Authors: J. Zern   P. Massimino   J. Alakuijala
            Google LLC   *Google LLC*   *Google LLC*

RFC 9649
WebP Image Format

Abstract

This document defines the WebP image format and registers a media type supporting its use.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

WebP is an image file format based on the Resource Interchange File Format (RIFF) [RIFF-spec] (Section 2) that supports lossless and lossy compression as well as alpha (transparency) and animation. It covers use cases similar to JPEG [JPEG-spec], PNG [RFC2083], and the Graphics Interchange Format (GIF) [GIF-spec].

WebP consists of two compression algorithms used to reduce the size of image pixel data, including alpha (transparency) information. Lossy compression is achieved using VP8 intra-frame encoding [RFC6386]. The lossless algorithm (Section 3) stores and restores the pixel values exactly, including the color values for fully transparent pixels. A universal algorithm for sequential data compression [LZ77], prefix coding [Huffman], and a color cache are used for compression of the bulk data.

2. WebP Container Specification

> Note that this section is based on the documentation in the libwebp source repository [webp-riff-src].

2.1. Introduction (from "WebP Container Specification")

WebP is an image format that uses either (i) the VP8 intra-frame encoding [RFC6386] to compress image data in a lossy way or (ii) the WebP lossless encoding (Section 3). These encoding schemes should make it more efficient than older formats, such as JPEG, GIF, and PNG. It is optimized for fast image transfer over the network (for example, for websites). The WebP format has feature parity (color profile, metadata, animation, etc.) with other formats as well. This section describes the structure of a WebP file.

The WebP container (that is, the RIFF container for WebP) allows feature support over and above the basic use case of WebP (that is, a file containing a single image encoded as a VP8 key frame). The WebP container provides additional support for the following:

- Lossless Compression: An image can be losslessly compressed, using the WebP lossless format.
- Metadata: An image may have metadata stored in Exchangeable Image File Format [Exif] or Extensible Metadata Platform [XMP] format.
- Transparency: An image may have transparency, that is, an alpha channel.
- Color Profile: An image may have an embedded ICC profile (ICCP) [ICC].
- Animation: An image may have multiple frames with pauses between them, making it an animation.

2.2. Terminology & Basics

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

A WebP file contains either a still image (that is, an encoded matrix of pixels) or an animation (Section 2.7.1.1). Optionally, it can also contain transparency information, a color profile, and metadata. We refer to the matrix of pixels as the *canvas* of the image.

Bit numbering in chunk diagrams starts at 0 for the most significant bit ('MSB 0'), as described in [RFC1166].

Below are additional terms used throughout this section:

Reader/Writer
> Code that reads WebP files is referred to as a *reader*, while code that writes them is referred to as a *writer*.

uint16
> A 16-bit, little-endian, unsigned integer.

uint24
> A 24-bit, little-endian, unsigned integer.

uint32
> A 32-bit, little-endian, unsigned integer.

FourCC
> A four-character code (FourCC) is a uint32 created by concatenating four ASCII characters in little-endian order. This means 'aaaa' (0x61616161) and 'AAAA' (0x41414141) are treated as different FourCCs.

1-based
> An unsigned integer field storing values offset by -1, for example, such a field would store value *25* as *24*.

ChunkHeader('ABCD')
> Used to describe the *FourCC* and *Chunk Size* header of individual chunks, where 'ABCD' is the FourCC for the chunk. This element's size is 8 bytes.

2.3. RIFF File Format

The WebP file format is based on the RIFF [RIFF-spec] document format.

The basic element of a RIFF file is a *chunk*. It consists of:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Chunk FourCC                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Chunk Size                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         Chunk Payload                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
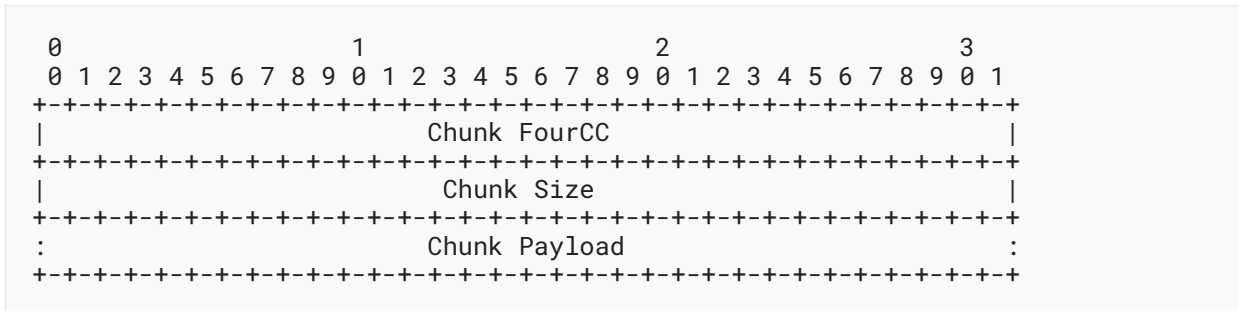
Figure 1: 'RIFF' Chunk Structure

Chunk FourCC: 32 bits
    ASCII four-character code used for chunk identification.

Chunk Size: 32 bits (*uint32*)
    The size of the chunk in bytes, not including this field, the chunk identifier, or padding.

Chunk Payload: *Chunk Size* bytes
    The data payload. If *Chunk Size* is odd, a single padding byte -- which **MUST** be 0 to conform
    with RIFF [RIFF-spec] -- is added.

> Note: RIFF has a convention that all uppercase chunk FourCCs are standard chunks
> that apply to any RIFF file format, while FourCCs specific to a file format are all
> lowercase. WebP does not follow this convention.

2.4. WebP File Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'R'      |      'I'      |      'F'      |      'F'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          File Size                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'W'      |      'E'      |      'B'      |      'P'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: WebP File Header Chunk

'RIFF': 32 bits
    The ASCII characters 'R', 'I', 'F', 'F'.

File Size: 32 bits (*uint32*)
> The size of the file in bytes, starting at offset 8. The maximum value of this field is 2^{32} minus 10 bytes, and thus the size of the whole file is at most 4 GiB minus 2 bytes.

'WEBP': 32 bits
> The ASCII characters 'W', 'E', 'B', 'P'.

A WebP file **MUST** begin with a RIFF header with the FourCC 'WEBP'. The file size in the header is the total size of the chunks that follow plus 4 bytes for the 'WEBP' FourCC. The file **SHOULD NOT** contain any data after the data specified by *File Size*. Readers **MAY** parse such files, ignoring the trailing data. As the size of any chunk is even, the size given by the RIFF header is also even. The contents of individual chunks are described in the following sections.

2.5. Simple File Format (Lossy)

This layout **SHOULD** be used if the image requires lossy encoding and does not require transparency or other advanced features provided by the extended format. Files with this layout are smaller and supported by older software.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                  WebP file header (12 bytes)                  |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                         'VP8 ' Chunk                          :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
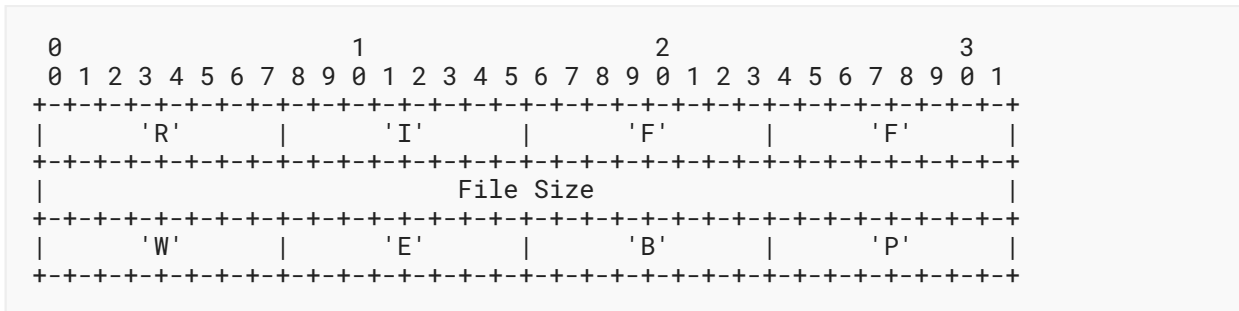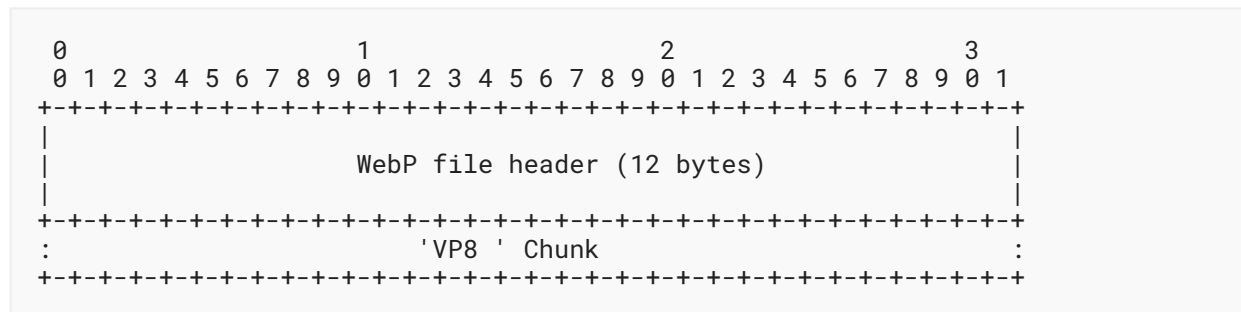
Figure 3: Simple WebP (Lossy) File Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       ChunkHeader('VP8 ')                     |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           VP8 data                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

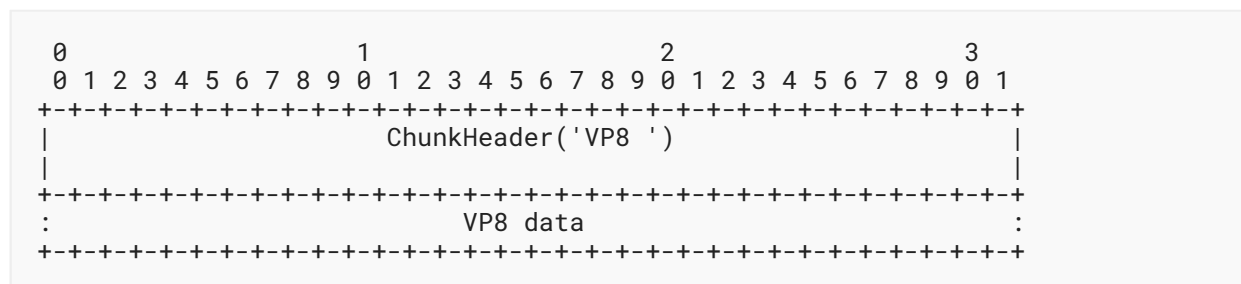*Figure 4: 'VP8 ' Chunk*

VP8 data: *Chunk Size* bytes
> VP8 bitstream data.

> > Note that the fourth character in the 'VP8 ' FourCC is an ASCII space (0x20).

The VP8 bitstream format specification is described in [RFC6386].

> Note that the VP8 frame header contains the VP8 frame width and height. That is assumed to be the width and height of the canvas.

The VP8 specification describes how to decode the image into Y'CbCr format. To convert to RGB, Recommendation 601 [rec601] **SHOULD** be used. Applications **MAY** use another conversion method, but visual results may differ among decoders.

2.6. Simple File Format (Lossless)

> Note: Older readers may not support files using the lossless format.

This layout **SHOULD** be used if the image requires lossless encoding (with an optional transparency channel) and does not require advanced features provided by the extended format.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                    WebP file header (12 bytes)                |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                          'VP8L' Chunk                          :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
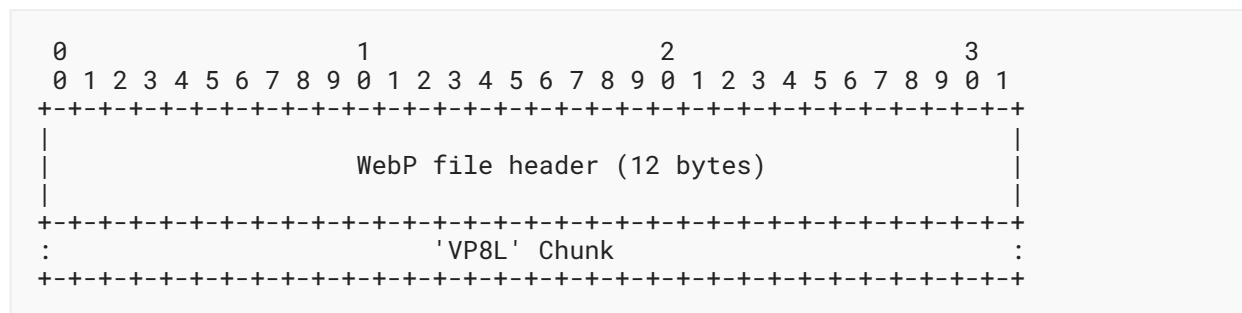
Figure 5: Simple WebP (Lossless) File Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('VP8L')                      |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                           VP8L data                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
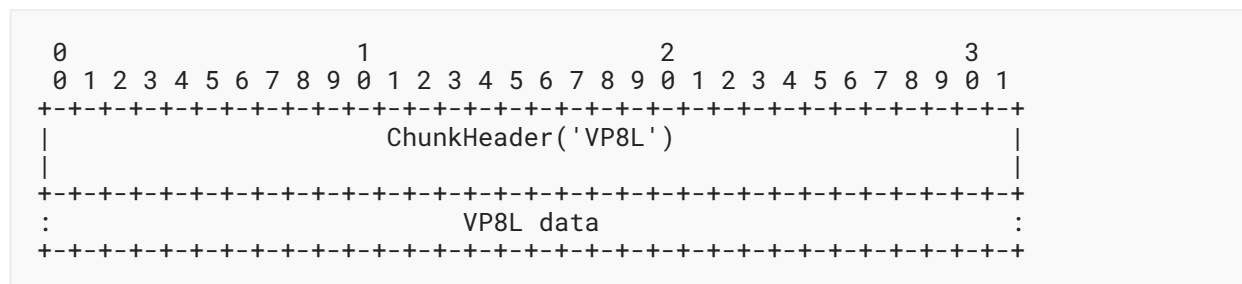
Figure 6: 'VP8L' Chunk

VP8L data: *Chunk Size* bytes
    VP8L bitstream data.

The specification of the VP8L bitstream can be found in Section 3.

> Note that the VP8L header contains the VP8L image width and height. That is assumed to be the width and height of the canvas.

2.7. Extended File Format

> Note: Older readers may not support files using the extended format.

An extended format file consists of:

- A 'VP8X' Chunk with information about features used in the file.
- An optional 'ICCP' Chunk with a color profile.
- An optional 'ANIM' Chunk with animation control data.
- Image data.
- An optional 'EXIF' Chunk with Exif metadata.
- An optional 'XMP ' Chunk with XMP metadata.
- An optional list of unknown chunks (Section 2.7.1.6).

For a *still image*, the *image data* consists of a single frame, which is made up of:

- An optional alpha subchunk (Section 2.7.1.2).
- A bitstream subchunk (Section 2.7.1.3).

For an *animated image*, the *image data* consists of multiple frames. More details about frames can be found in Section 2.7.1.1.

All chunks necessary for reconstruction and color correction, that is, 'VP8X', 'ICCP', 'ANIM', 'ANMF', 'ALPH', 'VP8 ', and 'VP8L', **MUST** appear in the order described earlier. Readers **SHOULD** fail when chunks necessary for reconstruction and color correction are out of order.

Metadata (Section 2.7.1.5) and unknown chunks (Section 2.7.1.6) MAY appear out of order.

> Rationale: The chunks necessary for reconstruction should appear first in the file to allow a reader to begin decoding an image before receiving all of the data. An application may benefit from varying the order of metadata and custom chunks to suit the implementation.
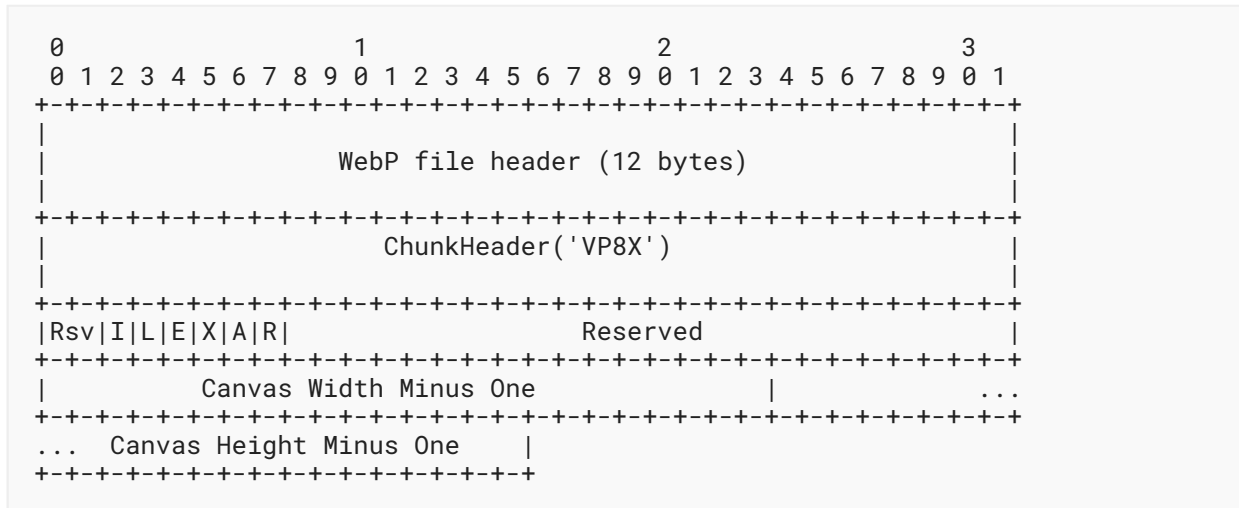
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                   WebP file header (12 bytes)                 |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     ChunkHeader('VP8X')                       |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Rsv|I|L|E|X|A|R|                Reserved                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Canvas Width Minus One        |                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
...  Canvas Height Minus One    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: Extended WebP File Header

Reserved (Rsv): 2 bits
    MUST be 0. Readers **MUST** ignore this field.

ICC profile (I): 1 bit
    Set if the file contains an 'ICCP' Chunk.

Alpha (L): 1 bit
    Set if any of the frames of the image contain transparency information ("alpha").

Exif metadata (E): 1 bit
    Set if the file contains Exif metadata.

XMP metadata (X): 1 bit
    Set if the file contains XMP metadata.

Animation (A): 1 bit
    Set if this is an animated image. Data in 'ANIM' and 'ANMF' Chunks should be used to control the animation.

Reserved (R): 1 bit
    MUST be 0. Readers **MUST** ignore this field.

Reserved: 24 bits
    MUST be 0. Readers **MUST** ignore this field.

Canvas Width Minus One: 24 bits
    1-based width of the canvas in pixels. The actual canvas width is 1 + Canvas Width Minus One.

Canvas Height Minus One: 24 bits
    1-based height of the canvas in pixels. The actual canvas height is `1 + Canvas Height Minus One`.

The product of *Canvas Width* and *Canvas Height* **MUST** be at most $2^{32} - 1$.

Future specifications may add more fields. Unknown fields **MUST** be ignored.

2.7.1. Chunks

2.7.1.1. Animation

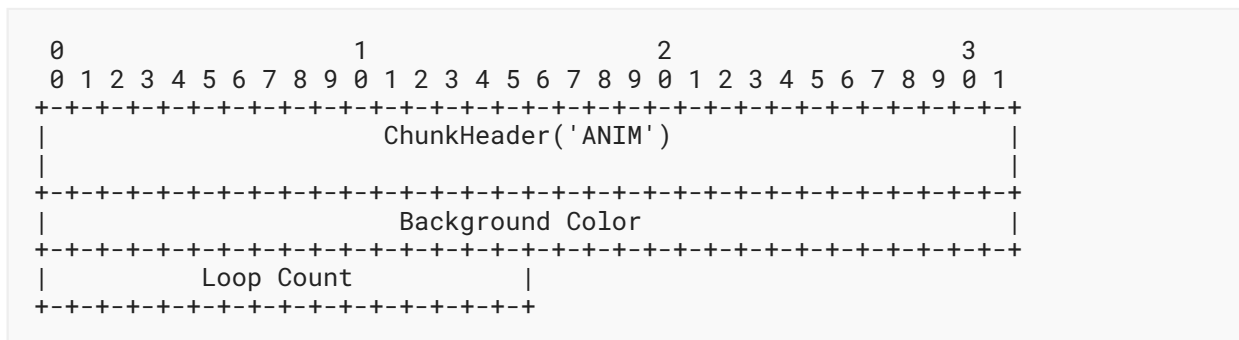An animation is controlled by 'ANIM' and 'ANMF' Chunks.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('ANIM')                      |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Background Color                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Loop Count          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: 'ANIM' Chunk

For an animated image, this chunk contains the *global parameters* of the animation.

Background Color: 32 bits (*uint32*)
    The default background color of the canvas in [Blue, Green, Red, Alpha] byte order. This color **MAY** be used to fill the unused space on the canvas around the frames, as well as the transparent pixels of the first frame. The background color is also used when the Disposal method is 1.

    Note:

    - The background color **MAY** contain a nonopaque alpha value, even if the *Alpha* flag in the 'VP8X' Chunk (Figure 7) is unset.
    - Viewer applications **SHOULD** treat the background color value as a hint and are not required to use it.
    - The canvas is cleared at the start of each loop. The background color **MAY** be used to achieve this.

Loop Count: 16 bits (*uint16*)
    The number of times to loop the animation. If it is 0, this means infinitely.

This chunk **MUST** appear if the *Animation* flag in the 'VP8X' Chunk is set. If the *Animation* flag is not set and this chunk is present, it **MUST** be ignored.
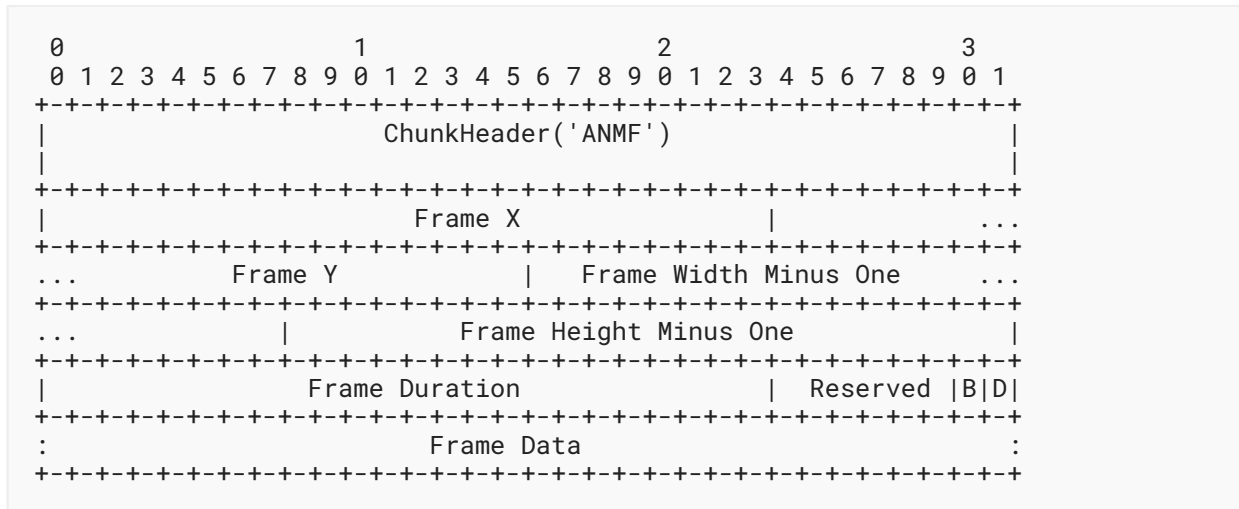
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('ANMF')                      |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Frame X           |            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 ...          Frame Y           |     Frame Width Minus One     ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 ...          |            Frame Height Minus One            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Frame Duration               |  Reserved |B|D|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                        Frame Data                            :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9: 'ANMF' Chunk

For animated images, this chunk contains information about a *single* frame. If the *Animation flag* is not set, then this chunk **SHOULD NOT** be present.

Frame X: 24 bits (*uint24*)
> The X coordinate of the upper left corner of the frame is `Frame X * 2`.

Frame Y: 24 bits (*uint24*)
> The Y coordinate of the upper left corner of the frame is `Frame Y * 2`.

Frame Width Minus One: 24 bits (*uint24*)
> The *1-based* width of the frame. The frame width is `1 + Frame Width Minus One`.

Frame Height Minus One: 24 bits (*uint24*)
> The *1-based* height of the frame. The frame height is `1 + Frame Height Minus One`.

Frame Duration: 24 bits (*uint24*)
> The time to wait before displaying the next frame, in 1-millisecond units. Note that the interpretation of the Frame Duration of 0 (and often <= 10) is defined by the implementation. Many tools and browsers assign a minimum duration similar to GIF.

Reserved: 6 bits
> **MUST** be 0. Readers **MUST** ignore this field.

Blending method (B): 1 bit
> Indicates how transparent pixels of *the current frame* are to be blended with corresponding pixels of the previous canvas:
>
> - 0: Use alpha-blending. After disposing of the previous frame, render the current frame on the canvas using alpha-blending (Section 2.7.1.1, Paragraph 8, Item 16.4.2). If the current frame does not have an alpha channel, assume the alpha value is 255, effectively replacing the rectangle.
> - 1: Do not blend. After disposing of the previous frame, render the current frame on the canvas by overwriting the rectangle covered by the current frame.

Disposal method (D): 1 bit
> Indicates how *the current frame* is to be treated after it has been displayed (before rendering the next frame) on the canvas:
>
> - 0: Do not dispose. Leave the canvas as is.
> - 1: Dispose to the background color. Fill the *rectangle* on the canvas covered by the *current frame* with the background color specified in the 'ANIM' Chunk (Figure 8).
>
> Notes:
>
> - The frame disposal only applies to the *frame rectangle*, that is, the rectangle defined by *Frame X*, *Frame Y*, *frame width*, and *frame height*. It may or may not cover the whole canvas.
> - Alpha-blending:
>
> Given that each of the R, G, B, and A channels is 8 bits and the RGB channels are *not premultiplied* by alpha, the formula for blending 'dst' onto 'src' is:
>
> ```
> blend.A = src.A + dst.A * (1 - src.A / 255)
> if blend.A = 0 then
>   blend.RGB = 0
> else
>   blend.RGB =
>       (src.RGB * src.A +
>        dst.RGB * dst.A * (1 - src.A / 255)) / blend.A
> ```
>
> - Alpha-blending **SHOULD** be done in linear color space by taking into account the color profile (Section 2.7.1.4) of the image. If the color profile is not present, standard RGB (sRGB) is to be assumed. (Note that sRGB also needs to be linearized due to a gamma of ~2.2.)

Frame Data: *Chunk Size* bytes - 16
> Consists of:
>
> - An optional alpha subchunk (Section 2.7.1.2) for the frame.
> - A bitstream subchunk (Section 2.7.1.3) for the frame.

• An optional list of unknown chunks (Section 2.7.1.6).

> Note: The 'ANMF' payload, *Frame Data*, consists of individual *padded* chunks, as
> described by the RIFF file format (Section 2.3).

2.7.1.2. Alpha

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     ChunkHeader('ALPH')                       |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Rsv| P | F | C |     Alpha Bitstream...                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 10: 'ALPH' Chunk

Reserved (Rsv): 2 bits
  MUST be 0. Readers **MUST** ignore this field.

Preprocessing (P): 2 bits
  These informative bits are used to signal the preprocessing that has been performed during
  compression. The decoder can use this information to, for example, dither the values or
  smooth the gradients prior to display.

  • 0: No preprocessing.
  • 1: Level reduction.

  Decoders are not required to use this information in any specified way.

Filtering method (F): 2 bits
  The filtering methods used are described as follows:

  • 0: None.
  • 1: Horizontal filter.
  • 2: Vertical filter.
  • 3: Gradient filter.

  For each pixel, filtering is performed using the following calculations. Assume the alpha
  values surrounding the current X position are labeled as:

```
  C | B |
 ---+---+
  A | X |
```

Figure 11: Pixels Used in Alpha Filtering

We seek to compute the alpha value at position X. First, a prediction is made depending on the filtering method:

- Method `0`: predictor = 0
- Method `1`: predictor = A
- Method `2`: predictor = B
- Method `3`: predictor = clip(A + B - C)

where `clip(v)` is equal to:

- 0 if v < 0,
- 255 if v > 255, or
- v otherwise.

The final value is derived by adding the decompressed value X to the predictor and using modulo-256 arithmetic to wrap the [256..511] range into the [0..255] one:

```
  alpha = (predictor + X) % 256
```

There are special cases for the left-most and top-most pixel positions.

For example, the top-left value at location (0, 0) uses 0 as the predictor value. Otherwise:

- For horizontal or gradient filtering methods, the left-most pixels at location (0, y) are predicted using the location (0, y-1) just above.
- For vertical or gradient filtering methods, the top-most pixels at location (x, 0) are predicted using the location (x-1, 0) on the left.

Compression method (C): 2 bits
The compression method used:

- `0`: No compression.
- `1`: Compressed using the WebP lossless format.

Alpha bitstream: *Chunk Size* bytes - `1`
Encoded alpha bitstream.

This optional chunk contains encoded alpha data for this frame. A frame containing a 'VP8L' Chunk **SHOULD NOT** contain this chunk.

> Rationale: The transparency information is already part of the 'VP8L' Chunk.

The alpha channel data is stored as uncompressed raw data (when the compression method is '0') or compressed using the lossless format (when the compression method is '1').

- Raw data: This consists of a byte sequence of length = width * height, containing all the 8-bit transparency values in scan order.
- Lossless format compression: The byte sequence is a compressed image-stream (as described in Section 3) of implicit dimensions width x height. That is, this image-stream does NOT contain any headers describing the image dimensions.

  Rationale: The dimensions are already known from other sources, so storing them again would be redundant and prone to errors.

  Once the image-stream is decoded into Alpha, Red, Green, Blue (ARGB) color values, following the process described in the lossless format specification, the transparency information must be extracted from the green channel of the ARGB quadruplet.

  Rationale: The green channel is allowed extra transformation steps in the specification -- unlike the other channels -- that can improve compression.

2.7.1.3. Bitstream (VP8/VP8L)

This chunk contains compressed bitstream data for a single frame.

A bitstream chunk may be either (i) a 'VP8 ' Chunk, using 'VP8 ' (note the significant fourth-character space) as its FourCC, *or* (ii) a 'VP8L' Chunk, using 'VP8L' as its FourCC.

The formats of' VP8 ' and 'VP8L' Chunks are as described in Sections 2.5 and 2.6, respectively.
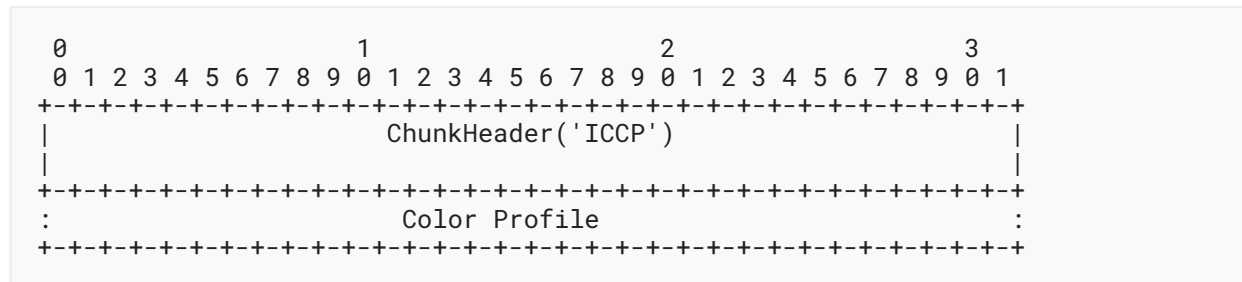
2.7.1.4. Color Profile

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('ICCP')                      |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                        Color Profile                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12: 'ICCP' Chunk

Color Profile: *Chunk Size* bytes
    ICC profile.

This chunk **MUST** appear before the image data.

There **SHOULD** be at most one such chunk. If there are more such chunks, readers **MAY** ignore all except the first one. See the ICC specification [ICC] for details.

If this chunk is not present, sRGB **SHOULD** be assumed.

2.7.1.5. Metadata

Metadata can be stored in 'EXIF' or 'XMP ' Chunks.

There **SHOULD** be at most one chunk of each type ('EXIF' and 'XMP '). If there are more such chunks, readers **MAY** ignore all except the first one.

The chunks are defined as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('EXIF')                      |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                        Exif Metadata                         :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
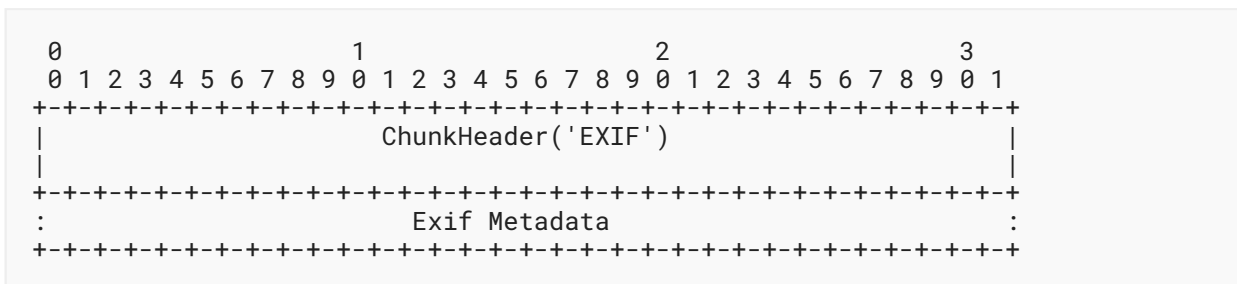
Figure 13: 'EXIF' Chunk

Exif Metadata: *Chunk Size* bytes
     Image metadata in [Exif] format.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      ChunkHeader('XMP ')                      |
|                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                        XMP Metadata                          :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
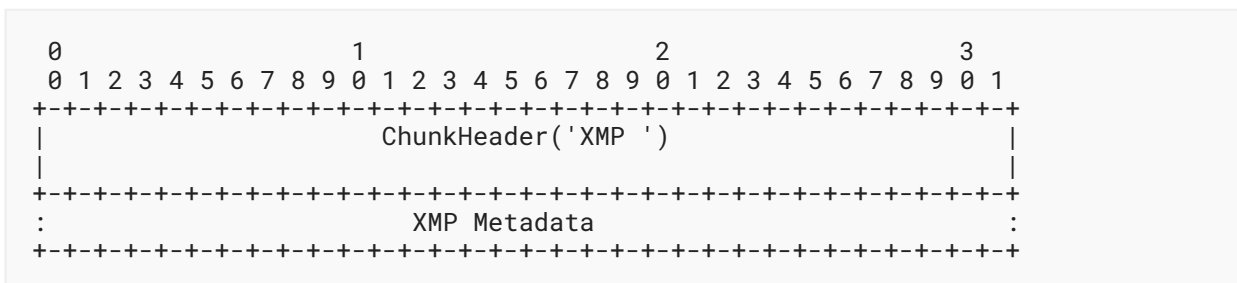
Figure 14: 'XMP ' Chunk

XMP Metadata: *Chunk Size* bytes
     Image metadata in [XMP] format.

> Note that the fourth character in the 'XMP ' FourCC is an ASCII space (0x20).

Additional guidance about handling metadata can be found in the Metadata Working Group's "Guidelines For Handling Image Metadata" [MWG].

2.7.1.6. Unknown Chunks

A RIFF chunk (described in Section 2.3) whose *FourCC* is different from any of the chunks described in this section is considered an *unknown chunk*.

> Rationale: Allowing unknown chunks gives a provision for future extension of the format and also allows storage of any application-specific data.

A file **MAY** contain unknown chunks:

- at the end of the file, as described in Section 2.7, or
- at the end of 'ANMF' Chunks, as described in Section 2.7.1.1.

Readers **SHOULD** ignore these chunks. Writers **SHOULD** preserve them in their original order (unless they specifically intend to modify these chunks).

2.7.2. Canvas Assembly from Frames

Here, we provide an overview of how a reader **MUST** assemble a canvas in the case of an animated image.

The process begins with creating a canvas using the dimensions given in the 'VP8X' Chunk, `Canvas Width Minus One + 1` pixels wide by `Canvas Height Minus One + 1` pixels high. The `Loop Count` field from the 'ANIM' Chunk controls how many times the animation process is repeated. This is `Loop Count - 1` for nonzero `Loop Count` values or infinite if the `Loop Count` is zero.

At the beginning of each loop iteration, the canvas is filled using the background color from the 'ANIM' Chunk or an application-defined color.

'ANMF' Chunks contain individual frames given in display order. Before rendering each frame, the previous frame's `Disposal method` is applied.

The rendering of the decoded frame begins at the Cartesian coordinates (`2 * Frame X`, `2 * Frame Y`), using the top-left corner of the canvas as the origin. `Frame Width Minus One + 1` pixels wide by `Frame Height Minus One + 1` pixels high are rendered onto the canvas using the `Blending method`.

The canvas is displayed for `Frame Duration` milliseconds. This continues until all frames given by 'ANMF' Chunks have been displayed. A new loop iteration is then begun, or the canvas is left in its final state if all iterations have been completed.

The following pseudocode illustrates the rendering process. The notation *VP8X.field* means the field in the 'VP8X' Chunk with the same description.

```
VP8X.flags.hasAnimation MUST be TRUE
canvas <- new image of size VP8X.canvasWidth x VP8X.canvasHeight with
         background color ANIM.background_color or
         application-defined color.
loop_count <- ANIM.loopCount
dispose_method <- Dispose to background color
if loop_count == 0:
  loop_count = inf
frame_params <- nil
next chunk in image_data is ANMF MUST be TRUE
for loop = 0..loop_count - 1
  clear canvas to ANIM.background_color or application-defined color
  until eof or non-ANMF chunk
    frame_params.frameX = Frame X
    frame_params.frameY = Frame Y
    frame_params.frameWidth = Frame Width Minus One + 1
    frame_params.frameHeight = Frame Height Minus One + 1
    frame_params.frameDuration = Frame Duration
    frame_right = frame_params.frameX + frame_params.frameWidth
    frame_bottom = frame_params.frameY + frame_params.frameHeight
    VP8X.canvasWidth >= frame_right MUST be TRUE
    VP8X.canvasHeight >= frame_bottom MUST be TRUE
    for subchunk in 'Frame Data':
      if subchunk.tag == "ALPH":
        alpha subchunks not found in 'Frame Data' earlier MUST be
          TRUE
        frame_params.alpha = alpha_data
      else if subchunk.tag == "VP8 " OR subchunk.tag == "VP8L":
        bitstream subchunks not found in 'Frame Data' earlier MUST
          be TRUE
        frame_params.bitstream = bitstream_data
    apply dispose_method.
    render frame with frame_params.alpha and frame_params.bitstream
      on canvas with top-left corner at (frame_params.frameX,
      frame_params.frameY), using Blending method
      frame_params.blendingMethod.
    canvas contains the decoded image.
    Show the contents of the canvas for
    frame_params.frameDuration * 1 ms.
    dispose_method = frame_params.disposeMethod
```

2.7.3. Example File Layouts

A lossy-encoded image with alpha may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- ALPH (alpha bitstream)
+- VP8 (bitstream)
```

Figure 15: A Lossy-Encoded Image with Alpha

A lossless-encoded image may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- VP8L (lossless bitstream)
+- XYZW (unknown chunk)
```

Figure 16: A Lossless-Encoded Image

A lossless image with an ICC profile and XMP metadata may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- ICCP (color profile)
+- VP8L (lossless bitstream)
+- XMP  (metadata)
```

Figure 17: A Lossless Image with an ICC Profile and XMP Metadata

An animated image with Exif metadata may look as follows:

```
RIFF/WEBP
+- VP8X (descriptions of features used)
+- ANIM (global animation parameters)
+- ANMF (frame1 parameters + data)
+- ANMF (frame2 parameters + data)
+- ANMF (frame3 parameters + data)
+- ANMF (frame4 parameters + data)
+- EXIF (metadata)
```

Figure 18: An Animated Image with Exif Metadata

3. Specification for WebP Lossless Bitstream

> Note that this section is based on the documentation in the libwebp source repository [webp-lossless-src].

3.1. Abstract (from "Specification for WebP Lossless Bitstream")

WebP lossless is an image format for lossless compression of ARGB images. The lossless format stores and restores the pixel values exactly, including the color values for pixels whose alpha value is 0. The format uses subresolution images, recursively embedded into the format itself, for storing statistical data about the images, such as the used entropy codes, spatial predictors, color space conversion, and color table. A universal algorithm for sequential data compression [LZ77],

prefix coding, and a color cache are used for compression of the bulk data. Decoding speeds faster than PNG have been demonstrated, as well as 25% denser compression than can be achieved using today's PNG format [webp-lossless-study].

3.2. Introduction (from "Specification for WebP Lossless Bitstream")

This section describes the compressed data representation of a WebP lossless image.

In this section, we extensively use C programming language syntax [ISO.9899.2018] to describe the bitstream and assume the existence of a function for reading bits, `ReadBits(n)`. The bytes are read in the natural order of the stream containing them, and bits of each byte are read in least-significant-bit-first order. When multiple bits are read at the same time, the integer is constructed from the original data in the original order. The most significant bits of the returned integer are also the most significant bits of the original data. Thus, the statement

```
b = ReadBits(2);
```

is equivalent with the two statements below:

```
b = ReadBits(1);
b |= ReadBits(1) << 1;
```

We assume that each color component (that is, alpha, red, blue, and green) is represented using an 8-bit byte. We define the corresponding type as uint8. A whole ARGB pixel is represented by a type called uint32, which is an unsigned integer consisting of 32 bits. In the code showing the behavior of the transforms, these values are codified in the following bits: alpha in bits 31..24, red in bits 23..16, green in bits 15..8, and blue in bits 7..0; however, implementations of the format are free to use another representation internally.

Broadly, a WebP lossless image contains header data, transform information, and actual image data. Headers contain the width and height of the image. A WebP lossless image can go through four different types of transforms before being entropy encoded. The transform information in the bitstream contains the data required to apply the respective inverse transforms.

3.3. Nomenclature

ARGB
> A pixel value consisting of alpha, red, green, and blue values.

ARGB image
> A two-dimensional array containing ARGB pixels.

color cache
> A small hash-addressed array to store recently used colors to be able to recall them with shorter codes.

color indexing image
> A one-dimensional image of colors that can be indexed using a small integer (up to 256 within WebP lossless).

color transform image
> A two-dimensional subresolution image containing data about correlations of color components.

distance mapping
> Changes LZ77 distances to have the smallest values for pixels in two-dimensional proximity.

entropy image
> A two-dimensional subresolution image indicating which entropy coding should be used in a respective square in the image, that is, each pixel is a meta prefix code.

[LZ77]
> A dictionary-based sliding window compression algorithm that either emits symbols or describes them as sequences of past symbols.

meta prefix code
> A small integer (up to 16 bits) that indexes an element in the meta prefix table.

predictor image
> A two-dimensional subresolution image indicating which spatial predictor is used for a particular square in the image.

prefix code
> A classic way to do entropy coding where a smaller number of bits are used for more frequent codes.

prefix coding
> A way to entropy code larger integers, which codes a few bits of the integer using an entropy code and codifies the remaining bits raw. This allows for the descriptions of the entropy codes to remain relatively small even when the range of symbols is large.

scan-line order
> A processing order of pixels (left to right and top to bottom), starting from the left-hand-top pixel. Once a row is completed, continue from the left-hand column of the next row.

3.4. RIFF Header

The beginning of the header has the RIFF container. This consists of the following 21 bytes:

1. String 'RIFF'.
2. A little-endian, 32-bit value of the chunk length, which is the whole size of the chunk controlled by the RIFF header. Normally, this equals the payload size (file size minus 8 bytes: 4 bytes for the 'RIFF' identifier and 4 bytes for storing the value itself).
3. String 'WEBP' (RIFF container name).
4. String 'VP8' (FourCC for lossless-encoded image data).

5. A little-endian, 32-bit value of the number of bytes in the lossless stream.

6. 1-byte signature 0x2f.

The first 28 bits of the bitstream specify the width and height of the image. Width and height are decoded as 14-bit integers as follows:

```
int image_width = ReadBits(14) + 1;
int image_height = ReadBits(14) + 1;
```

The 14-bit precision for image width and height limits the maximum size of a WebP lossless image to 16384x16384 pixels.

The alpha_is_used bit is a hint only and **SHOULD NOT** impact decoding. It **SHOULD** be set to 0 when all alpha values are 255 in the picture and 1 otherwise.

```
int alpha_is_used = ReadBits(1);
```

The version_number is a 3-bit code that **MUST** be set to 0. Any other value **MUST** be treated as an error.

```
int version_number = ReadBits(3);
```

3.5. Transforms

The transforms are reversible manipulations of the image data that can reduce the remaining symbolic entropy by modeling spatial and color correlations. They can make the final compression more dense.

An image can go through four types of transforms. A 1 bit indicates the presence of a transform. Each transform is allowed to be used only once. The transforms are used only for the main-level ARGB image; the subresolution images (color transform image, entropy image, and predictor image) have no transforms, not even the 0 bit indicating the end of transforms.

> Typically, an encoder would use these transforms to reduce the Shannon entropy in the residual image. Also, the transform data can be decided based on entropy minimization.

```
while (ReadBits(1)) {  // Transform present.
  // Decode transform type.
  enum TransformType transform_type = ReadBits(2);
  // Decode transform data.
  ...
}

// Decode actual image data.
```

If a transform is present, then the next two bits specify the transform type. There are four types of transforms.

Transform	Bit
PREDICTOR_TRANSFORM	0
COLOR_TRANSFORM	1
SUBTRACT_GREEN_TRANSFORM	2
COLOR_INDEXING_TRANSFORM	3

Table 1: Transform Types

The transform type is followed by the transform data. Transform data contains the information required to apply the inverse transform and depends on the transform type. The inverse transforms are applied in the reverse order that they are read from the bitstream, that is, last one first.

Next, we describe the transform data for different types.

3.5.1. Predictor Transform

The predictor transform can be used to reduce entropy by exploiting the fact that neighboring pixels are often correlated. In the predictor transform, the current pixel value is predicted from the pixels already decoded (in scan-line order) and only the residual value (actual - predicted) is encoded. The green component of a pixel defines which of the 14 predictors is used within a particular block of the ARGB image. The *prediction mode* determines the type of prediction to use. We divide the image into squares, and all the pixels in a square use the same prediction mode.

The first 3 bits of prediction data define the block width and height in number of bits.

```
int size_bits = ReadBits(3) + 2;
int block_width = (1 << size_bits);
int block_height = (1 << size_bits);
#define DIV_ROUND_UP(num, den) (((num) + (den) - 1) / (den))
int transform_width = DIV_ROUND_UP(image_width, 1 << size_bits);
```

The transform data contains the prediction mode for each block of the image. It is a subresolution image where the green component of a pixel defines which of the 14 predictors is used for all the `block_width * block_height` pixels within a particular block of the ARGB image. This subresolution image is encoded using the same techniques described in Section 3.6.

The number of block columns, `transform_width`, is used in two-dimensional indexing. For a pixel (x, y), one can compute the respective filter block address by:

```
int block_index = (y >> size_bits) * transform_width +
                  (x >> size_bits);
```

There are 14 different prediction modes. In each prediction mode, the current pixel value is predicted from one or more neighboring pixels whose values are already known.

We chose the neighboring pixels (TL, T, TR, and L) of the current pixel (P) as follows:

```
O    O    O    O    O    O    O    O    O    O    O
O    O    O    O    O    O    O    O    O    O    O
O    O    O    O    TL   T    TR   O    O    O    O
O    O    O    O    L    P    X    X    X    X    X
X    X    X    X    X    X    X    X    X    X    X
X    X    X    X    X    X    X    X    X    X    X
```

Figure 19: Neighboring Pixels of the Current Pixel (P)

where TL means top-left, T means top, TR means top-right, and L means left. At the time of predicting a value for P, all O, TL, T, TR, and L pixels have already been processed, and the P pixel and all X pixels are unknown.

Given the preceding neighboring pixels, the different prediction modes are defined as follows.

Mode	Predicted Value of Each Channel of the Current Pixel
0	0xff000000 (represents solid black color in ARGB)
1	L
2	T
3	TR
4	TL
5	Average2(Average2(L, TR), T)
6	Average2(L, TL)

Mode	Predicted Value of Each Channel of the Current Pixel
7	Average2(L, T)
8	Average2(TL, T)
9	Average2(T, TR)
10	Average2(Average2(L, TL), Average2(T, TR))
11	Select(L, T, TL)
12	ClampAddSubtractFull(L, T, TL)
13	ClampAddSubtractHalf(Average2(L, T), TL)

Table 2: Prediction Modes

`Average2` is defined as follows for each ARGB component:

```
uint8 Average2(uint8 a, uint8 b) {
  return (a + b) / 2;
}
```

The Select predictor is defined as follows:

```
uint32 Select(uint32 L, uint32 T, uint32 TL) {
  // L = left pixel, T = top pixel, TL = top-left pixel.

  // ARGB component estimates for prediction.
  int pAlpha = ALPHA(L) + ALPHA(T) - ALPHA(TL);
  int pRed = RED(L) + RED(T) - RED(TL);
  int pGreen = GREEN(L) + GREEN(T) - GREEN(TL);
  int pBlue = BLUE(L) + BLUE(T) - BLUE(TL);

  // Manhattan distances to estimates for left and top pixels.
  int pL = abs(pAlpha - ALPHA(L)) + abs(pRed - RED(L)) +
           abs(pGreen - GREEN(L)) + abs(pBlue - BLUE(L));
  int pT = abs(pAlpha - ALPHA(T)) + abs(pRed - RED(T)) +
           abs(pGreen - GREEN(T)) + abs(pBlue - BLUE(T));

  // Return either left or top, the one closer to the prediction.
  if (pL < pT) {
    return L;
  } else {
    return T;
  }
}
```

The functions `ClampAddSubtractFull` and `ClampAddSubtractHalf` are performed for each ARGB component as follows:

```
// Clamp the input value between 0 and 255.
int Clamp(int a) {
  return (a < 0) ? 0 : (a > 255) ? 255 : a;
}

int ClampAddSubtractFull(int a, int b, int c) {
  return Clamp(a + b - c);
}

int ClampAddSubtractHalf(int a, int b) {
  return Clamp(a + (a - b) / 2);
}
```

There are special handling rules for some border pixels. If there is a predictor transform, regardless of the mode [0..13] for these pixels, the predicted value for the left-topmost pixel of the image is 0xff000000, all pixels on the top row are L-pixel, and all pixels on the leftmost column are T-pixel.

Addressing the TR-pixel for pixels on the rightmost column is exceptional. The pixels on the rightmost column are predicted by using the modes [0..13], just like pixels not on the border, but the leftmost pixel on the same row as the current pixel is instead used as the TR-pixel.

The final pixel value is obtained by adding each channel of the predicted value to the encoded residual value.

```
void PredictorTransformOutput(uint32 residual, uint32 pred,
                              uint8* alpha, uint8* red,
                              uint8* green, uint8* blue) {
  *alpha = ALPHA(residual) + ALPHA(pred);
  *red = RED(residual) + RED(pred);
  *green = GREEN(residual) + GREEN(pred);
  *blue = BLUE(residual) + BLUE(pred);
}
```

3.5.2. Color Transform

The goal of the color transform is to decorrelate the R, G, and B values of each pixel. The color transform keeps the green (G) value as it is, transforms the red (R) value based on the green value, and transforms the blue (B) value based on the green value and then on the red value.

As is the case for the predictor transform, first the image is divided into blocks, and the same transform mode is used for all the pixels in a block. For each block, there are three types of color transform elements.

```
typedef struct {
  uint8 green_to_red;
  uint8 green_to_blue;
  uint8 red_to_blue;
} ColorTransformElement;
```

The actual color transform is done by defining a color transform delta. The color transform delta depends on the `ColorTransformElement`, which is the same for all the pixels in a particular block. The delta is subtracted during the color transform. The inverse color transform then is just adding those deltas.

The color transform function is defined as follows:

```
void ColorTransform(uint8 red, uint8 blue, uint8 green,
                    ColorTransformElement *trans,
                    uint8 *new_red, uint8 *new_blue) {
  // Transformed values of red and blue components
  int tmp_red = red;
  int tmp_blue = blue;

  // Applying the transform is just subtracting the transform deltas
  tmp_red  -= ColorTransformDelta(trans->green_to_red,  green);
  tmp_blue -= ColorTransformDelta(trans->green_to_blue, green);
  tmp_blue -= ColorTransformDelta(trans->red_to_blue, red);

  *new_red = tmp_red & 0xff;
  *new_blue = tmp_blue & 0xff;
}
```

`ColorTransformDelta` is computed using a signed 8-bit integer representing a 3.5-fixed-point number and a signed 8-bit RGB color channel (c) [-128..127] and is defined as follows:

```
int8 ColorTransformDelta(int8 t, int8 c) {
  return (t * c) >> 5;
}
```

A conversion from the 8-bit unsigned representation (`uint8`) to the 8-bit signed one (`int8`) is required before calling `ColorTransformDelta()`. The signed value should be interpreted as an 8-bit two's complement number (that is: uint8 range [128..255] is mapped to the [-128..-1] range of its converted int8 value).

The multiplication is to be done using more precision (with at least 16-bit precision). The sign extension property of the shift operation does not matter here; only the lowest 8 bits are used from the result, and in these bits, the sign extension shifting and unsigned shifting are consistent with each other.

Now, we describe the contents of color transform data so that decoding can apply the inverse color transform and recover the original red and blue values. The first 3 bits of the color transform data contain the width and height of the image block in number of bits, just like the predictor transform:

```
int size_bits = ReadBits(3) + 2;
int block_width = 1 << size_bits;
int block_height = 1 << size_bits;
```

The remaining part of the color transform data contains `ColorTransformElement` instances, corresponding to each block of the image. Each `ColorTransformElement 'cte'` is treated as a pixel in a subresolution image whose alpha component is 255, red component is `cte.red_to_blue`, green component is `cte.green_to_blue`, and blue component is `cte.green_to_red`.

During decoding, `ColorTransformElement` instances of the blocks are decoded and the inverse color transform is applied on the ARGB values of the pixels. As mentioned earlier, that inverse color transform is just adding `ColorTransformElement` values to the red and blue channels. The alpha and green channels are left as is.

```
void InverseTransform(uint8 red, uint8 green, uint8 blue,
                      ColorTransformElement *trans,
                      uint8 *new_red, uint8 *new_blue) {
  // Transformed values of red and blue components
  int tmp_red = red;
  int tmp_blue = blue;

  // Applying the inverse transform is just adding the
  // color transform deltas
  tmp_red  += ColorTransformDelta(trans->green_to_red, green);
  tmp_blue += ColorTransformDelta(trans->green_to_blue, green);
  tmp_blue +=
      ColorTransformDelta(trans->red_to_blue, tmp_red & 0xff);

  *new_red = tmp_red & 0xff;
  *new_blue = tmp_blue & 0xff;
}
```

3.5.3. Subtract Green Transform

The subtract green transform subtracts green values from red and blue values of each pixel. When this transform is present, the decoder needs to add the green value to both the red and blue values. There is no data associated with this transform. The decoder applies the inverse transform as follows:

```
void AddGreenToBlueAndRed(uint8 green, uint8 *red, uint8 *blue) {
  *red  = (*red  + green) & 0xff;
  *blue = (*blue + green) & 0xff;
}
```

This transform is redundant, as it can be modeled using the color transform, but since there is no additional data here, the subtract green transform can be coded using fewer bits than a full-blown color transform.

3.5.4. Color Indexing Transform

If there are not many unique pixel values, it may be more efficient to create a color index array and replace the pixel values by the array's indices. The color indexing transform achieves this. (In the context of WebP lossless, we specifically do not call this a palette transform because a similar but more dynamic concept exists in WebP lossless encoding: color cache.)

The color indexing transform checks for the number of unique ARGB values in the image. If that number is below a threshold (256), it creates an array of those ARGB values, which is then used to replace the pixel values with the corresponding index: the green channel of the pixels are replaced with the index, all alpha values are set to 255, and all red and blue values are set to 0.

The transform data contains the color table size and the entries in the color table. The decoder reads the color indexing transform data as follows:

```
// 8-bit value for the color table size
int color_table_size = ReadBits(8) + 1;
```

The color table is stored using the image storage format itself. The color table can be obtained by reading an image, without the RIFF header, image size, and transforms, assuming the height of 1 pixel and the width of `color_table_size`. The color table is always subtraction-coded to reduce image entropy. The deltas of palette colors contain typically much less entropy than the colors themselves, leading to significant savings for smaller images. In decoding, every final color in the color table can be obtained by adding the previous color component values by each ARGB component separately and storing the least significant 8 bits of the result.

The inverse transform for the image is simply replacing the pixel values (which are indices to the color table) with the actual color table values. The indexing is done based on the green component of the ARGB color.

```
// Inverse transform
argb = color_table[GREEN(argb)];
```

If the index is equal to or larger than `color_table_size`, the argb color value should be set to 0x00000000 (transparent black).

When the color table is small (equal to or less than 16 colors), several pixels are bundled into a single pixel. The pixel bundling packs several (2, 4, or 8) pixels into a single pixel, reducing the image width respectively.

> Pixel bundling allows for a more efficient joint distribution entropy coding of neighboring pixels and gives some arithmetic coding-like benefits to the entropy code, but it can only be used when there are 16 or fewer unique values.

`color_table_size` specifies how many pixels are combined:

color_table_size	width_bits value
1..2	3
3..4	2
5..16	1
17..256	0

Table 3: Color Table Size to Bundled Pixel Bit Width Mapping

`width_bits` has a value of 0, 1, 2, or 3. A value of 0 indicates no pixel bundling is to be done for the image. A value of 1 indicates that two pixels are combined, and each pixel has a range of [0..15]. A value of 2 indicates that four pixels are combined, and each pixel has a range of [0..3]. A value of 3 indicates that eight pixels are combined, and each pixel has a range of [0..1], that is, a binary value.

The values are packed into the green component as follows:

- `width_bits` = 1: For every x value, where x = 2k + 0, a green value at x is positioned into the 4 least significant bits of the green value at x / 2, and a green value at x + 1 is positioned into the 4 most significant bits of the green value at x / 2.
- `width_bits` = 2: For every x value, where x = 4k + 0, a green value at x is positioned into the 2 least significant bits of the green value at x / 4, and green values at x + 1 to x + 3 are positioned in order to the more significant bits of the green value at x / 4.
- `width_bits` = 3: For every x value, where x = 8k + 0, a green value at x is positioned into the least significant bit of the green value at x / 8, and green values at x + 1 to x + 7 are positioned in order to the more significant bits of the green value at x / 8.

After reading this transform, `image_width` is subsampled by `width_bits`. This affects the size of subsequent transforms. The new size can be calculated using `DIV_ROUND_UP`, as defined in [Section 3.5.1](#).

```
image_width = DIV_ROUND_UP(image_width, 1 << width_bits);
```

3.6. Image Data

Image data is an array of pixel values in scan-line order.

3.6.1. Roles of Image Data

We use image data in five different roles:

1. ARGB image: Stores the actual pixels of the image.

2. Entropy image: Stores the meta prefix codes (see "Decoding of Meta Prefix Codes" (Section 3.7.2.2)).

3. Predictor image: Stores the metadata for the predictor transform (see "Predictor Transform" (Section 3.5.1)).

4. Color transform image: Created by `ColorTransformElement` values (defined in "Color Transform" (Section 3.5.2)) for different blocks of the image.

5. Color indexing image: An array of the size of `color_table_size` (up to 256 ARGB values) that stores the metadata for the color indexing transform (see "Color Indexing Transform" (Section 3.5.4)).

3.6.2. Encoding of Image Data

The encoding of image data is independent of its role.

The image is first divided into a set of fixed-size blocks (typically 16x16 blocks). Each of these blocks are modeled using their own entropy codes. Also, several blocks may share the same entropy codes.

> Rationale: Storing an entropy code incurs a cost. This cost can be minimized if statistically similar blocks share an entropy code, thereby storing that code only once. For example, an encoder can find similar blocks by clustering them using their statistical properties or by repeatedly joining a pair of randomly selected clusters when it reduces the overall amount of bits needed to encode the image.

Each pixel is encoded using one of the three possible methods:

1. Prefix-coded literals: Each channel (green, red, blue, and alpha) is entropy-coded independently.

2. LZ77 backward reference: A sequence of pixels are copied from elsewhere in the image.

3. Color cache code: Using a short multiplicative hash code (color cache index) of a recently seen color.

The following subsections describe each of these in detail.

3.6.2.1. Prefix-Coded Literals

The pixel is stored as prefix-coded values of green, red, blue, and alpha (in that order). See Section 3.7.2.3 for details.

3.6.2.2. LZ77 Backward Reference

Backward references are tuples of *length* and *distance code*:

• Length indicates how many pixels in scan-line order are to be copied.

• Distance code is a number indicating the position of a previously seen pixel, from which the pixels are to be copied. The exact mapping is described below (Section 3.6.2.2.1).

The length and distance values are stored using **LZ77 prefix coding**.

LZ77 prefix coding divides large integer values into two parts: the *prefix code* and the *extra bits*. The prefix code is stored using an entropy code, while the extra bits are stored as they are (without an entropy code).

> Rationale: This approach reduces the storage requirement for the entropy code. Also, large values are usually rare, so extra bits would be used for very few values in the image. Thus, this approach results in better compression overall.

The following table denotes the prefix codes and extra bits used for storing different ranges of values.

> Note: The maximum backward reference length is limited to 4096. Hence, only the first 24 prefix codes (with the respective extra bits) are meaningful for length values. For distance values, however, all the 40 prefix codes are valid.

Value Range	Prefix Code	Extra Bits
1	0	0
2	1	0
3	2	0
4	3	0
5..6	4	1
7..8	5	1
9..12	6	2
13..16	7	2
...	...	...
3072..4096	23	10
...	...	...
524289..786432	38	18
786433..1048576	39	18

Table 4: Value to Prefix Code and Extra Bits Mapping

The pseudocode to obtain a (length or distance) value from the prefix code is as follows:

```
if (prefix_code < 4) {
  return prefix_code + 1;
}
int extra_bits = (prefix_code - 2) >> 1;
int offset = (2 + (prefix_code & 1)) << extra_bits;
return offset + ReadBits(extra_bits) + 1;
```

3.6.2.2.1. Distance Mapping

As noted previously, a distance code is a number indicating the position of a previously seen pixel, from which the pixels are to be copied. This subsection defines the mapping between a distance code and the position of a previous pixel.

Distance codes larger than 120 denote the pixel distance in scan-line order, offset by 120.

The smallest distance codes [1..120] are special and are reserved for a close neighborhood of the current pixel. This neighborhood consists of 120 pixels:

- Pixels that are 1 to 7 rows above the current pixel and are up to 8 columns to the left or up to 7 columns to the right of the current pixel [Total such pixels = 7 * (8 + 1 + 7) = 112].
- Pixels that are in the same row as the current pixel and are up to 8 columns to the left of the current pixel [8 such pixels].

The mapping between distance code `distance_code` and the neighboring pixel offset (`xi`, `yi`) is as follows:

```
(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)
```

Figure 20: Distance Code to Neighboring Pixel Offset Mapping

For example, the distance code 1 indicates an offset of (0, 1) for the neighboring pixel, that is, the pixel above the current pixel (0 pixel difference in the X direction and 1 pixel difference in the Y direction). Similarly, the distance code 3 indicates the top-left pixel.

The decoder can convert a distance code `distance_code` to a scan-line order distance `dist` as follows:

```
(xi, yi) = distance_map[distance_code - 1]
dist = xi + yi * image_width
if (dist < 1) {
  dist = 1
}
```

where `distance_map` is the mapping noted above, and `image_width` is the width of the image in pixels.

3.6.2.3. Color Cache Coding

Color cache stores a set of colors that have been recently used in the image.

> Rationale: This way, the recently used colors can sometimes be referred to more efficiently than emitting them using the other two methods (described in Sections 3.6.2.1 and 3.6.2.2).

Color cache codes are stored as follows. First, there is a 1-bit value that indicates if the color cache is used. If this bit is 0, no color cache codes exist, and they are not transmitted in the prefix code that decodes the green symbols and the length prefix codes. However, if this bit is 1, the color cache size is read next:

```
int color_cache_code_bits = ReadBits(4);
int color_cache_size = 1 << color_cache_code_bits;
```

`color_cache_code_bits` defines the size of the color cache (1 << `color_cache_code_bits`). The range of allowed values for `color_cache_code_bits` is [1..11]. Compliant decoders MUST indicate a corrupted bitstream for other values.

A color cache is an array of size `color_cache_size`. Each entry stores one ARGB color. Colors are looked up by indexing them by (0x1e35a7bd * color) >> (32 - color_cache_code_bits). Only one lookup is done in a color cache; there is no conflict resolution.

In the beginning of decoding or encoding of an image, all entries in all color cache values are set to zero. The color cache code is converted to this color at decoding time. The state of the color cache is maintained by inserting every pixel, be it produced by backward referencing or as literals, into the cache in the order they appear in the stream.

3.7. Entropy Code

3.7.1. Overview

Most of the data is coded using a canonical prefix code [Huffman]. Hence, the codes are transmitted by sending the *prefix code lengths*, as opposed to the actual *prefix codes*.

In particular, the format uses **spatially variant prefix coding**. In other words, different blocks of the image can potentially use different entropy codes.

> Rationale: Different areas of the image may have different characteristics. So, allowing them to use different entropy codes provides more flexibility and potentially better compression.

3.7.2. Details

The encoded image data consists of several parts:

1. Decoding and building the prefix codes.
2. Meta prefix codes.
3. Entropy-coded image data.

For any given pixel (x, y), there is a set of five prefix codes associated with it. These codes are (in bitstream order):

- **Prefix code #1**: Used for green channel, backward-reference length, and color cache.
- **Prefix code #2, #3, and #4**: Used for red, blue, and alpha channels, respectively.
- **Prefix code #5**: Used for backward-reference distance.

From here on, we refer to this set as a **prefix code group**.

3.7.2.1. Decoding and Building the Prefix Codes

This section describes how to read the prefix code lengths from the bitstream.

The prefix code lengths can be coded in two ways. The method used is specified by a 1-bit value.

- If this bit is 1, it is a *simple code length code.*
- If this bit is 0, it is a *normal code length code.*

In both cases, there can be unused code lengths that are still part of the stream. This may be inefficient, but it is allowed by the format. The described tree must be a complete binary tree. A single leaf node is considered a complete binary tree and can be encoded using either the simple code length code or the normal code length code. When coding a single leaf node using the *normal code length code*, all but one code length are zeros, and the single leaf node value is marked with the length of 1 -- even when no bits are consumed when that single leaf node tree is used.

3.7.2.1.1. Simple Code Length Code

This variant is used in the special case when only 1 or 2 prefix symbols are in the range [0..255] with code length 1. All other prefix code lengths are implicitly zeros.

The first bit indicates the number of symbols:

```
int num_symbols = ReadBits(1) + 1;
```

The following are the symbol values. This first symbol is coded using 1 or 8 bits, depending on the value of `is_first_8bits`. The range is [0..1] or [0..255], respectively. The second symbol, if present, is always assumed to be in the range [0..255] and coded using 8 bits.

```
int is_first_8bits = ReadBits(1);
symbol0 = ReadBits(1 + 7 * is_first_8bits);
code_lengths[symbol0] = 1;
if (num_symbols == 2) {
  symbol1 = ReadBits(8);
  code_lengths[symbol1] = 1;
}
```

> The two symbols should be different. Duplicate symbols are allowed, but inefficient.

> Note: Another special case is when *all* prefix code lengths are *zeros* (an empty prefix code). For example, a prefix code for distance can be empty if there are no backward references. Similarly, prefix codes for alpha, red, and blue can be empty if all pixels within the same meta prefix code are produced using the color cache. However, this case doesn't need special handling, as empty prefix codes can be coded as those containing a single symbol 0.

3.7.2.1.2. Normal Code Length Code

The code lengths of the prefix code fit in 8 bits and are read as follows. First, `num_code_lengths` specifies the number of code lengths.

```
int num_code_lengths = 4 + ReadBits(4);
```

The code lengths are themselves encoded using prefix codes; lower-level code lengths, `code_length_code_lengths`, first have to be read. The rest of those `code_length_code_lengths` (according to the order in `kCodeLengthCodeOrder`) are zeros.

```
int kCodeLengthCodes = 19;
int kCodeLengthCodeOrder[kCodeLengthCodes] = {
  17, 18, 0, 1, 2, 3, 4, 5, 16, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
};
int code_length_code_lengths[kCodeLengthCodes] = { 0 };  // All zeros
for (i = 0; i < num_code_lengths; ++i) {
  code_length_code_lengths[kCodeLengthCodeOrder[i]] = ReadBits(3);
}
```

Next, if `ReadBits(1) == 0`, the maximum number of different read symbols (`max_symbol`) for each symbol type (A, R, G, B, and distance) is set to its alphabet size:

- G channel: 256 + 24 + `color_cache_size`
- Other literals (A, R, and B): 256
- Distance code: 40

Otherwise, it is defined as:

```
int length_nbits = 2 + 2 * ReadBits(3);
int max_symbol = 2 + ReadBits(length_nbits);
```

If `max_symbol` is larger than the size of the alphabet for the symbol type, the bitstream is invalid.

A prefix table is then built from `code_length_code_lengths` and used to read up to `max_symbol` code lengths.

- Code [0..15] indicates literal code lengths.
  ◦ Value 0 means no symbols have been coded.
  ◦ Values [1..15] indicate the bit length of the respective code.
- Code 16 repeats the previous nonzero value [3..6] times, that is, `3 + ReadBits(2)` times. If code 16 is used before a nonzero value has been emitted, a value of 8 is repeated.
- Code 17 emits a streak of zeros of length [3..10], that is, `3 + ReadBits(3)` times.
- Code 18 emits a streak of zeros of length [11..138], that is, `11 + ReadBits(7)` times.

Once code lengths are read, a prefix code for each symbol type (A, R, G, B, and distance) is formed using their respective alphabet sizes.

3.7.2.2. Decoding of Meta Prefix Codes

As noted earlier, the format allows the use of different prefix codes for different blocks of the image. *Meta prefix codes* are indexes identifying which prefix codes to use in different parts of the image.

Meta prefix codes may be used *only* when the image is being used in the role (Section 3.6.1) of an *ARGB image*.

There are two possibilities for the meta prefix codes, indicated by a 1-bit value:

- If this bit is zero, there is only one meta prefix code used everywhere in the image. No more data is stored.
- If this bit is one, the image uses multiple meta prefix codes. These meta prefix codes are stored as an *entropy image* (described below).

The red and green components of a pixel define a 16-bit meta prefix code used in a particular block of the ARGB image.

3.7.2.2.1. Entropy Image

The entropy image defines which prefix codes are used in different parts of the image.

The first 3 bits contain the `prefix_bits` value. The dimensions of the entropy image are derived from `prefix_bits`:

```
int prefix_bits = ReadBits(3) + 2;
int prefix_image_width =
    DIV_ROUND_UP(image_width, 1 << prefix_bits);
int prefix_image_height =
    DIV_ROUND_UP(image_height, 1 << prefix_bits);
```

where `DIV_ROUND_UP` is as defined in Section 3.5.1.

The next bits contain an entropy image of width `prefix_image_width` and height `prefix_image_height`.

3.7.2.2.2. Interpretation of Meta Prefix Codes

The number of prefix code groups in the ARGB image can be obtained by finding the *largest meta prefix code* from the entropy image:

```
int num_prefix_groups = max(entropy image) + 1;
```

where `max(entropy image)` indicates the largest prefix code stored in the entropy image.

As each prefix code group contains five prefix codes, the total number of prefix codes is:

```
int num_prefix_codes = 5 * num_prefix_groups;
```

Given a pixel (x, y) in the ARGB image, we can obtain the corresponding prefix codes to be used as follows:

```
int position =
    (y >> prefix_bits) * prefix_image_width + (x >> prefix_bits);
int meta_prefix_code = (entropy_image[position] >> 8) & 0xffff;
PrefixCodeGroup prefix_group = prefix_code_groups[meta_prefix_code];
```

where we have assumed the existence of `PrefixCodeGroup` structure, which represents a set of five prefix codes. Also, `prefix_code_groups` is an array of `PrefixCodeGroup` (of size `num_prefix_groups`).

The decoder then uses prefix code group `prefix_group` to decode the pixel (x, y), as explained in Section 3.7.2.3.

3.7.2.3. Decoding Entropy-Coded Image Data

For the current position (x, y) in the image, the decoder first identifies the corresponding prefix code group (as explained in the last section). Given the prefix code group, the pixel is read and decoded as follows.

Next, read symbol S from the bitstream using prefix code #1.

> Note that S is any integer in the range 0 to (`256 + 24 + color_cache_size - 1`).
> See Section 3.6.2.3 for details about `color_cache_size`.

The interpretation of S depends on its value:

1. If S < 256

    i. Use S as the green component.
    ii. Read red from the bitstream using prefix code #2.
    iii. Read blue from the bitstream using prefix code #3.
    iv. Read alpha from the bitstream using prefix code #4.

2. If S >= 256 & S < 256 + 24

    i. Use S - 256 as a length prefix code.
    ii. Read extra bits for the length from the bitstream.
    iii. Determine backward-reference length L from length prefix code and the extra bits read.
    iv. Read the distance prefix code from the bitstream using prefix code #5.
    v. Read extra bits for the distance from the bitstream.
    vi. Determine backward-reference distance D from the distance prefix code and the extra bits read.
    vii. Copy L pixels (in scan-line order) from the sequence of pixels starting at the current position minus D pixels.

3. If S >= 256 + 24

    i. Use S - (256 + 24) as the index into the color cache.

ii. Get ARGB color from the color cache at that index.

3.8. Overall Structure of the Format

Below is a view into the format in Augmented Backus-Naur Form [RFC5234] [RFC7405]. It does not cover all details. The end-of-image (EOI) is only implicitly coded into the number of pixels (image_width * image_height).

> Note that `*element` means `element` can be repeated 0 or more times. `5element` means `element` is repeated exactly 5 times. `%b` represents a binary value.

3.8.1. Basic Structure

```
format        = RIFF-header image-header image-stream
RIFF-header   = %s"RIFF" 4OCTET %s"WEBPVP8L" 4OCTET
image-header  = %x2F image-size alpha-is-used version
image-size    = 14BIT 14BIT ; width - 1, height - 1
alpha-is-used = 1BIT
version       = 3BIT ; 0
image-stream  = optional-transform spatially-coded-image
```

3.8.2. Structure of Transforms

```
optional-transform    =  (%b1 transform optional-transform) / %b0
transform             =  predictor-tx / color-tx / subtract-green-tx
transform             =/ color-indexing-tx

predictor-tx          =  %b00 predictor-image
predictor-image       =  3BIT ; sub-pixel code
                         entropy-coded-image

color-tx              =  %b01 color-image
color-image           =  3BIT ; sub-pixel code
                         entropy-coded-image

subtract-green-tx     =  %b10

color-indexing-tx     =  %b11 color-indexing-image
color-indexing-image  =  8BIT ; color count
                         entropy-coded-image
```

3.8.3. Structure of the Image Data

```
spatially-coded-image =  color-cache-info meta-prefix data
entropy-coded-image   =  color-cache-info data

color-cache-info      =  %b0
color-cache-info      =/ (%b1 4BIT) ; 1 followed by color cache size

meta-prefix           =  %b0 / (%b1 entropy-image)

data                  =  prefix-codes lz77-coded-image
entropy-image         =  3BIT ; subsample value
                         entropy-coded-image

prefix-codes          =  prefix-code-group *prefix-codes
prefix-code-group     =
    5prefix-code ; See "Interpretation of Meta Prefix Codes" to
                 ; understand what each of these five prefix
                 ; codes are for.

prefix-code           =  simple-prefix-code / normal-prefix-code
simple-prefix-code    =  ; see "Simple Code Length Code" for details
normal-prefix-code    =  ; see "Normal Code Length Code" for details

lz77-coded-image      =
    *((argb-pixel / lz77-copy / color-cache-code) lz77-coded-image)
```

The following is a possible example sequence:

```
RIFF-header image-size %b1 subtract-green-tx
%b1 predictor-tx %b0 color-cache-info
%b0 prefix-codes lz77-coded-image
```

4. Security Considerations

Implementations of this format face security risks, such as integer overflows, out-of-bounds reads and writes to both heap and stack, uninitialized data usage, null pointer dereferences, resource (disk or memory) exhaustion, and extended resource usage (long running time) as part of the demuxing and decoding process. In particular, implementations reading this format are likely to take input from unknown and possibly unsafe sources -- both clients (for example, web browsers or email clients) and servers (for example, applications that accept uploaded images). These may result in arbitrary code execution, information leakage (memory layout and contents), or crashes and thereby allow a device to be compromised or cause a denial of service to an application using the format [cve.mitre.org-libwebp] [issues-security].

The format does not employ "active content" but does allow metadata (for example, [XMP] and [Exif]) and custom chunks to be embedded in a file. Applications that interpret these chunks may be subject to security considerations for those formats.

5. Interoperability Considerations

The format is defined using little-endian byte ordering (see Section 3.1 of [RFC2781]), but demuxing and decoding are possible on platforms using a different ordering with the appropriate conversion. The container is based on RIFF and allows extension via user-defined chunks, but nothing beyond the chunks defined by the container format (Section 2) are required for decoding of the image. These have been finalized but were extended in the format's early stages, so some older readers may not support lossless or animated image decoding.

6. IANA Considerations

IANA has registered the 'image/webp' media type [RFC2046].

6.1. The 'image/webp' Media Type

This section contains the media type registration details per [RFC6838].

6.1.1. Registration Details

Type name: image

Subtype name: webp

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Binary. The Base64 encoding [RFC4648] should be used on transports that cannot accommodate binary data directly.

Security considerations: See RFC 9649, Section 4.

Interoperability considerations: See RFC 9649, Section 5.

Published specification: RFC 9649

Applications that use this media type: Applications that are used to display and process images, especially when smaller image file sizes are important.

Fragment identifier considerations: N/A

Additional information:

    Deprecated alias names for this type: N/A
    Magic number(s):   The first 4 bytes are 0x52, 0x49, 0x46, 0x46 ('RIFF'), followed by 4 bytes for the 'RIFF' Chunk size. The next 7 bytes are 0x57, 0x45, 0x42, 0x50, 0x56, 0x50, 0x38 ('WEBPVP8').
    File extension(s):   webp
    Apple Uniform Type Identifier: org.webmproject.webp conforms to public.image

Object Identifiers: N/A

Person & email address to contact for further information: James Zern <jzern@google.com>

Intended usage: COMMON

Restrictions on usage: N/A

Author: James Zern <jzern@google.com>

Change controller: IETF

Provisional registration?:   No

7. References

7.1. Normative References

[Exif]   Camera & Imaging Products Association (CIPA) and Japan Electronics and Information Technology Industries Association (JEITA), "Exchangeable image file format for digital still cameras: Exif Version 2.3", CIPA DC-008-2012, JEITA CP-3451C, December 2012, <https://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf>.

[ICC]   International Color Consortium, "Image technology colour management -- Architecture, profile format, and data structure", Profile version 4.3.0.0, REVISION of ICC.1:2004-10, Specification ICC.1:2010, December 2010, <https://www.color.org/specification/ICC1v43_2010-12.pdf>.

[ISO.9899.2018]   International Organization for Standardization, "Information technology -- Programming languages -- C", Fourth Edition, ISO/IEC 9899:2018, June 2018, <https://www.iso.org/standard/74528.html>.

[rec601]   ITU, "Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios", ITU-R Recommendation BT.601, March 2011, <https://www.itu.int/rec/R-REC-BT.601/>.

[RFC1166] Kirkpatrick, S., Stahl, M., and M. Recker, "Internet numbers", RFC 1166, DOI 10.17487/RFC1166, July 1990, <https://www.rfc-editor.org/info/rfc1166>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2781]   Hoffman, P. and F. Yergeau, "UTF-16, an encoding of ISO 10646", RFC 2781, DOI 10.17487/RFC2781, February 2000, <https://www.rfc-editor.org/info/rfc2781>.

[RFC4648]   Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6386]   Bankoski, J., Koleszar, J., Quillio, L., Salonen, J., Wilkins, P., and Y. Xu, "VP8 Data Format and Decoding Guide", RFC 6386, DOI 10.17487/RFC6386, November 2011, <https://www.rfc-editor.org/info/rfc6386>.

[RFC6838]   Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7405]   Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc7405>.

[RFC8174]   Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[XMP]   Adobe Inc., "XMP Specification", <https://www.adobe.com/devnet/xmp.html>.

7.2. Informative References

[cve.mitre.org-libwebp]   "libwebp CVE List", <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libwebp>.

[GIF-spec]   CompuServe Incorporated, "Graphics Interchange Format(sm)", Version 89a, July 1990, <https://www.w3.org/Graphics/GIF/spec-gif89a.txt>.

[Huffman]   Huffman, D., "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the Institute of Radio Engineers, Vol. 40, Issue 9, pp. 1098-1101, DOI 10.1109/JRPROC.1952.273898, September 1952, <https://doi.org/10.1109/JRPROC.1952.273898>.

[issues-security]   "libwebp Security Issues", <https://issues.webmproject.org/issues?q=componentid:1618983%2B%20(%22Restrict-View-Security%22%20OR%20type:vulnerability)>.

[JPEG-spec]   "Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines", ITU-T Recommendation T.81, ISO/IEC 10918-1, September 1992, <https://www.w3.org/Graphics/JPEG/itu-t81.pdf>.

[LZ77]   Ziv, J. and A. Lempel, "A Universal Algorithm for Sequential Data Compression",
            IEEE Transactions on Information Theory, Vol. 23, Issue 3, pp. 337-343, DOI
            10.1109/TIT.1977.1055714, May 1977, <https://doi.org/10.1109/TIT.1977.1055714>.

[MWG]   Metadata Working Group, "Guidelines For Handling Image Metadata", Version
            2.0, November 2010, <https://web.archive.org/web/20180919181934/http://
            www.metadataworkinggroup.org/pdf/mwg_guidance.pdf>.

[RFC2083]   Boutell, T., "PNG (Portable Network Graphics) Specification Version 1.0", RFC
            2083, DOI 10.17487/RFC2083, March 1997, <https://www.rfc-editor.org/info/
            rfc2083>.

[RIFF-spec] "RIFF (Resource Interchange File Format)", <https://www.loc.gov/preservation/
            digital/formats/fdd/fdd000025.shtml>.

[webp-lossless-src]   Alakuijala, J., "WebP Lossless Bitstream Specification", July 2024, <https://
            chromium.googlesource.com/webm/libwebp/+/refs/tags/webp-rfc9649/doc/webp-
            lossless-bitstream-spec.txt>.

[webp-lossless-study]   Alakuijala, J. and V. Rabaud, "Lossless and Transparency Encoding in
            WebP", August 2017, <https://developers.google.com/speed/webp/docs/
            webp_lossless_alpha_study>.

[webp-riff-src]   Google LLC, "WebP RIFF Container", July 2024, <https://
            chromium.googlesource.com/webm/libwebp/+/refs/tags/webp-rfc9649/doc/webp-
            container-spec.txt>.

Authors' Addresses

James Zern
Google LLC
1600 Amphitheatre Parkway
Mountain View, CA 94043
United States of America
Phone: +1 650 253-0000
Email: jzern@google.com

Pascal Massimino
Google LLC
Email: pascal.massimino@gmail.com

Jyrki Alakuijala
Google LLC
Email: jyrki.alakuijala@gmail.com