RDD 36:2022
Revision of SMPTE RDD 36:2015

SMPTE REGISTERED DISCLOSURE DOCUMENT

Apple ProRes Bitstream Syntax and Decoding Process

The attached document is a Registered Disclosure Document prepared by the sponsor identified below. It has been examined by the appropriate SMPTE Technology Committee and is believed to contain adequate information to satisfy the objectives defined in the Scope, and to be technically consistent.

This document is NOT a Standard, Recommended Practice or Engineering Guideline, and does NOT imply a finding or representation of the Society.

Every attempt has been made to ensure that the information contained in this document is accurate. Errors in this document should be reported to the proponent identified below, with a copy to eng@smpte.org.

All other inquiries in respect of this document, including inquiries as to intellectual property requirements that may be attached to use of the disclosed technology, should be addressed to the proponent identified below.

Proponent contact information:

ProRes Program Office
Apple Inc.
1 Infinite Loop, MS: 77-2YAK
Cupertino, CA 95014
USA

Email: ProRes@apple.com

Approved September 9, 2022

Table of Contents Page

Introduction

Apple ProRes is a video compression scheme developed by Apple Inc. for use in workflows that require high quality and efficient performance. It is an intra-frame codec that can encode progressive or interlaced frames with arbitrary dimensions and either 4:2:2 or 4:4:4 chroma sampling. It operates on Y′CbCr video data; the pixel component samples can have bit depths of 12 or even more bits per sample, which enables ProRes to be used for RGB video data (via conversion to Y′CbCr) with high quality results. Frames can also include an alpha channel, with up to 16 bits per alpha sample, which ProRes encodes losslessly.

1 Scope

This SMPTE Registered Disclosure Document (RDD) includes specifications for the Apple ProRes bitstream syntax, the bitstream element semantics, and the decoding process used to produce decompressed images. A reference implementation that reads ProRes bitstreams from a file and decompresses the bitstreams is part of the contribution. Sample bitstreams and the resulting decompressed images have also been contributed for exercising the reference implementation. This RDD does not describe the Apple QuickTime file format or the details of storing ProRes bitstreams in QuickTime files.

2 References

IEEE Std 1180-1990, *IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform*.

ISO/IEC 13818-2:2013, *Information technology — Generic coding of moving pictures and associated audio information: Video*.

Recommendation ITU-R BT.601-7, *Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*.

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

Recommendation ITU-R BT.2020-2, *Parameter values for ultra-high definition television systems for production and international programme exchange*.

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

Recommendation ITU-T H.264 (08/2021), *Advanced video coding for generic audiovisual services*.

Recommendation ITU-T H.273 (07/2021), *Coding-independent code points for video signal type identification*.

SMPTE ST 2084:2014, *High Dynamic Range Electro-Optical Transfer Function of Mastering Reference Displays*.

3 Notation

3.1 Arithmetic Operators

+	Addition
−	Subtraction (as a binary operator) or negation (as a unary prefix operator)

*	Multiplication
÷	Division (used in mathematical equations where no truncation or rounding is intended)
$\frac{x}{y}$	Division, x ÷ y
/	Integer division with truncation of the result toward negative infinity: x / y = floor(x ÷ y)
n mod m	Modulo operator with modulus m. Defined only for integers n and m with m > 0. Result is remainder r after integer division of n by m, r = n − floor(n ÷ m) * m; 0 ≤ r ≤ m − 1.
x^y	Exponentiation, x raised to the power y
$\sum_{i=n_1}^{n_2} f(i)$	Summation of f(i), where i, n_1, and n_2 are integers and $n_1 \le i \le n_2$

3.2 Logical Operators

&&	Boolean logical “and”
||	Boolean logical “or”
!	Boolean logical “not”

3.3 Relational Operators

>	Greater than
≥, >=	Greater than or equal to
<	Less than
≤, <=	Less than or equal to
==	Equal to
!=	Not equal to

3.4 Bitwise Operators

&	Bitwise “and”
|	Bitwise “or”
<<	Left shift; vacated bits are set to ‘0’

3.5 Assignment Operators

=	Assignment
++	Increment: x++ performs x = x + 1 and, if used in an expression, evaluates to the value of x prior to the increment
−−	Decrement: x−− performs x = x − 1 and, if used in an expression, evaluates to the value of x prior to the decrement
+=	Increment assignment: x += y performs x = x + y
−=	Decrement assignment: x −= y performs x = x − y

3.6 Mathematical Functions

$\lvert x \rvert$	Absolute value of x: x if $x \geq 0$, $-x$ if $x < 0$
$\sqrt{x}$	Square root of x
ceil(x)	Least integer greater than or equal to x
cos(x)	Cosine of x, with x in units of radians
floor(x)	Greatest integer less than or equal to x
$\log_2(x)$	Base-2 logarithm of x
round(x)	Integer nearest to x, for example floor(x + ½). If $x = n + ½$ for some integer n, either n or n+1 is acceptable as the result.

3.7 Constants

π	3.14159 26535 89793 23846…

4 ProRes Frame Structure

ProRes can encode either progressive or interlaced frames. A progressive frame comprises a single picture, while an interlaced frame comprises two pictures, one for each field. In the interlaced case, the order of the compressed pictures in the bitstream matches the temporal order of the corresponding fields.

Pictures are divided into macroblocks and then further into blocks. Macroblocks are 16×16 arrays of pixels, and blocks are 8×8 arrays of video component samples. For 4:2:2 sampling, each macroblock thus consists of four Y′ (luma) blocks, two Cb blocks, and two Cr blocks; for 4:4:4 sampling, each consists of four Y′ blocks, four Cb blocks, and four Cr blocks.

The macroblock width and height of encoded pictures will both be integral. If the luma sample width of the source picture is not a multiple of 16, encoders will append sufficient pixels to the end (right) of each row of the source picture to rectify this; decoders shall discard those excess pixels. Likewise, if the luma sample height of the source picture is not a multiple of 16, encoders will append sufficient rows of pixels to the end (bottom) of the source picture to rectify this while decoders shall discard those excess rows.

Finally, the macroblocks of a picture are grouped into slices. Each slice consists of 1, 2, 4, or 8 contiguous macroblocks all in the same macroblock row. The slice layout is identical for each macroblock row and is determined by a bitstream parameter indicating the desired slice size: starting from the beginning (left) of each macroblock row, slices with the desired number of macroblocks are defined until there are insufficient remaining macroblocks in the row, at which point slices are defined with the next-largest permissible number of macroblocks, and so forth until the row is exhausted. Each slice can be encoded and decoded independently, which facilitates parallel encoding and decoding.

As an example, consider a progressive frame 720 pixels wide by 486 pixels high, with a desired slice size of 8 macroblocks. The sole source picture will be the frame itself. The width of the source picture is a multiple of 16, making the encoded picture 720 ÷ 16 = 45 macroblocks wide. The height of the source picture, however, is not a multiple of 16, so the encoded picture will be ceil(486 ÷ 16) = 31 macroblocks high; encoders will pad the source picture with 31 * 16 – 486 = 10 rows of pixels at the bottom, while decoders shall discard those same 10 rows of pixels. Starting from the left, the 45 macroblocks in each macroblock row will be grouped into five slices

of 8 macroblocks each, leaving 5 remaining macroblocks in the macroblock row. The next slice will then consist of 4 macroblocks, and finally there will be one last slice consisting of a single macroblock. In total each macroblock row will contain seven slices with sizes of (from left to right) 8, 8, 8, 8, 8, 4, and 1. This is illustrated in Figure 1.

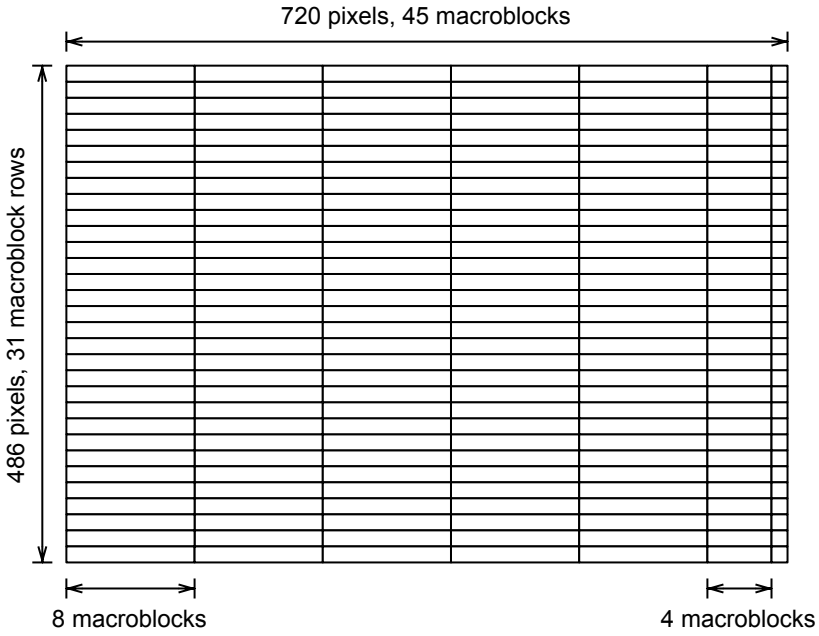

Figure 1 – Example picture slice arrangement

5 Bitstream Syntax

The formal description of ProRes bitstream syntax is provided by the tables in this section. The syntax tables are similar in style to those in ISO/IEC 13818-2 and ITU-T H.264. In particular, the tables describe syntax using pseudo-code based on the C programming language.

The syntax description involves several special constructs. Syntax elements are the fundamental parameters that describe the compressed image or direct the exact nature of the decoding process. There are two types of syntax elements: bitstream syntax elements and derived syntax elements. The former appear directly in ProRes bitstreams, while the latter are calculated from bitstream syntax elements or other derived syntax elements. Both are denoted by names using all lowercase letters with underscore characters separating words; the names of bitstream syntax elements appear in boldface in the syntax tables where they occur in the bitstream (though not in subsequent usage) and in the headings of their semantic descriptions.

Syntax structures are collections of syntax elements, other syntax structures, or both. They are denoted by functions whose names use all lowercase letters with underscore characters separating words. The functions can take arguments, which identify a specific instance of the syntax structure or provide information relevant to the parsing or decoding process of the structure. The sizes of some syntax structures are specified in ProRes bitstreams as described subsequently. Decoders shall use these sizes to determine the start of the immediately following syntax structure. (See Section 6.4, “Bitstream Versions, Version Variants, and Compatibility,” for more detail.)

Finally, syntax variables and syntax functions are used within syntax structures as part of the description of those structures. To distinguish them from syntax elements and structures, their names use a mixture of lowercase and uppercase letters and do not include underscore characters. Syntax variables are defined implicitly by their use in the syntax tables.

The syntax function endOfData() takes an argument that specifies the size in bytes of a syntax structure and returns true if the number of remaining bits—the difference between the structure's size in bits and the number of bits representing the preceding bitstream syntax elements of the structure—is 31 or less and the remaining bits, if any, all have value 0; it returns false if 32 or more bits remain or if any of the remaining bits has the value 1. The syntax function byteAligned() returns true if the number of bits representing the preceding bitstream syntax elements of the associated syntax structure is a multiple of eight and returns false if not. The syntax function endOfStructure() takes an argument that specifies the size in bytes of a syntax structure and returns true if the number of remaining bits is 0 and false if one or more bits remain. The syntax function isModuloAlphaDifference(), which returns true or false, is used in the alpha channel decoding process and is defined in Section 7.1.2, "Scanned Alpha."

ProRes bitstream syntax elements belong to one of three categories: fixed-length bit strings, fixed-length numerical values, or variable-length codes. In the syntax tables, the "Descriptor" column indicates the category to which each bitstream syntax element belongs. Fixed-length bit strings are denoted by "f(n)," where n is the number of bits in the string. Fixed-length numerical values are unsigned integers and are denoted by "u(n)," where n is the number of bits used to represent the value. Variable-length codes are denoted by "vlc." Bit strings and variable-length codes appear in the bitstream left bit first; numerical values appear most-significant bit first.

Figure 2 gives an overview of the hierarchy of ProRes bitstream syntax structures.

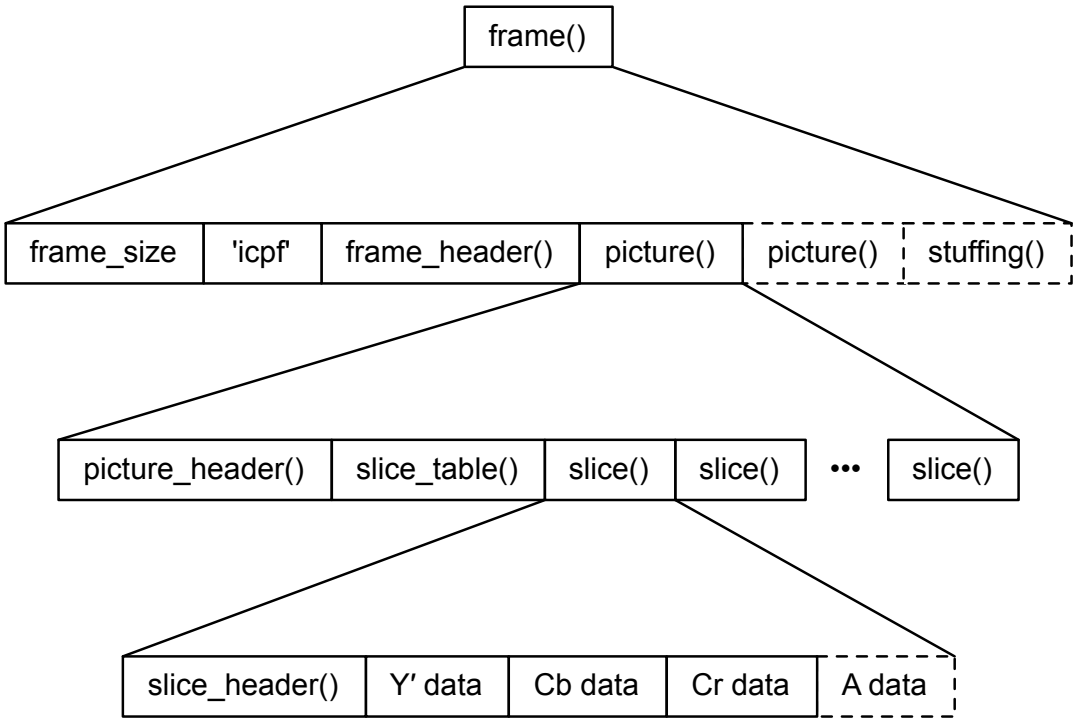

Figure 2 – ProRes bitstream syntax structure hierarchy

5.1 Frame Syntax

frame() {	Descriptor
frame_size	u(32)
frame_identifier	f(32)
frame_header()	
picture(“first”)	
if (interlace_mode == 1 || interlace_mode == 2)	
picture(“second”)	
if (stuffing_size > 0)	
stuffing()	
}	

5.1.1 Frame Header Syntax

frame_header() {	Descriptor
frame_header_size	u(16)
reserved	u(8)
bitstream_version	u(8)
encoder_identifier	f(32)
horizontal_size	u(16)
vertical_size	u(16)
chroma_format	u(2)
reserved	u(2)
interlace_mode	u(2)
reserved	u(2)
aspect_ratio_information	u(4)
frame_rate_code	u(4)
color_primaries	u(8)
transfer_characteristic	u(8)
matrix_coefficients	u(8)
reserved	u(4)
alpha_channel_type	u(4)
reserved	u(14)
load_luma_quantization_matrix	u(1)
load_chroma_quantization_matrix	u(1)
if (load_luma_quantization_matrix) {	
for (v = 0; v < 8; v++)	
for (u = 0; u < 8; u++)	
luma_quantization_matrix[v][u]	u(8)
}	
if (load_chroma_quantization_matrix) {	
for (v = 0; v < 8; v++)	
for (u = 0; u < 8; u++)	
chroma_quantization_matrix[v][u]	u(8)
}	
}	

5.1.2 Stuffing Syntax

stuffing() {	Descriptor
for (m = 0; m < stuffing_size; m++)	
zero_byte /* Equal to 0x00 */	f(8)
}	

5.2 Picture Syntax

picture(temporalOrder) {	Descriptor
picture_header()	
slice_table()	
for (i = 0; i < height_in_mb; i++)	
for (j = 0; j < number_of_slices_per_mb_row; j++)	
slice(i, j)	
}	

5.2.1 Picture Header Syntax

picture_header() {	Descriptor
picture_header_size	u(5)
reserved	u(3)
picture_size	u(32)
deprecated_number_of_slices	u(16)
reserved	u(2)
log2_desired_slice_size_in_mb	u(2)
reserved	u(4)
}	

5.2.2 Slice Table Syntax

slice_table () {	Descriptor
for (i = 0; i < height_in_mb; i++)	
for (j = 0; j < number_of_slices_per_mb_row; j++)	
coded_size_of_slice[i][j]	u(16)
}	

5.3 Slice Syntax

slice(i, j) {	Descriptor
slice_header()	
numYBlocks = 4 * slice_size_in_mb[j]	
if (chroma_format == 3) /* 4:4:4 */	
numCBlocks = 4 * slice_size_in_mb[j]	
else /* 4:2:2 */	
numCBlocks = 2 * slice_size_in_mb[j]	
codedYDataSize = coded_size_of_y_data	
codedCbDataSize = coded_size_of_cb_data	
if (alpha_channel_type != 0)	
codedCrDataSize = coded_size_of_cr_data	
else	
codedCrDataSize = coded_size_of_slice[i][j] - slice_header_size - codedYDataSize - codedCbDataSize	
scanned_coefficients(scannedYCoeffs, numYBlocks, codedYDataSize)	
scanned_coefficients(scannedCbCoeffs, numCBlocks, codedCbDataSize)	
scanned_coefficients(scannedCrCoeffs, numCBlocks, codedCrDataSize)	
if (alpha_channel_type != 0) {	
if (i < height_in_mb – 1)	
sliceVerticalSize = 16	
else	
sliceVerticalSize = picture_vertical_size - 16 * (height_in_mb – 1)	
numAlphaValues = 16 * slice_size_in_mb[j] * sliceVerticalSize	
scanned_alpha(scannedAlphaValues, numAlphaValues)	
}	
}	

5.3.1 Slice Header Syntax

slice_header() {	Descriptor
slice_header_size	u(5)
reserved	u(3)
quantization_index	u(8)
coded_size_of_y_data	u(16)
coded_size_of_cb_data	u(16)
if (alpha_channel_type != 0)	
coded_size_of_cr_data	u(16)
}	

5.3.2 Scanned Coefficients Syntax

scanned_coefficients(coefficients, numBlocks, dataSize) {	**Descriptor**
first_dc_coeff	vlc
coefficients[0] = first_dc_coeff	
previousDCCoeff = first_dc_coeff	
n = 1	
while (n < numBlocks) {	
dc_coeff_difference	vlc
DCCoeff = previousDCCoeff + dc_coeff_difference	
coefficients[n++] = DCCoeff	
previousDCCoeff = DCCoeff	
}	
while (!endOfData(dataSize)) {	
run	vlc
for (m = 0; m < run; m++)	
coefficients[n++] = 0	
abs_level_minus_1	vlc
absLevel = abs_level_minus_1 + 1	
sign	u(1)
coefficients[n++] = absLevel * (1 – 2 * sign)	
}	
numCoefficients = numBlocks * 64	
while (n < numCoefficients)	
coefficients[n++] = 0	
while (!byteAligned())	
zero_bit /* Equal to 0 */	f(1)
for (m = 0; m < 3; m++) {	
if (!endOfStructure(dataSize))	
zero_byte /* Equal to 0x00 */	f(8)
}	
}	

5.3.3 Scanned Alpha Syntax

scanned_alpha(alphaValues, numValues) {	Descriptor
if (alpha_channel_type == 1) /* 8-bit alpha */	
mask = 0xFF	
else /* 16-bit alpha */	
mask = 0xFFFF	
n = 0	
previousAlpha = -1	
do {	
alpha_difference	vlc
alpha = previousAlpha + alpha_difference	
if (isModuloAlphaDifference())	
alpha = alpha & mask	
previousAlpha = alpha	
run	vlc
for (m = 0; m < run; m++)	
alphaValues[n++] = alpha	
} while (n < numValues)	
while (!byteAligned())	
zero_bit /* Equal to 0 */	f(1)
}	

6 Bitstream Semantics

6.1 Frame Semantics

frame_size

The total size of the compressed frame in bytes (including the frame_size element itself and, if present, stuffing).

frame_identifier

A four-character code that identifies the bitstream as a ProRes frame. This will be ‘icpf’ (0x69637066).

6.1.1 Frame Header Semantics

frame_header_size

The total size of the frame header in bytes (including the frame_header_size element itself). Decoders shall use this value to determine the start of the compressed picture following the frame header in the bitstream.

reserved

Bits reserved for use with future bitstream versions and/or version variants. Encoders will set the value of this element to 0; decoders shall ignore it (in particular shall not expect its value to be 0).

bitstream_version

The version number of the bitstream. The version number is incremented when a change is made to bitstream syntax or semantics that breaks compatibility with existing decoders. A decoder shall abort if it encounters a bitstream with an unsupported bitstream_version value. If 0, the value of the chroma_format syntax element will be 2 (4:2:2 sampling) and the value of the alpha_channel_type element will be 0 (no encoded alpha); if 1, any permissible value can be used for those syntax elements.

encoder_identifier

A four-character code that identifies the encoder vendor or product that generated the compressed frame. Apple maintains a registry of the codes for encoder licensees. Decoders should ignore this element.

horizontal_size

The width of the frame in luma samples.

vertical_size

The height of the frame in luma samples.

chroma_format

A code specifying the sampling format of the frame. Values and their meanings are listed in Table 1.

Table 1 – Meaning of chroma_format

chroma_format	Meaning
0	Reserved
1	Reserved
2	4:2:2
3	4:4:4

interlace_mode

A code specifying whether the frame is progressive or interlaced (as well as field order in the latter case). Values and their meanings are listed in Table 2.

Table 2 – Meaning of interlace_mode

interlace_mode	Meaning
0	Progressive frame (frame contains one full-height picture)
1	Interlaced frame (first picture is top field)
2	Interlaced frame (second picture is top field)
3	Reserved

aspect_ratio_information

A code indicating the pixel or image aspect ratio. Values and their meanings are listed in Table 3.

Table 3 – Meaning of aspect_ratio_information

aspect_ratio_information	Meaning
0	Unknown/unspecified
1	Square pixels
2	4:3 image aspect ratio
3	16:9 image aspect ratio
4	Reserved
…	…
15	Reserved

frame_rate_code

A code indicating frame rate. Values and their meanings are listed in Table 4.

Table 4 – Meaning of frame_rate_code

frame_rate_code	Meaning
0	Unknown/unspecified
1	24 ÷ 1.001 (23.976…)
2	24
3	25
4	30 ÷ 1.001 (29.97…)
5	30
6	50
7	60 ÷ 1.001 (59.94…)
8	60
9	100
10	120 ÷ 1.001 (119.88…)
11	120
12	Reserved
…	…
15	Reserved

color_primaries

A code indicating the chromaticity coordinates of the source primaries and white point. Values and their meanings are listed in Table 5. Note: The (nonreserved) values in the table agree with the values in Table 2 of ITU-T H.273; the intent was to avoid inconsistency between the values used here and the ones in that Recommendation.

Table 5 – Meaning of color_primaries

color_primaries	Meaning			
0	Unknown/unspecified			
1		x	y	(ITU-R BT.709)
	Red	0.640	0.330	
	Green	0.300	0.600	
	Blue	0.150	0.060	
	White (D65)	0.3127	0.3290	
2	Unknown/unspecified			
3	Reserved			
4	Reserved			
5		x	y	(ITU-R BT.601 625)
	Red	0.640	0.330	
	Green	0.290	0.600	
	Blue	0.150	0.060	
	White (D65)	0.3127	0.3290	
6		x	y	(ITU-R BT.601 525)
	Red	0.630	0.340	
	Green	0.310	0.595	
	Blue	0.155	0.070	
	White (D65)	0.3127	0.3290	
7	Reserved			
8	Reserved			
9		x	y	(ITU-R BT.2020)
	Red	0.708	0.292	
	Green	0.170	0.797	
	Blue	0.131	0.046	
	White (D65)	0.3127	0.3290	
10	Reserved			
11		x	y	(DCI P3)
	Red	0.680	0.320	
	Green	0.265	0.690	
	Blue	0.150	0.060	
	White (DCI)	0.314	0.351	
12		x	y	(P3 D65)
	Red	0.680	0.320	
	Green	0.265	0.690	
	Blue	0.150	0.060	
	White (D65)	0.3127	0.3290	
13	Reserved			
…	…			
255	Reserved			

transfer_characteristic

A code indicating the opto-electronic transfer characteristic of the source video data. The values 0 and 2 mean unknown/unspecified; the value 1 signifies the function specified by ITU-R BT.601/BT.709/BT.2020, namely

$$V = \begin{cases} \alpha * L^{0.45} - (\alpha - 1), & \beta \le L \le 1 \\ 4.5 * L, & 0 \le L \le \beta, \end{cases}$$

where L is normalized linear optical intensity, V is the corresponding nonlinear (gamma pre-corrected) signal value, α = 1.099 296 826 809 44…, and β = 0.018 053 968 510 807…; the value 16 signifies the Inverse-EOTF formula in Section 5.3 of SMPTE ST 2084:2014, namely

$$V = \left(\frac{c_1 + c_2 L^{m_1}}{1 + c_3 L^{m_1}} \right)^{m_2},$$

where L and V are as above (here L is normalized so that L = 1 corresponds to an absolute optical intensity of 10,000 cd/m^2), m_1 = 0.25 * (2610 ÷ 4096), m_2 = 128 * (2523 ÷ 4096), $c_1 = c_3 - c_2 + 1$ = 3424 ÷ 4096, c_2 = 32 * (2413 ÷ 4096), and c_3 = 32 * (2392 ÷ 4096); and the value 18 signifies the HLG Reference OETF from Table 5 of ITU-R BT.2100-2, namely

$$V = \begin{cases} \sqrt{3\,L}, & 0 \le L \le 1/12 \\ a * \ln(12\,L - b) + c, & 1/12 \le L \le 1, \end{cases}$$

where again L and V are as above (L has no prescribed normalization here), a = 0.17883277, b = 1 − 4a (= 0.28466892), and c = ½ − a * ln(4a) (= 0.55991073). All other values are reserved. Note: The (nonreserved) values specified above agree with the values in Table 3 of ITU-T H.273; the intent was to avoid inconsistency between the values used here and the ones in that Recommendation.

matrix_coefficients

A code indicating the matrix coefficients used to derive luma and chroma values from the red, green, and blue primaries. Values and their meanings are listed in Table 6. For values that specify luma coefficients (K_R, K_G, and K_B), the derivation is $E'_Y = K_R * E'_R + K_G * E'_G + K_B * E'_B$, $E'_{Cb} = (E'_B - E'_Y) \div (2 * (1 - K_B))$, and $E'_{Cr} = (E'_R - E'_Y) \div (2 * (1 - K_R))$, where E'_Y is the normalized luma value and E'_{Cb} and E'_{Cr} are the normalized chroma values corresponding to the normalized gamma pre-corrected primary values E'_R, E'_G, and E'_B. Note: The (nonreserved) values in Table 6 agree with the values in Table 4 of ITU-T H.273; the intent was to avoid inconsistency between the values used here and the ones in that Recommendation.

Table 6 – Meaning of matrix_coefficients

matrix_coefficients	Meaning
0	Unknown/unspecified
1	K_R = 0.2126, K_G = 0.7152, K_B = 0.0722 (ITU-R BT.709)
2	Unknown/unspecified
3	Reserved
4	Reserved
5	Reserved
6	K_R = 0.299, K_G = 0.587, K_B = 0.114 (ITU-R BT.601)
7	Reserved
8	Reserved
9	K_R = 0.2627, K_G = 0.6780, K_B = 0.0593 (ITU-R BT.2020)
10	Reserved
…	…
255	Reserved

alpha_channel_type

A code specifying the type of alpha channel data encoded in the bitstream, if any. Values and their meanings are listed in Table 7. If the value of the bitstream_version syntax element is 0, the value of this element will be 0. There are no such restrictions for other values of bitstream_version; in particular the value of this element can be 0 when the value of bitstream_version is not 0. Note: Slice syntax is affected by the value of this element.

Table 7 – Meaning of alpha_channel_type

alpha_channel_type	Meaning
0	No encoded alpha data present in bitstream
1	8 bits/sample integral alpha
2	16 bits/sample integral alpha
3	Reserved
…	…
15	Reserved

load_luma_quantization_matrix

A flag indicating whether a custom luma quantization matrix is specified. If 0, the default matrix shall be used.

load_chroma_quantization_matrix

A flag indicating whether a custom chroma quantization matrix is specified. If 0, the luma matrix shall be used (i.e., the specified custom luma quantization matrix if load_luma_quantization_matrix is 1 or the default matrix otherwise).

luma_quantization_matrix

Custom quantization weight matrix for luma coefficients. Each entry of the matrix will be in the range 2, 3, …, 63.

chroma_quantization_matrix

Custom quantization weight matrix for chroma coefficients. Each entry of the matrix will be in the range 2, 3, …, 63.

6.1.2 Stuffing Semantics

stuffing_size

The size of stuffing in bytes. Can be zero. It is calculated as follows:

```
frameDataSize = 4 + 4 + frame_header_size
                  + picture("first").picture_size;
if (interlace_mode == 1 || interlace_mode == 2)
    frameDataSize += picture("second").picture_size;
stuffing_size = frame_size – frameDataSize;
```

zero_byte

An eight-bit number with value zero (0x00). Optionally used to pad the compressed frame up to a desired size.

6.2 Picture Semantics

picture_vertical_size

The height of the picture in luma samples. Derived from vertical_size and interlace_mode:

```
if (interlace_mode == 0)
    picture_vertical_size = vertical_size
else {
    topFieldVerticalSize = (vertical_size + 1) / 2
    bottomFieldVerticalSize = vertical_size / 2
    if ((interlace_mode == 1 && temporalOrder == "first") ||
        (interlace_mode == 2 && temporalOrder == "second"))
        picture_vertical_size = topFieldVerticalSize
    else
        picture_vertical_size = bottomFieldVerticalSize
}
```

width_in_mb

The width of the encoded picture in macroblocks. Derived from horizontal_size:

```
width_in_mb = (horizontal_size + 15) / 16
```

If the encoded picture width in luma samples, 16 * width_in_mb, exceeds horizontal_size, encoders will append 16 * width_in_mb − horizontal_size additional pixels to the end (right) of each row of the source picture; decoders shall discard the excess pixel(s) from the end (right) of each row of the decoded picture. Note: When a frame consists of two pictures, width_in_mb is the same for each.

height_in_mb

The height of the encoded picture in macroblocks. Derived from picture_vertical_size:

```
height_in_mb = (picture_vertical_size + 15) / 16
```

If the encoded picture height in luma samples, 16 * height_in_mb, exceeds picture_vertical_size, encoders will append 16 * height_in_mb − picture_vertical_size additional rows of pixels to the end (bottom) of the source picture; decoders shall discard the excess row(s) of pixels from the end (bottom) of the decoded picture.

slice_size_in_mb

Array of sizes (in macroblocks) of slices within a single macroblock row of the encoded picture, starting with the first (leftmost) slice and ending with the last (rightmost) one. The array pertains to all macroblock rows. The entries are calculated as follows:

```
j = 0
sliceSize = 1 << log2_desired_slice_size_in_mb
numMbsRemainingInRow = width_in_mb
do {
    while (numMbsRemainingInRow >= sliceSize) {
        slice_size_in_mb[j++] = sliceSize
        numMbsRemainingInRow -= sliceSize
    }
    sliceSize /= 2
} while (numMbsRemainingInRow > 0)
number_of_slices_per_mb_row = j
```

number_of_slices_per_mb_row

The number of slices in a single macroblock row of the encoded picture. This corresponds to the number of entries in the slice_size_in_mb array; like that array, it is the same for every macroblock row. It is calculated as part of the procedure to determine the slice_size_in_mb array entries, as shown above.

6.2.1 Picture Header Semantics

picture_header_size

The total size of the picture header in bytes (including the picture_header_size element itself). Decoders shall use this value to determine the start of the slice table following the picture header in the bitstream.

picture_size

The total size of the compressed picture in bytes (including the picture header). If the compressed frame contains two compressed pictures, decoders shall use this value from the first compressed picture to determine the start of the second compressed picture in the bitstream.

deprecated_number_of_slices

The product of the picture height in macroblocks and the number of slices per macroblock row when that product is 65535 or less, otherwise 0. Decoders shall ignore this element.

log2_desired_slice_size_in_mb

The base-2 logarithm of the desired number of macroblocks constituting a slice. Permissible values for this element are 0, 1, 2, and 3, which correspond respectively to 1, 2, 4, and 8 macroblocks per slice.

6.2.2 Slice Table Semantics

coded_size_of_slice

Two-dimensional array of compressed slice sizes in bytes. These follow the same ordering as the slices themselves: coded_size_of_slice[i][j] gives the size of slice(i, j) in the bitstream.

6.3 Slice Semantics

6.3.1 Slice Header Semantics

slice_header_size

The total size of the slice header in bytes (including the slice_header_size element itself). Decoders shall use this value to determine the start of the compressed luma component data following the slice header in the bitstream.

quantization_index

A code that specifies the quantization scale factor, qScale. Permissible values are 1, …, 224; all other values are reserved.

coded_size_of_y_data

The size of the compressed luma (Y′) component data in bytes.

coded_size_of_cb_data

The size of the compressed blue chroma (Cb) component data in bytes.

coded_size_of_cr_data

The size of the compressed red chroma (Cr) component data in bytes. Note: This element is present only if the value of the alpha_channel_type syntax element is nonzero.

6.3.2 Scanned Coefficients Semantics

first_dc_coeff

The first quantized DC coefficient in the scanned coefficient array.

dc_coeff_difference

The difference between the current quantized DC coefficient and the previous one.

run

The number of consecutive zero-valued quantized AC coefficients in the scanned coefficient array preceding one that is nonzero.

abs_level_minus_1

One less than the absolute value of the nonzero quantized AC coefficient that terminates the preceding run of zero-valued coefficients.

sign

A code indicating the sign of the nonzero quantized AC coefficient that terminates the preceding run of zero-valued coefficients. A value of 0 means the coefficient is positive; a value of 1 means it is negative.

zero_bit

A single bit with value 0. Used to ensure that the compressed color component data comprise an integral number of bytes.

zero_byte

An eight-bit number with value zero (0x00). This syntax element serves no useful purpose but will occasionally appear in ProRes bitstreams produced by older encoders.

6.3.3 Scanned Alpha Semantics

alpha_difference

The difference (exact or modulo) between the current alpha value and the previous one.

run

The number of consecutive occurrences of the current alpha value in the scanned alpha value array.

zero_bit

A single bit with value 0. Used to ensure that the compressed alpha component data comprise an integral number of bytes.

6.4 Bitstream Versions, Version Variants, and Compatibility

ProRes bitstreams can incorporate future changes to syntax or semantics. There are two approaches to accommodating this: bitstream versioning and version variants. Different bitstream versions denote intrinsic changes to the decoding process, while different version variants correspond to nonessential distinctions.

A new bitstream version is required if a desired change in bitstream syntax or semantics breaks compatibility with existing decoders, i.e., if existing decoders cannot properly decode such bitstreams. The bitstream version will be incremented for each such change. A decoder that can decode a ProRes bitstream with a particular bitstream version shall be able to decode a bitstream with any earlier (lower) bitstream version. A decoder shall refuse to decode a ProRes bitstream with an unsupported bitstream version. To maximize decoder compatibility, encoders should use the lowest bitstream version appropriate for the frame being encoded and the encoding parameters in effect.

Version variants correspond to the addition of informative data to ProRes bitstreams. Such additional data will not include information that is required for correct decoding, and furthermore will be added in a manner that does not prevent correct decoding by existing decoders that would otherwise be capable of decoding the bitstream. As a consequence all version variants of a ProRes bitstream version can be decoded by any decoder compatible with that bitstream version.

ProRes bitstreams contain no explicit identification of version variant. Because unrecognized version variant data can be present in a ProRes bitstream, for syntax structures with size specified in the bitstream, decoders shall use the specified size—rather than inference from the syntax itself—to determine the start of the immediately following syntax structure.

This specification describes bitstream versions 0 and 1. Version 0 bitstreams will have a value of 2 (4:2:2 sampling) for the chroma_format syntax element and a value of 0 (no encoded alpha) for the alpha_channel_type element; version 1 bitstreams can have any permissible value for those elements. No version variants have been defined for either bitstream version.

7 Decoding Process

This section describes the process that a decoder shall follow to reconstruct a frame from a ProRes bitstream. The process is carried out for each compressed slice in the bitstream and consists of these steps:

- Entropy decoding is applied to each of the compressed video components of the slice to produce arrays of scanned color component quantized discrete cosine transform (DCT) coefficients and, if the ProRes bitstream includes an encoded alpha channel, an array of raster-scanned alpha values;
- Inverse scanning is applied to each of the scanned color component quantized DCT coefficient arrays to produce blocks of color component quantized DCT coefficients;
- Inverse quantization is applied to each of the color component quantized DCT coefficient blocks to produce blocks of color component DCT coefficients;
- An inverse discrete cosine transform (IDCT) is applied to each of the color component DCT coefficient blocks to produce blocks of reconstructed color component values;
- Each of the reconstructed color component values is converted to an integral sample of desired bit depth and is written to the appropriate location in the decoded frame buffer (as are the decoded alpha values, if any).

7.1 Entropy Decoding

7.1.1 Scanned Coefficients

A ProRes compressed slice contains an entropy-coded array of scanned quantized DCT coefficients for each color component (Y′, Cb, and Cr). Quantized DC coefficients are encoded differentially, while quantized AC coefficients are run-length encoded. Variable-length coding is applied to the results using codebooks based on the Golomb-Rice and exponential-Golomb coding schemes.

7.1.1.1 Golomb Combination Codes

Golomb-Rice and exponential-Golomb codes are families of codebooks parameterized by a nonnegative integer order. All members of both families have as their symbol alphabets the nonnegative integers. Their codewords consist of three parts: a unary prefix consisting solely of ‘0’ bits (the length of which is referred to as the code level); a separator consisting of a single ‘1’ bit; and a binary suffix. Decoding is accomplished by counting the number of prefix bits and then appropriately combining that count with the suffix to recover the encoded symbol.

The Golomb-Rice code of order k encodes a nonnegative integer symbol n by first calculating the quotient and remainder of n with respect to 2^k, q = floor(n ÷ 2^k) and r = n mod 2^k. Then the codeword for n consists of a prefix of q ‘0’ bits, a single ‘1’ (separator) bit, and a k-bit suffix containing the binary representation of r; the length of the codeword is q + 1 + k. To decode an order-k Golomb-Rice codeword, the quotient/code level q is determined by counting the number of ‘0’ bits preceding the first ‘1’ bit, ignoring that ‘1’ bit, taking the last k bits as the remainder r, and finally reconstructing the encoded symbol n as q * 2^k + r.

The exponential-Golomb codes have a slightly more complex structure. For these the number of ‘0’ bits in the codeword prefix—the code level—is q = floor($\log_2$(n + 2^k)) − k, where again n is the nonnegative integer symbol being encoded and k is the code order. The suffix is the (q+k)-bit binary representation of r = n + 2^k − 2^{q+k}, making the codeword the binary representation of

the sum $n + 2^k$, zero-extended by q bits for a codeword length of $q + 1 + (q + k) = 2q + k + 1$. The encoded symbol n can be decoded from an order-k exponential-Golomb codeword by simply subtracting 2^k from the codeword.

ProRes variable-length coding uses exponential-Golomb codes but does not directly use Golomb-Rice codes. Rather, it uses combinations of the two, where nonnegative integer symbols n with small values are encoded with a Golomb-Rice code and those with larger values are encoded with an exponential-Golomb code. These Golomb-Rice/exponential-Golomb combination codes require three parameters to specify: the order of the associated Golomb-Rice code, k_{Rice}; the order of the associated exponential-Golomb code, k_{Exp}; and an indication of where to transition from one code to the other, lastRiceQ, defined as the largest value of q for which the Golomb-Rice code still applies. Values of n less than $(lastRiceQ + 1) * 2^{k_{Rice}}$ are encoded with the order-k_{Rice} Golomb-Rice code. For values of n greater than or equal to $(lastRiceQ + 1) * 2^{k_{Rice}}$, the combination codeword is obtained by concatenating first lastRiceQ + 1 '0' bits and then the result of encoding $n - (lastRiceQ + 1) * 2^{k_{Rice}}$ (itself a nonnegative integer) with the order-k_{Exp} exponential-Golomb code. Note: For all nonnegative integers k, the Golomb-Rice/exponential-Golomb combination code with lastRiceQ = 0, $k_{Rice} = k$, and $k_{Exp} = k + 1$ is identical to the exponential-Golomb code of order k.

To decode a codeword from the Golomb-Rice/exponential-Golomb combination code with parameters lastRiceQ, k_{Rice}, and k_{Exp}, first determine the code level q by counting the number of '0' bits preceding the first '1' bit. If q is lastRiceQ or less, decode the codeword as an order-k_{Rice} Golomb-Rice codeword to obtain the encoded symbol n. Otherwise discard the first lastRiceQ + 1 bits of the codeword (which are '0's), decode the remaining bits as an order-k_{Exp} exponential-Golomb codeword, and finally add $(lastRiceQ + 1) * 2^{k_{Rice}}$ to the result to obtain the encoded symbol n.

7.1.1.2 Signed Golomb Combination Codes

The symbol alphabets of the various Golomb codes are the nonnegative integers. To apply these codes to signed integers, a signed integer-to-symbol mapping S(n) is used. For ProRes this mapping is defined as

$$S(n) = \begin{cases} 2|n|, & n \geq 0 \\ 2|n| - 1, & n < 0, \end{cases}$$

as illustrated in Table 8.

Table 8 – Signed integer-to-symbol mapping S(n)

n	S(n)
0	0
−1	1
+1	2
−2	3
+2	4
−3	5
+3	6
…	…

Notice that even symbols correspond to nonnegative integers while odd symbols correspond to negative ones. The inverse mapping is thus

$$n = \begin{cases} S(n) / 2, & S(n) \text{ even} \\ -((S(n) + 1) / 2), & S(n) \text{ odd.} \end{cases}$$

7.1.1.3 DC Coefficients

The first quantized DC coefficient in the scanned coefficient array, first_dc_coeff, is determined by decoding an order-5 exponential-Golomb code codeword from the bitstream, then applying the inverse of the signed integer-to-symbol mapping S(n) to the decoded symbol.

The remaining quantized DC coefficients in the scanned coefficient array are encoded differentially. Variable-length coding is done adaptively, with the codebook for one dc_coeff_difference syntax element determined by the absolute value of the previous one. The adaptation is specified in Table 9, where EXP_GOLOMB_CODE(k) denotes the exponential-Golomb code of order k, RICE_EXP_COMBO_CODE(lastRiceQ, k_{Rice}, k_{Exp}) denotes the Golomb-Rice/ exponential-Golomb combination code with the indicated parameters, and previousDCDiff refers to the value of the previous dc_coeff_difference.

Table 9 – Codebook adaptation for dc_coeff_difference syntax element

|previousDCDiff|	Codebook
0	EXP_GOLOMB_CODE(0)
1	EXP_GOLOMB_CODE(1)
2	RICE_EXP_COMBO_CODE(1, 2, 3)
3 and above	EXP_GOLOMB_CODE(3)

To determine the value of a dc_coeff_difference syntax element, use the entry in Table 9 corresponding to the absolute value of previousDCDiff to decode a codeword from the bitstream, then apply the inverse of the signed integer-to-symbol mapping S(n) to the decoded symbol to obtain a signed integer n; dc_coeff_difference is n if previousDCDiff ≥ 0 and −n if previousDCDiff < 0. For each scanned coefficient array the value of previousDCDiff is initially set to 3 (so that the exponential-Golomb code of order 3 is used for the first dc_coeff_difference syntax element of the array) and is updated as each dc_coeff_difference syntax element is determined.

7.1.1.4 AC Coefficients

The quantized AC coefficients in the scanned coefficient array are run-length encoded. Runs consist of consecutive array elements with value zero. They are terminated either by a nonzero array element—a level—or by reaching the end of the array. Only runs that are terminated by levels are encoded into the bitstream; a final run, if there is one, is implicit.

Variable-length coding of runs and levels is done separately and adaptively. The codebook for one run syntax element is determined by the value of the previous one according to Table 10.

Table 10 – Codebook adaptation for scanned coefficients run syntax element

previousRun	Codebook
0	RICE_EXP_COMBO_CODE(2, 0, 1)
1	RICE_EXP_COMBO_CODE(2, 0, 1)
2	RICE_EXP_COMBO_CODE(1, 0, 1)
3	RICE_EXP_COMBO_CODE(1, 0, 1)
4	EXP_GOLOMB_CODE(0)
5	RICE_EXP_COMBO_CODE(1, 1, 2)
6	RICE_EXP_COMBO_CODE(1, 1, 2)
7	RICE_EXP_COMBO_CODE(1, 1, 2)
8	RICE_EXP_COMBO_CODE(1, 1, 2)
9	EXP_GOLOMB_CODE(1)
10	EXP_GOLOMB_CODE(1)
11	EXP_GOLOMB_CODE(1)
12	EXP_GOLOMB_CODE(1)
13	EXP_GOLOMB_CODE(1)
14	EXP_GOLOMB_CODE(1)
15 and above	EXP_GOLOMB_CODE(2)

The value of a run syntax element is the symbol obtained by decoding a codeword from the bitstream using the entry in Table 10 corresponding to previousRun (the value of the previous run syntax element). For each scanned coefficient array the value of previousRun is initially set to 4 (so that the exponential-Golomb code of order 0 is used for the first run syntax element of the array) and is updated as each run syntax element is determined.

Levels are encoded in sign-magnitude fashion. The abs_level_minus_1 syntax element provides the level symbol, which is one less than the absolute value of the level, and the sign syntax element indicates the sign of the level. The codebook for one abs_level_minus_1 syntax element is determined by the value of the previous one according to Table 11.

Table 11 – Codebook adaptation for abs_level_minus_1 syntax element

previousLevelSymbol	Codebook
0	RICE_EXP_COMBO_CODE(2, 0, 2)
1	RICE_EXP_COMBO_CODE(1, 0, 1)
2	RICE_EXP_COMBO_CODE(2, 0, 1)
3	EXP_GOLOMB_CODE(0)
4	EXP_GOLOMB_CODE(1)
5	EXP_GOLOMB_CODE(1)
6	EXP_GOLOMB_CODE(1)
7	EXP_GOLOMB_CODE(1)
8 and above	EXP_GOLOMB_CODE(2)

The value of an abs_level_minus_1 syntax element is the symbol obtained by decoding a codeword from the bitstream using the entry in Table 11 corresponding to previousLevelSymbol

(the value of the previous abs_level_minus_1 syntax element). For each scanned coefficient array the initial previous level is taken to be 2. The value of previousLevelSymbol is thus initially set to |2| − 1 = 1 (so that the Golomb-Rice/exponential-Golomb combination code with lastRiceQ 1, Rice order 0, and exponential order 1 is used for the first abs_level_minus_1 syntax element of the array) and is updated as each abs_level_minus_1 syntax element is determined. The level itself is calculated from the values of the abs_level_minus_1 and sign syntax elements according to the formula

level = (abs_level_minus_1 + 1) * (1 − 2 * sign).

7.1.2 Scanned Alpha

When the value of the alpha_channel_type syntax element is not zero, ProRes compressed slices contain an entropy-coded array of raster-scanned alpha (transparency) component values. The values in the array are run-length encoded, differences between alpha values of successive runs are calculated, and variable-length coding is applied to the results.

Runs consist of consecutive array elements with the same value (and hence always have length one or more). They terminate when one of the following occurs: two consecutive array elements have different values; the end of the array is reached; or the maximum permissible run length, 2048, is achieved. Variable-length coding of run lengths uses the code in Table 12.

Table 12 – Variable-length code for scanned alpha run lengths

Run length	Codeword
1	1
2	00001
3	00010
…	…
15	01110
16	01111
17	0000000000010000
18	0000000000010001
…	…
2047	0000011111111110
2048	0000011111111111

Run lengths of one are assigned a codeword consisting of a single ‘1’ bit. Run lengths between 2 and 16, inclusive, are assigned five-bit codewords that begin with a ‘0’ bit and contain at least one ‘1’ bit. Run lengths between 17 and 2048, inclusive, are assigned codewords consisting of an escape code of five ‘0’ bits followed by an eleven-bit fixed-length code containing the binary representation of one less than the run length. (Note: Each five-bit codeword also happens to be the binary representation of one less than the run length to which it corresponds.) The value of a run syntax element is the run length obtained by decoding a codeword from the bitstream using Table 12.

Run alpha values are encoded differentially. Each alpha_difference syntax element provides either the exact or the modulo difference between the alpha value of its corresponding run and that of the immediately previous run. For the first run, the previous alpha value is taken to be −1. Reconstruction of the alpha value of a run—denoted simply by alpha—from that

of the immediately previous run—denoted by previousAlpha—and the value of the run's alpha_difference syntax element is done according to the formula

alpha = previousAlpha + alpha_difference

when the value of the syntax function isModuloAlphaDifference() is false or else the formula

alpha = (previousAlpha + alpha_difference) & mask

when the value of isModuloAlphaDifference() is true, where mask is either the 8-bit value 0xFF (when the value of the alpha_channel_type syntax element is 1) or the 16-bit value 0xFFFF (when alpha_channel_type is 2). A two's complement representation shall be used for the value of the sum so that the bitwise-and corresponds to a modulo operation. Note: If it permits a simpler or more efficient implementation, the bitwise-and (modulo) operation can also be done when isModuloAlphaDifference() is false, as it has no effect in that case.

Variable-length coding of alpha differences uses one of two different codes depending on the value of alpha_channel_type. When alpha_channel_type is 1 (8-bit alpha), the code in Table 13 is used.

Table 13 – Variable-length code for alpha differences, alpha_channel_type = 1 (8-bit alpha)

Alpha difference	Codeword
+1	00000
−1	00001
+2	00010
−2	00011
…	…
+8	01110
−8	01111
Other	Escaped FLC

Differences with absolute value between 1 and 8, inclusive, are assigned codewords consisting of a '0' bit followed first by the three-bit binary representation of one less than the absolute value of the difference and then by a one-bit indicator of the sign of the difference ('0' for positive, '1' for negative). Differences with absolute value greater than 8, or equal to 0, are assigned codewords consisting of an escape bit of '1' followed by an eight-bit fixed-length code (FLC) containing the binary representation of the value of the difference mod 256. To determine the value of an alpha_difference syntax element and that of the associated syntax function isModuloAlphaDifference() when alpha_channel_type is 1, first observe whether the next bit of the bitstream is a '0' or a '1'. If '0', isModuloAlphaDifference() is false and the subsequent four bits of the bitstream encode the absolute value and sign of alpha_difference according to Table 13 and the above description. If '1', isModuloAlphaDifference() is true and the subsequent eight bits of the bitstream directly provide the binary representation of alpha_difference. Note: If it permits a simpler or more efficient implementation, when the first bit of the codeword representing alpha_difference is '1' the full nine-bit codeword can also be used as the binary representation of alpha_difference; the effect of including the escape bit will be eliminated by the bitwise-and (modulo) operation in the reconstruction calculation.

When alpha_channel_type is 2 (16-bit alpha), the code in Table 14 is used for alpha differences.

Table 14 – Variable-length code for alpha differences, alpha_channel_type = 2 (16-bit alpha)

Alpha difference	Codeword
+1	00000000
−1	00000001
+2	00000010
−2	00000011
…	…
+64	01111110
−64	01111111
Other	Escaped FLC

Differences with absolute value between 1 and 64, inclusive, are assigned codewords consisting of a ‘0’ bit followed first by the six-bit binary representation of one less than the absolute value of the difference and then by a one-bit indicator of the sign of the difference (‘0’ for positive, ‘1’ for negative). Differences with absolute value greater than 64, or equal to 0, are assigned codewords consisting of an escape bit of ‘1’ followed by a sixteen-bit fixed-length code containing the binary representation of the value of the difference mod 65536. To determine the value of an alpha_difference syntax element and that of the associated syntax function isModuloAlphaDifference() when alpha_channel_type is 2, first observe whether the next bit of the bitstream is a ‘0’ or a ‘1’. If ‘0’, isModuloAlphaDifference() is false and the subsequent seven bits of the bitstream encode the absolute value and sign of alpha_difference according to Table 14 and the above description. If ‘1’, isModuloAlphaDifference() is true and the subsequent sixteen bits of the bitstream directly provide the binary representation of alpha_difference. Note: If it permits a simpler or more efficient implementation, when the first bit of the codeword representing alpha_difference is ‘1’ the full seventeen-bit codeword can also be used as the binary representation of alpha_difference; the effect of including the escape bit will be eliminated by the bitwise-and (modulo) operation in the reconstruction calculation.

7.2 Inverse Scanning

ProRes uses a combination of block and slice scanning to obtain each array of scanned color component quantized DCT coefficients from the individual quantized DCT coefficient blocks of the corresponding color component.

7.2.1 Slice Scanning

Slice scanning groups all quantized DCT coefficients with the same scanned frequency index together in the scanned coefficient array, ordered from those with the lowest scanned frequency index (0) at the start of the array to those with the highest (63) at the end. For each scanned frequency index n, the quantized DCT coefficients in the array follow the order of the corresponding macroblock within the slice, and among blocks of the same macroblock, the order of the corresponding block within the macroblock. This arrangement is illustrated in Figure 3.

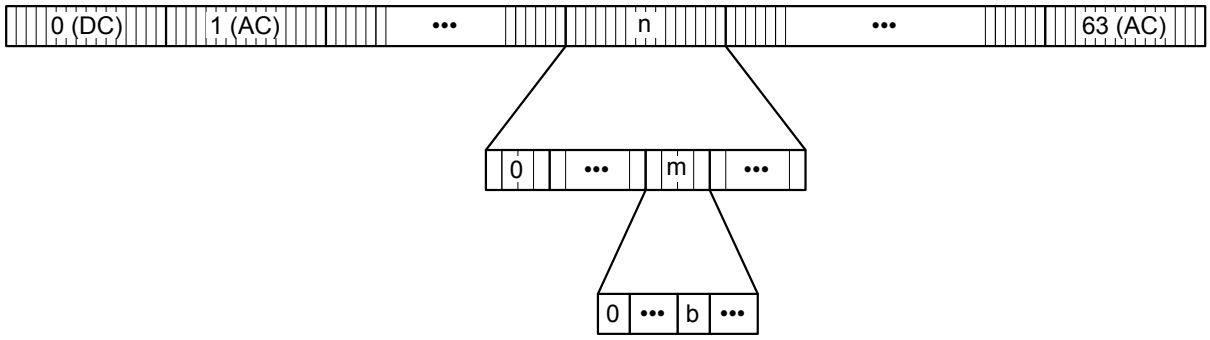

Figure 3 – Arrangement of quantized DCT coefficients in scanned coefficient array

Let sliceSizeInMb refer to the number of macroblocks forming the slice and nB to the number of blocks forming each macroblock; sliceSizeInMb is slice_size_in_mb[j] for slice(i, j) and nB is 4 for the luma (Y′) component and nC for the chroma (Cb, Cr) components, where nC is 2 when the value of chroma_format is 2 (4:2:2 sampling) and 4 when the value of that syntax element is 3 (4:4:4 sampling). Calling the scanned coefficient array scannedCoeffs[] and using $QFS_{m,b}$[] to refer to the scanned quantized DCT coefficient block with macroblock index m ($0 \le m < sliceSizeInMb$) and block index b ($0 \le b < nB$), then, inverse slice scanning is given by the formula

$$QFS_{m,b}[n] = scannedCoeffs[nB * sliceSizeInMb * n + nB * m + b]$$

($0 \le n < 64$). Note: Because nB and sliceSizeInMb are each a power of two and furthermore there are no carries among the addends, the above indexing calculation can be carried out with bit shift and bitwise-or operations instead of multiplication and addition; conversely, n, m, and b can be determined from a scanned coefficient array index using only bit shift and bit mask operations.

7.2.2 Block Scanning

Block scanning orders each 8×8 block of quantized DCT coefficients QF[][] into a corresponding scanned quantized DCT coefficient block QFS[] (the macroblock and block index subscripts have been dropped as there is no need to distinguish among them). The same scan pattern is used for all blocks and color components. The pattern does depend, however, on the value of the interlace_mode syntax element. When the value of interlace_mode is 0—when the block is part of a frame picture—the progressive scan pattern of Figure 4 is used.

v \ u	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	21	22
1	2	3	6	7	18	20	23	28
2	8	9	12	13	19	24	27	29
3	10	11	14	15	25	26	30	31
4	32	33	37	38	45	46	53	54
5	34	36	39	44	47	52	55	60
6	35	40	43	48	51	56	59	61
7	41	42	49	50	57	58	62	63

Figure 4 – Block scan pattern, interlace_mode = 0 (progressive)

When on the other hand the value of interlace_mode is not 0—when the block is part of a field picture—the interlaced scan pattern of Figure 5 is used.

v \ u	0	1	2	3	4	5	6	7
0	0	2	8	10	32	34	35	41
1	1	3	9	11	33	36	40	42
2	4	6	12	14	37	39	43	49
3	5	7	13	15	38	44	48	50
4	16	18	19	25	45	47	51	57
5	17	20	24	26	46	52	56	58
6	21	23	27	30	53	55	59	62
7	22	28	29	31	54	60	61	63

Figure 5 – Block scan pattern, interlace_mode not 0 (interlaced)

Setting scan[][] to the relevant scan pattern, inverse block scanning is given by the formula

QF[v][u] = QFS[scan[v][u]]

($0 \le u, v < 8$).

7.3 Inverse Quantization

Inverse quantization produces a block F[][] of dequantized DCT coefficients from a quantized counterpart QF[][] using a quantization weight matrix W[][] and a quantization scale factor qScale. The latter sets the overall level of quantization for the block while the former permits frequency-dependent variation. Quantization weight matrices are specified in the frame header and apply to all blocks of the frame, while quantization scale factors are specified in each slice header and apply to all blocks of the corresponding slice.

The quantization weights W[v][u] are integers between 2 and 63, inclusive. There are distinct quantization weight matrices for the luma (Y′) and chroma (Cb, Cr) video components, denoted

respectively by lumaQuantizationMatrix[][] and chromaQuantizationMatrix[][]. If the value of the load_luma_quantization_matrix syntax element is 1, then the frame header includes the luma_quantization_matrix syntax element and this is used for the luma quantization weight matrix, lumaQuantizationMatrix[v][u] = luma_quantization_matrix[v][u] for 0 ≤ u, v < 8. If the value of load_luma_quantization_matrix is 0, on the other hand, a default matrix with all 64 weights set to 4 is used, i.e., lumaQuantizationMatrix[v][u] = 4 for 0 ≤ u, v < 8. Likewise, if the value of the load_chroma_quantization_matrix syntax element is 1, the chroma_quantization_matrix syntax element included in the frame header is used for the chroma quantization weight matrix, chromaQuantizationMatrix[v][u] = chroma_quantization_matrix[v][u] for 0 ≤ u, v < 8. However if the value of load_chroma_quantization_matrix is 0, the luma quantization weight matrix is used for the chroma one as well, chromaQuantizationMatrix[v][u] = lumaQuantizationMatrix[v][u] for 0 ≤ u, v < 8. W[][] is then lumaQuantizationMatrix[][] for the luma component and chromaQuantizationMatrix[][] for the chroma components.

The quantization scale factor qScale for a slice is determined by the value of the quantization_index syntax element in the slice header—which is restricted to be between 1 and 224 inclusive—according to Table 15.

Table 15 – Quantization scale factor qScale as a function of quantization_index

quantization_index	qScale
1	1
2	2
…	…
126	126
127	127
128	128
129	132
130	136
…	…
223	508
224	512

In compact form the relationship is

$$\text{qScale} = \begin{cases} \text{quantization\_index}, & 1 \le \text{quantization\_index} \le 128 \\ 128 + 4 * (\text{quantization\_index} - 128), & 129 \le \text{quantization\_index} \le 224. \end{cases}$$

The dequantized DCT coefficients are calculated from the quantized ones, the quantization weights, and the quantization scale factor using the formula

$$F[v][u] = (QF[v][u] * W[v][u] * qScale) \div 8$$

(0 ≤ u, v < 8). Because the results of this calculation are not always integral, they require a fixed-point or floating-point representation. Notice that the quantized DCT coefficients, the quantization weights, and the quantization scale factor are all integral, so the dequantized DCT coefficients will always be multiples of 1/8; three fraction bits are sufficient to provide exact representation. At least two fraction bits (i.e., quarter-integer precision) shall be retained for the subsequent inverse transform.

7.4 Inverse Transform

Reconstructed color component values f[y][x] are obtained from a block of dequantized DCT coefficients F[][] by applying an 8×8 two-dimensional inverse discrete cosine transform (IDCT),

$$f[y][x] = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)\, C(v)\, F[v][u]\, \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

($0 \le x, y < 8$), where $C(0) = 1/\sqrt{2}$ and $C(n) = 1$ for $n = 1, \ldots, 7$. The IDCT follows the conventions of IEEE Std 1180-1990, specifically the expectation that the DCT coefficients have the range of 12-bit signed integers while the reconstructed values have the range of 9-bit signed integers.

Either a fixed-point or a floating-point implementation of the IDCT is acceptable. The implementation shall be capable of accommodating DCT coefficients with at least two bits of fractional precision. To ensure that the IDCT calculation is sufficiently accurate, the implementation shall comply with Annex A, "IDCT Implementation Accuracy Qualification." All fraction bits of the reconstructed values arising from the implementation should be preserved for conversion of the reconstructed values to pixel component samples.

7.5 Pixel Component Sample Generation and Pixel Output

The final step of the decoding process is to generate (integral) pixel component samples of desired bit depth from the reconstructed color component values and decoded alpha values (if any) and then write them to the appropriate locations in the decoded frame buffer.

7.5.1 Color Component Samples

The reconstructed color component values produced by the IDCT are offset and normalized to a range comparable to that of 9-bit signed integers, specifically greater than or equal to −256 and less than 256. Converting these to pixel component samples requires restoring mid-range bias and scaling to the bit depth of the component samples. Using v to denote a (fixed-point or floating-point) reconstructed color component value and s to denote the corresponding (unsigned integer) pixel component sample, this is

$$s = \text{clamp}(\text{round}(2^b * (v + 256) \div 512)),$$

where b is the number of bits per component sample and clamp(n) restricts its argument to the appropriate range,

$$\text{clamp}(n) = \begin{cases} n_{min}, & n < n_{min} \\ n_{max}, & n > n_{max} \\ n, & \text{otherwise.} \end{cases}$$

The clamping bounds n_{min} and n_{max} may be set respectively to 0 and $2^b - 1$ to produce pixel component samples that utilize all available quantization levels or to the smallest and largest permissible video quantization levels for b-bit samples if avoidance of ITU-R BT.601/BT.709 synchronization/timing reference quantization levels is desired.

The correspondence between the quantization levels of the resulting pixel component samples and component video signal levels is consistent with those of ITU-R BT.601 and ITU-R BT.709. For sample bit depths $b \ge 8$, black and nominal peak white luma signal levels correspond respectively to the luma sample quantization levels $16 * 2^{b-8}$ and $235 * 2^{b-8}$; achromatic and

nominal peak chroma signal levels correspond respectively to the chroma sample quantization levels $128 * 2^{b-8}$, $16 * 2^{b-8}$, and $240 * 2^{b-8}$.

7.5.2 Alpha Component Samples

ProRes encodes alpha channel information losslessly. If the pixel component sample bit depth matches that of the decoded alpha values, the pixel alpha component samples shall be the decoded values themselves and will be identical to the encoded samples. Otherwise the decoded alpha values shall be promoted or demoted as necessary to the pixel component sample bit depth. This should be done by treating the smallest and largest possible alpha sample values as respectively signifying opacities of exactly 0.0 and 1.0. For 8-bit decoded alpha values (when the value of the alpha_channel_type syntax element is 1), the conversion is

$$\text{alphaSample} = \text{round}((2^b - 1) * \text{alpha} \div 255),$$

while for 16-bit decoded alpha values (when alpha_channel_type is 2) it is

$$\text{alphaSample} = \text{round}((2^b - 1) * \text{alpha} \div 65535),$$

where alpha is a decoded alpha value, alphaSample is the corresponding pixel alpha component sample, and b is the number of bits per pixel component sample.

7.5.3 Pixel Arrangement

The individual rows of pixels and pixels within each row of a frame, picture, slice, macroblock, or block are identified using zero-based indexing sequenced from top to bottom (pixel rows) or left to right (pixels within a row). For a frame, for instance, row 0 refers to the top row of the frame, row 1 to the row immediately below the top one, and so forth; pixel 0 refers to the leftmost pixel of a row of the frame, pixel 1 to the pixel immediately to the right of pixel 0, and so forth.

If the frame being decoded is progressive, it consists of a single picture, and that picture is identical to the frame: pixel n of row r of the picture is pixel n of row r of the frame. If the frame is interlaced, however, it consists of two pictures, each of which consists of alternate rows of the frame. Specifically, pixel n of row r of the top field picture is pixel n of row 2r of the frame while pixel n of row r of the bottom field picture is pixel n of row 2r + 1 of the frame.

Slices are identified by the macroblock row i they occupy in their encoded picture, $0 \le i <$ height_in_mb, and an index j indicating their position within the macroblock row, $0 \le j <$ number_of_slices_per_mb_row. Calling $\text{mbOffset}(j) = \sum_{j'=0}^{j-1} \text{slice\_size\_in\_mb}[j']$ (for $j \ge 1$; $\text{mbOffset}(0) = 0$), pixel n of row r of slice(i, j) is pixel 16 * mbOffset(j) + n of row 16i + r of the encoded picture that incorporates the slice. Keep in mind that the dimensions of the encoded picture can exceed those of the source picture. If 16 * width_in_mb > horizontal_size, the last 16 * width_in_mb − horizontal_size pixels of each row of the decoded slices with j = number_of_slices_per_mb_row – 1 shall be discarded, i.e., not written to the decoded frame buffer; if 16 * height_in_mb > picture_vertical_size, the last 16 * height_in_mb − picture_vertical_size rows of pixels of the decoded slices with i = height_in_mb − 1 shall not be written to the decoded frame buffer.

The macroblocks that constitute a slice are indexed from left to right starting with index 0. Hence pixel n of row r of macroblock m of a slice is pixel 16m + n of row r of that slice. Within a macroblock, the constituent blocks are indexed separately for each (color) component. The four luma (Y′) blocks are arranged by block index as shown in Figure 6.

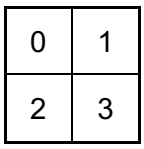

Figure 6 – Order of luma (Y′) blocks within macroblock

For 4:2:2 sampling (when the value of the chroma_format syntax element is 2), the two Cb blocks and two Cr blocks are each arranged by block index as shown in Figure 7.

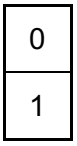

Figure 7 – Order of chroma (Cb, Cr) blocks within macroblock, 4:2:2 sampling

For 4:4:4 sampling (when chroma_format is 3), the four Cb blocks and four Cr blocks are each arranged by block index as shown in Figure 8. Note: The arrangement of the 4:4:4 chroma blocks (Figure 8) differs from that of the luma blocks (Figure 6).

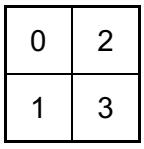

Figure 8 – Order of chroma (Cb, Cr) blocks within macroblock, 4:4:4 sampling

Within a slice alpha component data are organized not as macroblocks and blocks but rather as an array of raster-scanned values. Calling the array alphaValues[], for slice(i, j) the decoded alpha value corresponding to pixel n of row r is alphaValues[16 * slice_size_in_mb[j] * r + n]. The array includes alpha values—which shall be discarded—for the excess pixel(s) at the end of each row of slices with j = number_of_slices_per_mb_row – 1 when 16 * width_in_mb > horizontal_size but does not include alpha values for the excess row(s) of pixels at the bottom of slices with i = height_in_mb – 1 when 16 * height_in_mb > picture_vertical_size.

Annex A IDCT Implementation Accuracy Qualification

ProRes decoders are not required to use any particular implementation of the inverse discrete cosine transform (IDCT) calculation. To ensure the quality of decoded image data, however, only implementations that are sufficiently accurate are acceptable. This annex specifies how to qualify an IDCT implementation for use in a ProRes decoder.

The qualification procedure is based on the one in Section 3 of IEEE Std 1180-1990. As in Section 3.2 of that standard, there are seven steps involved in measuring the accuracy of a proposed IDCT implementation. They are:

(1) Generate random integer data values in the range −L to +H using the random number generator in the appendix of IEEE Std 1180-1990. Arrange the integer data values into 8×8 blocks of “reference-precision” (i.e., at least 64-bit) floating-point pixels by converting them to reference-precision floating-point numbers, dividing by 8, and populating the blocks in row-major order. Data sets of 10,000 blocks each shall be generated for (L = 2048, H = 2047), (L = H = 40), and (L = H = 2400).
(2) For each 8×8 block of reference-precision floating-point pixels produced by step 1, perform a separable, orthogonal FDCT (as defined in Eq 1 of IEEE Std 1180-1990) using reference-precision floating-point arithmetic.
(3) For each 8×8 block of transformed coefficients produced by step 2, round the 64 coefficients to the nearest quarter-integer by multiplying by 4, rounding to the nearest integer, and then dividing by 4. Clip the rounded coefficients to the range −2048 to +2047.75. These blocks, which consist of values that can be represented using a signed binary fixed-point format with 12 or more integer bits and 2 or more fraction bits, are the input data to the inverse transforms.
(4) For each 8×8 block of data produced by step 3, perform a separable, orthogonal IDCT using reference-precision floating-point arithmetic. Retaining full reference precision—specifically, *without* rounding to integers or any other lesser precision—clip the resulting values to the range −256 to +256. These blocks of 8×8 reference-precision floating-point pixels are the “reference” IDCT output data.
(5) For each 8×8 block of data produced by step 3, perform an IDCT using the proposed implementation (or a bit-accurate equivalent). Retaining the full precision of the results, promote them as necessary to reference-precision floating-point numbers and clip those values to the range −256 to +256. These blocks of 8×8 reference-precision floating-point pixels are the “test” IDCT output data.
(6) For each of the 64 IDCT output pixels and for each of the 10,000 block data sets generated by steps 1–5, measure the peak, mean, and mean square errors between the “reference” data and the “test” data. The error calculation and accumulation shall be carried out with reference-precision floating-point arithmetic.
(7) Rerun the measurements using exactly the same integer data values of step 1, but change the sign on each one.

Note: The distinctions between the procedure described above and the one in Section 3.2 of IEEE Std 1180-1990 are that here the pixel data produced by step 1 have three fraction bits (rather than being integers), the transformed coefficients produced by step 2 are rounded to the nearest quarter-integer (rather than integer) and clipped to the range −2048 to +2047.75 (rather than −2048 to +2047), the IDCT output data retain the full precision of their respective calculations (rather than being rounded to integers) and are clipped to the range −256 to +256 (rather than −256 to +255), and the error measurements use reference-precision floating-point (rather than integer) arithmetic.

Using the error term definitions in Section 3.3 of IEEE Std 1180-1990, the (reference-precision floating-point) errors measured according to the above procedure shall meet the following criteria:

(1) For any pixel location, the peak error (*ppe*) shall not exceed 0.15 in magnitude.
(2) For any pixel location, the mean square error (*pmse*) shall not exceed 0.002.
(3) Overall, the mean square error (*omse*) shall not exceed 0.001.
(4) For any pixel location, the mean error (*pme*) shall not exceed 0.0015 in magnitude.
(5) Overall, the mean error (*ome*) shall not exceed 0.00015 in magnitude.