ï»¿

FFV1 Video Coding Format Versions 0, 1, and 3

Abstract

   This document defines FFV1, a lossless, intra-frame video encoding
   format. FFV1 is designed to efficiently compress video data in a
   variety of pixel formats. Compared to uncompressed video, FFV1
   offers storage compression, frame fixity, and self-description, which
   makes FFV1 useful as a preservation or intermediate video format.

Status of This Memo

   This document is not an Internet Standards Track specification; it is
   published for informational purposes.

   This document is a product of the Internet Engineering Task Force
   (IETF). It represents the consensus of the IETF community. It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG). Not all documents
   approved by the IESG are candidates for any level of Internet
   Standard; see Section 2 of RFC 7841.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   https://www.rfc-editor.org/info/rfc9043.

Table of Contents

1. Introduction

   This document describes FFV1, a lossless video encoding format. The
   design of FFV1 considers the storage of image characteristics, data
   fixity, and the optimized use of encoding time and storage
   requirements. FFV1 is designed to support a wide range of lossless
   video applications such as long-term audiovisual preservation,
   scientific imaging, screen recording, and other video encoding
   scenarios that seek to avoid the generational loss of lossy video
   encodings.

   This document defines versions 0, 1, and 3 of FFV1. The distinctions
   of the versions are provided throughout the document, but in summary:

   * Version 0 of FFV1 was the original implementation of FFV1 and was
      flagged as stable on April 14, 2006 [FFV1_V0].

   * Version 1 of FFV1 adds support of more video bit depths and was
      flagged as stable on April 24, 2009 [FFV1_V1].

   * Version 2 of FFV1 only existed in experimental form and is not
      described by this document, but it is available as a LyX file at
      <https://github.com/FFmpeg/FFV1/
      blob/8ad772b6d61c3dd8b0171979a2cd9f11924d5532/ffv1.lyx>.

   * Version 3 of FFV1 adds several features such as increased
      description of the characteristics of the encoding images and
      embedded Cyclic Redundancy Check (CRC) data to support fixity
      verification of the encoding. Version 3 was flagged as stable on
      August 17, 2013 [FFV1_V3].

   This document assumes familiarity with mathematical and coding
   concepts such as Range encoding [Range-Encoding] and YCbCr color
   spaces [YCbCr].

   This specification describes the valid bitstream and how to decode
   it. Nonconformant bitstreams and the nonconformant handling of
   bitstreams are outside this specification. A decoder can perform any
   action that it deems appropriate for an invalid bitstream: reject the
   bitstream, attempt to perform error concealment, or re-download or
   use a redundant copy of the invalid part.

2. Notation and Conventions

   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.

2.1. Definitions

   FFV1: The chosen name of this video encoding format, which is the
      short version of "FF Video 1". The letters "FF" come from
      "FFmpeg", which is the name of the reference decoder whose first
      letters originally meant "Fast Forward".

   Container: A format that encapsulates Frames (see Section 4.4) and
      (when required) a "Configuration Record" into a bitstream.

   Sample: The smallest addressable representation of a color component
      or a luma component in a Frame. Examples of Sample are Luma (Y),
      Blue-difference Chroma (Cb), Red-difference Chroma (Cr),

Transparency, Red, Green, and Blue.

Symbol: A value stored in the bitstream, which is defined and
   decoded through one of the methods described in Table 4.

Line: A discrete component of a static image composed of Samples
   that represent a specific quantification of Samples of that image.

Plane: A discrete component of a static image composed of Lines that
   represent a specific quantification of Lines of that image.

Pixel: The smallest addressable representation of a color in a
   Frame. It is composed of one or more Samples.

MSB: Most Significant Bit, the bit that can cause the largest change
   in magnitude of the symbol.

VLC: Variable Length Code, a code that maps source symbols to a
   variable number of bits.

RGB: A reference to the method of storing the value of a pixel by
   using three numeric values that represent Red, Green, and Blue.

YCbCr: A reference to the method of storing the value of a pixel by
   using three numeric values that represent the luma of the pixel
   (Y) and the chroma of the pixel (Cb and Cr). The term YCbCr is
   used for historical reasons and currently references any color
   space relying on one luma Sample and two chroma Samples, e.g.,
   YCbCr (luma, blue-difference chroma, red-difference chroma),
   YCgCo, or ICtCp (intensity, blue-yellow, red-green).

2.2. Conventions

2.2.1. Pseudocode

The FFV1 bitstream is described in this document using pseudocode.
Note that the pseudocode is used to illustrate the structure of FFV1
and is not intended to specify any particular implementation. The
pseudocode used is based upon the C programming language
[ISO.9899.2018] and uses its "if/else", "while", and "for" keywords
as well as functions defined within this document.

In some instances, pseudocode is presented in a two-column format
such as shown in Figure 1. In this form, the "type" column provides
a symbol as defined in Table 4 that defines the storage of the data
referenced in that same line of pseudocode.

```
pseudocode                                                   | type
-------------------------------------------------------------|-----
ExamplePseudoCode( ) {                                       |
    value                                                    | ur
}                                                            |
```

   Figure 1: A depiction of type-labeled pseudocode used within this
                           document.

2.2.2. Arithmetic Operators

Note: the operators and the order of precedence are the same as used
in the C programming language [ISO.9899.2018], with the exception of
">>" (removal of implementation-defined behavior) and "^" (power
instead of XOR) operators, which are redefined within this section.

"a + b" means a plus b.

"a - b" means a minus b.

"-a" means negation of a.

"a * b" means a multiplied by b.

"a / b" means a divided by b.

"a ^ b" means a raised to the b-th power.

"a & b" means bitwise "and" of a and b.

"a | b" means bitwise "or" of a and b.

"a >> b" means arithmetic right shift of the two's complement integer representation of a by b binary digits. This is equivalent to dividing a by 2, b times, with rounding toward negative infinity.

"a << b" means arithmetic left shift of the two's complement integer representation of a by b binary digits.

2.2.3. Assignment Operators

"a = b" means a is assigned b.

"a++" is equivalent to a is assigned a + 1.

"a--" is equivalent to a is assigned a - 1.

"a += b" is equivalent to a is assigned a + b.

"a -= b" is equivalent to a is assigned a - b.

"a *= b" is equivalent to a is assigned a * b.

2.2.4. Comparison Operators

"a > b" is true when a is greater than b.

"a >= b" is true when a is greater than or equal to b.

"a < b" is true when a is less than b.

"a <= b" is true when a is less than or equal b.

"a == b" is true when a is equal to b.

"a != b" is true when a is not equal to b.

"a && b" is true when both a is true and b is true.

"a || b" is true when either a is true or b is true.

"!a" is true when a is not true.

"a ? b : c" if a is true, then b, otherwise c.

2.2.5. Mathematical Functions

"floor(a)" means the largest integer less than or equal to a.

"ceil(a)" means the smallest integer greater than or equal to a.

"sign(a)" extracts the sign of a number, i.e., if a < 0 then -1, else if a > 0 then 1, else 0.

"abs(a)" means the absolute value of a, i.e., "abs(a)" = "sign(a) * a".

"log2(a)" means the base-two logarithm of a.

"min(a,b)" means the smaller of two values a and b.

"max(a,b)" means the larger of two values a and b.

"median(a,b,c)" means the numerical middle value in a data set of a, b, and c, i.e., "a+b+c-min(a,b,c)-max(a,b,c)".

"a ==> b" means a implies b.

"a <==> b" means a ==> b, b ==> a.

"a_b" means the b-th value of a sequence of a.

"a_(b,c)" means the 'b,c'-th value of a sequence of a.

2.2.6. Order of Operation Precedence

When order of precedence is not indicated explicitly by use of
parentheses, operations are evaluated in the following order (from
top to bottom, operations of same precedence being evaluated from
left to right). This order of operations is based on the order of
operations used in Standard C.

```
a++, a--
!a, -a
a ^ b
a * b, a / b
a + b, a - b
a << b, a >> b
a < b, a <= b, a > b, a >= b
a == b, a != b
a & b
a | b
a && b
a || b
a ? b : c
a = b, a += b, a -= b, a *= b
```

2.2.7. Range

"a...b" means any value from a to b, inclusive.

2.2.8. NumBytes

"NumBytes" is a nonnegative integer that expresses the size in 8-bit
octets of a particular FFV1 "Configuration Record" or "Frame". FFV1
relies on its container to store the "NumBytes" values; see
Section 4.3.3.

2.2.9. Bitstream Functions

2.2.9.1. remaining_bits_in_bitstream

"remaining_bits_in_bitstream(NumBytes)" means the count of
remaining bits after the pointer in that "Configuration Record" or
"Frame". It is computed from the "NumBytes" value multiplied by 8
minus the count of bits of that "Configuration Record" or "Frame"
already read by the bitstream parser.

2.2.9.2. remaining_symbols_in_syntax

"remaining_symbols_in_syntax( )" is true as long as the range coder
has not consumed all the given input bytes.

2.2.9.3. byte_aligned

"byte_aligned( )" is true if "remaining_bits_in_bitstream(NumBytes
)" is a multiple of 8, otherwise false.

2.2.9.4. get_bits

"get_bits(i)" is the action to read the next "i" bits in the
bitstream, from most significant bit to least significant bit, and to
return the corresponding value. The pointer is increased by "i".

3. Sample Coding

For each "Slice" (as described in Section 4.5) of a Frame, the
Planes, Lines, and Samples are coded in an order determined by the
color space (see Section 3.7). Each Sample is predicted by the
median predictor as described in Section 3.3 from other Samples
within the same Plane, and the difference is stored using the method
described in Section 3.8.

3.1. Border

A border is assumed for each coded "Slice" for the purpose of the
median predictor and context according to the following rules:

* One column of Samples to the left of the coded Slice is assumed as
   identical to the Samples of the leftmost column of the coded Slice
   shifted down by one row. The value of the topmost Sample of the
   column of Samples to the left of the coded Slice is assumed to be
   "0".

* One column of Samples to the right of the coded Slice is assumed
   as identical to the Samples of the rightmost column of the coded
   Slice.

* An additional column of Samples to the left of the coded Slice and
   two rows of Samples above the coded Slice are assumed to be "0".

Figure 2 depicts a Slice of nine Samples "a,b,c,d,e,f,g,h,i" in a
three-by-three arrangement along with its assumed border.

```
+---+---+---+---+---+---+---+---+
| 0 | 0 |   | 0 | 0 | 0 |   | 0 |
+---+---+---+---+---+---+---+---+
| 0 | 0 |   | 0 | 0 | 0 |   | 0 |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| 0 | 0 |   | a | b | c |   | c |
+---+---+---+---+---+---+---+---+
| 0 | a |   | d | e | f |   | f |
+---+---+---+---+---+---+---+---+
| 0 | d |   | g | h | i |   | i |
+---+---+---+---+---+---+---+---+
```

Figure 2: A depiction of FFV1's assumed border for a set of
example Samples.

3.2. Samples

Relative to any Sample "X", six other relatively positioned Samples
from the coded Samples and presumed border are identified according
to the labels used in Figure 3. The labels for these relatively
positioned Samples are used within the median predictor and context.

```
+---+---+---+---+
|   |   | T |   |
+---+---+---+---+
|   |tl | t |tr |
+---+---+---+---+
| L | l | X |   |
+---+---+---+---+
```

Figure 3: A depiction of how relatively positioned Samples are
referenced within this document.

The labels for these relative Samples are made of the first letters
of the words Top, Left, and Right.

3.3. Median Predictor

The prediction for any Sample value at position "X" may be computed
based upon the relative neighboring values of "l", "t", and "tl" via
this equation:

```
median(l, t, l + t - tl)
```

Note that this prediction template is also used in [ISO.14495-1.1999]
and [HuffYUV].

3.3.1. Exception

If "colorspace_type == 0 && bits_per_raw_sample == 16 && (coder_type
== 1 || coder_type == 2)" (see Sections 4.2.5, 4.2.7, and 4.2.3),
the following median predictor MUST be used:

```
median(left16s, top16s, left16s + top16s - diag16s)
```

where:

```
left16s = l  >= 32768 ? ( l  - 65536 ) : l
top16s  = t  >= 32768 ? ( t  - 65536 ) : t
diag16s = tl >= 32768 ? ( tl - 65536 ) : tl
```

Background: a two's complement 16-bit signed integer was used for
storing Sample values in all known implementations of FFV1 bitstream
(see Appendix C). So in some circumstances, the most significant bit
was wrongly interpreted (used as a sign bit instead of the 16th bit
of an unsigned integer). Note that when the issue was discovered,
the only impacted configuration of all known implementations was the
16-bit YCbCr with no pixel transformation and with the range coder
coder type, as the other potentially impacted configurations (e.g.,
the 15/16-bit JPEG 2000 Reversible Color Transform (RCT)
[ISO.15444-1.2019] with range coder or the 16-bit content with the
Golomb Rice coder type) were not implemented. Meanwhile, the 16-bit
JPEG 2000 RCT with range coder was deployed without this issue in one
implementation and validated by one conformance checker. It is
expected (to be confirmed) that this exception for the median
predictor will be removed in the next version of the FFV1 bitstream.

3.4. Quantization Table Sets

Quantization Tables are used on Sample Differences (see Section 3.8),
so Quantized Sample Differences are stored in the bitstream.

The FFV1 bitstream contains one or more Quantization Table Sets.
Each Quantization Table Set contains exactly five Quantization Tables
with each Quantization Table corresponding to one of the five
Quantized Sample Differences. For each Quantization Table, both the
number of quantization steps and their distribution are stored in the
FFV1 bitstream; each Quantization Table has exactly 256 entries, and
the eight least significant bits of the Quantized Sample Difference
are used as an index:

```
Q_j[k] = quant_tables[i][j][k&255]
```

   Figure 4: Description of the mapping from sample differences to the
              corresponding Quantized Sample Differences.

In this formula, "i" is the Quantization Table Set index, "j" is the
Quantized Table index, and "k" is the Quantized Sample Difference
(see Section 4.1.1).

3.5. Context

Relative to any Sample "X", the Quantized Sample Differences "L-l",
"l-tl", "tl-t", "T-t", and "t-tr" are used as context:

```
context = Q_0[l - tl] +
          Q_1[tl - t] +
          Q_2[t - tr] +
          Q_3[L - l]  +
          Q_4[T - t]
```

        Figure 5: Description of the computing of the Context.

If "context >= 0" then "context" is used, and the difference between
the Sample and its predicted value is encoded as is; else "-context"
is used, and the difference between the Sample and its predicted
value is encoded with a flipped sign.

3.6. Quantization Table Set Indexes

For each Plane of each Slice, a Quantization Table Set is selected
from an index:

* For Y Plane, "quant_table_set_index[0]" index is used.

* For Cb and Cr Planes, "quant_table_set_index[1]" index is used.

* For extra Plane, "quant_table_set_index[(version <= 3 ||
   chroma_planes) ? 2 : 1]" index is used.

Background: in the first implementations of the FFV1 bitstream, the
index for Cb and Cr Planes was stored even if it was not used
("chroma_planes" set to 0), this index is kept for "version <= 3" in
order to keep compatibility with FFV1 bitstreams in the wild.

3.7. Color Spaces

FFV1 supports several color spaces. The count of allowed coded
Planes and the meaning of the extra Plane are determined by the
selected color space.

The FFV1 bitstream interleaves data in an order determined by the
color space. In YCbCr for each Plane, each Line is coded from top to
bottom, and for each Line, each Sample is coded from left to right.
In JPEG 2000 RCT for each Line from top to bottom, each Plane is
coded, and for each Plane, each Sample is encoded from left to right.

3.7.1. YCbCr

This color space allows one to four Planes.

The Cb and Cr Planes are optional, but if they are used, then they
MUST be used together. Omitting the Cb and Cr Planes codes the
frames in gray scale without color data.

An optional transparency Plane can be used to code transparency data.

An FFV1 Frame using YCbCr MUST use one of the following arrangements:

* Y

* Y, Transparency

* Y, Cb, Cr

* Y, Cb, Cr, Transparency

The Y Plane MUST be coded first. If the Cb and Cr Planes are used,
then they MUST be coded after the Y Plane. If a transparency Plane
is used, then it MUST be coded last.

3.7.2. RGB

This color space allows three or four Planes.

An optional transparency Plane can be used to code transparency data.

JPEG 2000 RCT is a Reversible Color Transform that codes RGB (Red,
Green, Blue) Planes losslessly in a modified YCbCr color space
[ISO.15444-1.2019]. Reversible pixel transformations between YCbCr
and RGB use the following formulae:

Cb = b - g

```
Cr = r - g
Y = g + (Cb + Cr) >> 2
```

      Figure 6: Description of the transformation of pixels from RGB
            color space to coded, modified YCbCr color space.

```
g = Y - (Cb + Cr) >> 2
r = Cr + g
b = Cb + g
```

   Figure 7: Description of the transformation of pixels from coded,
            modified YCbCr color space to RGB color space.

Cb and Cr are positively offset by "1 << bits_per_raw_sample" after
the conversion from RGB to the modified YCbCr, and they are
negatively offset by the same value before the conversion from the
modified YCbCr to RGB in order to have only nonnegative values after
the conversion.

When FFV1 uses the JPEG 2000 RCT, the horizontal Lines are
interleaved to improve caching efficiency since it is most likely
that the JPEG 2000 RCT will immediately be converted to RGB during
decoding. The interleaved coding order is also Y, then Cb, then Cr,
and then, if used, transparency.

As an example, a Frame that is two pixels wide and two pixels high
could comprise the following structure:

```
+------------------------+------------------------+
| Pixel(1,1)             | Pixel(2,1)             |
| Y(1,1) Cb(1,1) Cr(1,1) | Y(2,1) Cb(2,1) Cr(2,1) |
+------------------------+------------------------+
| Pixel(1,2)             | Pixel(2,2)             |
| Y(1,2) Cb(1,2) Cr(1,2) | Y(2,2) Cb(2,2) Cr(2,2) |
+------------------------+------------------------+
```

In JPEG 2000 RCT, the coding order is left to right and then top to
bottom, with values interleaved by Lines and stored in this order:

Y(1,1) Y(2,1) Cb(1,1) Cb(2,1) Cr(1,1) Cr(2,1) Y(1,2) Y(2,2) Cb(1,2)
Cb(2,2) Cr(1,2) Cr(2,2)

3.7.2.1. RGB Exception

If "bits_per_raw_sample" is between 9 and 15 inclusive and
"extra_plane" is 0, the following formulae for reversible conversions
between YCbCr and RGB MUST be used instead of the ones above:

```
Cb = g - b
Cr = r - b
Y = b + (Cb + Cr) >> 2
```

      Figure 8: Description of the transformation of pixels from RGB
        color space to coded, modified YCbCr color space (in case of
                              exception).

```
b = Y - (Cb + Cr) >> 2
r = Cr + b
g = Cb + b
```

   Figure 9: Description of the transformation of pixels from coded,
        modified YCbCr color space to RGB color space (in case of
                              exception).

Background: At the time of this writing, in all known implementations
of the FFV1 bitstream, when "bits_per_raw_sample" was between 9 and
15 inclusive and "extra_plane" was 0, Green Blue Red (GBR) Planes
were used as Blue Green Red (BGR) Planes during both encoding and
decoding. Meanwhile, 16-bit JPEG 2000 RCT was implemented without
this issue in one implementation and validated by one conformance
checker. Methods to address this exception for the transform are

under consideration for the next version of the FFV1 bitstream.

3.8. Coding of the Sample Difference

Instead of coding the n+1 bits of the Sample Difference with Huffman or Range coding (or n+2 bits, in the case of JPEG 2000 RCT), only the n (or n+1, in the case of JPEG 2000 RCT) least significant bits are used, since this is sufficient to recover the original Sample. In Figure 10, the term "bits" represents "bits_per_raw_sample + 1" for JPEG 2000 RCT or "bits_per_raw_sample" otherwise:

```
coder_input = ((sample_difference + 2 ^ (bits - 1)) &
               (2 ^ bits - 1)) - 2 ^ (bits - 1)
```

Figure 10: Description of the coding of the Sample Difference in
the bitstream.

3.8.1. Range Coding Mode

Early experimental versions of FFV1 used the Context-Adaptive Binary Arithmetic Coding (CABAC) coder from H.264 as defined in [ISO.14496-10.2020], but due to the uncertain patent/royalty situation, as well as its slightly worse performance, CABAC was replaced by a range coder based on an algorithm defined by G. Nigel N. Martin in 1979 [Range-Encoding].

3.8.1.1. Range Binary Values

To encode binary digits efficiently, a range coder is used. A range coder encodes a series of binary symbols by using a probability estimation within each context. The sizes of each of the two subranges are proportional to their estimated probability. The Quantization Table is used to choose the context used from the surrounding image sample values for the case of coding the Sample Differences. The coding of integers is done by coding multiple binary values. The range decoder will read bytes until it can determine into which subrange the input falls to return the next binary symbol.

To describe Range coding for FFV1, the following values are used:

C_i the i-th context.

B_i the i-th byte of the bytestream.

R_i the Range at the i-th symbol.

r_i the boundary between two subranges of R_i: a subrange of r_i values and a subrange $R_i - r_i$ values.

L_i the Low value of the Range at the i-th symbol.

l_i a temporary variable to carry over or adjust the Low value of the Range between range coding operations.

t_i a temporary variable to transmit subranges between range coding operations.

b_i the i-th range-coded binary value.

$S_{(0, i)}$ the i-th initial state.

j_n the length of the bytestream encoding n binary symbols.

The following range coder state variables are initialized to the following values. The Range is initialized to a value of 65,280 (expressed in base 16 as 0xFF00) as depicted in Figure 11. The Low is initialized according to the value of the first two bytes as depicted in Figure 12. j_i tracks the length of the bytestream encoding while incrementing from an initial value of j_0 to a final value of j_n. j_0 is initialized to 2 as depicted in Figure 13.

$R\_0 = 65280$

Figure 11: The initial value for the Range.

$L\_0 = 2 \verb|^| 8 * B\_0 + B\_1$

Figure 12: The initial value for Low is set according to the
first two bytes of the bytestream.

$j\_0 = 2$

Figure 13: The initial value for "j", the length of the
bytestream encoding.

The following equations define how the range coder variables evolve
as it reads or writes symbols.

$r\_i = floor((R\_i * S\_(i, C\_i)) / 2 \verb|^| 8)$

Figure 14: This formula shows the positioning of range split
based on the state.

```
         b_i =  0                         <==>
         L_i <  R_i - r_i                 ==>
 S_(i+1,C_i) =  zero_state_(S_(i, C_i))   AND
         l_i =  L_i                       AND
         t_i =  R_i - r_i

         b_i =  1                         <==>
         L_i >= R_i - r_i                 ==>
 S_(i+1,C_i) =  one_state_(S_(i, C_i))    AND
         l_i =  L_i - R_i + r_i           AND
         t_i =  r_i
```

Figure 15: This formula shows the linking of the decoded symbol
(represented as b_i), the updated state (represented as
S_(i+1,C_i)), and the updated range (represented as a range from
l_i to t_i).

$C\_i != k ==> S\_(i + 1, k) = S\_(i, k)$

Figure 16: If the value of "k" is unequal to the i-th value of
context, in other words, if the state is unchanged from the last
symbol coding, then the value of the state is carried over to the
next symbol coding.

```
t_i         <  2 ^ 8                      ==>
R_(i + 1) =  2 ^ 8 * t_i                  AND
L_(i + 1) =  2 ^ 8 * l_i + B_(j_i)        AND
j_(i + 1) =  j_i + 1

t_i         >= 2 ^ 8                      ==>
R_(i + 1) =  t_i                          AND
L_(i + 1) =  l_i                          AND
j_(i + 1) =  j_i
```

Figure 17: This formula shows the linking of the range coder with
the reading or writing of the bytestream.

```
    range = 0xFF00;
    end   = 0;
    low   = get_bits(16);
    if (low >= range) {
        low = range;
        end = 1;
    }
```

Figure 18: A pseudocode description of the initialization of
range coder variables in Range binary mode.

```
refill() {
    if (range < 256) {
        range = range * 256;
        low   = low * 256;
        if (!end) {
            c.low += get_bits(8);
            if (remaining_bits_in_bitstream( NumBytes ) == 0) {
                end = 1;
            }
        }
    }
}
```

Figure 19: A pseudocode description of refilling the binary value
buffer of the range coder.

```
get_rac(state) {
    rangeoff  = (range * state) / 256;
    range    -= rangeoff;
    if (low < range) {
        state = zero_state[state];
        refill();
        return 0;
    } else {
        low   -= range;
        state  = one_state[state];
        range  = rangeoff;
        refill();
        return 1;
    }
}
```

Figure 20: A pseudocode description of the read of a binary value
in Range binary mode.

3.8.1.1.1. Termination

The range coder can be used in three modes:

* In Open mode when decoding, every symbol the reader attempts to
   read is available. In this mode, arbitrary data can have been
   appended without affecting the range coder output. This mode is
   not used in FFV1.

* In Closed mode, the length in bytes of the bytestream is provided
   to the range decoder. Bytes beyond the length are read as 0 by
   the range decoder. This is generally one byte shorter than the
   Open mode.

* In Sentinel mode, the exact length in bytes is not known, and thus
   the range decoder MAY read into the data that follows the range-
   coded bytestream by one byte. In Sentinel mode, the end of the
   range-coded bytestream is a binary symbol with state 129, which
   value SHALL be discarded. After reading this symbol, the range
   decoder will have read one byte beyond the end of the range-coded
   bytestream. This way the byte position of the end can be
   determined. Bytestreams written in Sentinel mode can be read in
   Closed mode if the length can be determined. In this case, the
   last (sentinel) symbol will be read uncorrupted and be of value 0.

The above describes the range decoding. Encoding is defined as any
process that produces a decodable bytestream.

There are three places where range coder termination is needed in
FFV1. The first is in the "Configuration Record", which in this case
the size of the range-coded bytestream is known and handled as Closed
mode. The second is the switch from the "Slice Header", which is
range coded to Golomb-coded Slices as Sentinel mode. The third is
the end of range-coded Slices, which need to terminate before the CRC
at their end. This can be handled as Sentinel mode or as Closed mode
if the CRC position has been determined.

3.8.1.2. Range Nonbinary Values

To encode scalar integers, it would be possible to encode each bit
separately and use the past bits as context. However, that would
mean 255 contexts per 8-bit symbol, which is not only a waste of
memory but also requires more past data to reach a reasonably good
estimate of the probabilities. Alternatively, it would also be
possible to assume a Laplacian distribution and only deal with its
variance and mean (as in Huffman coding). However, for maximum
flexibility and simplicity, the chosen method uses a single symbol to
encode if a number is 0, and if the number is nonzero, it encodes the
number using its exponent, mantissa, and sign. The exact contexts
used are best described by Figure 21.

```
int get_symbol(RangeCoder *c, uint8_t *state, int is_signed) {
    if (get_rac(c, state + 0) {
        return 0;
    }

    int e = 0;
    while (get_rac(c, state + 1 + min(e, 9)) { //1..10
        e++;
    }

    int a = 1;
    for (int i = e - 1; i >= 0; i--) {
        a = a * 2 + get_rac(c, state + 22 + min(i, 9));   // 22..31
    }

    if (!is_signed) {
        return a;
    }

    if (get_rac(c, state + 11 + min(e, 10))) { //11..21
        return -a;
    } else {
        return a;
    }
}
```

Figure 21: A pseudocode description of the contexts of Range
nonbinary values.

"get_symbol" is used for the read out of "sample_difference"
indicated in Figure 10.

"get_rac" returns a boolean computed from the bytestream as described
by the formula found in Figure 14 and by the pseudocode found in
Figure 20.

3.8.1.3. Initial Values for the Context Model

When the "keyframe" value (see Section 4.4) is 1, all range coder
state variables are set to their initial state.

3.8.1.4. State Transition Table

In Range Coding Mode, a state transition table is used, indicating to
which state the decoder will move based on the current state and the
value extracted from Figure 20.

```
one_state_i =
        default_state_transition_i + state_transition_delta_i
```

Figure 22: Description of the coding of the state transition
table for a "get_rac" readout value of 1.

```
zero_state_i = 256 - one_state_(256-i)
```

Figure 23: Description of the coding of the state transition

3.8.1.5. default_state_transition

By default, the following state transition table is used:

```
  0,  0,  0,  0,  0,  0,  0,  0, 20, 21, 22, 23, 24, 25, 26, 27,
 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 37, 38, 39, 40, 41, 42,
 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 56, 57,
 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
 74, 75, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
 89, 90, 91, 92, 93, 94, 94, 95, 96, 97, 98, 99,100,101,102,103,
104,105,106,107,108,109,110,111,112,113,114,114,115,116,117,118,
119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,133,
134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,
150,151,152,152,153,154,155,156,157,158,159,160,161,162,163,164,
165,166,167,168,169,170,171,171,172,173,174,175,176,177,178,179,
180,181,182,183,184,185,186,187,188,189,190,190,191,192,194,194,
195,196,197,198,199,200,201,202,202,204,205,206,207,208,209,209,
210,211,212,213,215,215,216,217,218,219,220,220,222,223,224,225,
226,227,227,229,229,230,231,232,234,234,235,236,237,238,239,240,
241,242,243,244,245,246,247,248,248,  0,  0,  0,  0,  0,  0,  0,
```

Figure 24: Default state transition table for Range coding.

3.8.1.6. Alternative State Transition Table

The alternative state transition table has been built using iterative
minimization of frame sizes and generally performs better than the
default. To use it, the "coder_type" (see Section 4.2.3) MUST be set
to 2, and the difference to the default MUST be stored in the
"Parameters", see Section 4.2. At the time of this writing, the
reference implementation of FFV1 in FFmpeg uses Figure 25 by default
when Range coding is used.

```
  0, 10, 10, 10, 10, 16, 16, 16, 28, 16, 16, 29, 42, 49, 20, 49,
 59, 25, 26, 26, 27, 31, 33, 33, 33, 34, 34, 37, 67, 38, 39, 39,
 40, 40, 41, 79, 43, 44, 45, 45, 48, 48, 64, 50, 51, 52, 88, 52,
 53, 74, 55, 57, 58, 58, 74, 60,101, 61, 62, 84, 66, 66, 68, 69,
 87, 82, 71, 97, 73, 73, 82, 75,111, 77, 94, 78, 87, 81, 83, 97,
 85, 83, 94, 86, 99, 89, 90, 99,111, 92, 93,134, 95, 98,105, 98,
105,110,102,108,102,118,103,106,106,113,109,112,114,112,116,125,
115,116,117,117,126,119,125,121,121,123,145,124,126,131,127,129,
165,130,132,138,133,135,145,136,137,139,146,141,143,142,144,148,
147,155,151,149,151,150,152,157,153,154,156,168,158,162,161,160,
172,163,169,164,166,184,167,170,177,174,171,173,182,176,180,178,
```

```
175,189,179,181,186,183,192,185,200,187,191,188,190,197,193,196,

197,194,195,196,198,202,199,201,210,203,207,204,205,206,208,214,

209,211,221,212,213,215,224,216,217,218,219,220,222,228,223,225,

226,224,227,229,240,230,231,232,233,234,235,236,238,239,237,242,

241,243,242,244,245,246,247,248,249,250,251,252,252,253,254,255,
```

        Figure 25: Alternative state transition table for Range coding.

3.8.2. Golomb Rice Mode

   The end of the bitstream of the Frame is padded with zeroes until the
   bitstream contains a multiple of eight bits.

3.8.2.1. Signed Golomb Rice Codes

   This coding mode uses Golomb Rice codes. The VLC is split into two
   parts: the prefix and suffix. The prefix stores the most significant
   bits or indicates if the symbol is too large to be stored (this is
   known as the ESC case, see Section 3.8.2.1.1). The suffix either
   stores the k least significant bits or stores the whole number in the
   ESC case.

```
int get_ur_golomb(k) {
    for (prefix = 0; prefix < 12; prefix++) {
        if (get_bits(1)) {
            return get_bits(k) + (prefix << k);
        }
    }
    return get_bits(bits) + 11;
}
```

        Figure 26: A pseudocode description of the read of an unsigned
                      integer in Golomb Rice mode.

```
int get_sr_golomb(k) {
    v = get_ur_golomb(k);
    if (v & 1) return − (v >> 1) − 1;
    else       return   (v >> 1);
}
```

         Figure 27: A pseudocode description of the read of a signed
                      integer in Golomb Rice mode.

3.8.2.1.1. Prefix

bits	value
1	0
01	1
...	...
0000 0000 01	9
0000 0000 001	10
0000 0000 0001	11
0000 0000 0000	ESC

                 Table 1: Description
                 of the coding of the
                   prefix of signed

Golomb Rice codes.

   ESC is an ESCape symbol to indicate that the symbol to be stored is
   too large for normal storage and that an alternate storage method is
   used.

3.8.2.1.2. Suffix

         +---------+-------------------------------------+
         | non-ESC | the k least significant bits MSB first |
         +---------+-------------------------------------+
         | ESC | the value - 11, in MSB first order |
         +---------+-------------------------------------+

              Table 2: Description of the coding of the suffix
                       of signed Golomb Rice codes.

   ESC MUST NOT be used if the value can be coded as non-ESC.

3.8.2.1.3. Examples

   Table 3 shows practical examples of how signed Golomb Rice codes are
   decoded based on the series of bits extracted from the bitstream as
   described by the method above:

         +=====+=====================+=======+
         | k | bits | value |
         +=====+=====================+=======+
         | 0 | 1 | 0 |
         +-----+---------------------+-------+
         | 0 | 001 | 2 |
         +-----+---------------------+-------+
         | 2 | 1 00 | 0 |
         +-----+---------------------+-------+
         | 2 | 1 10 | 2 |
         +-----+---------------------+-------+
         | 2 | 01 01 | 5 |
         +-----+---------------------+-------+
         | any | 000000000000 10000000 | 139 |
         +-----+---------------------+-------+

                      Table 3: Examples of decoded,
                         signed Golomb Rice codes.

3.8.2.2. Run Mode

   Run mode is entered when the context is 0 and left as soon as a
   nonzero difference is found. The Sample Difference is identical to
   the predicted one. The run and the first different Sample Difference
   are coded as defined in Section 3.8.2.4.1.

3.8.2.2.1. Run Length Coding

   The run value is encoded in two parts. The prefix part stores the
   more significant part of the run as well as adjusting the "run_index"
   that determines the number of bits in the less significant part of
   the run. The second part of the value stores the less significant
   part of the run as it is. The "run_index" is reset to zero for each
   Plane and Slice.

   log2_run[41] = {
    0, 0, 0, 0, 1, 1, 1, 1,
    2, 2, 2, 2, 3, 3, 3, 3,
    4, 4, 5, 5, 6, 6, 7, 7,
    8, 9,10,11,12,13,14,15,
   16,17,18,19,20,21,22,23,
   24,
   };

   if (run_count == 0 && run_mode == 1) {
       if (get_bits(1)) {

```
            run_count = 1 << log2_run[run_index];
            if (x + run_count <= w) {
                run_index++;
            }
        } else {
            if (log2_run[run_index]) {
                run_count = get_bits(log2_run[run_index]);
            } else {
                run_count = 0;
            }
            if (run_index) {
                run_index--;
            }
            run_mode = 2;
        }
    }
```

The "log2_run" array is also used within [ISO.14495-1.1999].

3.8.2.3. Sign Extension

"sign_extend" is the function of increasing the number of bits of an
input binary number in two's complement signed number representation
while preserving the input number's sign (positive/negative) and
value, in order to fit in the output bit width. It MAY be computed
with the following:

```
sign_extend(input_number, input_bits) {
    negative_bias = 1 << (input_bits - 1);
    bits_mask = negative_bias - 1;
    output_number = input_number & bits_mask; // Remove negative bit
    is_negative = input_number & negative_bias; // Test negative bit
    if (is_negative)
        output_number -= negative_bias;
    return output_number
}
```

3.8.2.4. Scalar Mode

Each difference is coded with the per context mean prediction removed
and a per context value for "k".

```
get_vlc_symbol(state) {
    i = state->count;
    k = 0;
    while (i < state->error_sum) {
        k++;
        i += i;
    }

    v = get_sr_golomb(k);

    if (2 * state->drift < -state->count) {
        v = -1 - v;
    }

    ret = sign_extend(v + state->bias, bits);

    state->error_sum += abs(v);
    state->drift     += v;

    if (state->count == 128) {
        state->count     >>= 1;
        state->drift     >>= 1;
        state->error_sum >>= 1;
    }
    state->count++;
    if (state->drift <= -state->count) {
        state->bias = max(state->bias - 1, -128);

        state->drift = max(state->drift + state->count,
```

```
                         -state->count + 1);
       } else if (state->drift > 0) {
           state->bias = min(state->bias + 1, 127);

           state->drift = min(state->drift - state->count, 0);
       }

       return ret;
   }
```

3.8.2.4.1. Golomb Rice Sample Difference Coding

   Level coding is identical to the normal difference coding with the
   exception that the 0 value is removed as it cannot occur:

```
       diff = get_vlc_symbol(context_state);
       if (diff >= 0) {
           diff++;
       }
```

   Note that this is different from JPEG-LS (lossless JPEG), which
   doesn't use prediction in run mode and uses a different encoding and
   context model for the last difference. On a small set of test
   Samples, the use of prediction slightly improved the compression
   rate.

3.8.2.5. Initial Values for the VLC Context State

   When "keyframe" (see Section 4.4) value is 1, all VLC coder state
   variables are set to their initial state.

```
       drift     = 0;
       error_sum = 4;
       bias      = 0;
       count     = 1;
```

4. Bitstream

   An FFV1 bitstream is composed of a series of one or more Frames and
   (when required) a "Configuration Record".

   Within the following subsections, pseudocode as described in
   Section 2.2.1 is used to explain the structure of each FFV1 bitstream
   component. Table 4 lists symbols used to annotate that pseudocode in
   order to define the storage of the data referenced in that line of
   pseudocode.

symbol	definition
u(n)	Unsigned, big-endian integer symbol using n bits
br	Boolean (1-bit) symbol that is range coded with the method described in Section 3.8.1.1
ur	Unsigned scalar symbol that is range coded with the method described in Section 3.8.1.2
sr	Signed scalar symbol that is range coded with the method described in Section 3.8.1.2
sd	Sample Difference symbol that is coded with the method described in Section 3.8

       Table 4: Definition of pseudocode symbols for this document.

   The following MUST be provided by external means during the
   initialization of the decoder:

   "frame_pixel_width" is defined as Frame width in pixels.

"frame_pixel_height" is defined as Frame height in pixels.

Default values at the decoder initialization phase:

"ConfigurationRecordIsPresent" is set to 0.

4.1. Quantization Table Set

   The Quantization Table Sets store a sequence of values that are equal
   to one less than the count of equal concurrent entries for each set
   of equal concurrent entries within the first half of the table
   (represented as "len - 1" in the pseudocode below) using the method
   described in Section 3.8.1.2. The second half doesn't need to be
   stored as it is identical to the first with flipped sign. "scale" and
   "len_count[i][j]" are temporary values used for the computing of
   "context_count[i]" and are not used outside Quantization Table Set
   pseudocode.

   Example:

   Table: 0 0 1 1 1 1 2 2 -2 -2 -2 -1 -1 -1 -1 0

   Stored values: 1, 3, 1

   "QuantizationTableSet" has its own initial states, all set to 128.

```
pseudocode                                                    | type
--------------------------------------------------------------|-----
QuantizationTableSet( i ) {                                   |
    scale = 1                                                 |
    for (j = 0; j < MAX_CONTEXT_INPUTS; j++) {                |
        QuantizationTable( i, j, scale )                      |
        scale *= 2 * len_count[ i ][ j ] - 1                  |
    }                                                         |
    context_count[ i ] = ceil( scale / 2 )                    |
}                                                             |
```

   "MAX_CONTEXT_INPUTS" is 5.

```
pseudocode                                                   | type
-------------------------------------------------------------|-----
QuantizationTable(i, j, scale) {                             |
    v = 0                                                    |
    for (k = 0; k < 128;) {                                  |
        len - 1                                              | ur
        for (n = 0; n < len; n++) {                          |
            quant_tables[ i ][ j ][ k ] = scale * v          |
            k++                                              |
        }                                                    |
        v++                                                  |
    }                                                        |
    for (k = 1; k < 128; k++) {                              |
        quant_tables[ i ][ j ][ 256 - k ] = \                |
        -quant_tables[ i ][ j ][ k ]                         |
    }                                                        |
    quant_tables[ i ][ j ][ 128 ] = \                        |
    -quant_tables[ i ][ j ][ 127 ]                           |
    len_count[ i ][ j ] = v                                  |
}                                                            |
```

4.1.1. "quant_tables"

   "quant_tables[i][j][k]" indicates the Quantization Table value
   of the Quantized Sample Difference "k" of the Quantization Table "j"
   of the Quantization Table Set "i".

4.1.2. "context_count"

   "context_count[i]" indicates the count of contexts for Quantization
   Table Set "i". "context_count[i]" MUST be less than or equal to

32768.

4.2. Parameters

The "Parameters" section, which could be in a global header of a
container file that may or may not be considered to be part of the
bitstream, contains significant characteristics about the decoding
configuration used for all instances of Frame (in FFV1 versions 0 and
1) or the whole FFV1 bitstream (other versions), including the stream
version, color configuration, and Quantization Tables. Figure 28
describes the contents of the bitstream.

"Parameters" has its own initial states, all set to 128.

```
pseudocode                                                     | type
---------------------------------------------------------------|-----
Parameters( ) {                                                |
    version                                                    | ur
    if (version >= 3) {                                        |
        micro_version                                          | ur
    }                                                          |
    coder_type                                                 | ur
    if (coder_type > 1) {                                      |
        for (i = 1; i < 256; i++) {                            |
            state_transition_delta[ i ]                        | sr
        }                                                      |
    }                                                          |
    colorspace_type                                            | ur
    if (version >= 1) {                                        |
        bits_per_raw_sample                                    | ur
    }                                                          |
    chroma_planes                                              | br
    log2_h_chroma_subsample                                    | ur
    log2_v_chroma_subsample                                    | ur
    extra_plane                                                | br
    if (version >= 3) {                                        |
        num_h_slices - 1                                       | ur
        num_v_slices - 1                                       | ur
        quant_table_set_count                                  | ur
    }                                                          |
    for (i = 0; i < quant_table_set_count; i++) {              |
        QuantizationTableSet( i )                              |
    }                                                          |
    if (version >= 3) {                                        |
        for (i = 0; i < quant_table_set_count; i++) {          |
            states_coded                                       | br
            if (states_coded) {                                |
                for (j = 0; j < context_count[ i ]; j++) {     |
                    for (k = 0; k < CONTEXT_SIZE; k++) {       |
                        initial_state_delta[ i ][ j ][ k ]     | sr
                    }                                          |
                }                                              |
            }                                                  |
        }                                                      |
        ec                                                     | ur
        intra                                                  | ur
    }                                                          |
}                                                              |
```

        Figure 28: A pseudocode description of the bitstream contents.

    CONTEXT_SIZE is 32.

4.2.1. "version"

"version" specifies the version of the FFV1 bitstream.

Each version is incompatible with other versions: decoders SHOULD
reject FFV1 bitstreams due to an unknown version.

Decoders SHOULD reject FFV1 bitstreams with "version <= 1 &&

ConfigurationRecordIsPresent == 1".

Decoders SHOULD reject FFV1 bitstreams with "version >= 3 &&
ConfigurationRecordIsPresent == 0".

```
+=======+=========================+
| value | version                 |
+=======+=========================+
| 0     | FFV1 version 0          |
+-------+-------------------------+
| 1     | FFV1 version 1          |
+-------+-------------------------+
| 2     | reserved*               |
+-------+-------------------------+
| 3     | FFV1 version 3          |
+-------+-------------------------+
| Other | reserved for future use |
+-------+-------------------------+
```

Table 5: The definitions for
"version" values.

* Version 2 was experimental and this document does not describe it.

4.2.2. "micro_version"

"micro_version" specifies the micro-version of the FFV1 bitstream.

After a version is considered stable (a micro-version value is
assigned to be the first stable variant of a specific version), each
new micro-version after this first stable variant is compatible with
the previous micro-version: decoders SHOULD NOT reject FFV1
bitstreams due to an unknown micro-version equal or above the micro-
version considered as stable.

Meaning of "micro_version" for "version" 3:

```
+=======+=========================+
| value | micro_version           |
+=======+=========================+
| 0...3 | reserved*               |
+-------+-------------------------+
| 4     | first stable variant    |
+-------+-------------------------+
| Other | reserved for future use |
+-------+-------------------------+
```

Table 6: The definitions for
"micro_version" values for FFV1
version 3.

* Development versions may be incompatible with the stable variants.

4.2.3. "coder_type"

"coder_type" specifies the coder used.

```
+=======+=================================================+
| value | coder used                                      |
+=======+=================================================+
| 0     | Golomb Rice                                     |
+-------+-------------------------------------------------+
| 1     | Range coder with default state transition table |
+-------+-------------------------------------------------+
| 2     | Range coder with custom state transition table  |
+-------+-------------------------------------------------+
| Other | reserved for future use                         |
+-------+-------------------------------------------------+
```

Table 7: The definitions for "coder_type" values.

Restrictions:

If "coder_type" is 0, then "bits_per_raw_sample" SHOULD NOT be > 8.

Background: At the time of this writing, there is no known implementation of FFV1 bitstream supporting the Golomb Rice algorithm with "bits_per_raw_sample" greater than eight, and range coder is preferred.

4.2.4. "state_transition_delta"

"state_transition_delta" specifies the range coder custom state transition table.

If "state_transition_delta" is not present in the FFV1 bitstream, all range coder custom state transition table elements are assumed to be 0.

4.2.5. "colorspace_type"

"colorspace_type" specifies the color space encoded, the pixel transformation used by the encoder, the extra Plane content, as well as interleave method.

value	color space encoded	pixel transformation	extra Plane content	interleave method
0	YCbCr	None	Transparency	Plane then Line
1	RGB	JPEG 2000 RCT	Transparency	Line then Plane
Other	reserved for future use	reserved for future use	reserved for future use	reserved for future use

Table 8: The definitions for "colorspace_type" values.

FFV1 bitstreams with "colorspace_type == 1 && (chroma_planes != 1 || log2_h_chroma_subsample != 0 || log2_v_chroma_subsample != 0)" are not part of this specification.

4.2.6. "chroma_planes"

"chroma_planes" indicates if chroma (color) Planes are present.

value	presence
0	chroma Planes are not present
1	chroma Planes are present

Table 9: The definitions for "chroma_planes" values.

4.2.7. "bits_per_raw_sample"

"bits_per_raw_sample" indicates the number of bits for each Sample. Inferred to be 8 if not present.

value	bits for each Sample
0	reserved*
Other	the actual bits for each Sample

```
                    +-------+-------------------------------+
```

                     Table 10: The definitions for
                     "bits_per_raw_sample" values.

   * Encoders MUST NOT store "bits_per_raw_sample = 0". Decoders SHOULD
   accept and interpret "bits_per_raw_sample = 0" as 8.

4.2.8. "log2_h_chroma_subsample"

   "log2_h_chroma_subsample" indicates the subsample factor, stored in
   powers to which the number 2 is raised, between luma and chroma width
   ("chroma_width = 2 ^ -log2_h_chroma_subsample * luma_width").

4.2.9. "log2_v_chroma_subsample"

   "log2_v_chroma_subsample" indicates the subsample factor, stored in
   powers to which the number 2 is raised, between luma and chroma
   height ("chroma_height = 2 ^ -log2_v_chroma_subsample *
   luma_height").

4.2.10. "extra_plane"

   "extra_plane" indicates if an extra Plane is present.

```
                +=======+============================+
                | value | presence                   |
                +=======+============================+
                | 0     | extra Plane is not present |
                +-------+----------------------------+
                | 1     | extra Plane is present     |
                +-------+----------------------------+
```

                    Table 11: The definitions for
                       "extra_plane" values.

4.2.11. "num_h_slices"

   "num_h_slices" indicates the number of horizontal elements of the
   Slice raster.

   Inferred to be 1 if not present.

4.2.12. "num_v_slices"

   "num_v_slices" indicates the number of vertical elements of the Slice
   raster.

   Inferred to be 1 if not present.

4.2.13. "quant_table_set_count"

   "quant_table_set_count" indicates the number of Quantization
   Table Sets. "quant_table_set_count" MUST be less than or equal to 8.

   Inferred to be 1 if not present.

   MUST NOT be 0.

4.2.14. "states_coded"

   "states_coded" indicates if the respective Quantization Table Set has
   the initial states coded.

   Inferred to be 0 if not present.

```
                +=======+=============================+
                | value | initial states             |
                +=======+=============================+
                | 0     | initial states are not present |
                |       | and are assumed to be all 128  |
```

```
                +-------+-------------------------------+
                | 1     | initial states are present    |
                +-------+-------------------------------+
```

                       Table 12: The definitions for
                          "states_coded" values.

4.2.15. "initial_state_delta"

   "initial_state_delta[i][j][k]" indicates the initial range
   coder state, and it is encoded using "k" as context index for the
   range coder and the following pseudocode:

   pred = j ? initial_states[i][j – 1][k] : 128

                  Figure 29: Predictor value for the coding of
                      "initial_state_delta[i][j][k]".

   initial_state[i][j][k] =
           (pred + initial_state_delta[i][j][k]) & 255

                   Figure 30: Description of the coding of
                     "initial_state_delta[i][j][k]".

4.2.16. "ec"

   "ec" indicates the error detection/correction type.

```
        +=======+=================================================+
        | value | error detection/correction type                 |
        +=======+=================================================+
        | 0     | 32-bit CRC in "ConfigurationRecord"             |
        +-------+-------------------------------------------------+
        | 1     | 32-bit CRC in "Slice" and "ConfigurationRecord" |
        +-------+-------------------------------------------------+
        | Other | reserved for future use                         |
        +-------+-------------------------------------------------+
```

                  Table 13: The definitions for "ec" values.

4.2.17. "intra"

   "intra" indicates the constraint on "keyframe" in each instance of
   Frame.

   Inferred to be 0 if not present.

```
     +=======+=========================================================+
     | value | relationship                                            |
     +=======+=========================================================+
     | 0     | "keyframe" can be 0 or 1 (non keyframes or keyframes)   |
     +-------+---------------------------------------------------------+
     | 1     | "keyframe" MUST be 1 (keyframes only)                   |
     +-------+---------------------------------------------------------+
     | Other | reserved for future use                                 |
     +-------+---------------------------------------------------------+
```

                 Table 14: The definitions for "intra" values.

4.3. Configuration Record

   In the case of a FFV1 bitstream with "version >= 3", a "Configuration
   Record" is stored in the underlying container as described in
   Section 4.3.3. It contains the "Parameters" used for all instances
   of Frame. The size of the "Configuration Record", "NumBytes", is
   supplied by the underlying container.

```
   pseudocode                                                  | type
   ------------------------------------------------------------|-----
   ConfigurationRecord( NumBytes ) {                           |
       ConfigurationRecordIsPresent = 1                        |
```

```
    Parameters( )
    while (remaining_symbols_in_syntax(NumBytes - 4)) {
        reserved_for_future_use                                    | br/ur/sr
    }
    configuration_record_crc_parity                               | u(32)
}
```

4.3.1. "reserved_for_future_use"

"reserved_for_future_use" is a placeholder for future updates of this
specification.

Encoders conforming to this version of this specification SHALL NOT
write "reserved_for_future_use".

Decoders conforming to this version of this specification SHALL
ignore "reserved_for_future_use".

4.3.2. "configuration_record_crc_parity"

"configuration_record_crc_parity" is 32 bits that are chosen so that
the "Configuration Record" as a whole has a CRC remainder of zero.

This is equivalent to storing the CRC remainder in the 32-bit parity.

The CRC generator polynomial used is described in Section 4.9.3.

4.3.3. Mapping FFV1 into Containers

This "Configuration Record" can be placed in any file format that
supports "Configuration Records", fitting as much as possible with
how the file format stores "Configuration Records". The
"Configuration Record" storage place and "NumBytes" are currently
defined and supported for the following formats:

4.3.3.1. Audio Video Interleave (AVI) File Format

The "Configuration Record" extends the stream format chunk ("AVI ",
"hdlr", "strl", "strf") with the "ConfigurationRecord" bitstream.

See [AVI] for more information about chunks.

"NumBytes" is defined as the size, in bytes, of the "strf" chunk
indicated in the chunk header minus the size of the stream format
structure.

4.3.3.2. ISO Base Media File Format

The "Configuration Record" extends the sample description box
("moov", "trak", "mdia", "minf", "stbl", "stsd") with a "glbl" box
that contains the "ConfigurationRecord" bitstream. See
[ISO.14496-12.2020] for more information about boxes.

"NumBytes" is defined as the size, in bytes, of the "glbl" box
indicated in the box header minus the size of the box header.

4.3.3.3. NUT File Format

The "codec_specific_data" element (in "stream_header" packet)
contains the "ConfigurationRecord" bitstream. See [NUT] for more
information about elements.

"NumBytes" is defined as the size, in bytes, of the
"codec_specific_data" element as indicated in the "length" field of
"codec_specific_data".

4.3.3.4. Matroska File Format

FFV1 SHOULD use "V_FFV1" as the Matroska "Codec ID". For FFV1
versions 2 or less, the Matroska "CodecPrivate" Element SHOULD NOT be
used. For FFV1 versions 3 or greater, the Matroska "CodecPrivate"

Element MUST contain the FFV1 "Configuration Record" structure and no
other data. See [Matroska] for more information about elements.

"NumBytes" is defined as the "Element Data Size" of the
"CodecPrivate" Element.

4.4. Frame

A "Frame" is an encoded representation of a complete static image.
The whole "Frame" is provided by the underlying container.

A "Frame" consists of the "keyframe" field, "Parameters" (if "version
<= 1"), and a sequence of independent Slices. The pseudocode below
describes the contents of a "Frame".

The "keyframe" field has its own initial state, set to 128.

```
pseudocode                                                    | type
------------------------------------------------------------- |-----
Frame( NumBytes ) {                                           |
    keyframe                                                  | br
    if (keyframe && !ConfigurationRecordIsPresent {           |
        Parameters( )                                         |
    }                                                         |
    while (remaining_bits_in_bitstream( NumBytes )) {         |
        Slice( )                                              |
    }                                                         |
}                                                             |
```

The following is an architecture overview of Slices in a Frame:

```
+--------------------------------------------------------------+
|  first Slice header                                          |
+--------------------------------------------------------------+
|  first Slice content                                         |
+--------------------------------------------------------------+
|  first Slice footer                                          |
+--------------------------------------------------------------+
| ------------------------------------------------------------ |
+--------------------------------------------------------------+
|  second Slice header                                         |
+--------------------------------------------------------------+
|  second Slice content                                        |
+--------------------------------------------------------------+
|  second Slice footer                                         |
+--------------------------------------------------------------+
| ------------------------------------------------------------ |
+--------------------------------------------------------------+
|  ...                                                         |
+--------------------------------------------------------------+
| ------------------------------------------------------------ |
+--------------------------------------------------------------+
|  last Slice header                                           |
+--------------------------------------------------------------+
|  last Slice content                                          |
+--------------------------------------------------------------+
|  last Slice footer                                           |
+--------------------------------------------------------------+
```
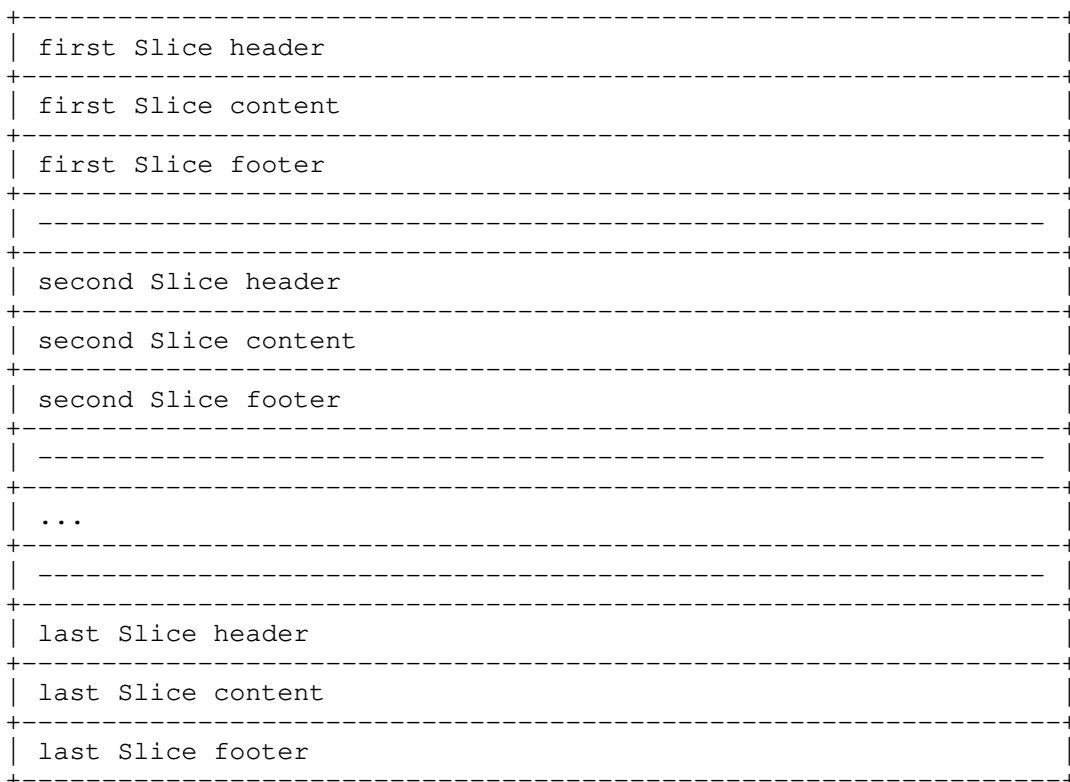
4.5. Slice

A "Slice" is an independent, spatial subsection of a Frame that is
encoded separately from another region of the same Frame. The use of
more than one "Slice" per Frame provides opportunities for taking
advantage of multithreaded encoding and decoding.

A "Slice" consists of a "Slice Header" (when relevant), a "Slice
Content", and a "Slice Footer" (when relevant). The pseudocode below
describes the contents of a "Slice".

pseudocode | type

```
    --------------------------------------------------------------|-----
    Slice( ) {                                                    |
        if (version >= 3) {                                       |
            SliceHeader( )                                        |
        }                                                         |
        SliceContent( )                                           |
        if (coder_type == 0) {                                    |
            while (!byte_aligned()) {                             |
                padding                                           | u(1)
            }                                                     |
        }                                                         |
        if (version <= 1) {                                       |
            while (remaining_bits_in_bitstream( NumBytes ) != 0) {|
                reserved                                          | u(1)
            }                                                     |
        }                                                         |
        if (version >= 3) {                                       |
            SliceFooter( )                                        |
        }                                                         |
    }                                                             |
```

"padding" specifies a bit without any significance and used only for
byte alignment. "padding" MUST be 0.

"reserved" specifies a bit without any significance in this
specification but may have a significance in a later revision of this
specification.

Encoders SHOULD NOT fill "reserved".

Decoders SHOULD ignore "reserved".

4.6. Slice Header

A "Slice Header" provides information about the decoding
configuration of the "Slice", such as its spatial position, size, and
aspect ratio. The pseudocode below describes the contents of the
"Slice Header".

"Slice Header" has its own initial states, all set to 128.

```
    pseudocode                                                    | type
    --------------------------------------------------------------|-----
    SliceHeader( ) {                                              |
        slice_x                                                   | ur
        slice_y                                                   | ur
        slice_width - 1                                           | ur
        slice_height - 1                                          | ur
        for (i = 0; i < quant_table_set_index_count; i++) {       |
            quant_table_set_index[ i ]                            | ur
        }                                                         |
        picture_structure                                         | ur
        sar_num                                                   | ur
        sar_den                                                   | ur
    }                                                             |
```

4.6.1. "slice_x"

"slice_x" indicates the x position on the Slice raster formed by
"num_h_slices".

Inferred to be 0 if not present.

4.6.2. "slice_y"

"slice_y" indicates the y position on the Slice raster formed by
"num_v_slices".

Inferred to be 0 if not present.

4.6.3. "slice_width"

"slice_width" indicates the width on the Slice raster formed by
"num_h_slices".

Inferred to be 1 if not present.

4.6.4. "slice_height"

"slice_height" indicates the height on the Slice raster formed by
"num_v_slices".

Inferred to be 1 if not present.

4.6.5. "quant_table_set_index_count"

"quant_table_set_index_count" is defined as the following:

```
1 + ( ( chroma_planes || version <= 3 ) ? 1 : 0 )
     + ( extra_plane ? 1 : 0 )
```

4.6.6. "quant_table_set_index"

"quant_table_set_index" indicates the Quantization Table Set index to
select the Quantization Table Set and the initial states for the
"Slice Content".

Inferred to be 0 if not present.

4.6.7. "picture_structure"

"picture_structure" specifies the temporal and spatial relationship
of each Line of the Frame.

Inferred to be 0 if not present.

value	picture structure used
0	unknown
1	top field first
2	bottom field first
3	progressive
Other	reserved for future use

Table 15: The definitions for
"picture_structure" values.

4.6.8. "sar_num"

"sar_num" specifies the Sample aspect ratio numerator.

Inferred to be 0 if not present.

A value of 0 means that aspect ratio is unknown.

Encoders MUST write 0 if the Sample aspect ratio is unknown.

If "sar_den" is 0, decoders SHOULD ignore the encoded value and
consider that "sar_num" is 0.

4.6.9. "sar_den"

"sar_den" specifies the Sample aspect ratio denominator.

Inferred to be 0 if not present.

A value of 0 means that aspect ratio is unknown.

Encoders MUST write 0 if the Sample aspect ratio is unknown.

If "sar_num" is 0, decoders SHOULD ignore the encoded value and
consider that "sar_den" is 0.

4.7. Slice Content

A "Slice Content" contains all Line elements part of the "Slice".

Depending on the configuration, Line elements are ordered by Plane
then by row (YCbCr) or by row then by Plane (RGB).

```
pseudocode                                                      | type
----------------------------------------------------------------|-----
SliceContent( ) {                                               |
    if (colorspace_type == 0) {                                 |
        for (p = 0; p < primary_color_count; p++) {             |
            for (y = 0; y < plane_pixel_height[ p ]; y++) {     |
                Line( p, y )                                    |
            }                                                   |
        }                                                       |
    } else if (colorspace_type == 1) {                          |
        for (y = 0; y < slice_pixel_height; y++) {              |
            for (p = 0; p < primary_color_count; p++) {         |
                Line( p, y )                                    |
            }                                                   |
        }                                                       |
    }                                                           |
}                                                               |
```

4.7.1. "primary_color_count"

"primary_color_count" is defined as the following:

1 + (chroma_planes ? 2 : 0) + (extra_plane ? 1 : 0)

4.7.2. "plane_pixel_height"

"plane_pixel_height[p]" is the height in pixels of Plane p of the
"Slice". It is defined as the following:

chroma_planes == 1 && (p == 1 || p == 2)
    ? ceil(slice_pixel_height / (1 << log2_v_chroma_subsample))
    : slice_pixel_height

4.7.3. "slice_pixel_height"

"slice_pixel_height" is the height in pixels of the Slice. It is
defined as the following:

floor(
        (slice_y + slice_height)
        * slice_pixel_height
        / num_v_slices
) - slice_pixel_y.

4.7.4. "slice_pixel_y"

"slice_pixel_y" is the Slice vertical position in pixels. It is
defined as the following:

floor(slice_y * frame_pixel_height / num_v_slices)

4.8. Line

A "Line" is a list of the Sample Differences (relative to the
predictor) of primary color components. The pseudocode below
describes the contents of the "Line".

```
   pseudocode                                                 | type
   ----------------------------------------------------------|-----
   Line( p, y ) {                                            |
       if (colorspace_type == 0) {                           |
           for (x = 0; x < plane_pixel_width[ p ]; x++) {    |
               sample_difference[ p ][ y ][ x ]              | sd
           }                                                 |
       } else if (colorspace_type == 1) {                    |
           for (x = 0; x < slice_pixel_width; x++) {         |
               sample_difference[ p ][ y ][ x ]              | sd
           }                                                 |
       }                                                     |
   }                                                         |
```

4.8.1. "plane_pixel_width"

"plane_pixel_width[p]" is the width in pixels of Plane p of the
"Slice". It is defined as the following:

chroma_planes == 1 && (p == 1 || p == 2)
    ? ceil(slice_pixel_width / (1 << log2_h_chroma_subsample))
    : slice_pixel_width.

4.8.2. "slice_pixel_width"

"slice_pixel_width" is the width in pixels of the Slice. It is
defined as the following:

floor(
        (slice_x + slice_width)
        * slice_pixel_width
        / num_h_slices
) - slice_pixel_x

4.8.3. "slice_pixel_x"

"slice_pixel_x" is the Slice horizontal position in pixels. It is
defined as the following:

floor(slice_x * frame_pixel_width / num_h_slices)

4.8.4. "sample_difference"

"sample_difference[p][y][x]" is the Sample Difference for
Sample at Plane "p", y position "y", and x position "x". The Sample
value is computed based on median predictor and context described in
Section 3.2.

4.9. Slice Footer

A "Slice Footer" provides information about Slice size and
(optionally) parity. The pseudocode below describes the contents of
the "Slice Footer".

Note: "Slice Footer" is always byte aligned.

```
   pseudocode                                                 | type
   ----------------------------------------------------------|-----
   SliceFooter( ) {                                          |
       slice_size                                            | u(24)
       if (ec) {                                             |
           error_status                                      | u(8)
           slice_crc_parity                                  | u(32)
       }                                                     |
   }                                                         |
```

4.9.1. "slice_size"

"slice_size" indicates the size of the Slice in bytes.

Note: this allows finding the start of Slices before previous Slices

have been fully decoded and allows parallel decoding as well as error
resilience.

4.9.2. "error_status"

"error_status" specifies the error status.

```
        +=======+=======================================+
        | value | error status                          |
        +=======+=======================================+
        | 0     | no error                              |
        +-------+---------------------------------------+
        | 1     | Slice contains a correctable error    |
        +-------+---------------------------------------+
        | 2     | Slice contains an uncorrectable error  |
        +-------+---------------------------------------+
        | Other | reserved for future use               |
        +-------+---------------------------------------+
```

Table 16: The definitions for "error_status"
values.

4.9.3. "slice_crc_parity"

"slice_crc_parity" is 32 bits that are chosen so that the Slice as a
whole has a CRC remainder of 0.

This is equivalent to storing the CRC remainder in the 32-bit parity.

The CRC generator polynomial used is the standard IEEE CRC polynomial
(0x104C11DB7) with initial value 0, without pre-inversion, and
without post-inversion.

5. Restrictions

To ensure that fast multithreaded decoding is possible, starting with
version 3 and if "frame_pixel_width * frame_pixel_height" is more
than 101376, "slice_width * slice_height" MUST be less or equal to
"num_h_slices * num_v_slices / 4". Note: 101376 is the frame size in
pixels of a 352x288 frame also known as CIF (Common Intermediate
Format) frame size format.

For each Frame, each position in the Slice raster MUST be filled by
one and only one Slice of the Frame (no missing Slice position and no
Slice overlapping).

For each Frame with a "keyframe" value of 0, each Slice MUST have the
same value of "slice_x", "slice_y", "slice_width", and "slice_height"
as a Slice in the previous Frame.

6. Security Considerations

Like any other codec (such as [RFC6716]), FFV1 should not be used
with insecure ciphers or cipher modes that are vulnerable to known
plaintext attacks. Some of the header bits as well as the padding
are easily predictable.

Implementations of the FFV1 codec need to take appropriate security
considerations into account. Those related to denial of service are
outlined in Section 2.1 of [RFC4732]. It is extremely important for
the decoder to be robust against malicious payloads. Malicious
payloads MUST NOT cause the decoder to overrun its allocated memory
or to take an excessive amount of resources to decode. An overrun in
allocated memory could lead to arbitrary code execution by an
attacker. The same applies to the encoder, even though problems in
encoders are typically rarer. Malicious video streams MUST NOT cause
the encoder to misbehave because this would allow an attacker to
attack transcoding gateways. A frequent security problem in image
and video codecs is failure to check for integer overflows. An
example is allocating "frame_pixel_width * frame_pixel_height" in
pixel count computations without considering that the multiplication

result may have overflowed the range of the arithmetic type. The
range coder could, if implemented naively, read one byte over the
end. The implementation MUST ensure that no read outside allocated
and initialized memory occurs.

None of the content carried in FFV1 is intended to be executable.

7. IANA Considerations

   IANA has registered the following values.

7.1. Media Type Definition

   This registration is done using the template defined in [RFC6838] and
   following [RFC4855].

   Type name: video

   Subtype name: FFV1

   Required parameters: None.

   Optional parameters: These parameters are used to signal the
      capabilities of a receiver implementation. These parameters MUST
      NOT be used for any other purpose.

      "version": The "version" of the FFV1 encoding as defined by
         Section 4.2.1.

      "micro_version": The "micro_version" of the FFV1 encoding as
         defined by Section 4.2.2.

      "coder_type": The "coder_type" of the FFV1 encoding as defined by
         Section 4.2.3.

      "colorspace_type": The "colorspace_type" of the FFV1 encoding as
         defined by Section 4.2.5.

      "bits_per_raw_sample": The "bits_per_raw_sample" of the FFV1
         encoding as defined by Section 4.2.7.

      "max_slices": The value of "max_slices" is an integer indicating
         the maximum count of Slices within a Frame of the FFV1
         encoding.

   Encoding considerations: This media type is defined for
      encapsulation in several audiovisual container formats and
      contains binary data; see Section 4.3.3. This media type is
      framed binary data; see Section 4.8 of [RFC6838].

   Security considerations: See Section 6 of this document.

   Interoperability considerations: None.

   Published specification: RFC 9043.

   Applications that use this media type: Any application that requires
      the transport of lossless video can use this media type. Some
      examples are, but not limited to, screen recording, scientific
      imaging, and digital video preservation.

   Fragment identifier considerations: N/A.

   Additional information: None.

   Person & email address to contact for further information:
      Michael Niedermayer (mailto:michael@niedermayer.cc)

   Intended usage: COMMON

   Restrictions on usage: None.

Author: Dave Rice (mailto:dave@dericed.com)

   Change controller: IETF CELLAR Working Group delegated from the
      IESG.

8. References

8.1. Normative References

   [ISO.9899.2018]
              International Organization for Standardization,
              "Information technology - Programming languages - C", ISO/
              IEC 9899:2018, June 2018.

   [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>.

   [RFC4732] Handley, M., Ed., Rescorla, E., Ed., and IAB, "Internet
              Denial-of-Service Considerations", RFC 4732,
              DOI 10.17487/RFC4732, December 2006,
              <https://www.rfc-editor.org/info/rfc4732>.

   [RFC4855] Casner, S., "Media Type Registration of RTP Payload
              Formats", RFC 4855, DOI 10.17487/RFC4855, February 2007,
              <https://www.rfc-editor.org/info/rfc4855>.

   [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>.

   [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>.

8.2. Informative References

   [AddressSanitizer]
              Clang Project, "AddressSanitizer", Clang 12 documentation,
              <https://clang.llvm.org/docs/AddressSanitizer.html>.

   [AVI]   Microsoft, "AVI RIFF File Reference",
              <https://docs.microsoft.com/en-
              us/windows/win32/directshow/avi-riff-file-reference>.

   [FFV1GO]   Buitenhuis, D., "FFV1 Decoder in Go", 2019,
              <https://github.com/dwbuiten/go-ffv1>.

   [FFV1_V0] Niedermayer, M., "Commit to mark FFV1 version 0 as non-
              experimental", April 2006, <https://git.videolan.org/?p=ff
              mpeg.git;a=commit;h=b548f2b91b701e1235608ac882ea6df915167c
              7e>.

   [FFV1_V1] Niedermayer, M., "Commit to release FFV1 version 1", April
              2009, <https://git.videolan.org/?p=ffmpeg.git;a=commit;h=6
              8f8d33becbd73b4d0aa277f472a6e8e72ea6849>.

   [FFV1_V3] Niedermayer, M., "Commit to mark FFV1 version 3 as non-
              experimental", August 2013, <https://git.videolan.org/?p=f
              fmpeg.git;a=commit;h=abe76b851c05eea8743f6c899cbe5f7409b0f
              301>.

   [HuffYUV] Rudiak-Gould, B., "HuffYUV revisited", December 2003,
              <https://web.archive.org/web/20040402121343/
              http://cultact-server.novi.dk/kpo/huffyuv/huffyuv.html>.

   [ISO.14495-1.1999]
              International Organization for Standardization,

             "Information technology -- Lossless and near-lossless
             compression of continuous-tone still images: Baseline",
             ISO/IEC 14495-1:1999, December 1999.

   [ISO.14496-10.2020]
             International Organization for Standardization,
             "Information technology -- Coding of audio-visual objects
             -- Part 10: Advanced Video Coding", ISO/IEC 14496-10:2020,
             December 2020.

   [ISO.14496-12.2020]
             International Organization for Standardization,
             "Information technology -- Coding of audio-visual objects
             -- Part 12: ISO base media file format", ISO/IEC
             14496-12:2020, December 2020.

   [ISO.15444-1.2019]
             International Organization for Standardization,
             "Information technology -- JPEG 2000 image coding system:
             Core coding system", ISO/IEC 15444-1:2019, October 2019.

   [Matroska] Lhomme, S., Bunkus, M., and D. Rice, "Matroska Media
             Container Format Specifications", Work in Progress,
             Internet-Draft, draft-ietf-cellar-matroska-07, 12 April
             2021, <https://datatracker.ietf.org/doc/html/draft-ietf-
             cellar-matroska-07>.

   [MediaConch]
             MediaArea.net, "MediaConch", 2018,
             <https://mediaarea.net/MediaConch>.

   [NUT]   Niedermayer, M., "NUT Open Container Format", December
             2013, <https://ffmpeg.org/~michael/nut.txt>.

   [Range-Encoding]
             Martin, G. N. N., "Range encoding: an algorithm for
             removing redundancy from a digitised message", Proceedings
             of the Conference on Video and Data Recording, Institution
             of Electronic and Radio Engineers, Hampshire, England,
             July 1979.

   [REFIMPL] Niedermayer, M., "The reference FFV1 implementation / the
             FFV1 codec in FFmpeg",
             <https://ffmpeg.org/doxygen/trunk/ffv1_8h.html>.

   [RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the
             Opus Audio Codec", RFC 6716, DOI 10.17487/RFC6716,
             September 2012, <https://www.rfc-editor.org/info/rfc6716>.

   [Valgrind] Valgrind Developers, "Valgrind website",
             <https://valgrind.org/>.

   [YCbCr]   Wikipedia, "YCbCr", 25 May 2021,
             <https://en.wikipedia.org/w/
             index.php?title=YCbCr&oldid=1025097882>.

Appendix A. Multithreaded Decoder Implementation Suggestions

   This appendix is informative.

   The FFV1 bitstream is parsable in two ways: in sequential order as
   described in this document or with the pre-analysis of the footer of
   each Slice. Each Slice footer contains a "slice_size" field so the
   boundary of each Slice is computable without having to parse the
   Slice content. That allows multithreading as well as independence of
   Slice content (a bitstream error in a Slice header or Slice content
   has no impact on the decoding of the other Slices).

   After having checked the "keyframe" field, a decoder should parse
   "slice_size" fields, from "slice_size" of the last Slice at the end
   of the "Frame" up to "slice_size" of the first Slice at the beginning

of the "Frame" before parsing Slices, in order to have Slice
boundaries. A decoder may fall back on sequential order e.g., in
case of a corrupted "Frame" (e.g., frame size unknown or "slice_size"
of Slices not coherent) or if there is no possibility of seeking into
the stream.

Appendix B. Future Handling of Some Streams Created by Nonconforming
            Encoders

   This appendix is informative.

   Some bitstreams were found with 40 extra bits corresponding to
   "error_status" and "slice_crc_parity" in the "reserved" bits of
   "Slice". Any revision of this specification should avoid adding 40
   bits of content after "SliceContent" if "version == 0" or "version ==
   1", otherwise a decoder conforming to the revised specification could
   not distinguish between a revised bitstream and such buggy bitstream
   in the wild.

Appendix C. FFV1 Implementations

   This appendix provides references to a few notable implementations of
   FFV1.

C.1. FFmpeg FFV1 Codec

   This reference implementation [REFIMPL] contains no known buffer
   overflow or cases where a specially crafted packet or video segment
   could cause a significant increase in CPU load.

   The reference implementation [REFIMPL] was validated in the following
   conditions:

   * Sending the decoder valid packets generated by the reference
      encoder and verifying that the decoder's output matches the
      encoder's input.

   * Sending the decoder packets generated by the reference encoder and
      then subjected to random corruption.

   * Sending the decoder random packets that are not FFV1.

   In all of the conditions above, the decoder and encoder was run
   inside the Valgrind memory debugger [Valgrind] as well as the Clang
   AddressSanitizer [AddressSanitizer], which tracks reads and writes to
   invalid memory regions as well as the use of uninitialized memory.
   There were no errors reported on any of the tested conditions.

C.2. FFV1 Decoder in Go

   An FFV1 decoder [FFV1GO] was written in Go by Derek Buitenhuis during
   the work to develop this document.

C.3. MediaConch

   The developers of the MediaConch project [MediaConch] created an
   independent FFV1 decoder as part of that project to validate FFV1
   bitstreams. This work led to the discovery of three conflicts
   between existing FFV1 implementations and draft versions of this
   document. These issues are addressed by Section 3.3.1,
   Section 3.7.2.1, and Appendix B.

Authors' Addresses

   Michael Niedermayer

   Email: michael@niedermayer.cc

   Dave Rice

Email: dave@dericed.com

Jérôme Martinez

Email: jerome@mediaarea.net