field with the name in the **name** column is read from the codestream ([B.1.1](#)) according to the field type in the **type** column. Otherwise, the field is instead initialized to default value in the **default** column ([B.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 ([B.1.1](#)) in order of increasing index. Otherwise, each element is initialized to default ([B.1.2](#)). The name is potentially referenced in the condition or type of a subsequent row, or the condition of the same row.

B.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 [B.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.

B.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, the field is set to default, which is a valid value of the same type.

B.2 Field types

B.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 (more significant) 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.

B.2.2 $U32(d0, d1, d2, d3)$

In this subclause, “distribution” refers to one of the following three encodings of a range of values: u , $u(n)$, or $\text{offset} + u(n)$. The $d0, d1, d2, d3$ parameters represent distributions.

$U32(d0, d1, d2, d3)$ reads an unsigned 32-bit integer 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 u , value is the integer u . If d is $u(n)$, value is read as $u(n)$. If d is $\text{offset} + u(n)$, the decoder reads $v = u(n)$. The resulting value is $(\text{offset} + v) \text{ Umod } (1 \ll 32)$.

NOTE The value of u is implicitly defined and not stored explicitly in the codestream.

EXAMPLE For a field of type $U32(8, 16, 32, u(7))$, the bits 10 result in $\text{value} = 32$. For a $U32(u(2), u(4), u(6), u(8))$ field, the bits 010111 result in $\text{value} = 7$.

B.2.3 $U64()$

$U64()$ reads an unsigned 64-bit integer 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 $1 + u(4)$. If $s == 2$, value is $17 + u(8)$. Otherwise ($s == 3$), value is read 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.

B.2.4 $F16()$

$F16()$ reads a real value in $[-65504, 65504]$, 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.

B.2.5 $Bool()$

$Bool()$ reads a boolean value as $u(1) ? \text{true} : \text{false}$.

B.2.6 Enum(EnumTable)

Enum(EnumTable) reads $v = U32(0, 1, 2 + u(4), 18 + u(6))$. The value v does not exceed 63 and is defined by the table titled EnumTable. Such tables are structured according to [Table B.2](#), with one row per unique value.

Table B.2 — Example structure of a table describing an enumerated type

name	value	meaning
kD65	1	CIE Standard Illuminant D65: 0.3127, 0.3290
kCustom	2	Custom white point stored in <code>colour_encoding.white</code>
kE	10	CIE Standard Illuminant E (equal-energy): 1/3, 1/3
kDCI	11	DCI-P3 from SMPTE ST 428-1: 0.314, 0.351

An enumerated type is interpreted as having the meaning of the row with the corresponding value. The value in the **name** column (or where ambiguous, EnumTable.name) is an identifier used to refer to a particular enumerated value, and begins with the letter **k**, e.g., **kRGB**.

B.2.7 ZeroPadToByte()

The decoder skips to the next byte boundary if not already at a byte boundary. All skipped bits have value 0.

NOTE Since any padding bits are zero, they can serve as an additional indicator of codestream integrity.

B.3 Extensions

This bundle, specified in [Table B.3](#), is a field in bundles (ImageMetadata, FrameHeader, RestorationFilter) which can be extended (cf. [Annex N](#)).

Table B.3 — 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 N](#). Each extension is identified by an `ext_id` 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_id = 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_id`. `extension_bits[i]` indicates the number of bits stored for the extension whose `ext_id = i`, starting after `extension_bits` has been read. The decoder reads all these bits for all extensions which are present.

Annex C (normative)

Entropy decoding

C.1 Overview

This Annex specifies an entropy decoder. Other Annexes and their subclauses indicate how it is used for various elements of the codestream.

The codestream contains multiple independently entropy-coded streams. During decoding of such a stream, the decoder maintains an entropy coder state, which is initialized as specified in [C.2](#). This state has to persist and can be updated during the decoding of the entropy-coded stream, as specified in [C.3](#).

The following variables (which are specified in the following clauses) are part of the entropy decoder state:

- `num_dist`, the pre-clustering number of distributions (context identifiers), which is given;
- `num_clusters`, the post-clustering number of distributions;
- `lz77`, a bundle with LZ77 settings;
- `clusters[num_dist]`, describing the context clustering;
- `configs[num_clusters]`, hybrid integer configurations;
- `use_prefix_code`, a Boolean flag indicating if prefix coding or ANS is used;
- `lz_len_conf`, a hybrid integer configuration used to decode LZ77 lengths;
- `window[1 << 20]`, the LZ77 window of previously-decoded symbols (initialized to all zeroes);
- `num_to_copy`, `copy_pos`, and `num_decoded`: counters used for copying symbols from the LZ77 window;
- If `use_prefix_code`: a set of `num_clusters` post-clustered prefix codes;
- If `!use_prefix_code`, i.e. when the stream uses ANS:
 - `log_alphabet_size`, indicating the alphabet size of the entropy-coded ANS symbols;
 - a set of `num_clusters` post-clustered probability distributions $D[1 << \log_alphabet_size]$;
 - the alias mapping for each post-clustered distribution (`log_bucket_size`, `bucket_size`, `symbols`, `offsets`, and `cutoffs`);
 - `state`, a 32-bit unsigned integer that represents the ANS state.

NOTE As an implementation optimization, the variables `lz_len_conf`, `window`, `num_to_copy`, `copy_pos`, and `num_decoded` can be ignored in case `lz77.enabled` is false.

C.2 Distribution decoding

C.2.1 General

This subclause describes how to decode the probability distributions and other stream initialization information that is needed before the actual entropy decoding can start. When this Annex is referenced, `num_dist` is given; it denotes the number of pre-clustered probability distributions.

The decoder first reads the LZ77 settings `lz77` as specified in [Table C.1](#).

Table C.1 — LZ77Params bundle

condition	type	name
	Bool()	enabled
enabled	U32(224, 512, 4096, 8 + u(15))	min_symbol
enabled	U32(3, 4, 5 + u(2), 9 + u(8))	min_length

If `lz77.enabled`, the decoder sets `lz_dist_ctx = num_dist++`, and reads `lz_len_conf = HybridUintConfig(8)` as specified in [C.2.3](#).

The decoder then reads the mapping `clusters` of the `num_dist` distributions into `num_clusters` clusters as specified in [C.2.2](#). It then reads `use_prefix_code` as a Bool(). If `use_prefix_code` is false, the decoder sets `log_alphabet_size` to 5 + u(2); otherwise, it sets `log_alphabet_size` to 15. Then for each post-clustered distribution, in increasing order of index `i` in `[0, num_clusters)`, the decoder reads `configs[i] = HybridUintConfig(log_alphabet_size)` as specified in [C.2.3](#).

The decoder then reads `num_clusters` post-clustered distributions `D[i]` as follows. If `use_prefix_code` is true, then for `i` in `[0, num_clusters)`: if Bool() is false, `count[i] = 1`, otherwise `n = u(4)` is read, and `count[i] = 1 + (1 << n) + u(n)`, which is at most `1 << 15`. After reading the counts, the decoder reads each `D[i]` (implicitly described by a prefix code) as specified in [C.2.4](#), with `alphabet_size = count[i]`. If `use_prefix_code` is false, the distributions are read as specified in [C.2.5](#).

C.2.2 Distribution clustering

The probability distributions that symbols belong to can be clustered together. This subclause specifies how this clustering is read by the decoder.

The output of this procedure is an array `clusters` with `num_dist` entries (one for each pre-clustered context), with values in the range `[0, num_clusters)`. All integers in `[0, num_clusters)` are present in this array, and `num_clusters < 256`. Position `i` in this array indicates that context `i` is merged into the corresponding cluster.

If `num_dist == 1`, then `num_clusters = 1` and `clusters[0] = 0`, and the remainder of this subclause is skipped.

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

Otherwise, if `is_simple` is false, the decoder reads a Bool() `use_mtf`. The decoder then initializes a symbol decoder using a single distribution `D`, as specified in [C.2.1](#), where if `num_dist == 2` then in the recursive distribution decoding, `lz77.enabled` is false. For each pre-clustered distribution `i`, the decoder reads an integer as specified in [C.3.3](#). 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_dist]) {
  for (i = 0; i < 256; ++i) mtf[i] = i;
  for (i = 0; i < num_dist; ++i) {
    index = clusters[i]; /* index < 256 */
    clusters[i] = mtf[index];
    if (index != 0) MTF(mtf, index);
  }
}
```

C.2.3 Hybrid integer configuration

The configuration for the hybrid unsigned integer decoder of [C.3.3](#) is read from the bitstream as follows:

```

ReadUintConfig(log_alphabet_size) {
    config.split_exponent = u(ceil(log2(log_alphabet_size + 1)));
    if (config.split_exponent != log_alphabet_size) {
        nbits = ceil(log2(config.split_exponent + 1));
        config.msb_in_token = u(nbits);
        /* config.msb_in_token is <= config.split_exponent */
        nbits = ceil(log2(config.split_exponent - config.msb_in_token + 1));
        config.lsb_in_token = u(nbits);
    } else { config.msb_in_token = 0; config.lsb_in_token = 0; }
    /* config.lsb_in_token + config.msb_in_token is <= config.split_exponent */
    config.split = 1 << config.split_exponent;
    return config;
}

```

C.2.4 Histogram and prefix code

A Huffman histogram stream stores metadata required for decoding a Huffman symbol stream. See IETF RFC 7932:2016 sections 3.4 (“Simple Prefix Codes”) and 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, except in the special case of `alphabet_size == 1`, where no histogram is read, and all decoded symbols are zero without reading any bits at all.

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 a Huffman histogram.

C.2.5 ANS distribution decoding

ANS probabilities distributions are represented as an array with $1 << \log_alphabet_size$ elements with values in $[0, 1 << 12]$, one per symbol, that sum to $1 << 12$, i.e. they can be interpreted as representing symbol probabilities with an implicit denominator of $1 << 12$. Indexing the array with an out-of-bounds value results in a 0 value.

To decode a distribution D, the decoder follows the following procedure (where ‘read a number according to a prefix code’ refers to [C.2.4](#)):

```

U8() { // reads an integer value in the range [0, 256) using [1, 12) bits
    if (u(1) == 0) return 0;
    else { n = u(3); return u(n) + (1 << n); }
}

table_size = 1 << log_alphabet_size;
/* initialize D as array with table_size entries with value 0 */
if (Bool()) {
    if (Bool()) { v1 = U8(); v2 = U8(); /* v1 != v2 */
        D[v1] = u(12); D[v2] = (1 << 12) - D[v1];
        alphabet_size = 1 + max(v1, v2);
    } else { x = U8(); D[x] = 1 << 12; alphabet_size = 1 + x; }
    return;
}
if (Bool()) {
    alphabet_size = U8() + 1;
    for (i = 0; i < alphabet_size; i++) D[i] = (1 << 12) / alphabet_size;
    for (i = 0; i < ((1 << 12) / alphabet_size); i++) D[i]++;
    return;
}
len = 0; while (len < 3) { if (Bool()) len++; else break; }
shift = u(len) + (1 << len) - 1;
/* shift <= 13 */
alphabet_size = U8() + 3;

omit_log = -1; omit_pos = -1;
/* initialize same as array with alphabet_size entries with value 0 */
for (i = 0; i < alphabet_size; i++) {
    logcounts[i] = /* read a number according to the following prefix code:
        Number   Code       Number   Code (bits parsed from right to left)
        0         10001      7         010
    */
}

```

```

1      1011      8      101
2      1111      9      110
3      0011     10      000
4      1001     11     100001
5      0111     12     0000001
6      100      13     1000001
*/
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 */

```

C.2.6 Alias mapping

For a given probability distribution D of symbols in the range $[0, 1 \ll \log_alphabet_size)$, 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;
table_size = 1 << log_alphabet_size;
if (/* amount of symbols with nonzero probability in D */ == 1) {
    s = /* index of the symbol with nonzero probability */;
    for (i = 0; i < table_size; i++) {
        symbols.push_back(s); offsets.push_back(bucket_size * i); cutoffs.push_back(0);
    }
    return;
}
for (i = 0; i < alphabet_size; i++) {
    cutoffs.push_back(D[i]); symbols.push_back(i);
    if (cutoffs[i] > bucket_size) overfull.push_back(i);
    else if (cutoffs[i] < bucket_size) underfull.push_back(i);
}
for (i = alphabet_size; i < table_size; i++) {
    cutoffs[i] = 0;
    underfull.push_back(i);
}
while (overfull.size() > 0) {
    /* underfull.size() > 0 */
    o = overfull.back(); u = underfull.back(); overfull.pop_back(); underfull.pop_back();
    by = bucket_size - cutoffs[u];
    cutoffs[o] -= by; symbols[u] = o; offsets[u] = cutoffs[o];
    if (cutoffs[o] < bucket_size) underfull.push_back(o);
    else if (cutoffs[o] > bucket_size) 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 $[0, 1 \ll \log_alphabet_size)$ and an offset in $[0, 1 \ll 12)$.

C.3 Symbol decoding

The decoder reads integer symbols from an entropy coded stream as specified in [C.3.3](#). If `use_prefix_code`, entropy coded symbols are read according to [C.2.4](#), using the corresponding prefix code. Otherwise, they are read according to [C.3.2](#). In both cases the final decoded integers are reconstructed using a combination of context-dependent entropy decoding (using a prefix code or ANS), raw bits (read directly from the same stream), and possibly LZ77 (copying of previously decoded symbol sequences).

The ANS decoder keeps an internal 32-bit unsigned integer `state`. Upon initialization of a new ANS stream, the initial value of `state` is read as a `u(32)` from the codestream. At the end of an ANS stream, `state` is `0x130000`. There is only a single `state` variable per ANS stream; it is shared between different contexts.

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

An unsigned integer is read from an entropy coded stream given a pre-clustered context identifier `ctx` as specified by `DecodeHybridVarLenUInt` in the following code. `num_to_copy` is initialized to 0 and the array `window[]` is initialized to all zeroes when the first symbol is decoded from the stream. `dist_multiplier` is 0 unless otherwise specified.

```

ReadUInt(config, token) {
    if (token < config.split) return token;
    n = config.split_exponent - config.msb_in_token - config.lsb_in_token +
        ((token - config.split) >> (config.msb_in_token + config.lsb_in_token));
    /* n < 32 */
    low = 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);
    result = (((token << n) | u(n)) << config.lsb_in_token) | low;
    /* result < (1 << 32) */
    return result;
}

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[clusters[ctx]] */;

```

```

if (lz77.enabled and token >= lz77.min_symbol) {
    num_to_copy = ReadUint(lz_len_conf, token - lz77.min_symbol) + lz77.min_length;
    token = /* Read symbol using D[clusters[lz_dist_ctx]] */;
    distance = ReadUint(configs[clusters[lz_dist_ctx]], token);
    if (dist_multiplier == 0) distance++;
    else if (distance < 120) {
        offset = kSpecialDistances[distance][0];
        distance = offset + dist_multiplier * kSpecialDistances[distance][1];
        if (distance < 1) distance = 1;
    } else distance -= 119;
    distance = min(distance, num_decoded, 1 << 20);
    copy_pos = num_decoded - distance;
    return DecodeHybridVarLenUint(clusters[ctx]);
}
r = ReadUint(configs[clusters[ctx]], token);
}
window[(num_decoded++) & 0xFFFFF] = r;
return r;
}

```

NOTE It is not necessarily the case that `num_to_copy` is zero at the end of the stream.

Annex D (normative)

Image header

D.1 General

The decoder reads the Headers bundle ([Table D.1](#)) as specified in [B.1](#).

Table D.1 — Headers bundle

condition	type	name	subclause
	u(16)	signature	
	SizeHeader	size	D.2
	ImageMetadata	metadata	D.3

The `signature` consists of the fixed two-byte sequence `0xFF0A`, i.e. it is equal to the value 2815.

D.2 Image dimensions

[Table D.2](#) specifies the SizeHeader bundle.

Table D.2 — SizeHeader bundle

condition	type	default	name
	Bool()	false	div8
div8	1 + u(5)	0	h_div8
!div8	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	8 * h_div8	height
	u(3)	0	ratio
div8 and !ratio	1 + u(5)	0	w_div8
!div8 and !ratio	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	d_width	width

The default value `d_width` is defined as $(ratio == 0 ? 8 * w_div8 : AspectRatio(height, ratio))$, where `AspectRatio` is computed as follows:

```
AspectRatio(height, ratio) {
  if (ratio == 1) return height;
  if (ratio == 2) return height * 6 Idiv 5;
  if (ratio == 3) return height * 4 Idiv 3;
  if (ratio == 4) return height * 3 Idiv 2;
  if (ratio == 5) return height * 16 Idiv 9;
  if (ratio == 6) return height * 5 Idiv 4;
  if (ratio == 7) return height * 2;
}
```

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

D.3 Image metadata

D.3.1 ImageMetadata

The ImageMetadata bundle, defined in [Table D.3](#), holds information that applies to all Frames (including the preview); decoders use it to interpret pixel values and restore the original image.

Table D.3 — ImageMetadata bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	Bool()	false	extra_fields
extra_fields	1 + u(3)	1	orientation
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(0, 1, 2 + u(4), 1 + u(12))	0	num_extra
!all_default	ExtraChannelInfo		ec_info[num_extra]
!all_default	Bool()	true	xyb_encoded
!all_default	ColourEncoding		colour_encoding
extra_fields	ToneMapping		tone_mapping
!all_default	Extensions		extensions
	Bool()		default_m
!default_m and xyb_encoded	OpsinInverseMatrix		opsin_inverse_matrix
!default_m	u(3)	0	cw_mask
BitSet(cw_mask, 1)	F16()	d_up2	up2_weight[15]
BitSet(cw_mask, 2)	F16()	d_up4	up4_weight[55]
BitSet(cw_mask, 4)	F16()	d_up8	up8_weight[210]

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_frame`.

`have_animation` indicates whether an `AnimationHeader` is included (animation) and whether `FrameHeader` (F.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 Annex H). 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` is the number of additional image channels. If it is nonzero, then `ec_info` signals the semantics of each extra channel.

`xyb_encoded` is true if the stored image is in the XYZ colour space. In L.2 it is specified how to convert XYZ 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 XYZ 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. This bundle is specified in Annex E.

`tone_mapping` (Table E.9) has information for mapping HDR images to lower dynamic range displays. This bundle is specified in E.3.

OpsinInverseMatrix is defined in [L.2.1](#).

For the upsampling filters, custom weights are signalled depending on `cw_mask`. The default weights `d_up2`, `d_up4` and `d_up8` are defined in [K.2](#).

D.3.2 Orientation

The `orientation` indicates which orientation transform the decoder applies after decoding the image ([Table D.4](#)), including the preview frame if present.

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

Table D.4 — Orientation

orientation	first row	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

D.3.3 PreviewHeader

[Table D.5](#) specifies the PreviewHeader bundle.

Table D.5 — PreviewHeader bundle

condition	type	default	name
	Bool()		div8
div8	U32(16, 32, 1 + u(5), 33 + u(9))	1	h_div8
!div8	U32(1 + u(6), 65 + u(8), 321 + u(10), 1345 + u(12))	8 * h_div8	height
	u(3)		ratio
div8 and !ratio	U32(16, 32, 1 + u(5), 33 + u(9))	1	w_div8
!div8 and !ratio	U32(1 + u(6), 65 + u(8), 321 + u(10), 1345 + u(12))	d_width	width

The default value `d_width` is defined as in SizeHeader: `d_width = (ratio == 0 ? 8 * w_div8 : AspectRatio(height, ratio))`.

The preview `width` and `height` do not exceed 4096.

D.3.4 AnimationHeader

AnimationHeader, defined in [Table D.6](#), holds meta-information about an animation or image sequence that is present if `metadata.have_animation`.

Table D.6 — AnimationHeader bundle

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

`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.

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.

D.3.5 BitDepth

[Table D.7](#) specifies the BitDepth bundle.

Table D.7 — BitDepth bundle

condition	type	default	name
	Bool()	false	float_sample
<code>!float_sample</code>	U32(8, 10, 12, 1 + u(6))	8	bits_per_sample
<code>float_sample</code>	U32(32, 16, 24, 1 + u(6))	8	bits_per_sample
<code>float_sample</code>	1 + u(4)	0	exp_bits

This information describes how to interpret Modular-encoded integer samples, for extra channels and for the colour channels if `!metadata.xyb_encoded`, as specified in [G.4.2](#). For colour channels encoded with VarDCT and/or with `metadata.xyb_encoded`, this information is merely a suggestion for a sample representation that can be suitable for the image, but it does not impact the actual sample values.

`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 `float_sample`: for integers the value is in range [1, 31], for floating point the value is in range [5, 32].

If `float_sample` is false, then integer samples of the decoded image are divided by $(1 \ll \text{bits_per_sample}) - 1$.

If `float_sample` is true, then `exp_bits` is the number of bits used for the exponent of floating point sample values. Let `mantissa_bits = bits_per_sample - exp_bits - 1`. Integer samples of the decoded 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 ISO/IEC 60559 with an exponent bias of $(1 \ll (\text{exp_bits} - 1)) - 1$. The value of `exp_bits` is in range [2, 8], and the value of `mantissa_bits` is in range [2, 23]. If a sample's exponent has the highest representable value, the decoded value of this sample is unspecified.

NOTE 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.

D.3.6 ExtraChannelInfo

[Table D.8](#) specifies the ExtraChannelInfo bundle.

Table D.8 — ExtraChannelInfo bundle

condition	type	default	name
	Bool()	true	d_alpha
<code>!d_alpha</code>	Enum(ExtraChannelType)	kAlpha	type
<code>!d_alpha</code>	BitDepth		bit_depth
<code>!d_alpha</code>	U32(0, 3, 4, 1 + u(3))	0	dim_shift
<code>!d_alpha</code>	U32(0, u(4), 16 + u(5), 48 + u(10))	0	name_len
	u(8)	0	name[name_len]
<code>!d_alpha and type == kAlpha</code>	Bool()	false	alpha_associated

Table D.8 (continued)

condition	type	default	name
<code>type == kSpotColour</code>	F16()	0	red
<code>type == kSpotColour</code>	F16()	0	green
<code>type == kSpotColour</code>	F16()	0	blue
<code>type == kSpotColour</code>	F16()	0	solidity
<code>type == kCFA</code>	U32(1, u(2), 3 + u(4), 19 + u(8))	1	cfa_channel

Table D.9 specifies the meaning of the ExtraChannelType values.

Table D.9 — ExtraChannelType

name	value	meaning
kAlpha	0	Alpha transparency, where 0 means fully transparent
kDepth	1	Depth map. Higher values mean farther away from the camera.
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. Higher values mean warmer temperature.
kNonOptional	15	The decoder indicates it cannot safely interpret the semantics of this extra channel.
kOptional	16	Extra channel that can be safely ignored.

Extra channels are interpreted according to their type and are rendered as specified in L.4.

`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 $1 \ll \text{dim_shift}$ does not exceed the `group_dim` of any frame (F.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 E (normative)

Colour encoding

E.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 XYZ colour space, or as RGB triplets, YCbCr triplets or greyscale values which are to be interpreted according to the colour space described by a given colour profile.

NOTE Both ways are useful in different scenarios:

- XYZ 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 XYZ scenario, the signalled colour space does not give any information about how to interpret the encoded XYZ 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 XYZ 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 XYZ, see [F.2](#)).
- In the other scenario, the signalled colour space has direct meaning: it is the colour space of the RGB, YCbCr or greyscale 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 ([D.3](#)) indicates which of the two scenarios is followed. In the case where `metadata.xyb_encoded` is true, then `metadata.colour_encoding`, `metadata.bit_depth`, and the ICC profile described in this Annex (if present) are merely suggestions as to a colour encoding and sample representation 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 false, in which case it is not merely a suggestion but the colour space in which blending is performed, see [F.2](#)). If `xyb_encoded` is false, the decoded samples are to be interpreted according to `metadata.bit_depth` and the colour space defined 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 [E.2](#).
- Otherwise, the decoded samples are indicated to be interpreted according to an ICC profile decoded as specified in [E.4](#).