WebP Lossless Bitstream Specification

developers.google.com/speed/webp/docs/webp_lossless_bitstream_specification

Jyrki Alakuijala, Ph.D., Google, Inc., 2012-06-19

Paragraphs marked as [AMENDED] were amended on 2014-09-16.

Abstract

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 zero alpha pixels. 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. LZ77, Huffman 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.

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 2D proximity.

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

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

LZ77
Dictionary-based sliding window compression algorithm that either emits symbols or describes them as sequences of past symbols.

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

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

prefix coding
A way to entropy code larger integers that 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, top-to-bottom, starting from the left-hand-top pixel, proceeding to the right. Once a row is completed, continue from the left-hand column of the next row.

1 Introduction

This document describes the compressed data representation of a WebP lossless image. It is intended as a detailed reference for WebP lossless encoder and decoder implementation.

In this document, we extensively use C programming language syntax 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 (e.g. 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, an unsigned integer consisting of 32 bits. In the code showing the behavior of the transformations, alpha value is codified in bits 31..24, red in bits 23..16, green in bits 15..8 and blue in bits 7..0, but 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 width and height of the image. A WebP lossless image can go through four different types of transformation before being entropy encoded. The transform information in the bitstream contains the data required to apply the respective inverse transforms.

2 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 block length, the whole size of the block 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 "VP8L" (chunk tag for lossless encoded image data).
5. A little-endian 32-bit value of the number of bytes in the lossless stream.
6. One 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 dynamics for image size limit the maximum size of a WebP lossless image to 16384✕16384 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 should be treated as an error. [AMENDED]

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

3 Transformations

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

An image can go through four types of transformation. A 1 bit indicates the presence of a transform. Each transform is allowed to be used only once. The transformations are used only for the main level ARGB image: the subresolution images 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 (Section 4).
```

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

```
enum TransformType {
  PREDICTOR_TRANSFORM             = 0,
  COLOR_TRANSFORM                 = 1,
  SUBTRACT_GREEN                  = 2,
  COLOR_INDEXING_TRANSFORM        = 3,
};
```

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. Next we describe the transform data for different types.

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 *prediction mode* determines the type of prediction to use. We divide the image into squares and all the pixels in a square use same prediction mode.

The first 3 bits of prediction data define the block width and height in number of bits. The number of block columns, `block_xsize`, is used in indexing two-dimensionally.

```
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 block_xsize = DIV_ROUND_UP(image_width, 1 << size_bits);
```

The transform data contains the prediction mode for each block of the image. All the `block_width * block_height` pixels of a block use same prediction mode. The prediction modes are treated as pixels of an image and encoded using the same techniques described in Chapter 4.

For a pixel *x, y*, one can compute the respective filter block address by:

```
int block_index = (y >> size_bits) * block_xsize +
                  (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 choose 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
```

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

Given the above 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)
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)

Mode	Predicted value of each channel of the current pixel
12	ClampAddSubtractFull(L, T, TL)
13	ClampAddSubtractHalf(Average2(L, T), TL)

`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) {     // [AMENDED]
    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 prediction transform, regardless of the mode [0..13] for these pixels, the predicted value for the left-topmost pixel of the image is 0xff000000, L-pixel for all pixels on the top row, and T-pixel for all pixels on the leftmost column.

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 border, but by using the leftmost pixel on the same row as the current TR-pixel. The TR-pixel offset in memory is the same for border and non-border pixels.

Color Transform

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

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 transformation 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 added during color transform. The inverse color transform then is just subtracting 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
  uint32 tmp_red = red;
  uint32 tmp_blue = blue;

  // Applying transform is just adding 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(). It should be performed using 8-bit two's complement (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 dynamics). The sign extension property of the shift operation does not matter here: only the lowest 8 bits are used from the result, and there 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. `ColorTransformElement` instances are treated as pixels of an image and encoded using the methods described in Chapter 4.

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 subtracting `ColorTransformElement` values from the red and blue channels.

```
void InverseTransform(uint8 red, uint8 green, uint8 blue,
                      ColorTransformElement *p,
                      uint8 *new_red, uint8 *new_blue) {
  // Applying inverse transform is just subtracting the
  // color transform deltas
  red  -= ColorTransformDelta(p->green_to_red_,  green);
  blue -= ColorTransformDelta(p->green_to_blue_, green);
  blue -= ColorTransformDelta(p->red_to_blue_, red & 0xff);

  *new_red = red & 0xff;
  *new_blue = blue & 0xff;
}
```

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 red and blue. 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 it is still often useful. Since it can extend the dynamics of the color transform and there is no additional data here, the subtract green transform can be coded using fewer bits than a full-blown color transform.

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; all red and blue values to 0.

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

```
// 8 bit value for 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 a height of one pixel and a 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 or larger than color_table_size, the argb color value should be set to 0x00000000 (transparent black). [AMENDED]

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 a small number of unique values.

`color_table_size` specifies how many pixels are combined together:

```
int width_bits;
if (color_table_size <= 2) {
  width_bits = 3;
} else if (color_table_size <= 4) {
  width_bits = 2;
} else if (color_table_size <= 16) {
  width_bits = 1;
} else {
  width_bits = 0;
}
```

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

The values are packed into the green component as follows:

- `width_bits` = 1: for every x value where x ≡ 0 (mod 2), a green value at x is positioned into the 4 least-significant bits of the green value at x / 2, 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 ≡ 0 (mod 4), a green value at x is positioned into the 2 least-significant bits of the green value at x / 4, green values at x + 1 to x + 3 in order to the more significant bits of the green value at x / 4.
- `width_bits` = 3: for every x value where x ≡ 0 (mod 8), a green value at x is positioned into the least-significant bit of the green value at x / 8, green values at x + 1 to x + 7 in order to the more significant bits of the green value at x / 8.

4 Image Data

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

4.1 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 Huffman codes. The red and green components of a pixel define the meta Huffman code used in a particular block of the ARGB image.

3. Predictor image: Stores the metadata for Predictor Transform. The green component of a pixel defines which of the 14 predictors is used within a particular block of the ARGB image.

4. Color transform image. It is created by `ColorTransformElement` values (defined in Color Transform) for different blocks of the image. Each `ColorTransformElement` `'cte'` is treated as a pixel 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`.

5. Color indexing image: An array of of size `color_table_size` (up to 256 ARGB values) storing the metadata for the Color Indexing Transform. This is stored as an image of width `color_table_size` and height `1`.

4.2 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. Huffman coded literal: 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; or

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

The following sub-sections describe each of these in detail.

4.2.1 Huffman Coded Literals

The pixel is stored as Huffman coded values of green, red, blue and alpha (in that order). See this section for details.

4.2.2 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.

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, and so extra bits would be used for very few values in the image. Thus, this approach results in a better compression overall.

The following table denotes the prefix codes and extra bits used for storing different range 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

Value range	Prefix code	Extra bits
13..16	7	2
...	...	...
3072..4096	23	10
...	...	...
524289..786432	38	18
786433..1048576	39	18

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;
```

Distance Mapping:

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

The 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 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 `i` 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)
```

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

The decoder can convert a distances code 'i' to a scan-line order distance 'dist' as follows:

```
(xi, yi) = distance_map[i]
dist = x + y * xsize
if (dist < 1) {
  dist = 1
}
```

where 'distance_map' is the mapping noted above and `xsize` is the width of the image in pixels.

4.2.3 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 other two methods (described in 4.2.1 and 4.2.2).

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 Huffman 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 by (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.

5 Entropy Code

5.1 Overview

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

In particular, the format uses **spatially-variant Huffman 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 a better compression.

5.2 Details

The encoded image data consists of two parts:

1. Meta Huffman codes
2. Entropy-coded image data

5.2.1 Decoding of Meta Huffman Codes

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

Meta Huffman codes may be used *only* when the image is being used in the role of an *ARGB image*.

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

- If this bit is zero, there is only one meta Huffman code used everywhere in the image. No more data is stored.

- If this bit is one, the image uses multiple meta Huffman codes. These meta Huffman codes are stored as an *entropy image* (described below).

Entropy image:

The entropy image defines which Huffman codes are used in different parts of the image, as described below.

The first 3-bits contain the `huffman_bits` value. The dimensions of the entropy image are derived from 'huffman_bits'.

```
int huffman_bits = ReadBits(3) + 2;
int huffman_xsize = DIV_ROUND_UP(xsize, 1 << huffman_bits);
int huffman_ysize = DIV_ROUND_UP(ysize, 1 << huffman_bits);
```

where `DIV_ROUND_UP` is as defined earlier.

Next bits contain an entropy image of width `huffman_xsize` and height `huffman_ysize`.

Interpretation of Meta Huffman Codes:

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

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

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

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

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

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

As each Huffman code groups contains five Huffman codes, the total number of Huffman codes is:

```
int num_huff_codes = 5 * num_huff_groups;
```

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

```
int position = (y >> huffman_bits) * huffman_xsize + (x >> huffman_bits);
int meta_huff_code = (entropy_image[pos] >> 8) & 0xffff;
HuffmanCodeGroup huff_group = huffman_code_groups[meta_huff_code];
```

where, we have assumed the existence of `HuffmanCodeGroup` structure, which represents a set of five Huffman codes. Also, `huffman_code_groups` is an array of `HuffmanCodeGroup` (of size `num_huff_groups`).

The decoder then uses Huffman code group `huff_group` to decode the pixel (x, y) as explained in the next section.

5.2.2 Decoding Entropy-coded Image Data

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

Read next symbol S from the bitstream using Huffman code #1. [See next section for details on decoding the Huffman code lengths]. Note that S is any integer in the range `0` to `(256 + 24 + color_cache_size - 1)`.

The interpretation of S depends on its value:

1. if S < 256
    1. Use S as the green component
    2. Read red from the bitstream using Huffman code #2
    3. Read blue from the bitstream using Huffman code #3
    4. Read alpha from the bitstream using Huffman code #4
2. if S < 256 + 24
    1. Use S - 256 as a length prefix code
    2. Read extra bits for length from the bitstream
    3. Determine backward-reference length L from length prefix code and the extra bits read.
    4. Read distance prefix code from the bitstream using Huffman code #5
    5. Read extra bits for distance from the bitstream
    6. Determine backward-reference distance D from distance prefix code and the extra bits read.
    7. Copy the L pixels (in scan-line order) from the sequence of pixels prior to them by D pixels.
3. if S >= 256 + 24
    1. Use S - (256 + 24) as the index into the color cache.
    2. Get ARGB color from the color cache at that index.

Decoding the Code Lengths:

This section describes the details about reading a symbol from the bitstream by decoding the Huffman code length.

The Huffman 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*, and
- If this bit is 0, it is a *normal code length code*.

(i) Simple Code Length Code:

This variant is used in the special case when only 1 or 2 Huffman code lengths are non-zero, and are in the range of [0, 255]. All other Huffman code lengths are implicitly zeros.

The first bit indicates the number of non-zero code lengths:

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

The first code length is stored either using a 1-bit code for values of 0 and 1, or using an 8-bit code for values in range [0, 255]. The second code length, when present, is coded as an 8-bit code.

```
int is_first_8bits = ReadBits(1);
code_lengths[0] = ReadBits(1 + 7 * is_first_8bits);
if (num_code_lengths == 2) {
  code_lengths[1] = ReadBits(8);
}
```

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

(ii) Normal Code Length Code:

The code lengths of a Huffman code are read as follows: `num_code_lengths` specifies the number of code lengths; the rest of the 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_lengths[kCodeLengthCodes] = { 0 };  // All zeros.
int num_code_lengths = 4 + ReadBits(4);
for (i = 0; i < num_code_lengths; ++i) {
  code_lengths[kCodeLengthCodeOrder[i]] = ReadBits(3);
}
```

- Code length 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 non-zero value [3..6] times, i.e., 3 + `ReadBits(2)` times. If code 16 is used before a non-zero value has been emitted, a value of 8 is repeated.
- Code 17 emits a streak of zeros [3..10], i.e., 3 + `ReadBits(3)` times.
- Code 18 emits a streak of zeros of length [11..138], i.e., 11 + `ReadBits(7)` times.

6 Overall Structure of the Format

Below is a view into the format in Backus-Naur form. It does not cover all details. End-of-image (EOI) is only implicitly coded into the number of pixels (xsize * ysize).

Basic Structure

```
<format> ::= <RIFF header><image size><image stream>
<image stream> ::= <optional-transform><spatially-coded image>
```

Structure of Transforms

```
<optional-transform> ::= (1-bit value 1; <transform> <optional-transform>) |
                         1-bit value 0
<transform> ::= <predictor-tx> | <color-tx> | <subtract-green-tx> |
                <color-indexing-tx>
<predictor-tx> ::= 2-bit value 0; <predictor image>
<predictor image> ::= 3-bit sub-pixel code ; <entropy-coded image>
<color-tx> ::= 2-bit value 1; <color image>
<color image> ::= 3-bit sub-pixel code ; <entropy-coded image>
<subtract-green-tx> ::= 2-bit value 2
<color-indexing-tx> ::= 2-bit value 3; <color-indexing image>
<color-indexing image> ::= 8-bit color count; <entropy-coded image>
```

Structure of the Image Data

```
<spatially-coded image> ::= <meta huffman><entropy-coded image>
<entropy-coded image> ::= <color cache info><huffman codes><lz77-coded image>
<meta huffman> ::= 1-bit value 0 |
                   (1-bit value 1; <entropy image>)
<entropy image> ::= 3-bit subsample value; <entropy-coded image>
<color cache info> ::= 1 bit value 0 |
                       (1-bit value 1; 4-bit value for color cache size)
<huffman codes> ::= <huffman code group> | <huffman code group><huffman codes>
<huffman code group> ::= <huffman code><huffman code><huffman code>
                         <huffman code><huffman code>
                         See "Interpretation of Meta Huffman codes" to
                         understand what each of these five Huffman codes are
                         for.
<huffman code> ::= <simple huffman code> | <normal huffman code>
<simple huffman code> ::= see "Simple code length code" for details
<normal huffman code> ::= <code length code>; encoded code lengths
<code length code> ::= see section "Normal code length code"
<lz77-coded image> ::= ((<argb-pixel> | <lz77-copy> | <color-cache-code>)
                       <lz77-coded image>) | ""
```

A possible example sequence:

```
<RIFF header><image size>1-bit value 1<subtract-green-tx>
1-bit value 1<predictor-tx>1-bit value 0<meta huffman>
<color cache info><huffman codes>
<lz77-coded image>
```