Khronos Data Format Specification v1.3.1

Andrew Garrard

Khronos Data Format Specification License Information

Copyright (C) 2014-2019 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast, or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

This version of the Data Format Specification is published and copyrighted by Khronos, but is not a Khronos ratified specification. Accordingly, it does not fall within the scope of the Khronos IP policy, except to the extent that sections of it are normatively referenced in ratified Khronos specifications. Such references incorporate the referenced sections into the ratified specifications, and bring those sections into the scope of the policy for those specifications.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be reformatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group website should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents, or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, SYCL, SPIR, WebGL, EGL, COLLADA, StreamInput, OpenVX, OpenKCam, glTF, OpenKODE, OpenVG, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL and OpenMAX DL are trademarks and WebCL is a certification mark of the Khronos Group Inc. OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Revision History		
Revision 0.1	Jan 2015	AG
Initial sharing		
Revision 0.2	Feb 2015	AG
Added clarification, tables, examples		
Revision 0.3	Feb 2015	AG
Further cleanup		
Revision 0.4	Apr 2015	AG
Channel ordering standardized		
Revision 0.5	Apr 2015	AG
Typos and clarification		
Revision 1.0 rev 1	May 2015	AG
Submission for 1.0 release		
Revision 1.0 rev 2	Jun 2015	AG
Clarifications for 1.0 release		
Revision 1.0 rev 3	Jul 2015	AG
Added KHR_DF_SAMPLE_DATATYPE_LINEAR		
Revision 1.0 rev 4	Jul 2015	AG
Clarified KHR_DF_SAMPLE_DATATYPE_LINEAR		
Revision 1.0 rev 5	Mar 2019	AG
Clarification and typography		
Revision 1.1 rev 1	Nov 2015	AG
Added definitions of compressed texture formats		
Revision 1.1 rev 2	Jan 2016	AG
Added definitions of floating point formats		
Revision 1.1 rev 3	Feb 2016	AG
Fixed typo in sRGB conversion (thank you, Tom Grim!)		
Revision 1.1 rev 4	Mar 2016	AG

Fixed typo/clarified sRGB in ASTC, typographical improvements		
Revision 1.1 rev 5	Mar 2016	AG
Switch to official Khronos logo, removed scripts, restored title		
Revision 1.1 rev 6	Jun 2016	AG
ASTC block footprint note, fixed credits/changelog/contents		
Revision 1.1 rev 7	Sep 2016	AG
ASTC multi-point part and quint decode typo fixes		
Revision 1.1 rev 8	Jun 2017	AG
ETC2 legibility and table typo fix		
Revision 1.1 rev 9	Mar 2019	AG
Typo fixes and much reformatting		
Revision 1.2 rev 0	Sep 2017	AG
Added color conversion formulae and extra options		
Revision 1.2 rev 1	Mar 2019	AG
Typo fixes and much reformatting		
Revision 1.3	Oct 2019	AG/MC
Updates for KTX2/glTF. BC6h and ASTC table fixes and typo fixes. More examples.		
Revision 1.3.1	Mar 2020	MC
Fix swapped pre-multiplied alpha in DXT[2345] examples. Fix incorrect positions in BC7 bit layout table. Rewrite Unsized formats section.		

Table of Contents

1. Introduction
2. Formats and texel access
    - 2.1. 1-D texel addressing
    - 2.2. Simple 2-D texel addressing
    - 2.3. More complex 2-D texel addressing
    - 2.4. 3-dimensional texel addressing
    - 2.5. Downsampled channels
3. The Khronos Data Format Descriptor overview
    - 3.1. *Texel blocks* in the Khronos Data Format Descriptor
    - 3.2. *Planes* in the Khronos Data Format Specification
    - 3.3. Bit pattern interpretation and *samples*
    - 3.4. Canonical representation
    - 3.5. Related concepts outside the "format"
    - 3.6. Translation to API-specific representations
4. Khronos Data Format Descriptor
    - 4.1. Descriptor block
5. Khronos Basic Data Format Descriptor Block
    - 5.1. ***vendorId***
    - 5.2. ***descriptorType***
    - 5.3. ***versionNumber***
    - 5.4. ***descriptorBlockSize***
    - 5.5. ***colorModel***
    - 5.6. ***colorModel*** for compressed formats
    - 5.7. ***colorPrimaries***
    - 5.8. ***transferFunction***
    - 5.9. ***flags***
    - 5.10. ***texelBlockDimension[0..3]***
    - 5.11. ***bytesPlane[0..7]***
    - 5.12. Sample information
    - 5.13. Sample ***bitOffset***
    - 5.14. Sample ***bitLength***
    - 5.15. Sample ***channelType*** and qualifiers
    - 5.16. ***samplePosition[0..3]***
    - 5.17. ***sampleLower*** and ***sampleUpper***
    - 5.18. Paletted formats
    - 5.19. Unsized formats
    - 5.20. C99 struct mapping (informative)
6. Extension for more complex formats
7. Additional planes descriptor block
8. Additional dimensions descriptor block
9. Frequently Asked Questions
    - 9.1. Why have a binary format rather than a human-readable one?
    - 9.2. Why not use an existing representation such as those on FourCC.org?
    - 9.3. Why have a descriptive format?
    - 9.4. Why describe this standard within Khronos?
    - 9.5. Why should I use this descriptor if I don't need most of the fields?
    - 9.6. Why not expand each field out to be integer for ease of decoding?
    - 9.7. Can this descriptor be used for text content?

10. Floating-point formats
- 10.1. 16-bit floating-point numbers
- 10.2. Unsigned 11-bit floating-point numbers
- 10.3. Unsigned 10-bit floating-point numbers
- 10.4. Non-standard floating point formats

11. Example format descriptors
12. Introduction to color conversions
- 12.1. Color space composition
- 12.2. Operations in a color conversion

13. Transfer functions
- 13.1. About transfer functions (informative)
- 13.2. ITU transfer functions
- 13.3. sRGB transfer functions
- 13.4. BT.1886 transfer functions
- 13.5. BT.2100 HLG transfer functions
- 13.6. BT.2100 PQ transfer functions
- 13.7. DCI P3 transfer functions
- 13.8. Legacy NTSC transfer functions
- 13.9. Legacy PAL OETF
- 13.10. Legacy PAL 625-line EOTF
- 13.11. ST240/SMPTE240M transfer functions
- 13.12. Adobe RGB (1998) transfer functions
- 13.13. Sony S-Log transfer functions
- 13.14. Sony S-Log2 transfer functions
- 13.15. ACEScc transfer function
- 13.16. ACEScct transfer function

14. Color primaries
- 14.1. BT.709 color primaries
- 14.2. BT.601 625-line color primaries
- 14.3. BT.601 525-line color primaries
- 14.4. BT.2020 color primaries
- 14.5. NTSC 1953 color primaries
- 14.6. PAL 525-line analog color primaries
- 14.7. ACES color primaries
- 14.8. ACEScc color primaries
- 14.9. Display P3 color primaries
- 14.10. Adobe RGB (1998) color primaries
- 14.11. BT.709/BT.601 625-line primary conversion example
- 14.12. BT.709/BT.2020 primary conversion example

15. Color models
- 15.1. $Y'C_BC_R$ color model
- 15.2. $Y'_CC'_{BC}C'_{CR}$ constant luminance color model
- 15.3. IC_TC_P constant intensity color model

16. Quantization schemes
- 16.1. “Narrow range” encoding
- 16.2. “Full range” encoding
- 16.3. Legacy “full range” encoding.

17. Compressed Texture Image Formats
- 17.1. Terminology

18. S3TC Compressed Texture Image Formats
- 18.1. BC1 with no alpha
- 18.2. BC1 with alpha
- 18.3. BC2
- 18.4. BC3

19. RGTC Compressed Texture Image Formats
- 19.1. BC4 unsigned
- 19.2. BC4 signed
- 19.3. BC5 unsigned
- 19.4. BC5 signed

20. BPTC Compressed Texture Image Formats
- 20.1. BC7
- 20.2. BC6H

21. ETC1 Compressed Texture Image Formats
- 21.1. ETC1S

22. ETC2 Compressed Texture Image Formats
- 22.1. Format RGB ETC2
- 22.2. Format RGB ETC2 with sRGB encoding
- 22.3. Format RGBA ETC2
- 22.4. Format RGBA ETC2 with sRGB encoding

22.5. Format Unsigned R11 EAC
22.6. Format Unsigned RG11 EAC
22.7. Format Signed R11 EAC
22.8. Format Signed RG11 EAC
22.9. Format RGB ETC2 with punchthrough alpha
22.10. Format RGB ETC2 with punchthrough alpha and sRGB encoding
23. ASTC Compressed Texture Image Formats
23.1. What is ASTC?
23.2. Design Goals
23.3. Basic Concepts
23.4. Block Encoding
23.5. LDR and HDR Modes
23.6. Configuration Summary
23.7. Decode Procedure
23.8. Block Determination and Bit Rates
23.9. Block Layout
23.10. Block mode
23.11. Color Endpoint Mode
23.12. Integer Sequence Encoding
23.13. Endpoint Unquantization
23.14. LDR Endpoint Decoding
23.15. HDR Endpoint Decoding
23.16. Weight Decoding
23.17. Weight Unquantization
23.18. Weight Infill
23.19. Weight Application
23.20. Dual-Plane Decoding
23.21. Partition Pattern Generation
23.22. Data Size Determination
23.23. Void-Extent Blocks
23.24. Illegal Encodings
23.25. LDR PROFILE SUPPORT
23.26. HDR PROFILE SUPPORT
24. PVRTC Compressed Texture Image Formats
24.1. PVRTC Overview
24.2. Format PVRTC1 4bpp
24.3. Format PVRTC1 2bpp
24.4. Format PVRTC2 4bpp
24.5. Format PVRTC2 2bpp
25. External references
26. Contributors

Abstract

This document describes a data format specification for non-opaque (user-visible) representations of user data to be used by, and shared between, Khronos standards. The intent of this specification is to avoid replication of incompatible format descriptions between standards and to provide a definitive mechanism for describing data that avoids excluding useful information that may be ignored by other standards. Other APIs are expected to map internal formats to this standard scheme, allowing formats to be shared and compared. This document also acts as a reference for the memory layout of a number of common compressed texture formats, and describes conversion between a number of common color spaces.

1. Introduction

Many APIs operate on bulk data — buffers, images, volumes, etc. — each composed of many elements with a fixed and often simple representation. Frequently, multiple alternative representations of data are supported: vertices can be represented with different numbers of dimensions, textures may have different bit depths and channel orders, and so on. Sometimes the representation of the data is highly specific to the application, but there are many types of data that are common to multiple APIs — and these can reasonably be described in a portable manner. In this standard, the term *data format* describes the representation of data.

It is typical for each API to define its own enumeration of the data formats on which it can operate. This causes a problem when multiple APIs are in use: the representations are likely to be incompatible, even where the capabilities intersect. When additional format-specific capabilities are added to an API which was designed without them, the description of the data representation often becomes inconsistent and disjoint. Concepts that are unimportant to the core design of an API may be represented simplistically or inaccurately, which can be a problem as the API is enhanced or when data is shared.

Some APIs do not have a strict definition of how to interpret their data. For example, a rendering API may treat all color channels of a texture identically, leaving the interpretation of each channel to the user's choice of convention. This may be true even if color channels are given names that are associated with actual colors — in some APIs, nothing stops the user from storing the blue quantity in the red channel and the red quantity in the blue channel. Without enforcing a single data interpretation on such APIs, it is nonetheless often useful to offer a clear definition of the color interpretation convention that is in force, both for code maintenance and for communication with external APIs which do have a defined interpretation. Should the user wish to use an unconventional interpretation of the data, an appropriate descriptor can be defined that is specific to this choice, in order to simplify automated interpretation of the chosen representation and to provide concise documentation.

Where multiple APIs are in use, relying on an API-specific representation as an intermediary can cause loss of important information. For example, a camera API may associate color space information with a captured image, and a printer API may be able to operate with that color space, but if the data is passed through an intermediate compute API for processing and that API has no concept of a color space, the useful information may be discarded.

The intent of this standard is to provide a common, consistent, machine-readable way to describe those data formats which are amenable to non-proprietary representation. This standard provides a portable means of storing the most common descriptive information associated with data formats, and an extension mechanism that can be used when this common functionality must be supplemented.

While this standard is intended to support the description of many kinds of data, the most common class of bulk data used in Khronos standards represents color information. For this reason, the range of standard color representations used in Khronos standards is diverse, and a significant portion of this specification is devoted to color formats.

Later sections describe some of the common color space conversion operations and provide a description of the memory layout of a number of common texture compression formats.

2. Formats and texel access

This document describes a standard layout for a data structure that can be used to define the representation of simple, portable, bulk data. Using such a data structure has the following benefits:

- Ensuring a precise description of the portable data
- Simplifying the writing of generic functionality that acts on many types of data
- Offering portability of data between APIs

The "bulk data" may be, for example:

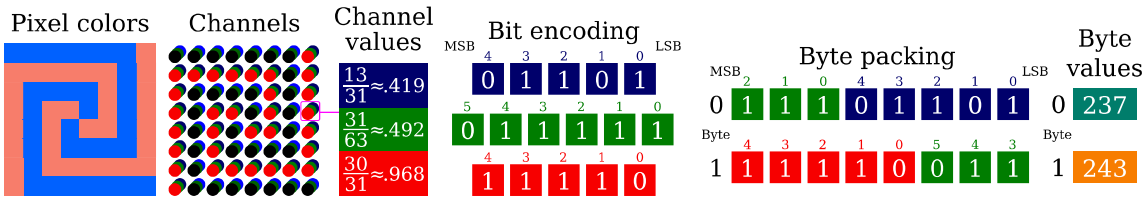
- Pixel/texel data
- Vertex data
- A buffer of simple type

The layout of proprietary data structures is beyond the remit of this specification, but the large number of ways to describe colors, vertices and other repeated data makes standardization useful. The widest variety of standard representations and the most common expected use of this API is to describe pixels or texels; as such the terms "texel" and "pixel" are used interchangeably in this specification when referring to elements of data, without intending to imply a restriction in use.

The data structure in this specification describes the elements in the bulk data in memory, not the layout of the whole. For example, it may describe the size, location and interpretation of color channels within a pixel, but is not responsible for determining the mapping between spatial coordinates and the location of pixels in memory. That is, two textures which share the same pixel layout can share the same descriptor as defined in this specification, but may have different sizes, line or plane strides, tiling or dimensionality; in common parlance, two images that describe (for example) color data in the same way but which are of different shapes or sizes are still described as having the same "format".

An example pixel representation is described in Figure 1: a single 5:6:5-bit pixel composed of a blue channel in the low 5 bits, a green channel in the next 6 bits, and red channel in the top 5 bits of a 16-bit word as laid out in memory on a little-endian machine (see Table 89).

Figure 1. A simple one-texel texel block



2.1. 1-D texel addressing

In bulk data, each element is interpreted first by addressing it in some form, then by interpreting the addressed values. Texels often represent a color (or other data) as a multi-dimensional set of values, each representing a *channel*. The bulk-data *image* or *buffer* then describes a number of these texels. Taking the simplest case of an array in the C programming language as an example, a developer might define the following structure to represent a color texel:

```
typedef struct _MyRGB {
  unsigned char red;
  unsigned char green;
  unsigned char blue;
} MyRGB;

MyRGB *myRGBarray = (MyRGB *) malloc(100 * sizeof(MyRGB));
```

To determine the location of, for example, the tenth element of `myRGBarray`, the compiler needs to know the base address of the array and `sizeof myRGB`. Extracting the `red`, `green` and `blue` components of `myRGBarray[9]` given its base address is, in a sense,

orthogonal to *finding* the base address of `myRGBarray[9]`.

Note also that `sizeof(MyRGB)` will often exceed the total size of `red`, `green` and `blue` due to padding; the difference in address between one `MyRGB` and the next can be described as the *pixel stride* in bytes.

Figure 2. (Trivial) 1D address offsets for 1-byte elements, start of buffer

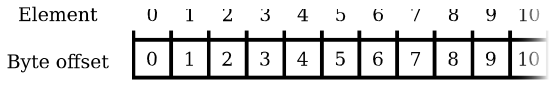


Figure 3. 1D address offsets for 2-byte elements, start of buffer

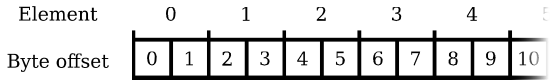
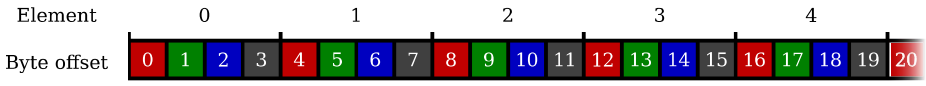


Figure 4. 1D address offsets for *R,G,B* elements (padding in gray), start of buffer



An alternative representation is a "structure of arrays", distinct from the "array of structures" `myRGBarray`:

```
typedef struct _MyRGBSoA {
  unsigned char *red;
  unsigned char *green;
  unsigned char *blue;
} MyRGBSoA;

MyRGBSoA myRGBSoA;
myRGBSoA.red = (unsigned char *) malloc(100);
myRGBSoA.green = (unsigned char *) malloc(100);
myRGBSoA.blue = (unsigned char *) malloc(100);
```

In this case, accessing a value requires the `sizeof` each channel element. The best approach depends on the operations performed: calculations on one whole `MyRGB` a time likely favor `MyRGB`, those processing multiple values from a single channel may prefer `MyRGBSoA`. A "pixel" need not fill an entire byte—nor need *pixel stride* be a whole number of bytes. For example, a C++ `std::vector<bool>` can be considered to be a 1-D bulk data structure of individual bits.

2.2. Simple 2-D texel addressing

The simplest way to represent two-dimensional data is in consecutive rows, each representing a one-dimensional array—as with a 2-D array in C. There may be padding after each row to achieve the required alignment: in some cases each row should begin at the start of a cache line, or rows may be deliberately offset to different cache lines to ensure that vertically-adjacent values can be cached concurrently. The offset from the start of one horizontal row to the next is a *line stride* or *row stride* (or just *stride* for short), and is necessarily at least the width of the row. If each row holds an whole number of pixels, row stride can be described either in bytes or pixels; it is rare not to start each row on a byte boundary. In a simple 2-D representation, the row stride and the offset from the start of the storage can be described as follows:

[Math Processing Error]

Figure 5 shows example coordinate byte offsets for a 13×4 buffer, padding row stride to a multiple of four elements.

Figure 5. 2D linear texel offsets from coordinates

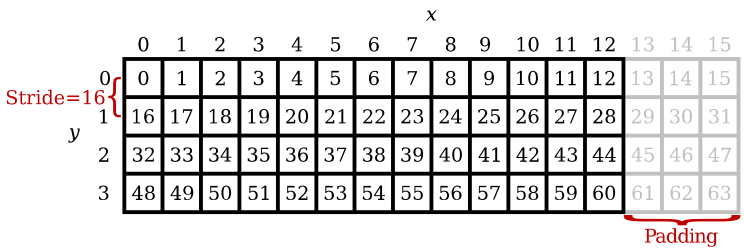
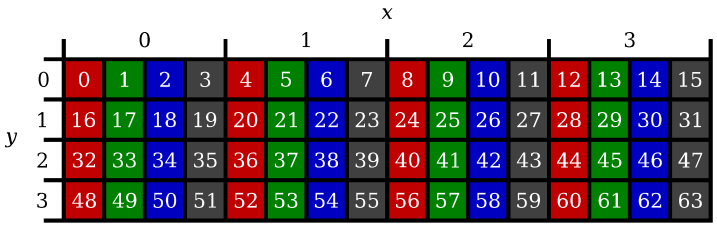


Table 1. Order of byte storage in memory for coordinates in a linear 5×3 buffer, padded (italics) to 8×3

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Coords	**0,0**	**1,0**	**2,0**	**3,0**	**4,0**	*5,0*	*6,0*	*7,0*	**0,1**	**1,1**	**2,1**	**3,1**	**4,1**	*5,1*	*6,1*	*7,1*	**0,2**	**1,2**	**2,2**	**3,2**	**4,2**	*5,2*	*6,2*	*7,2*

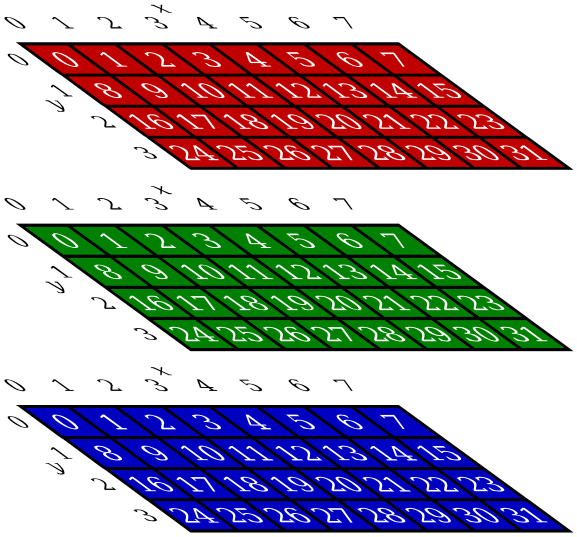
Figure 6. 2D *R,G,B* byte offsets (padding in gray) from coordinates for a 4×4 image



By convention, this "linear" layout is in *y*-major order (with the *x* axis changing most quickly). This does not necessarily imply that an API which accesses the 2D data will do so in this orientation, and there are image layouts which have *x* and *y* swapped. The direction of the axes (particularly *y*) in relation to the image orientation also varies between representations.

Each contiguous region described in this way can be considered as a *plane* of data. In the same way that 1-D data can be described in structure-of-arrays form, two-dimensional data can also be implemented as multiple *planes*.

Figure 7. 2D *R,G,B* plane offsets from coordinates for 8×4 texels



In early graphics, planes often contributed individual bits to the pixel. This allowed pixels of less than a byte, reducing bandwidth and storage; combining 1-bit planes allowed, for example, 64-color, 6-bit pixels. Storing six bits per pixel consecutively would make the number of memory bytes holding a pixel vary by alignment; a planar layout needs simpler hardware. This hardware could often process a subset of the planes — scrolling text may require blitting just one plane, and changes to the display controller start position for each plane independently provide cheap parallax effects.

In modern architectures, planes are typically byte-aligned and hold separate channels. One motivation is avoiding padding. Some common hardware can only access powers of two bytes for each pixel: for example, texels of three 1-byte channels may need padding to four bytes for alignment restrictions; separate planes of one byte per texel need no padding. Other benefits are supporting different downsampling of each channel (described below), and more efficient independent processing of each channel: many common operations (such as filtering and image compression) treat channels separately.

Note that, once all the data from any contributing channels have been combined, the interpretation of the resulting values is again orthogonal to addressing the data.

2.3. More complex 2-D texel addressing

In many applications, the simple "linear" layout of memory has disadvantages. The primary concern is that data which is vertically adjacent in two-dimensional space may be widely separated in memory address. Many computer graphics rendering operations involve accessing the frame buffer in an order not aligned with its x-axis; for example, traversing horizontally across the width of a textured triangle while writing to the frame buffer will result in a texture access pattern which depends on the orientation of the texture relative to the triangle and the triangle relative to the frame buffer. If each texel access processed a different cache line, the resulting performance would be heavily compromised. Modern GPUs process multiple texels in parallel, meaning that many nearby texels in different orientations may need to be accessed quickly. Additionally, texture filtering operations typically read a 2×2 quad of texels, inherently requiring access to texels which would be distant in memory in the linear representation.

One solution to this is to divide the image into smaller rectangular tiles of texels (of $tw \times th$) rather than horizontal rows. The texel ordering within the tiles may be treated as though each tile were an independent, smaller image, and the order of tiles in memory may be as in the linear layout:

[Math Processing Error]

This approach can be implemented efficiently in hardware, so long as the tile dimensions are powers of two, by interleaving some bits from the y-coordinate with the bits contributed by the x-coordinate. If `twb` = $\log_2(tw)$ and `thb` = $\log_2(th)$:

```
pixelOffset = (x & ((1 << twb) - 1)) | ((y & ((1 << thb) - 1)) << twb)
```

```
                | ((x & ~((1 << twb) - 1)) << thb) | ((y & ~((1 << thb)-1)) * lineStride);
```

For example, if a linear 16×12 image calculates a pixel offset relative to the start of the buffer as:

```
pixelOffset = x | (y * 16);
```

a 16×12 image comprised of 4×4 tiles may calculate the pixel offset relative to the start of the buffer as:

```
pixelOffset = (x & 3) | ((y & 3) << 2) | ((x & ~3) << 2) | ((y & ~3) * 16);
```

Figure 8. 4×4 tiled order texel offsets from coordinates (consecutive values linked in red to show the pattern)

y \ x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
1	4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
2	8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
3	12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
4	64	65	66	67	80	81	82	83	96	97	98	99	112	113	114	115
5	68	69	70	71	84	85	86	87	100	101	102	103	116	117	118	119
6	72	73	74	75	88	89	90	91	104	105	106	107	120	121	122	123
7	76	77	78	79	92	93	94	95	108	109	110	111	124	125	126	127

Textures which have dimensions that are not a multiple of the tile size require padding.

For so long as the size of a tile fits into an on-chip cache and can be filled efficiently by the memory subsystem, this approach has the benefit that only one in *[Math Processing Error]* transitions between vertically-adjacent lines requires accessing outside a tile of height *n*. The larger the tile, the greater the probability that vertically-adjacent accesses fall inside the same scan line. However, horizontally-adjacent texels that cross a tile boundary are made more distant in memory the larger the tile size, and tile sizes which do not conveniently fit the cache and bus sizes of the memory subsystem have inefficiencies; thus the tile size is an architectural trade-off.

While any non-linear representation is typically referred to as "tiling", some hardware implementations actually use a more complex layout in order to provide further locality of reference. One such scheme is Morton order:

Figure 9. Morton order texel offsets from coordinates (consecutive values linked in red)

y \ x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	4	5	16	17	20	21	64	65	68	69	80	81	84	85
1	2	3	6	7	18	19	22	23	66	67	70	71	82	83	86	87
2	8	9	12	13	24	25	28	29	72	73	76	77	88	89	92	93
3	10	11	14	15	26	27	30	31	74	75	78	79	90	91	94	95
4	32	33	36	37	48	49	52	53	96	97	100	101	112	113	116	117
5	34	35	38	39	50	51	54	55	98	99	102	103	114	115	118	119
6	40	41	44	45	56	57	60	61	104	105	108	109	120	121	124	125
7	42	43	46	47	58	59	62	63	106	107	110	111	122	123	126	127

```
uint32_t mortonOffset(uint32_t x, uint32_t y,
                      uint32_t width, uint32_t height) {

    const uint32_t minDim = (width <= height) ? width : height;
    uint32_t offset = 0, shift = 0, mask;

    // Tests xy bounds AND that width and height != 0
    assert(x < width && y < height);

    // Smaller size must be a power of 2
    assert((minDim & (minDim - 1)) == 0);

    // Larger size must be a multiple of the smaller
    assert(width % minDim == 0 && height % minDim == 0);

    for (mask = 1; mask < minDim; mask <<= 1) {
        offset |= (((y & mask) << 1) | (x & mask)) << shift;
        shift++;
    }

    // At least one of width and height will have run out of most-significant bits
```

```
    offset |= ((x | y) >> shift) << (shift * 2);
    return offset;
}
```

Note that this particular implementation assumes that the smaller of `width` and `height` is a power of two, and the larger is a multiple of the smaller. A practical hardware implementation of the calculation is likely much more efficient than this C code would imply.

Another approach, with even more correlation between locality in *x,y* space and memory offset, is Hilbert curve order:

Figure 10. Hilbert curve order texel offsets from coordinates (consecutive values linked in red)

y \ x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	3	4	5	58	59	60	63	64	67	68	69	122	123	124	127
1	1	2	7	6	57	56	61	62	65	66	71	70	121	120	125	126
2	14	13	8	9	54	55	50	49	78	77	72	73	118	119	114	113
3	15	12	11	10	53	52	51	48	79	76	75	74	117	116	115	112
4	16	17	30	31	32	33	46	47	80	81	94	95	96	97	110	111
5	19	18	29	28	35	34	45	44	83	82	93	92	99	98	109	108
6	20	23	24	27	36	39	40	43	84	87	88	91	100	103	104	107
7	21	22	25	26	37	38	41	42	85	86	89	90	101	102	105	106

```
uint32_t h(uint32_t size, uint32_t x, uint32_t y) {
    const uint32_t z = x ^ y;
    uint32_t offset = 0;

    while (size >>= 1) { // Iterate in decreasing block size order
        // Accumulate preceding blocks of size * size
        offset += size * (((size & x) << 1) + (size & z));
        y = z ^ x; // Transpose x and y
        if (!(size & y)) x = (size & x) ? ~y : y; // Conditionally swap/mirror
    }
    return offset;
}

uint32_t hilbertOffset(uint32_t x, uint32_t y, uint32_t width, uint32_t height) {
    const uint32_t minDim = (width <= height) ? width : height;

    // Tests xy bounds AND that width and height != 0
    assert(x < width && y < height);

    // Smaller size must be a power of 2
    assert((minDim & (minDim - 1)) == 0);

    // Larger size must be a multiple of the smaller
    assert(width % minDim == 0 && height % minDim == 0);

    if (width < height) return (width * width) * (y / width) + h(width, y % width, x);
    else return (height * height) * (x / height) + h(height, x % height, y);
}
```

This code assumes that the smaller of `width` and `height` is a power of two, with the larger a multiple of the smaller.

Some implementations will mix these approaches — for example, having linear tiles arranged in Morton order, or Morton order texels within tiles which are themselves in linear order. Indeed, for non-square areas, the tiling, Morton and Hilbert orderings shown here can be considered as a linear row of square areas with edges of the shorter dimension.

In all these cases, the relationship between coordinates and the memory storage location of the buffer depends on the image size and, other than needing to know the amount of memory occupied by a single texel, is orthogonal to the "format". Tiling schemes tend to be complemented by proprietary hardware that performs the coordinate-to-address mapping and depends on cache details, so many APIs expose only a linear layout to end-users, keeping the tiled representation opaque. The variety of possible mappings between coordinates and addresses mandates leaving the calculation to the application.

2.4. 3-dimensional texel addressing

These storage approaches can be extended to additional dimensions, for example treating a 3-D image as being made up of 2-D "planes", which are distinct from the concept of storing channels or bits of data in separate planes. In a linear layout, 3-dimensional textures can be accessed as:

[Math Processing Error]

Here, the *plane stride* is the offset between the start of one contiguous 2-D plane of data and the next. The *plane stride* is therefore *height* × *row stride* plus any additional padding.

Again, non-linear approaches can be used to increase the correlation between storage address and coordinates.

```
uint32_t tileOffset3D(uint32_t x, uint32_t y, uint32_t z,
                      uint32_t twb, uint32_t thb, uint32_t tdb,
                      uint32_t lineStride, uint32_t planeStride) {
  // twb = tile width bits (log2 of tile width)
  // thb = tile height bits (log2 of tile height)
  // tdb = tile depth bits (log2 of tile depth)
  return (x & ((1 << twb) - 1)) |
         ((y & ((1 << thb) - 1)) << twb) |
         ((z & ((1 << tdb) - 1)) << (twb + thb)) |
         ((x & ~((1 << twb) - 1)) << (thb + tdb)) |
         ((y & ~((1 << thb) - 1)) << tdb) * lineStride |
         ((z & ~((1 << tdb) - 1)) * planeStride);
}
```

```
uint32_t mortonOffset3D(uint32_t x, uint32_t y, uint32_t z,
                        uint32_t width, uint32_t height, uint32_t depth) {
    const uint32_t max = width | height | depth;
    uint32_t offset = 0, shift = 0, mask;
    for (mask = 1; max > mask; mask <<= 1) {
        if (width > mask)  offset |= (x & mask) << shift++;
        if (height > mask) offset |= (y & mask) << shift++;
        if (depth > mask)  offset |= (z & mask) << shift++;
        --shift;
    }
    return offset;
}
```

There are multiple 3D variations on the Hilbert curve; one such is this:

```
uint32_t hilbertOffset3D(uint32_t x, uint32_t y, uint32_t z, uint32_t size) {
    // "Harmonious" 3D Hilbert curve for cube of power-of-two edge "size":
    // http://herman.haverkort.net/recursive_tilings_and_space-filling_curves
    uint32_t offset = 0;
    while (size >>= 1) {
        uint32_t tx = (size & x) > 0, ty = (size & y) > 0, tz = (size & z) > 0;
        switch (tx + 2 * ty + 4 * tz) {
        case 0: tx =  z; ty =  y; tz =  x; break;
        case 1: tx =  x; ty =  z; tz =  y; offset += size * size * size; break;
        case 2: tx = ~y; ty = ~x; tz =  z; offset += size * size * size * 3; break;
        case 3: tx =  z; ty =  x; tz =  y; offset += size * size * size * 2; break;
        case 4: tx = ~z; ty =  y; tz = ~x; offset += size * size * size * 7; break;
        case 5: tx =  x; ty = ~z; tz = ~y; offset += size * size * size * 6; break;
        case 6: tx = ~y; ty = ~x; tz =  z; offset += size * size * size * 4; break;
        case 7: tx = ~z; ty =  x; tz = ~y; offset += size * size * size * 5; break;
        }
        x = tx; y = ty; z = tz;
    }
    return offset;
}
```

2.5. Downsampled channels

The examples provided so far have assumed that a unique value from each color channel is present at each access coordinate. However, some common representations break this assumption.

One reason for this variation comes from representing the image in the $Y'C_BC_R$ color model, described in Section 15.1; in this description, the Y' channel represents the intensity of light, with C_B and C_R channels describing how the color differs from a gray value of the same intensity in the blue and red axes. Since the human eye is more sensitive to high spatial frequencies in brightness than in the hue of a color, the C_B and C_R channels can be recorded at lower spatial resolution than the Y' channel with little loss in visual quality, saving storage space and bandwidth.

In one common representation known colloquially as *YUV* 4:2:2, each horizontal pair of Y' values has a single C_B and C_R value shared for the pair. For example, Figure 11 shows a 6-element 1-D buffer with one Y' value for each element, but as shown in Figure 12 the C_B and C_R values are shared across pairs of elements.

Figure 11. 1-D $Y'C_BC_R$ 4:2:2 buffer, texels associated with Y' values

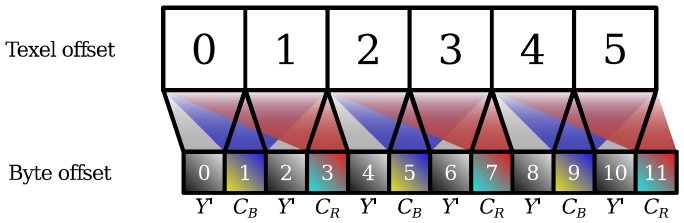
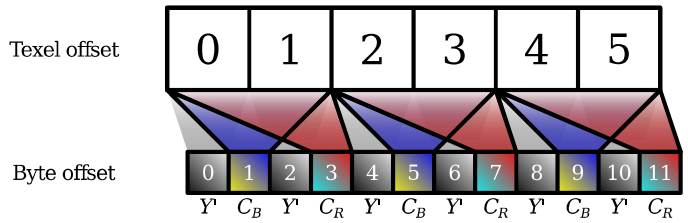


Figure 12. 1-D $Y'C_BC_R$ 4:2:2 buffer, texels associated with C_B and C_R values



In this case, we can say that the 2×1 coordinate region forms a *texel block* which contains two Y' values, one C_B value and one C_R value; our bulk-data buffer or image is composed of a repeating pattern of these texel blocks.

Note that this representation raises a question: while we have assumed so far that accessing a value at texel coordinates will provide the value contained in the texel, how should the shared C_B and C_R values relate to the coordinates of the Y' channel? Each texel coordinate represents a value in the coordinate space of the image or buffer, which can be considered as a *sample* of the value of a continuous surface at that location. The preferred means of reconstructing this surface is left to the application: since this specification only defines the values in the image and not how they are used, it is concerned with the sample values rather than the reconstruction algorithm or means of access. For example, in graphics APIs the coordinates used to access a 2-D texture may offset the sample locations of a texture by half a coordinate relative to the origin of sample space, and filtering between samples is typically used to implement antialiasing. However, to interpret the data correctly, any application still needs to know the theoretical location associated with the samples, so that information is within the remit of this specification.

Our Y' samples should fall naturally on our native coordinates. However, the C_B and C_R sample locations (which are typically at the same location as each other) could be considered as located coincident with one or other of the Y' values as shown in Figure 13, or could be defined as falling at the mid-point between them as in Figure 14. Different representations have chosen either of these alternatives — in some cases, choosing a different option for each coordinate axis. The application can choose how to treat these sample locations: in some cases it may suffice to duplicate C_B and C_R across the pair of Y' values, in others bilinear or bicubic filtering may be more appropriate.

Figure 13. 1-D $Y'C_BC_R$ 4:2:2 buffer, C_B and C_R cosited with even Y' values

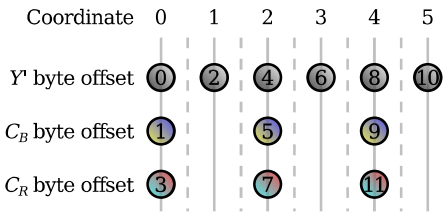
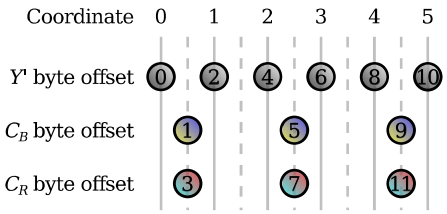


Figure 14. 1-D $Y'C_BC_R$ 4:2:2 buffer, C_B and C_R midpoint between Y' values

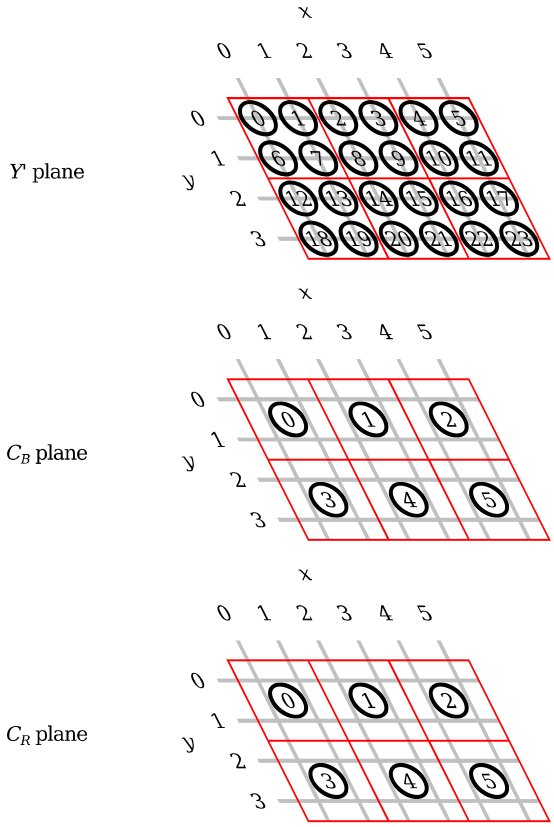


Traditional APIs have described the C_B and C_R as having 2×1 *downsampling* in this format (there are half as many samples available in the horizontal axis for these channels).

This concept can be extended to more dimensions. Commonly, a two-dimensional image stored in $Y'C_BC_R$ format may store the C_B and C_R channels downsampled by a factor of two in each dimension ("2×2 downsampling", also known for historical reasons as "4:2:0"). This approach is used in, for example, many JPEG images and MPEG video streams.

Because there are twice as many rows of Y' data as there are C_B and C_R data, it is convenient to record the channels as separate planes as shown in Figure 15 (with 2×2 texel blocks outlined in red); in addition, image compression schemes often work with the channels independently, which is amenable to planar storage.

Figure 15. 2-D $Y'C_BC_R$ 4:2:0 planes, downsampled C_B and C_R at midpoint between Y' values

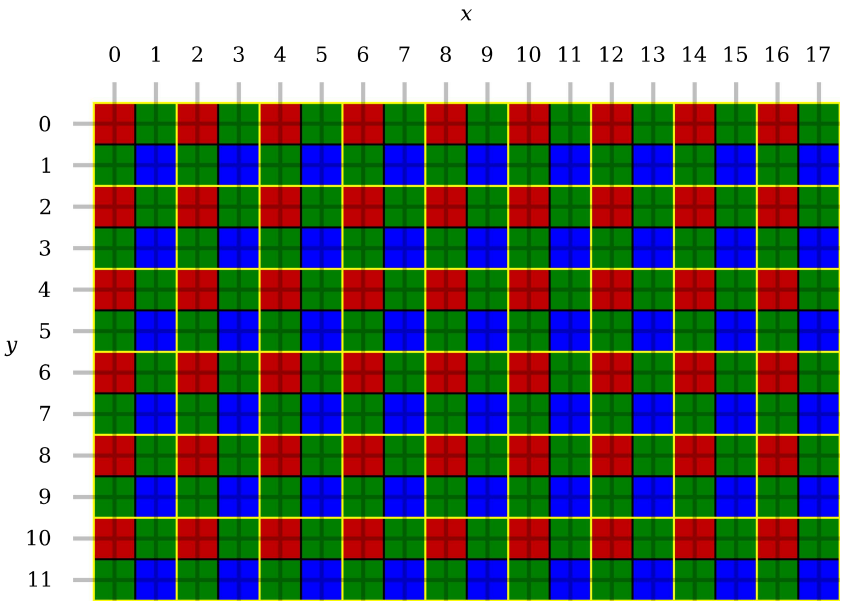


In this case, the C_B and C_R planes are half the width and half the height of the Y' plane, and also have half the line stride. Therefore if we store one byte per channel, the offsets for each plane in a linear representation can be calculated as:

[Math Processing Error]

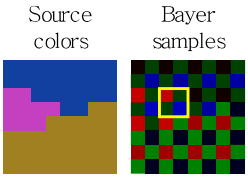
A description based on downsampling factors is sufficient in the case of common $Y'C_BC_R$ layouts, but does not extend conveniently to all representations. For example, one common representation used in camera sensors is a *Bayer pattern*, in which there is only one of the red, green and blue channels at each sample location: one red and blue value per 2×2 texel block, and two green values offset diagonally, as in Figure 16.

Figure 16. An 18×12 $^{RG}/_{GB}$ Bayer filter pattern (repeating pattern outlined in yellow)



A Bayer pattern can then be used to sample an image, as shown in Figure 17, and this sampled version can later be used to reconstruct the original image by relying on heuristic correlations between the channels. Technology for image processing continues to develop, so in many cases it is valuable to record the "raw" sensor data for later processing, and to pass the raw data unmodified to a range of algorithms; the choice of algorithm for reconstructing an image from samples is beyond the remit of this specification.

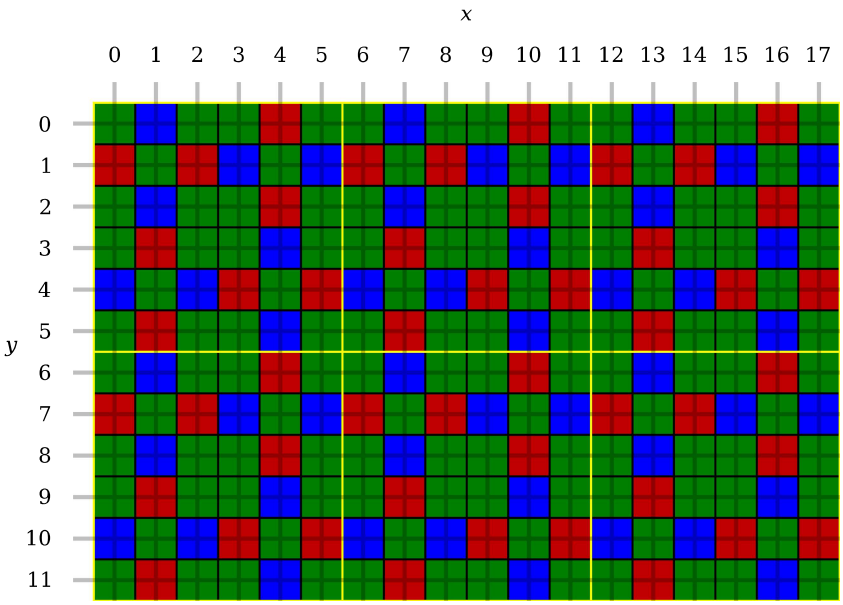
Figure 17. A Bayer-sampled image with a repeating 2×2 $^{RG}/_{GB}$ texel block (outlined in yellow)



In the Bayer representation, the red and blue channels can be considered to be downsampled by a factor of two in each dimension. The two green channels per 2×2 repeating block mean that the "downsampling factor" for the green channel is effectively *[Math Processing Error]* in each direction.

More complex layouts are not uncommon. For example, the X-Trans sample arrangement developed by Fujifilm for their digital camera range, shown in Figure 18, consists of 6×6 texel blocks, with each sample, as in Bayer, corresponding to a single channel. In X-Trans, each block contains eight red, eight blue and twenty green samples; red and blue are "downsampled" by *[Math Processing Error]* and green is "downsampled" by *[Math Processing Error]* .

Figure 18. An 18×12 X-Trans image (repeating 6×6 pattern outlined in yellow)



Allowing for possible alternative orientations of the samples (such as whether the Bayer pattern starts with a row containing red or blue samples, and whether the first sample is green or red/blue), trying to encode these sample patterns implicitly is difficult.

3. The Khronos Data Format Descriptor overview

The data structure defined in this specification is termed a *data format descriptor*. This is an extensible block of contiguous memory, with a defined layout. The size of the data format descriptor depends on its content, but is also stored in a field at the start of the descriptor, making it possible to copy the data structure without needing to interpret all possible contents.

The data format descriptor is divided into one or more *descriptor blocks*, each also consisting of contiguous data, as shown in Table 2. These descriptor blocks may, themselves, be of different sizes, depending on the data contained within. The size of a descriptor block is stored as part of its data structure, allowing an application to process a data format descriptor while skipping contained descriptor blocks that it does not need to understand. The data format descriptor mechanism is extensible by the addition of new descriptor blocks.

Table 2. Data format descriptor and descriptor blocks

Data format descriptor
Descriptor block 1
Descriptor block 2
:

The diversity of possible data makes a concise description that can support every possible format impractical. This document describes one type of descriptor block, a *basic descriptor block*, that is expected to be the first descriptor block inside the data format descriptor where present, and which is sufficient for a large number of common formats, particularly for pixels. Formats which cannot be described within this scheme can use additional descriptor blocks of other types as necessary.

Later sections of this specification describe several common color spaces, and provide a description of the in-memory representation of a number of common compressed texture formats.

3.1. *Texel blocks* in the Khronos Data Format Descriptor

Instead of an implicit representation, the descriptor specifies a *texel block*, which is described in terms of its finite extent in each of four dimensions, and which identifies the location of each sample within that region. The *texel block* is therefore a self-contained, repeating, axis-aligned pattern in the coordinate space of the image. This description conveniently corresponds to the concept of *compressed texel blocks* used in common block-based texture compression schemes, which similarly divide the image into (typically) small rectangular regions which are encoded collectively. The bounds of the texel block are chosen to be aligned to integer coordinates.

Although it is most common to consider one- and two-dimensional textures and texel blocks, it can be convenient to record additional dimensions; for example, the ASTC compressed texture format described in Section 23 can support compressed texel blocks with three dimensions. For convenience of encoding, this specification uses a four-dimensional space to define sample locations within a texel block — there is no meaning imposed on how those dimensions should be interpreted.

In many formats, all color channels refer to the same location, and the *texel block* defines a 1×1×1×1 region — that is, a single texel.

note

Tiling schemes for texel addressing can also be seen to break the image into rectangular (or higher-dimensional) sub-regions, and in many schemes these sub-regions are repeating and axis-aligned. Within reason, it is possible to define some coordinate-to-texel mapping in terms of a texel block; for example, instead of a simple tiled layout of 4×4 texels, it would be possible to describe the image in terms of a linear pattern of texel blocks, each of which contain 4×4 samples. In general, this is not a useful approach: it is very verbose to list each sample individually, it does not extend well to larger block sizes (or to infinite ranges in approaches such as Morton order), it does not handle special cases well (such as the "tail" of a mip chain), and encodings such as Hilbert order do not have a repeating mapping. In most contexts where these concepts exist, tiling is not considered to be part of a "format".

3.2. *Planes* in the Khronos Data Format Specification

The description above has shown that the data contributing to a texel may be stored in separate locations in memory; for example, R, G and B stored in separate planes may need to be combined to produce a single texel. For the purposes of this specification, a *plane* is defined to be **a contiguous region of bytes in memory that contributes to a single texel block**.

This interpretation contradicts the traditional interpretation of downsampled channels: if two rows of Y' data correspond to a single row of C_B and C_R (in a linear, non-tiled memory layout), the Y' channel contribution to the texel block is *not* "a contiguous region of bytes". Instead, each row of Y' contributing to a texel block can be treated as a separate "plane".

In linear layout, this can be represented by offsetting rows of Y' data with odd y coordinates by the row stride of the original Y' plane; each new Y' plane's stride is then double that of the original Y' plane, as in Figure 19 (c.f. Figure 15). If the planes of a 6×4 $Y'C_BC_R$ 4:2:0 texture are stored consecutively in memory with no padding, which might be described in a traditional API as Table 3, the representation used by this specification would be that shown in Table 4.

Table 3. Plane descriptors of a 6×4 $Y'C_BC_R$-format buffer in a conventional API

Plane	Byte offset	Byte stride	Downsample factor
Y'	0	6	1×1
C_B	24	3	2×2
C_R	30	3	2×2

Table 4. Plane descriptors of a 6×4 $Y'C_BC_R$-format buffer using this standard

Plane	Byte offset	Byte stride	Bytes per plane
Y' plane 1	0	12	2
Y' plane 2	6	12	2
C_B	24	3	1
C_R	30	3	1

note

There is no expectation that an API must actually use this representation in accessing the data: it is simple for an API with explicit support for the special case of integer chroma downsampling to detect interleaved planes and to deduce that they should be treated as a single plane of double vertical resolution. Many APIs will not support the full flexibility of formats supported by this specification, and will map to a more restrictive internal representation.

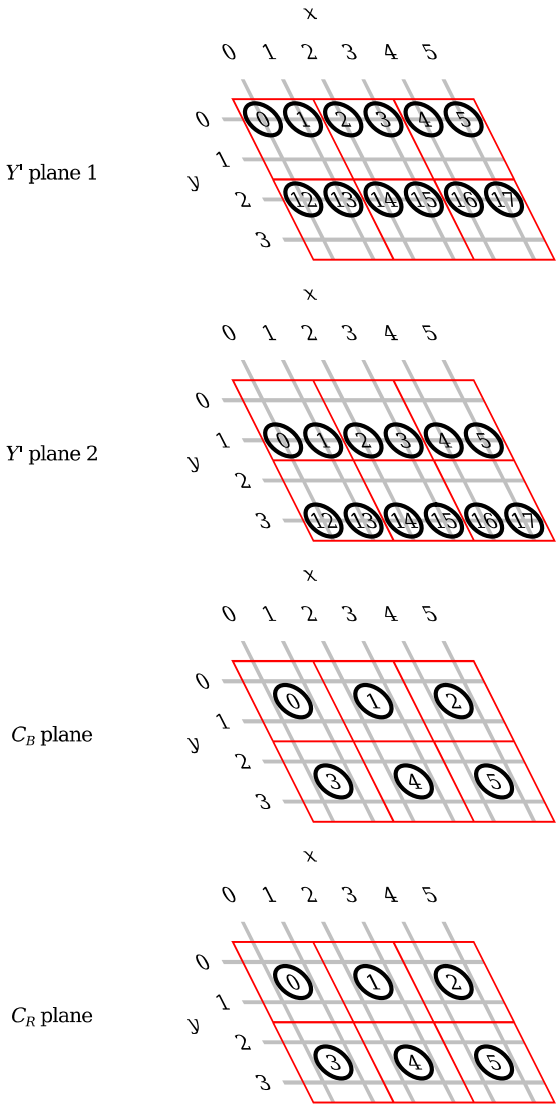
The Khronos Basic Descriptor Block indicates a number of bytes contributing to the texel block from each of up to eight planes — if more than eight planes are required, an extension is needed. Eight planes are enough to encode, for example:

- 8-bit data stored as individual bit planes
- Stereo planar R, G, B, A data
- 4× vertically-downsampled $Y'C_BC_R$ data, as might be produced by rotating a (relatively common) 4× horizontally-downsampled $Y'C_BC_R$ (4:1:1) video frame
- A 2×2×2×2 4-D texel block in a linear layout
- Interlaced $Y'C_BC_R$ 4:2:0 data with each field stored separately

If a plane contributes less than a byte to a texel (or a non-integer number of bytes), the texel block size should be expanded to cover more texels until a whole number of bytes are covered — q.v. Table 92.

Interlaced $Y'C_BC_R$ data may associate chroma channels with Y' samples only from the same field, not the frame as a whole. This distinction is not made explicit, in part because a number of interlacing schemes exist. One suggested convention for interlaced data is that the field of a sample be encoded in the fourth sample coordinate (the first field as ***samplePosition3*** = 0, the second field as ***samplePosition3*** = 1, etc.) This interpretation is not mandated to allow other reasons for encoding four-dimensional texels, although it is consistent with the fourth dimension representing "time".

Figure 19. 2-D $Y'C_BC_R$ 4:2:0 planes, with separate even and odd Y' planes



3.3. Bit pattern interpretation and *samples*

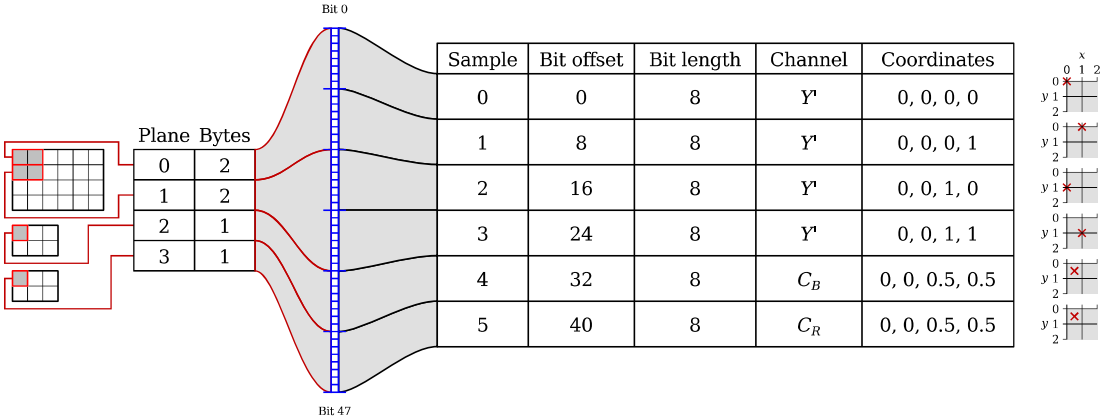
For the purposes of description, the bytes contributed by each plane are considered to be concatenated into a single contiguous *logical bit stream*. This "concatenation" of bits is purely conceptual for the purposes of determining the interpretation of the bits that contribute to the texel block, and is not expected to be the way that actual decoders would process the data.

That is, if planes zero and one contribute two bytes of data each and planes two and three contribute one byte of data each, this bit stream would consist of the two bytes from plane zero followed by the two bytes from plane one, then the byte from plane two, then the byte from plane three.

The data format descriptor then describes a series of *samples* that are contained within the texel block. Each sample represents a series of contiguous bits from the bit stream, interpreted in little-endian order, and is associated with a single channel and four-dimensional coordinate offset within the texel block; in formats for which only a single texel is being described, this coordinate offset will always be 0,0,0,0.

The descriptor block for a $Y'C_BC_R$ 4:2:0 layout is shown in Table 96. Figure 20 shows this representation graphically: the three disjoint regions for each channels and the texel block covering them are shown on the left, represented in this specification as four planes of contiguous bytes. These are concatenated into a 48-bit logical bit stream (shown in blue, in top-to-bottom order); these *samples* then describe the contributions from these logical bits, with geometric location of the sample at the right.

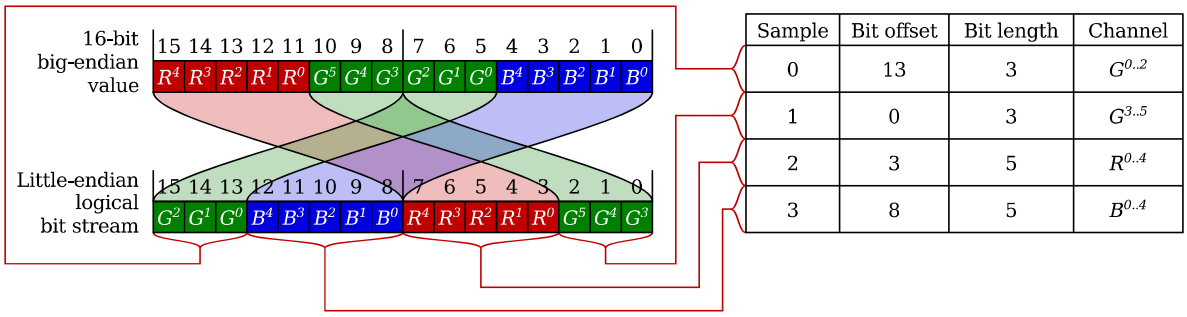
Figure 20. $Y'C_BC_R$ 4:2:0 described in six samples



Consecutive samples with the same channel, location and flags may describe additional bits that contribute to the same final value, and appear in increasing order. For example, a 16-bit big-endian word in memory can be described by one sample describing bits 0..7, coming from the higher byte address, followed by one sample describing bits 8..15, coming from the lower address. These samples comprise a single *virtual sample*, of 16 bits.

Figure 21 shows how multiple contributions to single value can be used to represent a channel which, due to a big-endian native word type (high bits stored in lower byte addresses), is not contiguous in a little-endian representation (low bits stored in lower byte addresses). Here, channels are comprised of consecutive bits from a 16-bit big-endian word; the bits of each channel cease to be contiguous when the memory bytes are interpreted in little-endian order for the logical bit stream. The bit contributions to each channel from each bit location are shown in superscript, as later in this specification; the channel contributions start at bit 0 with the first sample contributing to the value, and are deduced implicitly from the number of bits each sample contributes. This example assumes that we are describing a single texel, with the same 0,0,0,0 coordinates (not shown) for each sample. The descriptor block for this format is shown in Table 97.

Figure 21. *RGB* 565 big-endian encoding



In Figure 21, bit 0 of the logical bit stream corresponds to bit 3 of the virtual sample that describes the green channel; therefore the first channel to be encoded is green. Bits 0 to 2 of the green channel virtual sample correspond to bits 13..15 of the logical bit stream, so these are described by the first sample. The second sample continues describing bits 3..5 of the green channel virtual sample, in bits 0..2 of the logical bit stream. The next bit from the logical bit stream that has not yet been described is bit 3, which corresponds to bit 0 of the red channel. The third sample is therefore used to describe bits 3..7 of the logical bit stream as a 5-bit red channel. The last sample encodes the remaining 5-bit blue channel that forms bits 8..12 of the logical bit stream. Note that in the case where some bits of the data are ignored, they do not need to be covered by samples; bits may also appear repeatedly if they contribute to multiple samples.

The precision to which sample coordinates are specified depends on the size of the texel block: coordinates in a 256×256 texel block can be specified only to integer-coordinate granularity, but offsets within a texel block that is only a single coordinate wide are specified to the precision of *[Math Processing Error]* of a coordinate; this approach allows large texel blocks, half-texel offsets for representations such as $Y'C_BC_R$ 4:2:0, and precise fractional offsets for recording multisampled pattern locations.

The sequence of bits in the (virtual) sample defines a numerical range which may be interpreted as a fixed-point or floating-point value and signed or unsigned. Many common representations specify this range. For example, 8-bit *RGB* data may be interpreted in "unsigned normalized" form as meaning "0.0" (zero color contribution) when the channel bits are 0 and "1.0" (maximum color contribution) when the channel is 255. In $Y'C_BC_R$ "narrow-range" encoding, there is head-room and foot-room to the encoding: "black" is encoded in the Y' channel as the value 16, and "white" is encoded as 235, as shown in Section 16.1 encoding". To allow the bit pattern of simply-encoded numerical encodings to be mapped to the desired values, each sample has an *upper* and *lower* value associated with it, usually representing 1.0 and either 0.0 or -1.0 depending on whether the sample is signed.

Note that it is typically part of the "format" to indicate the numbers which are being encoded; how the application chooses to process these numbers is not part of the "format". For example, some APIs may have two separate "formats", in which the 8-bit pattern 0x01 may be interpreted as either the float value 1.0 or the integer value 1; for the purposes of this specification, these formats are identical — "1" means "1".

A sample has an associated color channel in the color model of the descriptor block — for example, if the descriptor block indicates an *RGB* color model, the sample's channel field could specify the *R* channel. The descriptor block enumerates a range of common color models, color primary values and transfer functions, which apply to the samples.

3.4. Canonical representation

There is some redundancy in the data format descriptor when it comes to the ordering of planes and samples. In the interests of simplifying comparison between formats, it is helpful to define a canonical ordering.

- Planes should be described first in order of the first channel with which they are associated, then in increasing raster order of coordinates, then in increasing bit order (in a little-endian interpretation). For example:
  - In the $Y'C_BC_R$ 4:2:0 format described above, the Y' planes should come before the C_B and C_R planes in that order, because of the channel order.
  - The Y' plane corresponding to even y addresses should come before the Y' plane corresponding to odd y coordinates, because row 0 is even.
  - Planes should be ordered such that sample values that are split across multiple planes should be ordered in increasing order — e.g. in an 8-bit format with one bit per plane, planes 0 through 7 should encode bits 0 through 7 in that order (thereby minimizing the number of samples required to describe the value).

The order of samples should be defined by the following rules:

- Samples sharing a channel and location appear consecutively in increasing order of bits contributed to the virtual sample.
  - For example, a big-endian 16-bit red channel at location 0,0 may be composed of two samples: one referencing the eight consecutive virtual bit stream bits from 0..7 (***bitOffset*** = 0, ***bitLength*** = 8) and the other referencing the eight consecutive virtual bit stream bits from 8..15 (***bitOffset*** = 8, ***bitLength*** = 8). Since the least-significant 8 bits of the virtual red value come from the sample that references bits 8..15 of the virtual bit stream, this sample should appear first, immediately followed by the sample that references bits 0..8, which define the most-significant 8 bits of the virtual value. Table 102 shows a similar virtual sample, split across three samples of contiguous bits.
- A minimum number of samples describes each sequence of contiguous virtual bit stream bits in a virtual sample value.
  - If an additional sample is required to represent the sample's ***sampleLower*** and ***sampleUpper*** values because more than 32 bits are encoded and the existing extension rules for ***sampleLower*** and ***sampleUpper*** do not result in the desired behavior, the earlier sample(s) should be limited to 32 bits, ensuring that the subsequent sample holds at least one bit (***bitLength*** is greater than or equal to 0).
  - Otherwise, if the contiguous sequence of bits from the virtual bit stream is longer than 256 bits, samples should be concatenated such that only the last sample describes fewer than 256 bits.
- Samples that qualify an existing virtual sample immediately follow it.
  - Specifically, in an explicitly-described floating-point format, any sample describing a sign bit immediately follows the unsigned mantissa, and any exponent follows thereafter, as seen in Table 103.
- Virtual samples are described in increasing order of the virtual bit stream bits to which they apply.
  - This means that if bit 0 of the virtual bit stream is part of a virtual sample, that virtual sample should be described first; this does not require that the first sample directly describes bit 0, as in the green channel of Table 97.
- If the same bit in the virtual bit stream in increasing bit order is the first to be associated with more than one virtual sample, virtual samples are listed in channel number order.
  - For example, an alpha channel may be encoded in bits 0..7 of the virtual bits stream, followed by red, green and blue channels sharing bits 8..15. The sample describing alpha should be listed first, since it uniquely describes bit 0 of the virtual bits stream. Bit 8 of the virtual bit stream is the first bit of the red, green and blue, and since red, green and blue are channel numbers 0, 1 and 2, the corresponding samples are described in this order. See, for example, Table 99 and Table 101.
- Virtual samples that share bits and channel number but not location would be extremely unusual, but would appear in increasing raster order of the location (that is, sorted first by coordinate three, then two, then one, then zero).

Finally:

- Fields which are irrelevant (for example, the alpha behavior for a format with no alpha channel) should be set to zero.
- Samples describing a channel of a single bit should have a linear transfer function — either by selecting the transfer function **`KHR_DF_TRANSFER_LINEAR`** for the descriptor as a whole if all samples are linear, or by setting the sample's **`KHR_DF_SAMPLE_DATATYPE_LINEAR`** bit if other samples have multiple bits and a non-linear transfer function is needed.

3.5. Related concepts outside the "format"

The data format descriptor describes only concepts which are conventionally part of the "format". Additional information is required to access the image data:

- A formula for mapping accessed coordinates to byte data for each channel.
- This may be expected to require, for each channel, a start address and a stride in each dimension.
- This transformation depends on the image size, and may be parameterized by the texel block dimensions and the number of bytes contributed by each plane.

- (Optionally) for each dimension, a maximum (and arguably minimum) range.
- Note that padding is independent of the "format".

For example, if texels are laid out in memory in linear order:

```
int numPlanes;
char *planeBaseAddresses[8];
unsigned int strides[8][4];

// Note: The strides here are assumed to be in units of the
// corresponding dimension of the texel block
char *planeAddress(uint32_t descriptor, int plane, float coords[4]) {
  return planeBaseAddresses[plane]
    // Block stride
    + ((int) (coords[0] / descriptorSize(descriptor, 0))) * strides[plane][0]
    // Row stride
    + ((int) (coords[1] / descriptorSize(descriptor, 1))) * strides[plane][1]
    // Plane stride
    + ((int) (coords[2] / descriptorSize(descriptor, 2))) * strides[plane][2]
    // Volume(?) stride
    + ((int) (coords[3] / descriptorSize(descriptor, 3))) * strides[plane][3];
}

decodeTexelGivenCoords(uint32_t *descriptor, float coords[4]) {
  char *addresses[8];
  for (int i = 0; i < numPlanes; ++i) {
    addresses[i] = planeAddress(i, coords);
  }

  // processTexel reads bytesPlane[] bytes from each address and
  // decodes the concatenated data according to the descriptor
  processTexel(descriptor, addresses, coords);
}
```

The `processTexel` function would typically operate on the coordinates having taken the remainder of dividing them by the texel block size. For example, if the texel block size is 2×2, the block and row stride provide the offset in bytes between adjacent blocks in the first two dimensions. `processTexel` would then work with the data corresponding to the 2×2 region with `coords[0]` and `coords[1]` in the range [0..2). For formats that describe a single texel, `coords` can be considered to be an integer. Note that for more complex formats such as Bayer layouts, reconstructing an *R,G,B* value at a location may require information from more than one texel block, depending on the algorithm used, in a manner analogous to sampling using bilinear filtering.

The stride values may be stored explicitly, or derived implicitly from the ***bytesPlane*** values and image size information, with application-specific knowledge about alignment constraints.

3.6. Translation to API-specific representations

Despite being designed to balance size against flexibility, the data format container described here is too unwieldy to be expected to be used directly in most APIs, which will generally support only a subset of possible formats. The expectation is that APIs and users will define data descriptors in memory, but have API-specific names for the formats that the API supports. If these names are enumeration values, a mapping can be provided by having an array of pointers to the data descriptors, indexed by the enumeration. It may commonly be necessary to provide API-specific supplementary information in the same array structure, particularly where the API natively associates concepts with the data which is not uniquely associated with the content.

In this approach, it is likely that an API would predefine a number of common data formats which are natively supported. If there is a desire to support dynamic creation of data formats, this array could be made extensible with a manager assigning unique handles.

Even where an API supports only a fixed set of formats, it is flexible to provide a comparison with user-provided format descriptors in order to establish whether a format is compatible (and differs only in a manner irrelevant to the API).

Some APIs have the concept of a *native data type* for a format, if each channel is stored separately. Since this specification describes a number of bytes per plane and separately contiguous bit sequences, there is no such explicit concept. However, if a sample's ***bitOffset*** and ***bitLength*** are byte-aligned and no further samples contribute to the same value, the ***bitLength*** trivially defines a little-endian native data type size. Big-endian data types can be identified by observing that in a big-endian format, a sequence of bits in the top bits of byte *n* may continue in the low bits of byte *n* - 1. Finally, "packed" formats consist of consecutive bit sequences per channel in either little- or big-endian order; little-endian sequences are a single stand-alone sample, and a big-endian sequence consists of a number of samples adjacent to byte boundaries in decreasing byte order (see Figure 21); the packed field size can typically be deduced from the ***bytesPlane0*** value. There is no way to distinguish a logically "packed", byte-aligned samples from distinct but consecutively-stored channels that have the same in-memory representation.

Glossary

Data format: The interpretation of individual elements in bulk data. Examples include the channel ordering and bit positions in pixel data or the configuration of samples in a Bayer image. The format describes the elements, not the bulk data itself: an image's size, stride, tiling, dimensionality, border control modes, and image reconstruction filter are not part of the format and are the responsibility of the application.

Data format descriptor: A contiguous block of memory containing information about how data is represented, in accordance with this specification. A data format descriptor is a container, within which can be found one or more descriptor blocks. This specification does not define where or how the the data format descriptor should be stored, only its content. For example, the descriptor may be directly prepended to the bulk data, perhaps as part of a file format header, or the descriptor may be stored in a CPU memory while the bulk data that it describes resides within GPU memory; this choice is application-specific.

(Data format) descriptor block: A contiguous block of memory with a defined layout, held within a data format descriptor. Each descriptor block has a common header that allows applications to identify and skip descriptor blocks that it does not understand, while continuing to process any other descriptor blocks that may be held in the data format descriptor.

Basic (data format) descriptor block: The initial form of descriptor block as described in this standard. Where present, it must be the first descriptor block held in the data format descriptor. This descriptor block can describe a large number of common formats and may be the only type of descriptor block that many portable applications will need to support.

Texel block: The units described by the Basic Data Format Descriptor: a repeating element within bulk data. In simple texture formats, a texel block may describe a single pixel. In formats where the bytes of each plane do not correspond uniquely to single pixels, as for example with subsampled channels, the texel block may cover several pixels. In a block-based compressed texture, the texel block typically describes the compression block unit. The basic descriptor block supports texel blocks of up to four dimensions.

Plane: In the Basic Data Format Descriptor, a plane describes a contiguous sequence of bytes that contribute to the texel block. The basic format descriptor block defines a texel block as being made of a number of concatenated bits which may come from different regions of memory, where each region is considered a separate *plane*. For common formats, it is sufficient to require that the contribution from each plane is an integer number of bytes. This specification places no requirements on the ordering of planes in memory — the plane locations are described outside the format. This allows support for multiplanar formats which have proprietary padding requirements that are hard to accommodate in a more terse representation.

Sample: In this standard, texel blocks are considered to be composed of contiguous bit patterns with a single channel or component type and a single spatial location. A typical *ARGB* pixel has four samples, one for each channel, held at the same coordinate. A texel block from a Bayer sensor might have a different location for different channels, and may have multiple samples representing the same channel at multiple locations. A $Y'C_BC_R$ buffer with downsampled chroma may have more luma samples than chroma, each at different locations.

4. Khronos Data Format Descriptor

The data format descriptor consists of a contiguous area of memory, as shown in Table 5, divided into one or more *descriptor blocks*, which are tagged by the type of descriptor that they contain. The size of the data format descriptor varies according to its content.

Table 5. Data Format Descriptor layout

`uint32_t`	***totalSize***
Descriptor block	*First descriptor*
Descriptor block	*Second descriptor (optional) etc.*

The ***totalSize*** field, measured in bytes, allows the full format descriptor to be copied without need for details of the descriptor to be interpreted. ***totalSize*** includes its own `uint32_t`, not just the following descriptor blocks. For example, we will see below that a four-sample Khronos Basic Data Format Descriptor Block occupies 88 bytes; if there are no other descriptor blocks in the data format descriptor, the ***totalSize*** field would then indicate 88 + 4 bytes (for the ***totalSize*** field itself) for a final value of 92.

For consistency of decode, each descriptor block should be aligned to a multiple of four bytes relative to the start of the descriptor; ***totalSize*** will therefore be a multiple of four.

This restriction was not present in versions of the Khronos Data Format Specification prior to version 1.3.

The layout of the data structures described here are comprised solely of 32-bit words, and for canonical communication between devices are assumed to be stored with a little-endian representation. For efficiency, applications may choose to convert the descriptor to the native endianness of the underlying hardware where all software using the data structure is prepared for this conversion. Extensions which are composed of quantities other than 32-bit words (for example if a data structure belonging to another standard is incorporated directly) may define the expected impact of endianness changes on the layout. Since the environment is expected to know its own endianness, there is no explicit means of automatically determining the endianness of a descriptor, although it can be observed that it is highly unlikely that a valid descriptor would be large enough for its size to need to be represented in more than 16 bits — meaning that the endianness of most descriptors can be deduced by which half of the `uint32_t` ***totalSize*** field is non-zero.

To avoid expanding the size of the data structure, there is no "magic identifier" for a data format descriptor: applications are expected to know the type of the data structure being accessed, and to provide their own means of identifying a data format descriptor if one is embedded in a multi-purpose byte stream.

4.1. Descriptor block

Each *descriptor block* has the same prefix, shown in Table 6.

Table 6. Descriptor Block layout

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
descriptorType															***vendorId***																
descriptorBlockSize																***versionNumber***															
Format-specific data																															

The ***vendorId*** is a 17-bit value uniquely assigned to organizations. If the organization has a 16-bit identifier assigned by the PCI SIG, this should occupy bits 0..15, with bit 16 set to 0. Other organizations should apply to Khronos of a unique identifier, which will be

assigned consecutively starting with 65536. The identifier 0x1FFFF is reserved for internal use which is guaranteed not to clash with third-party implementations; this identifier should not be shipped in libraries to avoid conflicts with development code.

The ***descriptorType*** is a unique 15-bit identifier defined by the vendor to distinguish between potential data representations.

> Prior to version 1.3 of the Khronos Data Format Specification, the ***vendorId*** field was 16-bit, and purely assigned through the auspices of this specification; the ***descriptorType*** was consequently also 16-bit. Since no vendor has requested an identifier and Khronos does not have a descriptor block with type 1, this change should not cause any ambiguity. This change is intended to allow consistency with the vendor IDs used by the Vulkan specification.

The ***versionNumber*** is vendor-defined, and is intended to allow for backwards-compatible updates to existing descriptor blocks.

The *DescriptorBlockSize* indicates the size in bytes of this Descriptor Block, remembering that there may be multiple Descriptor Blocks within one container, as shown in Table 7. The ***descriptorBlockSize*** therefore gives the offset between the start of the current Descriptor Block and the start of the next — so the size includes the ***vendorId***, ***descriptorType***, ***versionNumber*** and ***descriptorBlockSize*** fields, which collectively contribute 8 bytes.

Having an explicit ***descriptorBlockSize*** allows implementations to skip a descriptor block whose format is unknown, allowing known data to be interpreted and unknown information to be ignored. Some descriptor block types may not be of a uniform size, and may vary according to the content within.

This specification initially describes only one type of stand-alone descriptor block, plus two extension blocks which modify the description in the first. Future revisions may define additional descriptor block types for additional applications — for example, to describe data with a large number of channels or pixels described in an arbitrary color space. Vendors can also implement proprietary descriptor blocks to hold vendor-specific information within the standard descriptor.

Unless otherwise specified, descriptor blocks can appear in any order, to make it easier to add and remove additional informative descriptor blocks to a preexisting data format descriptor as part of processing. Descriptor blocks that provide additional capabilities beyond a basic scheme (such as the descriptor block for supporting additional planes beyond the Khronos Basic Descriptor Block) should not be present unless their additional capabilities are needed; that is, redundancy should be resolved so as to minimize the number of descriptor blocks in the data format descriptor.

Table 7. Data format descriptor header and descriptor block headers for two descriptor blocks

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize																															
First descriptor block																															
descriptorType															***vendorId***																
descriptorBlockSize																***versionNumber***															
Descriptor body																															
Second descriptor block																															
descriptorType															***vendorId***																
descriptorBlockSize																***versionNumber***															
Descriptor body																															

5. Khronos Basic Data Format Descriptor Block

A *basic descriptor block* (Table 8) is designed to encode common metadata associated with bulk data — especially image or texture data. While this descriptor holds more information about the data interpretation than is needed by many applications, a comprehensive encoding reduces the risk of metadata needed by different APIs being lost in translation.

The format is described in terms of a repeating axis-aligned *texel block* composed of *samples*. Each sample contains a single channel of information with a single spatial offset within the texel block, and consists of an amount of contiguous data. This *descriptor block* consists of information about the interpretation of the texel block as a whole, supplemented by a description of a number of samples taken from one or more *planes* of contiguous memory. For example, a 24-bit red/green/blue format may be described as a 1×1 pixel region, in one plane of three samples, one describing each channel. A $Y'C_BC_R$ 4:2:0 format may consist of a repeating 2×2 region: four Y' samples and one sample each of C_B and C_R.

Table 8. Basic Data Format Descriptor layout

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
descriptorType = 0															***vendorId*** = 0																
descriptorBlockSize = 24 + 16 × *#samples*																***versionNumber*** = 2															
flags								***transferFunction***								***colorPrimaries***								***colorModel***							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
bytesPlane3								***bytesPlane2***								***bytesPlane1***								***bytesPlane0***							
bytesPlane7								***bytesPlane6***								***bytesPlane5***								***bytesPlane4***							
Sample information for the first sample																															
Sample information for the second sample (optional), etc.																															

The Basic Data Format Descriptor Block should be the first descriptor block in any data format descriptor of which it is a component.

The fields of the Basic Data Format Descriptor Block are described in the following sections.

To simplify code using the Basic Data Format Descriptor Block, the header `khr_df.h` provides enums of the following form for accessing descriptor block fields:

Table 9. Field location information for field *xxx*

Word offset into basic descriptor block	**KHR_DF_WORD_xxx**	...
Word offset into descriptor	**KHR_DF_WORD_xxx** + 1	...
Start bit within word	**KHR_DF_SHIFT_xxx**	...
Bit mask of value	**KHR_DF_MASK_xxx**	...

If the basic descriptor block is treated as a **uint32_t** array bdb[], field ***xxx*** can be accessed as follows:

xxx = **KHR_DF_MASK_xxx** & (bdb[**KHR_DF_WORD_xxx**] >> **KHR_DF_SHIFT_xxx**);

The macro **KHR_DFDVAL**(BDB, X) is provided to perform this calculation. For example, **KHR_DFDVAL**(bdb, **MODEL**) returns the value:

KHR_DF_MASK_MODEL & (bdb[**KHR_DF_WORD_MODEL**] >> **KHR_DF_SHIFT_MODEL**)

5.1. *vendorId*

The ***vendorId*** for the Basic Data Format Descriptor Block is 0, defined as **KHR_DF_VENDORID_KHRONOS** in the enum **khr_df_vendorid_e**.

Table 10. Field location information for *vendorId*

Word offset into basic descriptor block	**KHR_DF_WORD_VENDORID**	0
Word offset into descriptor	**KHR_DF_WORD_VENDORID** + 1	1
Start bit within word	**KHR_DF_SHIFT_VENDORID**	0
Bit mask of value	**KHR_DF_MASK_VENDORID**	0x1FFFFU

khr_df_vendorid_e ***vendorId*** = **KHR_DF_MASK_VENDORID** & (bdb[**KHR_DF_WORD_VENDORID**] >> **KHR_DF_SHIFT_VENDORID**);

5.2. *descriptorType*

The ***descriptorType*** for the Basic Data Format Descriptor Block is 0, a value reserved in the enum of Khronos-specific descriptor types, **khr_df_khr_descriptortype_e**, as **KHR_DF_KHR_DESCRIPTORTYPE_BASICFORMAT**.

Table 11. Field location information for *descriptorType*

Word offset into basic descriptor block	**KHR_DF_WORD_DESCRIPTORTYPE**	0
Word offset into descriptor	**KHR_DF_WORD_DESCRIPTORTYPE** + 1	1
Start bit within word	**KHR_DF_SHIFT_DESCRIPTORTYPE**	17
Bit mask of value	**KHR_DF_MASK_DESCRIPTORTYPE**	0x7FFFU

khr_df_descriptortype_e ***descriptorType*** = **KHR_DF_MASK_DESCRIPTORTYPE** & (bdb[**KHR_DF_WORD_DESCRIPTORTYPE**] >> **KHR_DF_SHIFT_DESCRIPTORTYPE**);

5.3. *versionNumber*

The ***versionNumber*** relating to the Basic Data Format Descriptor Block as described in this specification is 2.

The ***versionNumber*** is incremented to indicate an incompatible change in the descriptor. The addition of enumerant values, for example to represent more compressed texel formats, does not constitute an "incompatible change", and implementations should be resilient against enumerants that have been added in later minor updates.

Table 12. Field location information for *versionNumber*

Word offset into basic descriptor block	**KHR_DF_WORD_VERSIONNUMBER**	1
Word offset into descriptor	**KHR_DF_WORD_VERSIONNUMBER** + 1	2
Start bit within word	**KHR_DF_SHIFT_VERSIONNUMBER**	0
Bit mask of value	**KHR_DF_MASK_VERSIONNUMBER**	0xFFFFU

uint32_t ***versionNumber*** = **KHR_DF_MASK_VERSIONNUMBER** & (bdb[**KHR_DF_WORD_VERSIONNUMBER**] >> **KHR_DF_SHIFT_VERSIONNUMBER**);

5.4. *descriptorBlockSize*

The memory size of the Basic Data Format Descriptor Block depends on the number of samples contained within it. The memory requirements for this format are 24 bytes of shared data plus 16 bytes per sample. The ***descriptorBlockSize*** is measured in bytes.

Table 13. Field location information for *descriptorBlockSize*

Word offset into basic descriptor block	KHR_DF_WORD_DESCRIPTORBLOCKSIZE	1
Word offset into descriptor	KHR_DF_WORD_DESCRIPTORBLOCKSIZE + 1	2
Start bit within word	KHR_DF_SHIFT_DESCRIPTORBLOCKSIZE	16
Bit mask of value	KHR_DF_MASK_DESCRIPTORBLOCKSIZE	0xFFFFU

uint32_t ***descriptorBlockSize*** = KHR_DF_MASK_DESCRIPTORBLOCKSIZE & (bdb[KHR_DF_WORD_DESCRIPTORBLOCKSIZE] >> KHR_DF_SHIFT_DESCRIPTORBLOCKSIZE);

5.5. *colorModel*

The ***colorModel*** determines the set of color (or other data) channels which may be encoded within the data, though there is no requirement that all of the possible channels from the ***colorModel*** be present. Most data fits into a small number of common color models, but compressed texture formats each have their own color model enumeration. Note that the data need not actually represent a color — this is just the most common type of content using this descriptor. Some standards use *color container* for this concept.

The available color models are described in the khr_df_model_e enumeration, and are represented as an unsigned 8-bit value.

Table 14. Field location information for *colorModel*

Word offset into basic descriptor block	KHR_DF_WORD_MODEL	2
Word offset into descriptor	KHR_DF_WORD_MODEL + 1	3
Start bit within word	KHR_DF_SHIFT_MODEL	0
Bit mask of value	KHR_DF_MASK_MODEL	0xFF

khr_df_model_e ***colorModel*** = KHR_DF_MASK_MODEL & (bdb[KHR_DF_WORD_MODEL] >> KHR_DF_SHIFT_MODEL);

Note that the numbering of the component channels is chosen such that those channel types which are common across multiple color models have the same enumeration value. That is, alpha is always encoded as channel ID 15, depth is always encoded as channel ID 14, and stencil is always encoded as channel ID 13. Luma/Luminance is always in channel ID 0. This numbering convention is intended to simplify code which can process a range of color models. Note that there is no guarantee that models which do not support these channels will not use this channel ID. Particularly, *RGB* formats do not have luma in channel 0, and a 16-channel undefined format is not obligated to represent alpha in any way in channel number 15.

The value of each enumerant is shown in parentheses following the enumerant name.

5.5.1. KHR_DF_MODEL_UNSPECIFIED (= 0)

When the data format is unknown or does not fall into a predefined category, utilities which perform automatic conversion based on an interpretation of the data cannot operate on it. This format should be used when there is no expectation of portable interpretation of the data using only the basic descriptor block.

For portability reasons, it is recommended that pixel-like formats with up to sixteen channels, but which cannot have those channels described in the basic block, be represented with a basic descriptor block with the appropriate number of samples from UNSPECIFIED channels, and then for the channel description to be stored in an extension block. This allows software which understands only the basic descriptor to be able to perform operations that depend only on channel location, not channel interpretation (such as image cropping). For example, a camera may store a raw format taken with a modified Bayer sensor, with *RGBW* (red, green, blue and white) sensor sites, or *RGBE* (red, green, blue and "emerald"). Rather than trying to encode the exact color coordinates of each sample in the basic descriptor, these formats could be represented by a four-channel UNSPECIFIED model, with an extension block describing the interpretation of each channel.

5.5.2. KHR_DF_MODEL_RGBSDA (= 1)

This color model represents additive colors of three channels, nominally red, green and blue, supplemented by channels for alpha, depth and stencil, as shown in Table 15. Note that in many formats, depth and stencil are stored in a completely independent buffer, but there are formats for which integrating depth and stencil with color data makes sense.

Table 15. Basic Data Format *RGBSDA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_RGBSDA_RED	Red
1	KHR_DF_CHANNEL_RGBSDA_GREEN	Green
2	KHR_DF_CHANNEL_RGBSDA_BLUE	Blue
13	KHR_DF_CHANNEL_RGBSDA_STENCIL	Stencil
14	KHR_DF_CHANNEL_RGBSDA_DEPTH	Depth
15	KHR_DF_CHANNEL_RGBSDA_ALPHA	Alpha (opacity)

Portable representation of additive colors with more than three primaries requires an extension to describe the full color space of the channels present. There is no practical way to do this portably without taking significantly more space.

5.5.3. KHR_DF_MODEL_YUVSDA (= 2)

This color model represents color differences with three channels, nominally luma (Y') and two color-difference chroma channels, U (C_B) and V (C_R), supplemented by channels for alpha, depth and stencil, as shown in Table 16. These formats are distinguished by C_B and C_R

being a delta between the Y' channel and the blue and red channels respectively, rather than requiring a full color matrix. The conversion between $Y'C_BC_R$ and *RGB* color spaces is defined in this case by the choice of value in the ***colorPrimaries*** field as described in Section 15.1.

Most single-channel luma/luminance monochrome data formats should select **KHR_DF_MODEL_YUVSDA** and use only the *Y* channel, unless there is a reason to do otherwise.

Table 16. Basic Data Format *YUVSDA* channels

Channel number	Name	Description
0	**KHR_DF_CHANNEL_YUVSDA_Y**	Y/Y' (luma/luminance)
1	**KHR_DF_CHANNEL_YUVSDA_CB**	C_B (alias for U)
1	**KHR_DF_CHANNEL_YUVSDA_U**	U (alias for C_B)
2	**KHR_DF_CHANNEL_YUVSDA_CR**	C_R (alias for V)
2	**KHR_DF_CHANNEL_YUVSDA_V**	V (alias for C_R)
13	**KHR_DF_CHANNEL_YUVSDA_STENCIL**	Stencil
14	**KHR_DF_CHANNEL_YUVSDA_DEPTH**	Depth
15	**KHR_DF_CHANNEL_YUVSDA_ALPHA**	Alpha (opacity)

Terminology for this color model is often abused. This model is based on the idea of creating a representation of monochrome light intensity as a weighted average of color channels, then calculating color differences by subtracting two of the color channels from this monochrome value. Proper names vary for each variant of the ensuing numbers, but *YUV* is colloquially used for all of them. In the television standards from which this terminology is derived, $Y'C_BC_R$ is more formally used to describe the representation of these color differences. See Section 15.1 for more detail.

5.5.4. KHR_DF_MODEL_YIQSDA (= 3)

This color model represents color differences with three channels, nominally luma (Y) and two color-difference chroma channels, I and Q, supplemented by channels for alpha, depth and stencil, as shown in Table 17. This format is distinguished by I and Q each requiring all three additive channels to evaluate. I and Q are derived from C_B and C_R by a 33-degree rotation.

Table 17. Basic Data Format *YIQSDA* channels

Channel number	Name	Description
0	**KHR_DF_CHANNEL_YIQSDA_Y**	Y (luma)
1	**KHR_DF_CHANNEL_YIQSDA_I**	I (in-phase)
2	**KHR_DF_CHANNEL_YIQSDA_Q**	Q (quadrature)
13	**KHR_DF_CHANNEL_YIQSDA_STENCIL**	Stencil
14	**KHR_DF_CHANNEL_YIQSDA_DEPTH**	Depth
15	**KHR_DF_CHANNEL_YIQSDA_ALPHA**	Alpha (opacity)

5.5.5. KHR_DF_MODEL_LABSDA (= 4)

This color model represents the ICC perceptually-uniform $L^*a^*b^*$ color space, combined with the option of an alpha channel, as shown in Table 18.

Table 18. Basic Data Format *LABSDA* channels

Channel number	Name	Description
0	**KHR_DF_CHANNEL_LABSDA_L**	L^* (luma)
1	**KHR_DF_CHANNEL_LABSDA_A**	a^*
2	**KHR_DF_CHANNEL_LABSDA_B**	b^*
13	**KHR_DF_CHANNEL_LABSDA_STENCIL**	Stencil
14	**KHR_DF_CHANNEL_LABSDA_DEPTH**	Depth
15	**KHR_DF_CHANNEL_LABSDA_ALPHA**	Alpha (opacity)

5.5.6. KHR_DF_MODEL_CMYKA (= 5)

This color model represents secondary (subtractive) colors and the combined key (black) channel, along with alpha, as shown in Table 19.

Table 19. Basic Data Format *CMYKA* channels

Channel number	Name	Description
0	**KHR_DF_CHANNEL_CMYKA_CYAN**	Cyan
1	**KHR_DF_CHANNEL_CMYKA_MAGENTA**	Magenta
2	**KHR_DF_CHANNEL_CMYKA_YELLOW**	Yellow
3	**KHR_DF_CHANNEL_CMYKA_KEY**	Key/Black
15	**KHR_DF_CHANNEL_CMYKA_ALPHA**	Alpha (opacity)

5.5.7. KHR_DF_MODEL_XYZW (= 6)

This "color model" represents channel data used for coordinate values, as shown in Table 20 — for example, as a representation of the surface normal in a bump map. Additional channels for higher-dimensional coordinates can be used by extending the channel number within the 4-bit limit of the ***channelType*** field.

Table 20. Basic Data Format *XYZW* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_XYZW_X	*X*
1	KHR_DF_CHANNEL_XYZW_Y	*Y*
2	KHR_DF_CHANNEL_XYZW_Z	*Z*
3	KHR_DF_CHANNEL_XYZW_W	*W*

5.5.8. KHR_DF_MODEL_HSVA_ANG (= 7)

This color model represents color differences with three channels, *value* (luminance or luma), *saturation* (distance from monochrome) and *hue* (dominant wavelength), supplemented by an alpha channel, as shown in Table 21. In this model, the hue relates to the angular offset on a color wheel.

Table 21. Basic Data Format angular *HSVA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_HSVA_ANG_VALUE	*V* (value)
1	KHR_DF_CHANNEL_HSVA_ANG_SATURATION	*S* (saturation)
2	KHR_DF_CHANNEL_HSVA_ANG_HUE	*H* (hue)
15	KHR_DF_CHANNEL_HSVA_ANG_ALPHA	Alpha (opacity)

5.5.9. KHR_DF_MODEL_HSLA_ANG (= 8)

This color model represents color differences with three channels, *lightness* (maximum intensity), *saturation* (distance from monochrome) and *hue* (dominant wavelength), supplemented by an alpha channel, as shown in Table 22. In this model, the hue relates to the angular offset on a color wheel.

Table 22. Basic Data Format angular *HSLA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_HSLA_ANG_LIGHTNESS	*L* (lightness)
1	KHR_DF_CHANNEL_HSLA_ANG_SATURATION	*S* (saturation)
2	KHR_DF_CHANNEL_HSLA_ANG_HUE	*H* (hue)
15	KHR_DF_CHANNEL_HSLA_ANG_ALPHA	Alpha (opacity)

5.5.10. KHR_DF_MODEL_HSVA_HEX (= 9)

This color model represents color differences with three channels, *value* (luminance or luma), *saturation* (distance from monochrome) and *hue* (dominant wavelength), supplemented by an alpha channel, as shown in Table 23. In this model, the hue is generated by interpolation between extremes on a color hexagon.

Table 23. Basic Data Format hexagonal *HSVA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_HSVA_HEX_VALUE	*V* (value)
1	KHR_DF_CHANNEL_HSVA_HEX_SATURATION	*S* (saturation)
2	KHR_DF_CHANNEL_HSVA_HEX_HUE	*H* (hue)
15	KHR_DF_CHANNEL_HSVA_HEX_ALPHA	Alpha (opacity)

5.5.11. KHR_DF_MODEL_HSLA_HEX (= 10)

This color model represents color differences with three channels, *lightness* (maximum intensity), *saturation* (distance from monochrome) and hue (dominant wavelength), supplemented by an alpha channel, as shown in Table 24. In this model, the hue is generated by interpolation between extremes on a color hexagon.

Table 24. Basic Data Format hexagonal *HSLA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_HSLA_HEX_LIGHTNESS	*L* (lightness)
1	KHR_DF_CHANNEL_HSLA_HEX_SATURATION	*S* (saturation)
2	KHR_DF_CHANNEL_HSLA_HEX_HUE	*H* (hue)
15	KHR_DF_CHANNEL_HSLA_HEX_ALPHA	Alpha (opacity)

5.5.12. KHR_DF_MODEL_YCGCOA (= 11)

This color model represents low-cost approximate color differences with three channels, nominally luma (*Y*) and two color-difference chroma channels, *Cg* (green/purple color difference) and *Co* (orange/cyan color difference), supplemented by a channel for alpha, as shown in Table 25.

Table 25. Basic Data Format *YCoCgA* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_YCGCOA_Y	*Y*
1	KHR_DF_CHANNEL_YCGCOA_CG	*Cg*
2	KHR_DF_CHANNEL_YCGCOA_CO	*Co*
15	KHR_DF_CHANNEL_YCGCOA_ALPHA	Alpha (opacity)

5.5.13. KHR_DF_MODEL_YCCBCCRC (= 12)

This color model represents the "Constant luminance" *[Math Processing Error]* color model defined as an optional representation in ITU-T BT.2020 and described in Section 15.2.

Table 26. Basic Data Format $Y'_C C'_{BC} C'_{RC}$ channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_YCCBCCRC_YC	*[Math Processing Error]* (luminance)
1	KHR_DF_CHANNEL_YCCBCCRC_CBC	*[Math Processing Error]*
2	KHR_DF_CHANNEL_YCCBCCRC_CRC	*[Math Processing Error]*
13	KHR_DF_CHANNEL_YCCBCCRC_STENCIL	Stencil
14	KHR_DF_CHANNEL_YCCBCCRC_DEPTH	Depth
15	KHR_DF_CHANNEL_YCCBCCRC_ALPHA	Alpha (opacity)

5.5.14. KHR_DF_MODEL_ICTCP (= 13)

This color model represents the "Constant intensity IC_TC_P color model" defined as an optional representation in ITU-T BT.2100 and described in Section 15.3.

Table 27. Basic Data Format IC_TC_P channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_ICTCP_I	*I* (intensity)
1	KHR_DF_CHANNEL_ICTCP_CT	C_T
2	KHR_DF_CHANNEL_ICTCP_CP	C_P
13	KHR_DF_CHANNEL_ICTCP_STENCIL	Stencil
14	KHR_DF_CHANNEL_ICTCP_DEPTH	Depth
15	KHR_DF_CHANNEL_ICTCP_ALPHA	Alpha (opacity)

5.5.15. KHR_DF_MODEL_CIEXYZ (= 14)

This color model represents channel data used to describe color coordinates in the CIE 1931 XYZ coordinate space, as shown in Table 28.

Table 28. Basic Data Format CIE *XYZ* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_CIEXYZ_X	*X*
1	KHR_DF_CHANNEL_CIEXYZ_Y	*Y*
2	KHR_DF_CHANNEL_CIEXYZ_Z	*Z*

5.5.16. KHR_DF_MODEL_CIEXYY (= 15)

This color model represents channel data used to describe chromaticity coordinates in the CIE 1931 xyY coordinate space, as shown in Table 29.

Table 29. Basic Data Format CIE *xyY* channels

Channel number	Name	Description
0	KHR_DF_CHANNEL_CIEXYZ_X	*x*
1	KHR_DF_CHANNEL_CIEXYZ_YCHROMA	*y*
2	KHR_DF_CHANNEL_CIEXYZ_YLUMA	*Y*

5.6. *colorModel* for compressed formats

A number of compressed formats are supported as part of **khr_df_model_e**. In general, these formats will have the texel block dimensions of the compression block size. Most contain a single sample of channel type 0 at offset 0,0 — where further samples are required, they should also be sited at 0,0. By convention, models which have multiple channels that are disjoint in memory have these channel locations described independently as separate samples; this can simplify some decoders.

The ASTC family of formats have a number of possible channels, and are distinguished by samples which reference some set of these channels. The ***texelBlockDimension*** fields determine the compression ratio for ASTC and PVRTC.

Compressed formats necessarily do not have an equivalent integer representation in which to describe the ***sampleLower*** and ***sampleUpper*** ranges — in particular, some have different ranges on a block-by-block format. Floating-point compressed formats have lower and upper limits specified in floating point format, since this representation indicates the output of compressed decoding. Integer compressed formats with a lower and upper of 0 and **UINT32_MAX** (for unsigned formats) or **INT32_MIN** and **INT32_MAX** (for signed formats) are assumed to map the full representable range to 0..1 or -1..1 respectively.

If a format has a non-linear transfer function, any samples with channel ID 15 (that is, the format has separate alpha encoding, for example **KHR_DF_BC2_ALPHA**) should set the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit for that sample.

Example descriptors for compressed formats are provided after each model in this section.

5.6.1. KHR_DF_MODEL_DXT1A/KHR_DF_MODEL_BC1A (= 128)

This model represents the DXT1 or BC1 format, described in Section 18. Each compressed texel block consists of 4×4 texels in 8 bytes. A single sample with channel ID 0 indicates that the "special value" should be interpreted as black, as described in Section 18.1 — a descriptor block representing this is shown in Table 31. A single sample with channel ID 1 indicates that the "special value" should represent transparency, as described in Section 18.2 — a descriptor block representing this is shown in Table 32.

Enumerant names for these channel ids are listed in Table 30.

Table 30. BC1A channel names

Enumerant	Value
KHR_DF_CHANNEL_DXT1A_COLOR	0
KHR_DF_CHANNEL_BC1A_COLOR	
KHR_DF_CHANNEL_DXT1A_ALPHAPRESENT	1
KHR_DF_CHANNEL_DXT1A_ALPHA	
KHR_DF_CHANNEL_BC1A_ALPHAPRESENT	
KHR_DF_CHANNEL_BC1A_ALPHA	

Table 31. Example DXT1A descriptor with no punch-through alpha

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: **ALPHA_STRAIGHT**								***transferFunction:*** **LINEAR**								***colorPrimaries:*** **BT709**								***colorModel:*** **DXT1A**							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	**COLOR**				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: **UINT32_MAX**																															

Table 32. Example DXT1A descriptor with punch-through alpha

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: **ALPHA_STRAIGHT**								***transferFunction:*** **LINEAR**								***colorPrimaries:*** **BT709**								***colorModel:*** **DXT1A**							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	**ALPHA**				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							

sampleLower: 0
sampleUpper: `UINT32_MAX`

5.6.2. `KHR_DF_MODEL_DXT2/3/KHR_DF_MODEL_BC2` (= 129)

This model represents the DXT2/3 format, also known as BC2, and described in Section 18.3. Each compressed texel block consists of 4×4 texels in 16 bytes. The alpha premultiplication state (the distinction between DXT2 and DXT3) is recorded separately in the descriptor in the ***flags*** field. This model has two channels, recorded as separate samples: Sample 0 with channel ID 15 contains the alpha information. Sample 1 with channel ID 0 contains the color information.

Enumerant names for these channel ids are listed in Table 33.

Table 33. BC2 channel names

Enumerant	Value
`KHR_DF_CHANNEL_DXT2_COLOR`	0
`KHR_DF_CHANNEL_DXT3_COLOR`	
`KHR_DF_CHANNEL_BC2_COLOR`	
`KHR_DF_CHANNEL_DXT2_ALPHA`	15
`KHR_DF_CHANNEL_DXT3_ALPHA`	
`KHR_DF_CHANNEL_BC2_ALPHA`	

The alpha channel is 64 bits and at offset 0; the color channel is 64 bits and at offset 64. No attempt is made to describe the 16 alpha samples for this position independently, since understanding the other channels for any pixel requires the whole texel block.

Table 34. Example DXT2 descriptor (premultiplied alpha)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `PREMULTIPLIED`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `DXT2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Alpha sample information																							
0	0	0	0	`ALPHA`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Color sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Table 35. Example DXT3 descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `DXT3`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Alpha sample information																							
0	0	0	0	`ALPHA`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Color sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

5.6.3. KHR_DF_MODEL_DXT4/5/KHR_DF_MODEL_BC3 (= 130)

This model represents the DXT4/5 format, also known as BC3, and described in Section 18.4. Each compressed texel block consists of 4×4 texels in 16 bytes. The alpha premultiplication state (the distinction between DXT4 and DXT5) is recorded separately in the descriptor in the ***flags*** field. This model has two channels, recorded as separate samples: Sample 0 with channel ID 15 contains the alpha information. Sample 1 with channel ID 0 contains the color information.

Enumerant names for these channel ids are listed in Table 36.

Table 36. BC3 channel names

Enumerant	Value
KHR_DF_CHANNEL_DXT4_COLOR	0
KHR_DF_CHANNEL_DXT5_COLOR	
KHR_DF_CHANNEL_BC3_COLOR	
KHR_DF_CHANNEL_DXT4_ALPHA	15
KHR_DF_CHANNEL_DXT5_ALPHA	
KHR_DF_CHANNEL_BC3_ALPHA	

The alpha channel is 64 bits and at offset 0; the color channel is 64 bits and at offset 64. No attempt is made to describe the 16 alpha samples for this position independently, since understanding the other channels for any pixel requires the whole texel block.

Table 37. Example DXT4 descriptor (premultiplied alpha)

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: PREMULTIPLIED								***transferFunction:*** LINEAR								***colorPrimaries:*** BT709								***colorModel:*** DXT4							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Alpha sample information																							
0	0	0	0	ALPHA				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: UINT32_MAX																															
F	***S***	***E***	***L***	***channelType***				Color sample information																							
0	0	0	0	COLOR				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: UINT32_MAX																															

Table 38. Example DXT5 descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: ALPHA_STRAIGHT								***transferFunction:*** LINEAR								***colorPrimaries:*** BT709								***colorModel:*** DXT5							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Alpha sample information																							
0	0	0	0	ALPHA				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: UINT32_MAX																															
F	***S***	***E***	***L***	***channelType***				Color sample information																							
0	0	0	0	COLOR				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: UINT32_MAX																															

5.6.4. KHR_DF_MODEL_BC4 (= 131)

This model represents the Direct3D BC4 format for single-channel interpolated 8-bit data, as described in Section 19.1.

Each compressed texel block consists of 4×4 texels in 8 bytes. The model has a single channel of id 0 with offset 0 and length 64 bits.

The enumerant name for this channel id is listed in Table 39.

Table 39. BC4 channel name

Enumerant	Value
KHR_DF_CHANNEL_BC4_DATA	0

Table 40. Example BC4 unsigned descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															*vendorId:* 0																
descriptorBlockSize: 24 + (16 × 1) = 40																*versionNumber:* 2															
flags: ALPHA_STRAIGHT								*transferFunction:* LINEAR								*colorPrimaries:* BT709								*colorModel:* BC4							
texelBlockDimension3								*texelBlockDimension2*								*texelBlockDimension1*								*texelBlockDimension0*							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								*bytesPlane2:* 0								*bytesPlane1:* 0								*bytesPlane0:* 8							
bytesPlane7: 0								*bytesPlane6:* 0								*bytesPlane5:* 0								*bytesPlane4:* 0							
F	*S*	*E*	*L*	*channelType*				Sample information																							
0	0	0	0	DATA				*bitLength:* 63 (= "64")								*bitOffset:* 0															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: UINT32_MAX																															

Table 41. Example BC4 signed descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															*vendorId:* 0																
descriptorBlockSize: 24 + (16 × 1) = 40																*versionNumber:* 2															
flags: ALPHA_STRAIGHT								*transferFunction:* LINEAR								*colorPrimaries:* BT709								*colorModel:* BC4							
texelBlockDimension3								*texelBlockDimension2*								*texelBlockDimension1*								*texelBlockDimension0*							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								*bytesPlane2:* 0								*bytesPlane1:* 0								*bytesPlane0:* 8							
bytesPlane7: 0								*bytesPlane6:* 0								*bytesPlane5:* 0								*bytesPlane4:* 0							
F	*S*	*E*	*L*	*channelType*				Sample information																							
0	1	0	0	DATA				*bitLength:* 63 (= "64")								*bitOffset:* 0															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: INT32_MIN																															
sampleUpper: INT32_MAX																															

5.6.5. KHR_DF_MODEL_BC5 (= 132)

This model represents the Direct3D BC5 format for dual-channel interpolated 8-bit data, as described in Section 19.3.

Each compressed texel block consists of 4×4 texels in 16 bytes. The model has two channels, 0 (red) and 1 (green), which should have their bit depths and offsets independently described: the red channel has offset 0 and length 64 bits and the green channel has offset 64 and length 64 bits.

Enumerant names for these channel ids are listed in Table 42.

Table 42. BC5 channel names

Enumerant	Value
KHR_DF_CHANNEL_BC5_RED	0
KHR_DF_CHANNEL_BC5_R	
KHR_DF_CHANNEL_BC5_GREEN	1
KHR_DF_CHANNEL_BC5_G	

Table 43. Example BC5 unsigned descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															

descriptorType: 0					***vendorId:*** 0	
descriptorBlockSize: 24 + (16 × 2) = 56					***versionNumber:*** 2	
flags: `ALPHA_STRAIGHT`				***transferFunction:*** `LINEAR`	***colorPrimaries:*** `BT709`	***colorModel:*** `BC5`
texelBlockDimension3				***texelBlockDimension2***	***texelBlockDimension1***	***texelBlockDimension0***
0				0	3 (= "4")	3 (= "4")
bytesPlane3: 0				***bytesPlane2:*** 0	***bytesPlane1:*** 0	***bytesPlane0:*** 16
bytesPlane7: 0				***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Red sample information	
0	0	0	0	`RED`	***bitLength:*** 63 (= "64")	***bitOffset:*** 0
samplePosition3				***samplePosition2***	***samplePosition1***	***samplePosition0***
0				0	0	0
sampleLower: 0						
sampleUpper: `UINT32_MAX`						
F	***S***	***E***	***L***	***channelType***	Green sample information	
0	0	0	0	`GREEN`	***bitLength:*** 63 (= "64")	***bitOffset:*** 64
samplePosition3				***samplePosition2***	***samplePosition1***	***samplePosition0***
0				0	0	0
sampleLower: 0						
sampleUpper: `UINT32_MAX`						

Table 44. Example BC5 signed descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `BC5`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Red sample information																							
0	1	0	0	`RED`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: `INT32_MIN`																															
sampleUpper: `INT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Green sample information																							
0	1	0	0	`GREEN`				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: `INT32_MIN`																															
sampleUpper: `INT32_MAX`																															

A legacy variant of this format known as "ATI2n" or "3Dc" swaps the location of the two channels, and can be encoded as follows:

Table 45. Example ATI2n unsigned descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `BC5`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Red sample information																							
0	0	0	0	`GREEN`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Green sample information																							
0	0	0	0	`RED`				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

5.6.6. `KHR_DF_MODEL_BC6H` (= 133)

This model represents the Direct3D BC6H format for *RGB* floating-point data, as described in Section 20.2.

Each compressed texel block consists of 4×4 texels in 16 bytes. The model has a single channel 0, representing all three channels, and occupying 128 bits.

The enumerant names for this channel id are listed in Table 46.

Table 46. BC6H channel names

Enumerant	Value
`KHR_DF_CHANNEL_BC6H_COLOR`	0
`KHR_DF_CHANNEL_BC6H_DATA`	

Table 47. Example BC6H signed descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `BC6H`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
1	1	0	0	`COLOR`				***bitLength:*** 127 (= "128")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0xBF800000U — -1.0f																															
sampleUpper: 0x7F800000U — 1.0f																															

Table 48. Example BC6H unsigned descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `BC6H`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
1	0	0	0	`COLOR`				***bitLength:*** 127 (= "128")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0xBF800000U — -1.0f																															
sampleUpper: 0x7F800000U — 1.0f																															

5.6.7. `KHR_DF_MODEL_BC7` (= 134)

This model represents the Direct3D BC7 format for *RGBA* data, as described in Section 20.1.

Each compressed texel block consists of 4×4 texels in 16 bytes. The model has a single channel 0, representing all four channels, and occupying 128 bits.

The enumerant names for this channel id are listed in Table 49.

Table 49. BC7 channel names

Enumerant	Value
`KHR_DF_CHANNEL_BC7_COLOR`	0
`KHR_DF_CHANNEL_BC7_DATA`	

Table 50. Example BC7 descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `BC7`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							

0					0	3 (= "4")	3 (= "4")
bytesPlane3: 0					***bytesPlane2:*** 0	***bytesPlane1:*** 0	***bytesPlane0:*** 16
bytesPlane7: 0					***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Sample information		
0	0	0	0	`COLOR`	***bitLength:*** 127 (= "128")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: `UINT32_MAX`							

5.6.8. `KHR_DF_MODEL_ETC1` (= 160)

This model represents the original Ericsson Texture Compression format, described in Section 21, with a guarantee that the format does not rely on the ETC2 extensions described in Section 22.

Each compressed texel block consists of 4×4 texels in 8 bytes. The model has a single channel 0, representing all three channels, and occupying 64 bits.

The enumerant names for this channel id are listed in Table 51.

Table 51. ETC1 channel names

Enumerant	Value
`KHR_DF_CHANNEL_ETC1_COLOR`	0
`KHR_DF_CHANNEL_ETC1_DATA`	

Table 52. Example ETC1 descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC1`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

5.6.9. `KHR_DF_MODEL_ETC2` (= 161)

This model represents the updated Ericsson Texture Compression format, ETC2, and also the related R11 EAC and RG11 EAC formats. Each compressed texel block consists of 4×4 texels.

The enumerant names for these channel ids are listed in Table 53.

Table 53. ETC2 channel names

Enumerant	Value
`KHR_DF_CHANNEL_ETC2_RED`	0
`KHR_DF_CHANNEL_ETC2_R`	
`KHR_DF_CHANNEL_ETC2_GREEN`	1
`KHR_DF_CHANNEL_ETC2_G`	
`KHR_DF_CHANNEL_ETC2_COLOR`	2
`KHR_DF_CHANNEL_ETC2_ALPHA`	15
`KHR_DF_CHANNEL_ETC2_A`	

Channel ID 0 represents red, and is used for the R11 EAC format, as described in Section 22.5; the texel block size in this format is 8 bytes, represented as a single 64-bit sample.

Table 54. Example R11 unsigned descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							

texelBlockDimension3					***texelBlockDimension2***	***texelBlockDimension1***	***texelBlockDimension0***
0					0	3 (= "4")	3 (= "4")
bytesPlane3: 0					***bytesPlane2:*** 0	***bytesPlane1:*** 0	***bytesPlane0:*** 8
bytesPlane7: 0					***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Sample information		
0	0	0	0	`RED`	***bitLength:*** 63 (= "64")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: `UINT32_MAX`							

Table 55. Example R11 signed descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	1	0	0	`RED`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: `INT32_MIN`																															
sampleUpper: `INT32_MAX`																															

Channel ID 1 represents green; the presence of samples for both red and green, in that order, indicates the RG11 EAC format as described in Section 22.6, which consists of a total of 16 bytes of data.

Table 56. Example RG11 unsigned descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Red sample information																							
0	0	0	0	`RED`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Green sample information																							
0	0	0	0	`GREEN`				***bitLength:*** 63 (= "64")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Table 57. Example RG11 signed descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 2) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Red sample information																							
0	1	0	0	`RED`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							

sampleLower: `INT32_MIN`							
sampleUpper: `INT32_MAX`							
F	***S***	***E***	***L***	***channelType***	Green sample information		
0	1	0	0	`GREEN`	***bitLength:*** 63 (= "64")	***bitOffset:*** 64	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: `INT32_MIN`							
sampleUpper: `INT32_MAX`							

Channel ID 2 represents *RGB* combined content, for the ETC2 format as described in Section 22.1. A single sample of ID 2 indicates RGB2 with no alpha, occupying 8 bytes.

Table 58. Example ETC2 descriptor (with no alpha)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Channel ID 15 indicates the presence of alpha. If the texel block size is 8 bytes and the *RGB* and alpha channels are co-sited, "punch through" alpha is supported as described in Section 22.9.

Table 59. Example ETC2 descriptor with punchthrough alpha

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Color sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															
F	***S***	***E***	***L***	***channelType***				Alpha sample information																							
0	0	0	0	`ALPHA`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Finally, if the texel block size is 16 bytes and the alpha channel appears in the first 8 bytes, followed by 8 bytes for the *RGB* channel, 8-bit separate alpha is supported, as described in Section 22.3.

Table 60. Example ETC2 descriptor with separate alpha

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 60																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 56																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							

bytesPlane7: 0					***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Alpha sample information		
0	0	0	0	`ALPHA`	***bitLength:*** 63 (= "64")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: `UINT32_MAX`							
F	***S***	***E***	***L***	***channelType***	Color sample information		
0	0	0	0	`COLOR`	***bitLength:*** 63 (= "64")	***bitOffset:*** 64	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: `UINT32_MAX`							

5.6.10. `KHR_DF_MODEL_ASTC` (= 162)

This model represents Adaptive Scalable Texture Compression as a single channel in a texel block of 16 bytes. ASTC HDR (high dynamic range) and LDR (low dynamic range) modes are distinguished by the ***channelId*** containing the flag `KHR_DF_SAMPLE_DATATYPE_FLOAT`: an ASTC texture that is guaranteed by the user to contain only LDR-encoded blocks should have the ***channelId*** `KHR_DF_SAMPLE_DATATYPE_FLOAT` bit clear, and an ASTC texture that may include HDR-encoded blocks should have the ***channelId*** `KHR_DF_SAMPLE_DATATYPE_FLOAT` bit set to 1. ASTC supports a number of compression ratios defined by different texel block sizes; these are selected by changing the texel block size fields in the data format.

ASTC encoding is described in Section 23.

The single sample, of ID 0, has a size of 128 bits.

The enumerant name for this channel id is listed in Table 61.

Table 61. ASTC channel name

Enumerant	Value
`KHR_DF_CHANNEL_ASTC_DATA`	0

Table 62. Example 4×4 ASTC LDR descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ASTC`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`DATA`				***bitLength:*** 127 (= "128")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Table 63. Example 8×5 ASTC HDR descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ASTC`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								4 (= "5")								7 (= "8")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
1	1	0	0	`DATA`				***bitLength:*** 127 (= "128")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0xBF800000U — -1.0f																															
sampleUpper: 0x7F800000U — 1.0f																															

5.6.11. `KHR_DF_MODEL_ETC1S` (= 163)

This model represents a subset of the original Ericsson Texture Compression format, described in Section 21.1, which is restricted in order to facilitate image compression.

Each compressed texel block consists of 4×4 texels in 8 bytes. The model has a single channel 0, representing all three channels, and occupying 64 bits.

The enumerant names for this channel id are listed in Table 64.

Table 64. ETC1S channel names

Enumerant	Value
`KHR_DF_CHANNEL_ETC1S_COLOR`	0
`KHR_DF_CHANNEL_ETC1S_DATA`	

Table 65. Example ETC1S descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `ETC1S`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

5.6.12. `KHR_DF_MODEL_PVRTC` (= 164)

This model represents the first generation of PowerVR Texture Compression as a single channel in a texel block of 8 bytes. 4-bit-per-pixel mode represents a 4×4 texel block; 2-bit-per-pixel mode represents an 8×4 texel block, and these can be distinguished by changing the texel block size fields in the data format. The single sample has a size of 64 bits.

The enumerant names for this channel id are listed in Table 66.

Table 66. PVRTC channel names

Enumerant	Value
`KHR_DF_CHANNEL_PVRTC_COLOR`	0
`KHR_DF_CHANNEL_PVRTC_DATA`	

Table 67. Example PVRTC 4bpp descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `PVRTC`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Table 68. Example PVRTC 2bpp descriptor

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															

uint32_t bit							
flags: `ALPHA_STRAIGHT`					***transferFunction:*** `LINEAR`	***colorPrimaries:*** `BT709`	***colorModel:*** `PVRTC`
texelBlockDimension3					***texelBlockDimension2***	***texelBlockDimension1***	***texelBlockDimension0***
0					0	3 (= "4")	7 (= "8")
bytesPlane3: 0					***bytesPlane2:*** 0	***bytesPlane1:*** 0	***bytesPlane0:*** 8
bytesPlane7: 0					***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Sample information		
0	0	0	0	`COLOR`	***bitLength:*** 63 (= "64")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: `UINT32_MAX`							

5.6.13. `KHR_DF_MODEL_PVRTC2` (= 165)

This model represents the second generation of PowerVR Texture Compression as a single channel in a texel block of 8 bytes. 4-bit-per-pixel mode represents a 4×4 texel block; 2-bit-per-pixel mode represents an 8×4 texel block, and these can be distinguished by changing the texel block size fields in the data format. The single sample has a size of 64 bits.

The enumerant names for this channel id are listed in Table 69.

Table 69. PVRTC2 channel names

Enumerant	Value
`KHR_DF_CHANNEL_PVRTC2_COLOR`	0
`KHR_DF_CHANNEL_PVRTC2_DATA`	

Table 70. Example PVRTC2 4bpp descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `PVRTC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								3 (= "4")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

Table 71. Example PVRTC2 2bpp descriptor

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `PVRTC2`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								3 (= "4")								7 (= "8")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 8							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`COLOR`				***bitLength:*** 63 (= "64")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: `UINT32_MAX`																															

5.7. *colorPrimaries*

It is not sufficient to define a buffer as containing, for example, additive primaries. Additional information is required to define what "red" is provided by the "red" channel. A full definition of primaries requires an extension which provides the full color space of the data, but a subset of common primary spaces can be identified by the `khr_df_primaries_e` enumeration, represented as an unsigned 8-bit integer.

More information about color primaries is provided in Section 14.

Table 72. Field location information for *colorPrimaries*

Word offset into basic descriptor block	**KHR_DF_WORD_PRIMARIES**	2
Word offset into descriptor	**KHR_DF_WORD_PRIMARIES** + 1	3
Start bit within word	**KHR_DF_SHIFT_PRIMARIES**	8
Bit mask of value	**KHR_DF_MASK_PRIMARIES**	0xFF

khr_df_primaries_e ***colorPrimaries*** = **KHR_DF_MASK_PRIMARIES** & (bdb[**KHR_DF_WORD_PRIMARIES**] >> **KHR_DF_SHIFT_PRIMARIES**);

5.7.1. KHR_DF_PRIMARIES_UNSPECIFIED (= 0)

This "set of primaries" identifies a data representation whose color representation is unknown or which does not fit into this list of common primaries. Having an "unspecified" value here precludes users of this data format from being able to perform automatic color conversion unless the primaries are defined in another way. Formats which require a proprietary color space — for example, raw data from a Bayer sensor that records the direct response of each filtered sample — can still indicate that samples represent "red", "green" and "blue", but should mark the primaries here as "unspecified" and provide a detailed description in an extension block.

5.7.2. KHR_DF_PRIMARIES_BT709 (= 1)

This value represents the Color Primaries defined by the ITU-R BT.709 specification and described in Section 14.1, which are also shared by sRGB.

RGB data is distinguished between BT.709 and sRGB by the Transfer Function. Conversion to and from BT.709 $Y'C_BC_R$ (*YUV*) representation uses the color conversion matrix defined in the BT.709 specification, and described in Section 15.1.1, except in the case of sYCC (which can be distinguished by the use of the sRGB transfer function), in which case conversion to and from BT.709 $Y'C_BC_R$ representation uses the color conversion matrix defined in the BT.601 specification, and described in Section 15.1.2. This is the preferred set of color primaries used by HDTV and sRGB, and likely a sensible default set of color primaries for common rendering operations.

KHR_DF_PRIMARIES_SRGB is provided as a synonym for **KHR_DF_PRIMARIES_BT709**.

5.7.3. KHR_DF_PRIMARIES_BT601_EBU (= 2)

This value represents the Color Primaries defined in the ITU-R BT.601 specification for standard-definition television, particularly for 625-line signals, and described in Section 14.2. Conversion to and from BT.601 $Y'C_BC_R$ (*YUV*) typically uses the color conversion matrix defined in the BT.601 specification and described in Section 15.1.2.

5.7.4. KHR_DF_PRIMARIES_BT601_SMPTE (= 3)

This value represents the Color Primaries defined in the ITU-R BT.601 specification for standard-definition television, particularly for 525-line signals, and described in Section 14.3. Conversion to and from BT.601 $Y'C_BC_R$ (*YUV*) typically uses the color conversion matrix defined in the BT.601 specification and described in Section 15.1.2.

5.7.5. KHR_DF_PRIMARIES_BT2020 (= 4)

This value represents the Color Primaries defined in the ITU-R BT.2020 specification for ultra-high-definition television and described in Section 14.4. Conversion to and from BT.2020 $Y'C_BC_R$ (*YUV* uses the color conversion matrix defined in the BT.2020 specification and described in Section 15.1.3.

5.7.6. KHR_DF_PRIMARIES_CIEXYZ (= 5)

This value represents the theoretical Color Primaries defined by the International Color Consortium for the ICC XYZ linear color space.

5.7.7. KHR_DF_PRIMARIES_ACES (= 6)

This value represents the Color Primaries defined for the Academy Color Encoding System and described in Section 14.7.

5.7.8. KHR_DF_PRIMARIES_ACESCC (= 7)

This value represents the Color Primaries defined for the Academy Color Encoding System compositor and described in Section 14.8.

5.7.9. KHR_DF_PRIMARIES_NTSC1953 (= 8)

This value represents the Color Primaries defined for the NTSC 1953 color television transmission standard and described in Section 14.5.

5.7.10. KHR_DF_PRIMARIES_PAL525 (= 9)

This value represents the Color Primaries defined for 525-line PAL signals, described in Section 14.6.

5.7.11. KHR_DF_PRIMARIES_DISPLAYP3 (= 10)

This value represents the Color Primaries defined for the Display P3 color space, described in Section 14.9.

5.7.12. KHR_DF_PRIMARIES_ADOBERGB (= 11)

This value represents the Color Primaries defined in Adobe RGB (1998), described in Section 14.10.

5.8. *transferFunction*

Many color representations contain a non-linear *transfer function* which maps between a linear (intensity-based) representation and a more perceptually-uniform encoding; more information is provided in Section 13. Common transfer functions are represented as an unsigned 8-bit integer and encoded in the enumeration **khr_df_transfer_e**. A fully-flexible transfer function requires an extension with a full color space definition. Where the transfer function can be described as a simple power curve, applying the function is commonly known as "gamma correction". The transfer function is applied to a sample only when the sample's **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit is 0; if this bit is 1, the sample is represented linearly irrespective of the ***transferFunction***.

When a color model contains more than one channel in a sample and the transfer function should be applied only to a subset of those channels, the convention of that model should be used when applying the transfer function. For example, ASTC stores both alpha and *RGB* data but is represented by a single sample; in ASTC, any sRGB transfer function is not applied to the alpha channel of the ASTC texture. In this case, the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit being zero means that the transfer function is "applied" to the ASTC sample in a way that only affects the *RGB* channels. This is not a concern for most color models, which explicitly store different channels in each sample.

If all the samples are linear, **KHR_DF_TRANSFER_LINEAR** should be used. In this case, no sample should have the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit set. If the samples encode a single bit, **KHR_DF_TRANSFER_LINEAR** or **KHR_DF_TRANSFER_UNSPECIFIED** should be used, since there are no intermediate values to which the transfer function should apply.

Table 73. Field location information for *transferFunction*

Word offset into basic descriptor block	**KHR_DF_WORD_TRANSFER**	2
Word offset into descriptor	**KHR_DF_WORD_TRANSFER** + 1	3
Start bit within word	**KHR_DF_SHIFT_TRANSFER**	16
Bit mask of value	**KHR_DF_MASK_TRANSFER**	0xFF

khr_df_transfer_e ***transferFunction*** = **KHR_DF_MASK_TRANSFER** & (bdb[**KHR_DF_WORD_TRANSFER**] >> **KHR_DF_SHIFT_TRANSFER**);

The enumerant value for each of the following transfer functions is shown in parentheses alongside the title.

5.8.1. KHR_DF_TRANSFER_UNSPECIFIED (= 0)

This value should be used when the transfer function is unknown, or specified only in an extension block, precluding conversion of color spaces and correct filtering of the data values using only the information in the basic descriptor block.

5.8.2. KHR_DF_TRANSFER_LINEAR (= 1)

This value represents a linear transfer function: for color data, there is a linear relationship between numerical pixel values and the intensity of additive colors. This transfer function allows for blending and filtering operations to be applied directly to the data values.

5.8.3. KHR_DF_TRANSFER_SRGB (= 2)

This value represents the non-linear transfer function defined in the sRGB specification for mapping between numerical pixel values and displayed light intensity, as described in Section 13.3.

Mapping from linear intensity to encoding	$EOTF^{-1}$	Section 13.3.2
Mapping from encoding to linear intensity	EOTF	Section 13.3.1

Encoded values outside the range 0..1 use the extended formulae for EOTF and $EOTF^{-1}$ described in Section 13.3.4.

5.8.4. KHR_DF_TRANSFER_ITU (= 3)

This value represents the non-linear transfer function defined by the ITU and used in the BT.601, BT.709 and BT.2020 specifications for mapping between represented scene light intensity and numerical pixel values, as described in Section 13.2.

Mapping from linear intensity to encoding	OETF	Section 13.2.1
Mapping from encoding to linear intensity	$OETF^{-1}$	Section 13.2.2

5.8.5. KHR_DF_TRANSFER_NTSC (= 4)

This value represents the non-linear transfer function defined by the original NTSC television broadcast specification for mapping between represented scene light intensity or display light intensity and numerical pixel values, as described in Section 13.8.

Mapping from linear intensity to encoding	$EOTF^{-1}$ / OETF
Mapping from encoding to linear intensity	EOTF / $OETF^{-1}$

More recent formulations of this transfer functions, such as that defined in SMPTE 170M-2004, use the "ITU" formulation described above.

5.8.6. KHR_DF_TRANSFER_SLOG (= 5)

This value represents a nonlinear Transfer Function between linear scene light intensity and nonlinear pixel values, used by some Sony video cameras to represent an increased dynamic range, and is described in Section 13.13.

Mapping from linear intensity to encoding	OETF
Mapping from encoding to linear intensity	$OETF^{-1}$

5.8.7. KHR_DF_TRANSFER_SLOG2 (= 6)

This value represents a nonlinear Transfer Function between linear scene light intensity and nonlinear pixel values, used by some Sony video cameras to represent a further increased dynamic range, and is described in Section 13.14.

Mapping from linear intensity to encoding	OETF
Mapping from encoding to linear intensity	$OETF^{-1}$

5.8.8. KHR_DF_TRANSFER_BT1886 (= 7)

This value represents the nonlinear *[Math Processing Error]* EOTF between encoded pixel values and linear image intensity defined in BT.1886 and described in Section 13.4.

Mapping from linear intensity to encoding	$EOTF^{-1}$	*[Math Processing Error]*
Mapping from encoding to linear intensity	EOTF	*[Math Processing Error]*

5.8.9. KHR_DF_TRANSFER_HLG_OETF (= 8)

This value represents the Hybrid Log Gamma OETF between linear scene light intensity and nonlinear pixel values, defined by the ITU in BT.2100 for high dynamic range television, and described in Section 13.5.

Mapping from linear intensity to encoding	OETF	Section 13.5.1
Mapping from encoding to linear intensity	$OETF^{-1}$	Section 13.5.2

5.8.10. KHR_DF_TRANSFER_HLG_EOTF (= 9)

This value represents the Hybrid Log Gamma EOTF between nonlinear pixel values and linear image light intensity, defined by the ITU in BT.2100 for high dynamic range television, and described in Section 13.5.

Mapping from linear intensity to encoding	$EOTF^{-1}$	Section 13.5.9
Mapping from encoding to linear intensity	EOTF	Section 13.5.8

5.8.11. KHR_DF_TRANSFER_PQ_EOTF (= 10)

This value represents the Perceptual Quantization EOTF between nonlinear pixel values and linear image light intensity, defined by the ITU in BT.2100 for high dynamic range television, and described in Section 13.6.

Mapping from linear intensity to encoding	$EOTF^{-1}$	Section 13.6.2
Mapping from encoding to linear intensity	EOTF	Section 13.6.1

5.8.12. KHR_DF_TRANSFER_PQ_OETF (= 11)

This value represents the Perceptual Quantization OETF between linear scene light intensity and nonlinear pixel values, defined by the ITU in BT.2100 for high dynamic range television, and described in Section 13.6.

Mapping from linear intensity to encoding	OETF	Section 13.6.4
Mapping from encoding to linear intensity	$OETF^{-1}$	Section 13.6.6

5.8.13. KHR_DF_TRANSFER_DCIP3 (= 12)

This value represents the transfer function between nonlinear pixel values and linear image light intensity defined in DCI P3 and described in Section 13.7.

Mapping from linear intensity to encoding	$EOTF^{-1}$
Mapping from encoding to linear intensity	EOTF

5.8.14. KHR_DF_TRANSFER_PAL_OETF (= 13)

This value represents the OETF between linear scene light intensity and nonlinear pixel values for legacy PAL systems described in Section 13.9.

Mapping from linear intensity to encoding	OETF
Mapping from encoding to linear intensity	$OETF^{-1}$

5.8.15. KHR_DF_TRANSFER_PAL625_EOTF (= 14)

This value represents the EOTF between nonlinear pixel values and linear image light intensity for legacy 625-line PAL systems described in Section 13.10.

Mapping from linear intensity to encoding	EOTF $^{-1}$
Mapping from encoding to linear intensity	EOTF

5.8.16. KHR_DF_TRANSFER_ST240 (= 15)

This value represents the transfer function between linear scene light intensity and nonlinear pixel values associated with the legacy ST-240 (SMPTE240M) standard, described in Section 13.11.

Mapping from linear intensity to encoding	EOTF $^{-1}$ / OETF
Mapping from encoding to linear intensity	EOTF / OETF $^{-1}$

5.8.17. KHR_DF_TRANSFER_ACESCC (= 16)

This value represents the nonlinear transfer function between linear scene light intensity and nonlinear pixel values used in the ACEScc Academy Color Encoding System logarithmic encoding system for use within Color Grading Systems, S-2014-003, defined in ACES. This is described in Section 13.15.

Mapping from linear intensity to encoding	OETF
Mapping from encoding to linear intensity	OETF $^{-1}$

5.8.18. KHR_DF_TRANSFER_ACESCCT (= 17)

This value represents the nonlinear transfer function between linear scene light intensity and nonlinear pixel values used in the ACEScc Academy Color Encoding System quasi-logarithmic encoding system for use within Color Grading Systems, S-2016-001, defined in ACES. This is described in Section 13.16.

Mapping from linear intensity to encoding	OETF
Mapping from encoding to linear intensity	OETF $^{-1}$

5.8.19. KHR_DF_TRANSFER_ADOBERGB (= 18)

This value represents the transfer function defined in the Adobe RGB (1998) specification and described in Section 13.12.

5.9. *flags*

The format supports some configuration options in the form of boolean flags; these are described in the enumeration **khr_df_flags_e** and represented in an unsigned 8-bit integer value.

Table 74. Field location information for *flags*

Word offset into basic descriptor block	KHR_DF_WORD_FLAGS	2
Word offset into descriptor	KHR_DF_WORD_FLAGS + 1	3
Start bit within word	KHR_DF_SHIFT_FLAGS	24
Bit mask of value	KHR_DF_MASK_FLAGS	0xFF

khr_df_flags_e ***flags*** = **KHR_DF_MASK_FLAGS** & (bdb[**KHR_DF_WORD_FLAGS**] >> **KHR_DF_SHIFT_FLAGS**);

5.9.1. KHR_DF_FLAG_ALPHA_PREMULTIPLIED (= 1)

If the **KHR_DF_FLAG_ALPHA_PREMULTIPLIED** bit is set, any color information in the data should be interpreted as having been previously scaled/modulated by the alpha channel when performing blending operations.

The value **KHR_DF_FLAG_ALPHA_STRAIGHT** (= 0) is provided to represent this flag not being set, which indicates that color values in the data should be interpreted as needing to be scaled by the alpha channel when performing blending operations. This flag has no effect if there is no alpha channel in the format.

5.10. *texelBlockDimension[0..3]*

The ***texelBlockDimension*** fields define an integer bound on the range of coordinates covered by the repeating block described by the samples; that is, the texel block covers an integer range in each dimension of coordinate space. Four separate values, represented as unsigned 8-bit integers, are supported, corresponding to successive dimensions: the Basic Data Format Descriptor Block supports up to four dimensions of encoding within a texel block, supporting, for example, a texture with three spatial dimensions and one temporal dimension. Nothing stops the data structure as a whole from having higher dimensionality: for example, a two-dimensional texel block can be used as an element in a six-dimensional look-up table.

The value held in each of these fields is one fewer than the size of the block in that dimension — that is, a value of 0 represents a size of 1, a value of 1 represents a size of 2, etc. A texel block which covers fewer than four dimensions should have a size of 1 in each dimension that it lacks, and therefore the corresponding fields in the representation should be 0.

For example, a $Y'C_BC_R$ 4:2:0 representation may use a Texel Block of 2×2 pixels in the nominal coordinate space, corresponding to the four Y' samples, as shown in Table 75. The texel block dimensions in this case would be 2×2×1×1 (in the X, Y, Z and T dimensions, if the fourth dimension is interpreted as T). The ***texelBlockDimension[0..3]*** values would therefore be:

Table 75. Example Basic Data Format *texelBlockDimension* values for $Y'C_BC_R$ 4:2:0

texelBlockDimension0	1

texelBlockDimension1	1
texelBlockDimension2	0
texelBlockDimension3	0

In the descriptor block examples in this specification, block dimensions larger than 1 (encoded as 0) are shown as the value to be stored in the ***texelBlockDimension*** field, but with the represented number in parentheses for clarity.

Table 76. Field location information for *texelBlockDimension[0..3]*

Word offset into basic descriptor block	`KHR_DF_WORD_TEXELBLOCKDIMENSION[0..3]`	3
Word offset into descriptor	`KHR_DF_WORD_TEXELBLOCKDIMENSION[0..3]` + 1	4
Start bit within word	`KHR_DF_SHIFT_TEXELBLOCKDIMENSION0`	0
	`KHR_DF_SHIFT_TEXELBLOCKDIMENSION1`	8
	`KHR_DF_SHIFT_TEXELBLOCKDIMENSION2`	16
	`KHR_DF_SHIFT_TEXELBLOCKDIMENSION3`	24
Bit mask of value	`KHR_DF_MASK_TEXELBLOCKDIMENSION[0..3]`	0xFF

`uint32_t` ***texelBlockDimension0*** = `KHR_DF_MASK_TEXELBLOCKDIMENSION0` & (bdb[`KHR_DF_WORD_TEXELBLOCKDIMENSION0`] >> `KHR_DF_SHIFT_TEXELBLOCKDIMENSION0`);

`uint32_t` ***texelBlockDimension1*** = `KHR_DF_MASK_TEXELBLOCKDIMENSION1` & (bdb[`KHR_DF_WORD_TEXELBLOCKDIMENSION1`] >> `KHR_DF_SHIFT_TEXELBLOCKDIMENSION1`);

`uint32_t` ***texelBlockDimension2*** = `KHR_DF_MASK_TEXELBLOCKDIMENSION2` & (bdb[`KHR_DF_WORD_TEXELBLOCKDIMENSION2`] >> `KHR_DF_SHIFT_TEXELBLOCKDIMENSION2`);

`uint32_t` ***texelBlockDimension3*** = `KHR_DF_MASK_TEXELBLOCKDIMENSION3` & (bdb[`KHR_DF_WORD_TEXELBLOCKDIMENSION3`] >> `KHR_DF_SHIFT_TEXELBLOCKDIMENSION3`);

5.11. *bytesPlane[0..7]*

The Basic Data Format Descriptor divides the image into a number of planes, each consisting of an integer number of consecutive bytes. The requirement that planes consist of consecutive data means that formats with distinct subsampled channels — such as $Y'C_BC_R$ 4:2:0 — may require multiple planes to describe a channel. A typical $Y'C_BC_R$ 4:2:0 image has *two* planes for the Y' channel in this representation, offset by one line vertically.

The use of byte granularity to define planes is a choice to allow large texel blocks. A consequence of this is that formats which are not byte-aligned on each addressable unit, such as 1-bit-per-pixel formats, need to represent a texel block of multiple samples, covering multiple texels — as, for example, in Table 92.

A maximum of eight independent planes is supported in the Basic Data Format Descriptor. Formats which require more than eight planes — which are rare — require an extension.

The ***bytesPlane[0..7]*** fields each contain an unsigned 8-bit integer which represents the number of bytes which a plane contributes to the format. If the top bit of a ***bytesPlane[n]*** field is set, bits 6..0 of the ***bytesPlane[n+1]*** field form bits 13..7 of the number of bytes in the plane (and the next plane is described by field ***bytesPlane[n+2]***). For example, if ***bytesPlane0*** is 0xC0 and ***bytesPlane1*** is 0x02, the first plane holds 0x40 + 128 × 0x02 = 0x140 bytes; ***bytesPlane2*** then describes the number of bytes in the second plane.. Since only sixteen bits are used to encode a bit offset for each sample, 14 bits (two bytes excluding the top bits) are sufficient to encode any useful number of bytes — there is no need to "extend" the higher byte. Few formats are expected to require this "extension bit", so for most of this specification, the number of bytes in a plane is considered to be synonymous with the ***bytesPlane*** value.

The first field which contains the value 0 indicates that only a subset of the 8 possible planes are present; that is, planes which are not present should be given the ***bytesPlane*** value of 0, and any ***bytesPlane*** values after the first 0 are ignored. If no ***bytesPlane*** value is zero, 8 planes are considered to exist.

Table 77. Field location information for *bytesPlane[0..7]*

Word offset into basic descriptor block	`KHR_DF_WORD_BYTESPLANE[0..3]`	4
	`KHR_DF_WORD_BYTESPLANE[4..7]`	5
Word offset into descriptor	`KHR_DF_WORD_BYTESPLANE[0..3]` + 1	5
	`KHR_DF_WORD_BYTESPLANE[4..7]` + 1	6
Start bit within word	`KHR_DF_SHIFT_BYTESPLANE0`	0
	`KHR_DF_SHIFT_BYTESPLANE1`	8
	`KHR_DF_SHIFT_BYTESPLANE2`	16
	`KHR_DF_SHIFT_BYTESPLANE3`	24
	`KHR_DF_SHIFT_BYTESPLANE4`	0
	`KHR_DF_SHIFT_BYTESPLANE5`	8
	`KHR_DF_SHIFT_BYTESPLANE6`	16
	`KHR_DF_SHIFT_BYTESPLANE7`	24
Bit mask of value	`KHR_DF_MASK_BYTESPLANE[0..7]`	0xFF

`uint32_t` ***bytesPlane[0..7]*** = `KHR_DF_MASK_BYTESPLANE[0..7]` & (bdb[`KHR_DF_WORD_BYTESPLANE[0..7]`] >> `KHR_DF_SHIFT_BYTESPLANE[0..7]`);

In versions of this specification prior to 1.3, there was no facility for the "extension bit", and a ***bytesPlane0*** value of 0 indicated a paletted format. The scheme for encoding paletted formats as of version 1.3 is described in Section 5.18.

5.12. Sample information

The layout and position of the information within each plane is determined by a number of *samples*, each consisting of a single channel of data and with a single corresponding position within the texel block, as shown in Table 78.

Table 78. Basic Data Format Descriptor Sample Information

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	***S***	***E***	***L***	***channelType***				***bitLength***								***bitOffset***															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
sampleLower																															
sampleUpper																															

Bits ***F***, ***S***, ***E*** and ***L*** are abbreviations for the following qualifier flags:

F	**`KHR_DF_SAMPLE_DATATYPE_FLOAT`**	0x80
S	**`KHR_DF_SAMPLE_DATATYPE_SIGNED`**	0x40
E	**`KHR_DF_SAMPLE_DATATYPE_EXPONENT`**	0x20
L	**`KHR_DF_SAMPLE_DATATYPE_LINEAR`**	0x10

The sample information begins at word **`KHR_DF_WORD_SAMPLESTART`** = 6, offset from the start of the basic descriptor block. Each sample occupies **`KHR_DF_WORD_SAMPLEWORDS`** = 4 32-bit words, and is stored consecutively.

The bytes from the plane data contributing to the format are treated as though they have been concatenated into a bit stream, with the first byte of the lowest-numbered plane providing the lowest bits of the result. Each sample consists of a number of consecutive bits from this bit stream.

If the content for a channel cannot be represented in a single sample, for example because the data for a channel is non-consecutive within this bit stream, additional samples with the same coordinate position and channel number should follow from the first, in order increasing from the least significant bits from the channel data; the corresponding bits from the bit stream are concatenated in the increasing order of reference to provide the value representing the channel.

For example, some native big-endian formats may need to be supported with multiple samples in a channel, since the constituent bits may not be consecutive in a little-endian interpretation. There is an example, Table 97, in the list of example format descriptors provided.

See Section 3.4 for more information about the order in which samples should appear in the descriptor block.

The number of samples present in the format is determined by the ***descriptorBlockSize*** field:

numSamples = (((**`KHR_DFDVAL`**(`BDB`, **`DESCRIPTORBLOCKSIZE`**) >> 2) - **`KHR_DF_WORD_SAMPLESTART`**) / **`KHR_DF_WORD_SAMPLEWORDS`**);

The macro **`KHR_DFDSAMPLECOUNT`**(`BDB`) is provided to perform this calculation.

There is no limit on the number of samples which may be present, other than the maximum size of the Data Format Descriptor Block. There is no requirement that samples should access unique parts of the bit-stream: formats such as combined intensity and alpha, or shared exponent formats, require that bits be reused. Nor is there a requirement that all the bits in a plane be used (a format may contain padding).

It is unusual but legal for a descriptor block to contain no samples provided the color model is **`KHR_DF_MODEL_UNSPECIFIED`**. See Section 5.19 for details.

To simplify code using the Basic Data Format Descriptor Block, the header `khr_df.h` provides enums of the following form for accessing sample fields:

Table 79. Field location information for sample field *xxx*

Word offset relative to start of sample	**`KHR_DF_SAMPLEWORD_xxx`** + 1	...
Start bit within word	**`KHR_DF_SAMPLESHIFT_xxx`**	...
Bit mask of value	**`KHR_DF_SAMPLEMASK_xxx`**	...

If the basic descriptor block is treated as a **`uint32_t`** array `bdb[]`, sample field ***xxx*** can be accessed as follows:

xxx = **`KHR_DF_SAMPLEMASK_xxx`** & (`bdb`[**`KHR_DF_WORD_SAMPLESTART`** + *sample* × **`KHR_DF_WORD_SAMPLEWORDS`** + **`KHR_DF_SAMPLEWORD_xxx`**] >> **`KHR_DF_SAMPLESHIFT_xxx`**);

The macro **`KHR_DFDSVAL`**(`BDB`, `S`, `X`) is provided to perform this calculation.

For example, **`KHR_DFDSVAL`**(`bdb`, 2, **`CHANNELID`**) returns the value:

KHR_DF_SAMPLEMASK_CHANNELID & (bdb[**KHR_DF_WORD_SAMPLESTART** + 2 × **KHR_DF_WORD_SAMPLEWORDS** + **KHR_DF_SAMPLEWORD_CHANNELID**] >> **KHR_DF_SAMPLESHIFT_CHANNELID**)

5.13. Sample *bitOffset*

The ***bitOffset*** field describes the offset of the least significant bit of this sample from the least significant bit of the least significant byte of the concatenated bit stream for the format. Typically the ***bitOffset*** of the first sample is therefore 0; a sample which begins at an offset of one byte relative to the data format would have a ***bitOffset*** of 8. The ***bitOffset*** is an unsigned 16-bit integer quantity.

In the special case that the ***bitOffset*** field contains the reserved value 0xFFFF, the sample contributes a constant value of the specified bit length, encoded in the ***sampleLower*** field. This mechanism notably supports values that are zero-extended.

Table 80. Field location information for sample *bitOffset*

Word offset relative to start of sample	**KHR_DF_SAMPLEWORD_BITOFFSET**	0
Start bit within word	**KHR_DF_SAMPLESHIFT_BITOFFSET**	0
Bit mask of value	**KHR_DF_SAMPLEMASK_BITOFFSET**	0xFFFFU

uint32_t ***bitoffset*** = **KHR_DF_SAMPLEMASK_BITOFFSET** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_BITOFFSET**]) >> **KHR_DF_SAMPLESHIFT_BITOFFSET**);

5.14. Sample *bitLength*

The ***bitLength*** field describes the number of consecutive bits from the concatenated bit stream that contribute to the sample. This field is an unsigned 8-bit integer quantity, and stores the number of bits contributed minus 1; thus a single-byte channel should have a ***bitLength*** field value of 7. If a ***bitLength*** of more than 256 is required, further samples should be added; the value for the sample is composed in increasing order from least to most significant bit as subsequent samples are processed.

Note that a large ***bitLength*** value means a sample can encode more bits than can be described in the ***sampleLower*** and ***sampleUpper*** fields. If the rules for expanding the ***sampleLower*** and ***sampleUpper*** values in this case produce the desired result, ***bitLength*** can exceed 32; otherwise the ***bitLength*** should be limited and multiple samples used to encode ***sampleLower*** and ***sampleUpper***: there is no way to indicate a sample contribution of zero bits for a sample that exists only to expand upon ***sampleLower*** and ***sampleUpper***.

Except in the case of a paletted texture (described in Section 5.18) or where the special ***bitOffset*** value 0xFFFF is used to indicate constant bits, the ***bitLength*** added to ***bitOffset*** should not be greater than eight times the total number of bytes contributed to the logical bit stream by the ***bytesPlane*** values.

In the descriptor block examples in this specification, bit lengths are shown as the value to be stored in the ***bitLength*** field, but with the represented number (without the -1 offset) in parentheses for clarity.

Table 81. Field location information for sample *bitLength*

Word offset relative to start of sample	**KHR_DF_SAMPLEWORD_BITLENGTH**	0
Start bit within word	**KHR_DF_SAMPLESHIFT_BITLENGTH**	16
Bit mask of value	**KHR_DF_SAMPLEMASK_BITLENGTH**	0xFF

uint32_t ***bitLength*** = **KHR_DF_SAMPLEMASK_BITLENGTH** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_BITLENGTH**]) >> **KHR_DF_SAMPLESHIFT_BITLENGTH**);

5.15. Sample *channelType* and qualifiers

The ***channelType*** field is an unsigned 8-bit quantity.

The bottom four bits of the ***channelType*** indicates which channel is being described by this sample. The list of available channels is determined by the ***colorModel*** field of the Basic Data Format Descriptor Block, and the ***channelType*** field contains the number of the required channel within this list — see the ***colorModel*** field for the list of channels for each model.

Table 82. Field location information for sample *channelType*

Word offset relative to start of sample	**KHR_DF_SAMPLEWORD_CHANNELID**	0
Start bit within word	**KHR_DF_SAMPLESHIFT_CHANNELID**	24
Bit mask of value	**KHR_DF_SAMPLEMASK_CHANNELID**	0xF

khr_df_model_channels_e ***channelType*** = **KHR_DF_SAMPLEMASK_CHANNELID** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_CHANNELID**]) >> **KHR_DF_SAMPLESHIFT_CHANNELID**);

The top four bits of the ***channelType*** are described by the **khr_df_sample_datatype_qualifiers_e** enumeration:

Table 83. Field location information for sample *qualifiers*

Word offset relative to start of sample	**KHR_DF_SAMPLEWORD_QUALIFIERS**	0
Start bit within word	**KHR_DF_SAMPLESHIFT_QUALIFIERS**	24
Bit mask of value	**KHR_DF_SAMPLEMASK_QUALIFIERS**	0xF0

khr_df_sample_datatype_qualifiers_e ***qualifiers*** = **KHR_DF_SAMPLEMASK_QUALIFIERS** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_QUALIFIERS**]) >> **KHR_DF_SAMPLESHIFT_QUALIFIERS**);

If the **KHR_DF_SAMPLE_DATATYPE_EXPONENT** bit, shown as ***E*** in Table 78, is not set, the sample contributes to a *base value*; multiple samples with the same color channel (bottom four bits of ***channelType***) and same ***samplePos*** values are accumulated into the virtual sample in increasing bit order from lowest to highest.

For samples in which the **KHR_DF_SAMPLE_DATATYPE_EXPONENT** bit is not set:

- If the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit, shown as ***L*** in Table 78, is not set, the final sample value (after any modifier has been applied to the base value) is modified by the transfer function defined in the ***transferFunction*** field of the descriptor; if this bit *is* set, the sample is considered to contain a linearly-encoded value irrespective of the format's ***transferFunction***. All samples referring to the same base value should have the same value stored in the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit.
- If the **KHR_DF_SAMPLE_DATATYPE_SIGNED** bit, shown as ***S*** in Table 78, is set, the sample holds a signed value in two's complement form. If this bit is *not* set, the sample holds an unsigned value. It is possible to represent a sign/magnitude integer value by having a sample of unsigned integer type with the same channel and sample location, as a 1-bit signed sample.
- If the **KHR_DF_SAMPLE_DATATYPE_FLOAT** bit, shown as ***F*** in Table 78, is set, the sample holds floating point data in a conventional format of 10, 11 or 16 bits, as described in Section 10, or of 32, or 64 bits as described in [IEEE 754]. Unless a genuine unsigned format is intended, **KHR_DF_SAMPLE_DATATYPE_SIGNED** (bit ***S***) should also be set. Less common floating point representations can be generated with multiple samples and a combination of signed integer, unsigned integer and exponent fields, as described above and in Section 10.4.

If the **KHR_DF_SAMPLE_DATATYPE_EXPONENT** bit, shown as ***E*** in Table 78, *is* set, the sample applies a *modifier* to the base value, with the interpretation of the modifier determined according to Table 84. In this case, the virtual sample contains both a base value and a modifier.

All samples contributing to a modifier for the same base value should contain the same ***L*** and ***F*** bits (it is not legal, for example, to define both a multiplier and a divisor). Samples which apply a modifier should directly follow the samples that describe the base value. If no samples have the ***E*** bit set for this channel and position, the base value directly represents the pixel value; it is not legal for a virtual sample to describe a modifier but no base value.

note

The same bits of the format may contribute to modifiers for more than one channel — this is commonly the case for high dynamic range formats with a shared exponent or divisor. The descriptor in this case should contain samples for each color channel in turn, with the description of the shared bits replicated for each channel, as shown in Table 98.

Table 84. Qualifier interpretation when KHR_DF_SAMPLE_DATATYPE_EXPONENT = 1

E	*L*	*F*	Interpretation	Formula
1	0	0	Exponent	*[Math Processing Error]*
1	0	1	Multiplier	*[Math Processing Error]*
1	1	0	Divisor	*[Math Processing Error]*
1	1	1	Power	*[Math Processing Error]*

For samples in which the **KHR_DF_SAMPLE_DATATYPE_EXPONENT** bit is set:

- If the **KHR_DF_SAMPLE_DATATYPE_LINEAR** and **KHR_DF_SAMPLE_DATATYPE_FLOAT** bits are clear, this modifier holds an exponent (in integer form) describing a floating-point offset for this channel. For example, this would be used to describe the exponent of a custom floating point format, as shown in Table 103, or a shared exponent location in shared exponent formats (with the exponent bits listed separately under each channel as shown in Table 98). If this modifer is used, the base value is considered to contain mantissa information and the samples describing it would not normally have the **KHR_DF_SAMPLE_DATATYPE_FLOAT** bit set. If the **KHR_DF_SAMPLE_DATATYPE_SIGNED** bit (***S***) is also set, the exponent is considered to be two's complement — otherwise it is treated as unsigned. The bias of the exponent can be determined by the exponent's ***sampleLower*** value. The presence or absence of an implicit leading digit in the mantissa of a format with an exponent can be determined by the ***sampleUpper*** value of the mantissa. The use of the exponent is described in more detail in Section 10.4.
- If the **KHR_DF_SAMPLE_DATATYPE_FLOAT** bit is set and the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit is clear, this sample holds a multiplier (in integer form) for this channel, such that the encoded value is a product of this modifier value and the base value. This approach is useful for encoding a shared multiplier as part of a high dynamic range color image, for example.
- If the **KHR_DF_SAMPLE_DATATYPE_LINEAR** bit is set and the **KHR_DF_SAMPLE_DATATYPE_FLOAT** bit is clear, this sample holds a divisor (in integer form) for this channel, such that the encoded value is the base value divided by this modifier value. This approach is useful for encoding a shared divisor as part of a high dynamic range color image, for example.
- If both **KHR_DF_SAMPLE_DATATYPE_FLOAT** and **KHR_DF_SAMPLE_DATATYPE_LINEAR** are set, this sample holds a power term (in integer form) for this channel, such that the encoded value is the base value raised to the power of the modifier value. This approach is useful for encoding a shared multiplier as part of a high dynamic range color image, for example.

Note that in the multiplier, divisor and power cases, the ***sampleLower*** and ***sampleUpper*** values allow the modifier value to be represented in fixed-point terms, and the values may be signed depending on whether the ***S*** bit is set.

5.16. *samplePosition[0..3]*

The sample has an associated location within the 4-dimensional space of the texel block. Therefore each sample has an offset relative to the 0,0 position of the texel block, represented as an 8-bit unsigned integer quantity.

Table 85. Field location information for sample *samplePosition[0..3]*

Word offset relative to start of sample	KHR_DF_SAMPLEWORD_SAMPLEPOSITION[0..3]	1
Start bit within word	KHR_DF_SAMPLESHIFT_SAMPLEPOSITION0	0
	KHR_DF_SAMPLESHIFT_SAMPLEPOSITION1	8
	KHR_DF_SAMPLESHIFT_SAMPLEPOSITION2	16
	KHR_DF_SAMPLESHIFT_SAMPLEPOSITION3	24
Bit mask of value	KHR_DF_SAMPLEMASK_SAMPLEPOSITION[0..3]	0xF

khr_df_model_channels_e ***samplePosition0*** = **KHR_DF_SAMPLEMASK_SAMPLEPOSITION0** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_SAMPLEPOSITION0**]) >> **KHR_DF_SAMPLESHIFT_SAMPLEPOSITION0**);

khr_df_model_channels_e ***samplePosition1*** = **KHR_DF_SAMPLEMASK_SAMPLEPOSITION1** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_SAMPLEPOSITION1**]) >> **KHR_DF_SAMPLESHIFT_SAMPLEPOSITION1**);

khr_df_model_channels_e ***samplePosition2*** = **KHR_DF_SAMPLEMASK_SAMPLEPOSITION2** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_SAMPLEPOSITION2**]) >> **KHR_DF_SAMPLESHIFT_SAMPLEPOSITION2**);

khr_df_model_channels_e ***samplePosition3*** = **KHR_DF_SAMPLEMASK_SAMPLEPOSITION3** & ((bdb[**KHR_DF_WORD_SAMPLESTART** + (*sample* × **KHR_DF_WORD_SAMPLEWORDS**) + **KHR_DF_SAMPLEWORD_SAMPLEPOSITION3**]) >> **KHR_DF_SAMPLESHIFT_SAMPLEPOSITION3**);

The interpretation of each ***samplePosition*** field depends on the corresponding ***texelBlockDimension*** value as follows:

[Math Processing Error]

For example, if ***texelBlockDimension0*** is 1 (indicating a texel block width of two units) ***samplePosition0*** is described in units of *[Math Processing Error]* . That is, a ***samplePosition0*** of 128 would encode an offset of "1.0" and a ***samplePosition0*** of 64 would encode an offset of "0.5". If ***texelBlockDimension0*** is 5 (indicating a texel block width of six units), ***samplePosition0*** is described in units of *[Math Processing Error]* . That is, a ***samplePosition0*** of 64 would encode an offset of "2.0" and a ***samplePosition0*** of 24 would encode an offset of "0.75".

The adjusted *coordinateOffset* must be less than the corresponding texel block dimension. That is, since *coordinateOffset* can represent fractional offsets, *coordinateOffset* < (***texelBlockDimension*** + 1) for each dimension.

This approach allows the common situation of downsampled channels to have samples conceptually sited at the midpoint between full resolution samples. The direction of the sample offsets is determined by the coordinate addressing scheme used by the API. There is no limit on the dimensionality of the data, but if more than four dimensions need to be contained within a single texel block, an extension will be required.

It is legal, but unusual, to use the same bits to represent multiple samples at different coordinate locations.

Versions of this specification prior to 1.3 always recorded the sample position in a 7.1 fixed-point format (with half-coordinate granularity). This change does not affect the representation of single-coordinate texel blocks; that is, a ***samplePosition*** of "0" still represents "0.0".

5.17. *sampleLower* and *sampleUpper*

The ***sampleLower*** and ***sampleUpper*** fields are used to define the mapping between the numerical value stored in the format and the conceptual numerical interpretation.

For unsigned formats, ***sampleLower*** typically represents the value which should be interpreted as zero (the black point). For signed formats, ***sampleLower*** typically represents "-1". For color difference models such as $Y'C_BC_R$, ***sampleLower*** for chroma channels represents the lower extent of the color difference range (which corresponds to an encoding of -0.5 in numerical terms).

sampleUpper typically represents the value which should be interpreted as "1.0" (the "white point"). For color difference models such as $Y'C_BC_R$, ***sampleUpper*** for chroma channels represents the upper extent of the color difference range (which corresponds to an encoding of 0.5 in numerical terms).

Equation 1. Sample range conversion rules

[Math Processing Error]

For example, the BT.709 television broadcast standard dictates that the *Y'* value stored in an 8-bit encoding should fall between the range 16 and 235, as described in Section 16.1 encoding". In this case, ***sampleLower*** should contain the value 16 and ***sampleUpper*** 235.

Sensor data from a camera typically does not cover the full range of the bit depth used to represent it. ***sampleUpper*** can be used to specify an upper limit on sensor brightness — or to specify the value which should map to white on the display, which may be less than the full dynamic range of the captured image.

There is no guarantee or expectation that image data be guaranteed to fall between ***sampleLower*** and ***sampleUpper*** unless the users of a format agree that convention. For example, high dynamic range video formats may define "1.0" as a nominal brightness level substantially lower than the maximum, and coordinates may encode an arbitrary range. In some formats, the integer value should be interpreted directly as a number, in which case ***sampleUpper*** and ***sampleLower*** should hold "1" and either "0" or "-1" depending on whether the format is signed, respectively.

If the channel encoding is an integer format, the ***sampleLower*** and ***sampleUpper*** values are represented as 32-bit integers — signed or unsigned according to whether the channel encoding is signed. Signed negative values should be sign-extended if the channel has fewer than 32 bits, such that the value encoded in ***sampleLower*** and ***sampleUpper*** values are themselves negative if the encoded values are negative. If the channel encoding is a floating point value, the ***sampleLower*** and ***sampleUpper*** values are also described in floating point.

If the number of bits in the sample is greater than 32, ***sampleLower*** and ***sampleUpper*** are converted to a data type of the actual number of bits as follows: A floating point value is converted to the native representation (for example, a `float` value is converted to `double` in the normal way). An integer value is expanded by preserving any sign bit and replicating the top non-sign bit (for example, signed 0x80000000 is extended to the 40-bit value 0x8000000000). If these rules do not produce the desired result, it may be necessary to describe the contribution to the channel in multiple samples of no more than 32 bits. In this case, the samples corresponding to lower ***bitOffset*** values should occupy 32 bits, with any residual bits encoded in samples corresponding to higher ***bitOffset*** values.

If multiple samples contribute to a single value, for example because the bits of a channel are non-contiguous in the logical bit stream, the ***sampleLower*** and ***sampleUpper*** fields of each sample are concatenated in increasing bit order to produce virtual ***sampleLower*** and ***sampleUpper*** values; each sample contributes a number of bits equal to the number of data bits that the sample describes. If this contribution is fewer than 32 bits and the value being encoded is signed, the sign bit of the final ***sampleLower*** and ***sampleUpper*** values should be used to pad the fields of the sample.

For example, if a signed 16-bit value with a minimum value of -32767 (0xFFFE) is described as two 8-bit values, the first sample should have a ***sampleLower*** value of 0xFFFFFFFE — of which the bottom 8 bits (0xFE) corresponds to the bottom 8 bits of the final minimum value and the upper 24 bits (0xFFFFFF00) are a result of replicating the sign of bit 15 of the final minimum value. The second sample should have a ***sampleLower*** value of 0xFFFFFFFF — of which the bottom 8 bits (0xFF) correspond to the top 8 bits of the final minimum value and the upper 24 bits (0xFFFFFF00) are a result of sign-extending this value. Only the sample corresponding to the top bits of the channel may have a ***sampleLower*** or ***sampleUpper*** occupying more bits than the input.

The ***sampleLower*** value for an exponent should represent the exponent bias — the value that should be subtracted from the encoded exponent to indicate that the mantissa's ***sampleUpper*** value will represent 1.0. The ***sampleUpper*** value for an exponent should represent the largest conventional legal exponent value. If the encoded exponent exceeds this value, the encoded floating point value encodes either an infinity or a *NaN* value, depending on the mantissa. See Section 10.4 for more detail on this.

If the channel encoding is the mantissa of a custom floating point format (that is, the encoding is integer but the same sample location and channel is shared by a sample that encodes an exponent), the presence of an implicit "1" digit can be represented by setting the ***sampleUpper*** value to a value one larger than can be encoded in the available bits for the mantissa, as described in Section 10.4.

In OpenGL terminology, a "normalized" channel contains an integer value which is mapped to the range 0..1.0; a channel which is not normalized contains an integer value which is mapped to a floating point equivalent of the integer value. Similarly an "snorm" channel is a signed normalized value mapping from -1.0 to 1.0. Setting ***sampleLower*** to the minimum signed integer value representable in the channel (which is often the negative version of the maximum signed integer, for example -127 rather than -128 for an 8-bit value in order to allow the value 0.0 to be represented exactly) is equivalent to defining an "snorm" texture. Setting ***sampleUpper*** to the maximum signed integer value representable in the channel for a signed channel type is equivalent to defining an "snorm" texture. Setting ***sampleUpper*** to the maximum unsigned value representable in the channel for an unsigned channel type is equivalent to defining a "normalized" texture. Setting ***sampleUpper*** to "1" is equivalent to defining an "unnormalized" texture.

In the special case that the sample ***bitOffset*** field is 0xFFFF, only the bottom 16 bits of the ***sampleLower*** field indicate a contribution to the sample lower limit; the upper 16 bits are taken as a constant contribution to the interpreted value; in this case, the ***bitLength*** field of the sample must be no more than 16. For example, a 4-bit value which is interpreted as being zero-extended to eight bits before conversion may have four bits with the value 0 stored in bits 19..16 of ***sampleLower***, indicated by a ***bitOffset*** of 0xFFFF.

These "virtual bits" may be needed to encode some numerical representations. For example, if an 8-bit integer encodes the value "-0.5" as 0 and "0.5" as 255 (in the manner of the color difference channel in Equation 1, but if we wish to apply this mapping to a channel other than color difference), Equation 1 suggests that ***sampleLower*** should hold -127.5 and ***sampleUpper*** should hold 382.5. This is impossible to encode, since ***sampleLower*** and ***sampleUpper*** are integers. However, if a virtual 0-bit is added below the encoded value (such that the effective range is 0..510), storing ***sampleLower*** as -255 and ***sampleUpper*** as 765 will have the desired effect. Note that in this case, the original 8-bits of data are unsigned, but the bounds must encode a negative value; this can be achieved by describing the data as a signed value that is guaranteed to be positive by another sample storing a virtual 0 bit in bit 9 (so the data is treated as a signed 10-bit quantity, of which the top and bottom bits are guaranteed to be 0). The cost of this flexibility is that multiple samples are needed per channel, which is why the common case of chroma channels which should map to the -0.5..0.5 range are treated specially.

5.18. Paletted formats

The storage of the palette is considered to be outside the remit of this specification; however, the "format" describes both the encoding of the bits which index into the palette and the format of the entries in the palette itself.

The convention for encoding paletted formats was different in revisions of this specification prior to 1.3.

If the ***bitOffset*** field of any of the samples equals eight times the total number of bytes indicated by the ***bytesPlane*** channels (that is, indicating an offset after the end of the logical bit stream), this sample indicates a palette entry. Samples with offsets within the range of the logical bit stream describe the index of the palette; ***sampleUpper*** is used to indicate the number of entries in the palette (which

may be lower than could be addressed by the number of bits available). The index can be comprised of multiple samples, supporting non-contiguous bits.

The four bits encoding ***channelType*** indicate a choice of palette in the index samples. The palette entries indicate which palette they are associated with by encoding the corresponding palette id in the ***samplePosition0*** field, with the other ***samplePosition*** fields set to 0. This approach allows both a simple palette in which each entry represents a complete color and per-channel look-up tables.

The index samples should be described first, as though the format were not paletted; samples for palette entries should then follow, sorted first by palette id, then by channel id.

The descriptor's ***colorModel***, ***colorPrimaries***, ***transferFunction*** and ***flags*** apply to the palette entries. The ***texelBlockDimension*** values on the other hand refer to the storage of per-texel indices.

Table 94 shows a 240-entry palette with an index encoded in eight bits, and six bits for each of *R*, *G* and *B* per palette entry. Table 95 shows a 24-bit format with 256-entry palettes indexed by each of three 256-bit channels, with a 10-bit *R*, *G* and *B* palette associated with the corresponding channels.

5.19. Unsized formats

The data format descriptor can be a convenient representation for describing some data which does not follow the constraint that texel blocks are of a fixed size. For example, it may be useful to use a descriptor to encode the color space of an image which is compressed with a variable-rate compressor (meaning there is no data-independent mapping between memory locations and corresponding pixels).

There are two ways to do this, each with its own uses. Only the color primaries, transfer function and flags may be encoded or these and the presence of color and alpha samples may be encoded.

In both cases an unsized format is indicated by setting all ***bytesPlane*** values to 0.

In the first case, **`KHR_DF_MODEL_UNSPECIFIED`** is used in a descriptor with no samples. All ***texelBlockDimension*** values must be zero.

In the second case an appropriate color model is used, e.g. **`KHR_DF_MODEL_RGBSDA`**. The presence of color and alpha samples is used to indicate the presence of these samples in the image. The ***texelBlockDimension*** values and ***bitOffset***, ***bitLength*** and ***samplePosition*** fields of the samples can all be set to 0 or the values prior to variable-rate compression can be retained.

5.20. C99 struct mapping (informative)

The basic descriptor block has been specified in terms of an array of `uint32_t` value. C99 provides a more direct representation, but this relies on the bit ordering of bitfields (which is implementation-defined) and *flexible array members*, which are not supported in C++.

In the interests of portability, the following summary (which assumes that bitfields are encoded starting at bit 0) is therefore provided for information, but is not canonical:

```
typedef struct _DFDSampleType {
  uint32_t bitOffset: 16;
  uint32_t bitLength: 8;
  uint32_t channelType: 8; // Includes qualifiers
  uint32_t samplePosition0: 8;
  uint32_t samplePosition1: 8;
  uint32_t samplePosition2: 8;
  uint32_t samplePosition3: 8;
  uint32_t lower;
  uint32_t upper;
} DFDSampleType;

typedef struct _BasicDataFormatDescriptor {
  uint32_t vendorId: 17;
  uint32_t descriptorType: 15;
  uint32_t model: 8;
  uint32_t primaries: 8;
  uint32_t transfer: 8;
  uint32_t flags: 8;
  uint32_t texelBlockDimension0: 8;
  uint32_t texelBlockDimension1: 8;
  uint32_t texelBlockDimension2: 8;
  uint32_t texelBlockDimension3: 8;
  uint32_t bytesPlane0: 8;
  uint32_t bytesPlane1: 8;
  uint32_t bytesPlane2: 8;
  uint32_t bytesPlane3: 8;
  uint32_t bytesPlane4: 8;
  uint32_t bytesPlane5: 8;
  uint32_t bytesPlane6: 8;
  uint32_t bytesPlane7: 8;
  DFDSampleType samples[];
} BasicDataFormatDescriptor;
```

6. Extension for more complex formats

Some formats will require more channels than can be described in the Basic Format Descriptor, or may have more specific color requirements. For example, it is expected than an extension will be available which places an ICC color profile block into the descriptor block, allowing more color channels to be specified in more precise ways. This will significantly enlarge the space required for the

descriptor, and is not expected to be needed for most common uses. A vendor may also use an extension block to associate metadata with the descriptor — for example, information required as part of hardware rendering. So long as software which uses the data format descriptor always uses the ***totalSize*** field to determine the size of the descriptor, this should be transparent to user code.

The extension mechanism is the preferred way to support even simple extensions such as additional color spaces transfer functions that can be supported by an additional enumeration. This approach improves compatibility with code which is unaware of the additional values. Simple extensions of this form that have cross-vendor support have a good chance of being incorporated more directly into future revisions of the specification, allowing application code to distinguish them by the ***versionId*** field.

If bit 13, **`KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_WRITE_BIT`**, of the ***descriptorType*** field of an extension is set, an application must understand the extension in order to write data coherently. If this bit is clear, copying the bits which correspond to one texel to another can be expected to result in a correct transfer of the texel even if the application does not understand the extension.

If bit 14, **`KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_DECODE_BIT`**, of the ***descriptorType*** field of an extension is set, an application must understand the extension in order to decode the contents of the texel coherently. If this bit is clear, the data held in the extension can be considered to be "informative" and that ignoring the extension will still result in correct values to the extent specified by the basic descriptor block. For example, an extension may associate an absolute brightness level with a format, but software which does not have need of this concept can continue processing the texel contents correctly.

As an example of the description of an extension, consider a single-channel 32-bit depth buffer, as shown in Table 86. A tiled renderer may wish to indicate that this buffer is "virtual": it will be allocated real memory only if needed, and will otherwise exist only a subset at a time in an on-chip representation. Someone developing such a renderer may choose to add a vendor-specific extension (with ID 0x1FFFF to indicate development work and avoid the need for a vendor ID) which uses a boolean to establish whether this depth buffer exists only in virtual form. Note that the mere presence or absence of this extension within the data format descriptor itself forms a boolean, but for this example we will assume that an extension block is always present, and that a boolean is stored within. We will give the enumeration 32 bits, in order to simplify the possible addition of further extensions and pad to the alignment requirements.

In this example (which should not be taken as an implementation suggestion), the data descriptor would first contain a descriptor block describing the depth buffer format as conventionally described, followed by a second descriptor block that contains only the enumeration. The descriptor itself has a ***totalSize*** that includes both of these descriptor blocks. Note that **`KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_WRITE_BIT`** is not set, indicating that depth data can be written without knowing about the extension, and **`KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_DECODE_BIT`** is not set, indicating that software can safely ignore the information about the form of allocation while reading texel values.

Table 86. Example of a depth buffer with an extension to indicate a virtual allocation

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 56 — total size of the two blocks plus one 32-bit value																															
Basic descriptor block																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags								***transferFunction***								***colorPrimaries***								***colorModel***							
`ALPHA_STRAIGHT`								**`UNSPECIFIED`**								**`UNSPECIFIED`**								**`RGBSDA`**							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 4							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information for depth																							
1	1	0	0	**`DEPTH`**				***bitLength:*** 31 (= "32")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0xBF800000U — -1.0f																															
sampleUpper: 0x7F800000U — 1.0f																															
Extension descriptor block																															
descriptorType: 0																***vendorId:*** 0x1FFFF															
descriptorBlockSize: 8 + (4 × 1) = 12																***versionNumber:*** 0															
Data specific to the extension follows																															
1 — buffer is "virtual"																															

It is possible for a vendor to use the extension block to store peripheral information required to access the image — plane base addresses, stride, etc. Since different implementations have different kinds of non-linear ordering and proprietary alignment requirements, this is not described as part of the standard. By many conventional definitions, this information is not part of the "format", and particularly it ensures that an identical copy of the image will have a different descriptor block (because the addresses will have changed) and so a simple bitwise comparison of two descriptor blocks will disagree even though the "format" matches. Additionally, many APIs will use the format descriptor only for external communication, and have an internal representation that is more concise and less flexible. In this case, it is likely that address information will need to be represented separately from the format anyway. For these reasons, it is an implementation choice whether to store this information in an extension block, and how to do so, rather than being specified in this standard.

7. Additional planes descriptor block

Under some relatively unusual circumstances, either the number of planes described by the basic descriptor block or the number of bytes that can be contributed to a texel may be insufficient to describe the memory layout of bulk data. For example, a format may describe 12-bit colors (4 bits each or red, green and blue), and have the contributing bits stored as separate planes. An extension

descriptor block, with ***vendorId*** = `KHR_DF_VENDORID_KHRONOS` and ***descriptorType*** = `KHR_DF_DESCRIPTORTYPE_ADDITIONAL_PLANES`, describes additional planes.

Table 87. Additional planes descriptor block

`uint32_t` bit	
descriptorType: 0x6001	***vendorId:*** 0
descriptorBlockSize: 8 + (4 × *number of planes*)	***versionNumber:*** 0
bytesPlane0	
bytesPlane1 *(optional)*	
(etc.)	

If this descriptor block is present, the ***bytesPlane[0..7]*** fields of the basic descriptor block are ignored, and the number of bytes for plane *n* is taken directly from word *n* + 2 of the descriptor block. The number of planes described can be determined by the descriptor block size.

This extension both allows an arbitrary number of planes and makes it easy to specify planes that contribute a large number of bytes to the virtual bit stream. Note that the sample ***bitOffset*** field remains limited to 16 bits, so the total texel block memory footprint is limited.

> Since knowing the bit stream contribution from all planes is necessary when interpreting data, this ***descriptorType*** sets `KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_DECODE_BIT`. Since the memory mapping is necessary when writing data, this ***descriptorType*** sets `KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_WRITE_BIT`.

This descriptor block should be used only if the Khronos Basic Descriptor Block is the first descriptor block in the data format descriptor, and cannot represent the format without extension.

8. Additional dimensions descriptor block

The basic descriptor block allows texel blocks of up to four non-trivial dimensions, and with a texel block size of up to 256 coordinate units, with sample positions described in precision up to *[Math Processing Error]* of a coordinate. Under rare circumstances, this may provide insufficient flexibility. An extension descriptor block, with ***vendorId*** = `KHR_DF_VENDORID_KHRONOS` and ***descriptorType*** = `KHR_DF_DESCRIPTORTYPE_ADDITIONAL_DIMENSIONS`, describes additional dimensions. Note that in some cases where this solution might be useful, a texel block is an inappropriate unit. For example, this extension block allows the direct representation of large texel tiling patterns, but it does so in a manner that is very inefficient, having much more potential flexibility than is needed by the users of the layout being described.

Table 88. Additional dimensions descriptor block

`uint32_t` bit	
descriptorType: 0x6002	***vendorId:*** 0
descriptorBlockSize: 8+(4 × *dimensions* × (*samples*+1))	***versionNumber:*** 0
`uint32_t` ***texelBlockDimension0***	
`uint32_t` ***texelBlockDimension1*** *(optional)*	
(etc.)	
`uint16_t` ***sample0Pos0***	`uint16_t` ***sample0Pos0Divisor***
`uint16_t` ***sample0Pos1***	`uint16_t` ***sample0Pos1Divisor***
(etc.)	

The fields ***texelBlockDimension[0..n]*** describe the size in coordinate units of the texel block in the corresponding dimension; as with the basic descriptor block, the value stored is the corresponding dimension minus 1 (so a stored value of "0" corresponds to a dimension of 1). The ***texelBlockDimension[0..7]*** fields of the basic descriptor block are ignored.

For each sample, the ***samplePos*** and ***samplePosDivisor*** fields store a numerator and denominator pair for the coordinate *[Math Processing Error]* for each dimension, with all dimensions for a sample described before describing the next sample. ***samplePos*** and ***samplePosDivisor*** should be minimized, leaving an irreducible fraction. The ***samplePos*** fields of the basic descriptor block are ignored. These fields are present even for samples corresponding to palette entries.

Since the number of samples in the texel block can be deduced from the size of the basic descriptor block, the number of dimensions that are to be described by the additional dimensions descriptor block can be deduced from its size: *numDimensions* = (***descriptorBlockSize*** - 8) / (*numSamples* + 1).

> Since knowing the mapping from coordinates to samples is necessary when interpreting data, this ***descriptorType*** sets `KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_DECODE_BIT`. Since the coordinate mapping is necessary for writing data, this ***descriptorType*** sets `KHR_DF_KHR_DESCRIPTORTYPE_NEEDED_FOR_WRITE_BIT`.

This descriptor block should be used only if the Khronos Basic Descriptor Block is the first descriptor block in the data format descriptor, and cannot represent the format without extension.

9. Frequently Asked Questions

9.1. Why have a binary format rather than a human-readable one?

While it is not expected that every new container will have a unique data descriptor or that analysis of the data format descriptor will be on a critical path in an application, it is still expected that comparison between formats may be time-sensitive. The data format descriptor is designed to allow relatively efficient queries for subsets of properties, to allow a large number of format descriptors to be stored, and to be amenable to hardware interpretation or processing in shaders. These goals preclude a text-based representation such as an XML schema.

9.2. Why not use an existing representation such as those on FourCC.org?

Formats in FourCC.org do not describe in detail sufficient information for many APIs, and are sometimes inconsistent.

9.3. Why have a descriptive format?

Enumerations are fast and easy to process, but are limited in that any software can only be aware of the enumeration values in place when it was defined. Software often behaves differently according to properties of a format, and must perform a look-up on the enumeration — if it knows what it is — in order to change behaviors. A descriptive format allows for more flexible software which can support a wide range of formats without needing each to be listed, and simplifies the programming of conditional behavior based on format properties.

9.4. Why describe this standard within Khronos?

Khronos supports multiple standards that have a range of internal data representations. There is no requirement that this standard be used specifically with other Khronos standards, but it is hoped that multiple Khronos standards may use this specification as part of a consistent approach to inter-standard operation.

9.5. Why should I use this descriptor if I don't need most of the fields?

While a library may not use all the data provided in the data format descriptor that is described within this standard, it is common for users of data — particularly pixel-like data — to have additional requirements. Capturing these requirements portably reduces the need for additional metadata to be associated with a proprietary descriptor. It is also common for additional functionality to be added retrospectively to existing libraries — for example, $Y'C_BC_R$ support is often an afterthought in rendering APIs. Having a consistent and flexible representation in place from the start can reduce the pain of retrofitting this functionality.

Note that there is no expectation that the format descriptor from this standard be used directly, although it can be. The impact of providing a mapping between internal formats and format descriptors is expected to be low, but offers the opportunity both for simplified access from software outside the proprietary library and for reducing the effort needed to provide a complete, unambiguous and accurate description of a format in human-readable terms.

9.6. Why not expand each field out to be integer for ease of decoding?

There is a trade-off between size and decoding effort. It is assumed that data which occupies the same 32-bit word may need to be tested concurrently, reducing the cost of comparisons. When transferring data formats, the packing reduces the overhead. Within these constraints, it is intended that most data can be extracted with low-cost operations, typically being byte-aligned (other than sample flags) and with the natural alignment applied to multi-byte quantities.

9.7. Can this descriptor be used for text content?

For simple ASCII content, there is no reason that plain text could not be described in some way, and this may be useful for image formats that contain comment sections. However, since many multilingual text representations do not have a fixed character size, this use is not seen as an obvious match for this standard.

10. Floating-point formats

Some common floating-point numeric representations are defined in [IEEE 754]. Additional floating point formats are defined in this section.

10.1. 16-bit floating-point numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value V of a 16-bit floating-point number is determined by the following:

[Math Processing Error]

If the floating-point number is interpreted as an unsigned 16-bit integer N, then

[Math Processing Error] [Math Processing Error] [Math Processing Error]

10.2. Unsigned 11-bit floating-point numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value V of an unsigned 11-bit floating-point number is determined by the following:

[Math Processing Error]

If the floating-point number is interpreted as an unsigned 11-bit integer *N*, then

[Math Processing Error] [Math Processing Error]

10.3. Unsigned 10-bit floating-point numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (*E*), and a 5-bit mantissa (*M*). The value *V* of an unsigned 10-bit floating-point number is determined by the following:

[Math Processing Error]

If the floating-point number is interpreted as an unsigned 10-bit integer *N*, then

[Math Processing Error] [Math Processing Error]

10.4. Non-standard floating point formats

Rather than attempting to enumerate every possible floating-point format variation in this specification, the data format descriptor can be used to describe the components of arbitrary floating-point data, as follows. Note that non-standard floating point formats do not use the **KHR_DF_SAMPLE_DATATYPE_FLOAT** bit.

An example of use of the 16-bit floating point format described in Section 10.1 but described in terms of a custom floating point format is provided in Table 103. Note that this is provided for example only, and this particular format would be better described using the standard 16-bit floating point format as documented in Table 104.

10.4.1. The mantissa

The mantissa of a custom floating point format should be represented as an integer ***channelType***. If the mantissa represents a signed quantity encoded in two's complement, the **KHR_DF_SAMPLE_DATATYPE_SIGNED** bit should be set. To encode a signed mantissa represented in sign-magnitude format, the main part of the mantissa should be represented as an unsigned integer quantity (with **KHR_DF_SAMPLE_DATATYPE_SIGNED** not set), and an additional one-bit sample *with* **KHR_DF_SAMPLE_DATATYPE_SIGNED** set should be used to identify the sign bit. By convention, a sign bit should be encoded in a later sample than the corresponding mantissa.

The ***sampleUpper*** and ***sampleLower*** values for the mantissa should be set to indicate the representation of 1.0 and 0.0 (for unsigned formats) or -1.0 (for signed formats) respectively when the exponent is in a 0 position after any bias has been corrected. If there is an implicit "1" bit, these values for the mantissa will exceed what can be represented in the number of available mantissa bits.

For example, the shared exponent formats shown in Table 98 does not have an implicit "1" bit, and therefore the ***sampleUpper*** values for the 9-bit mantissas are 256 — this being the mantissa value for 1.0 when the exponent is set to 0.

For the 16-bit signed floating point format described in Section 10.1, ***sampleUpper*** should be set to 1024, indicating the implicit "1" bit which is above the 10 bits representable in the mantissa. ***sampleLower*** should be 0 in this case, since the mantissa uses a sign-magnitude representation.

By convention, the ***sampleUpper*** and ***sampleLower*** values for a sign bit are 0 and -1 respectively.

10.4.2. The exponent

The **KHR_DF_SAMPLE_DATATYPE_EXPONENT** bit should be set in a sample which contains the exponent of a custom floating point format.

The ***sampleLower*** for the exponent should indicate the exponent bias. That is, the mantissa should be scaled by two raised to the power of the stored exponent minus this ***sampleLower*** value.

The ***sampleUpper*** for the exponent indicates the maximum legal exponent value. Values above this are used to encode infinities and not-a-number (*NaN*) values. ***sampleUpper*** can therefore be used to indicate whether or not the format supports these encodings.

10.4.3. Special values

Floating point values encoded with an exponent of 0 (before bias) and a mantissa of 0 are used to represent the value 0. An explicit sign bit can distinguish between +0 and -0.

Floating point values encoded with an exponent of 0 (before bias) and a non-zero mantissa are assumed to indicate a denormalized number, if the format has an implicit "1" bit. That is, when the exponent is 0, the "1" bit becomes explicit and the exponent is considered to be the negative sample bias minus one.

Floating point values encoded with an exponent larger than the exponent's ***sampleUpper*** value and with a mantissa of 0 are interpreted as representing +/- infinity, depending on the value of an explicit sign bit. Note that in some formats, no exponent above ***sampleUpper*** is possible — for example, Table 98.

Floating point values encoded with an exponent larger than the exponent's ***sampleUpper*** value and with a mantissa of non-0 are interpreted as representing not-a-number (*NaN*).

Note that these interpretations are compatible with the corresponding numerical representations in [IEEE 754].

10.4.4. Conversion formulae

Given an optional sign bit *S*, a mantissa value of *M* and an exponent value of *E*, a format with an implicit "1" bit can be converted from its representation to a real value as follows:

[Math Processing Error]

If there is no implicit "1" bit (that is, the ***sampleUpper*** value of the mantissa is representable in the number of bits assigned to the mantissa), the value can be converted to a real value as follows:

[Math Processing Error]

A descriptor block for a format without an implicit "1" (and with the added complication of having the same exponent bits shared across multiple channels, which is why an implicit "1" bit does not make sense) is shown in Table 98. In the case of this particular example, the above equations simplify to:

[Math Processing Error] [Math Processing Error] [Math Processing Error]

Where:

[Math Processing Error] [Math Processing Error]

Note that in general conversion from a real number *to* any representation may require rounding, truncation and special value management rules which are beyond the scope of a data format specification and may be documented in APIs which generate these formats.

11. Example format descriptors

Example data format descriptors for compressed formats can be found under the ***colorModel*** field in Section 5.6.

Table 89. 565 *RGB* packed 16-bit format as written to memory by a little-endian architecture

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 76																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 3) = 72																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 2							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				First sample: low five bits blue																							
0	0	0	0	`BLUE`				***bitLength:*** 4 (= "5")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 31																															
F	***S***	***E***	***L***	***channelType***				Second sample: middle six bits green																							
0	0	0	0	`GREEN`				***bitLength:*** 5 = ("6")								***bitOffset:*** 6															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 63																															
F	***S***	***E***	***L***	***channelType***				Third sample: top five bits red																							
0	0	0	0	`RED`				***bitLength:*** 4 (= "5")								***bitOffset:*** 11															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 31																															

Table 90. Four co-sited 8-bit sRGB channels, assuming premultiplied alpha

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 92																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 4) = 88																***versionNumber:*** 2															
flags: `PREMULTIPLIED`								***transferFunction:*** `SRGB`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 4							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information for the first sample																							
0	0	0	0	`RED`				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Sample information for the second sample																							
0	0	0	0	`GREEN`				***bitLength:*** 7 (= "8")								***bitOffset:*** 8															

samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 255							
F	***S***	***E***	***L***	***channelType***	Sample information for the third sample		
0	0	0	0	`BLUE`	***bitLength:*** 7 (= "8")	***bitOffset:*** 16	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 255							
F	***S***	***E***	***L***	***channelType***	Sample information for the fourth sample		
0	0	0	1	`ALPHA`	***bitLength:*** 7 (= "8")	***bitOffset:*** 24	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 255							

Table 91. A single 8-bit monochrome channel

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 1) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `ITU`								***colorPrimaries:*** `BT709`								***colorModel:*** `YUVSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 1							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
0	0	0	0	`Y`				***bitLength:*** 8								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															

Table 92. A single 1-bit monochrome channel, as an 8×1 texel block to allow byte-alignment

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 156																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 8) = 142																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `YUVSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								7 (= "8")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 1							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information for the first sample																							
0	0	0	0	`Y`				***bitLength:*** 0 (= "1")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 1																															
F	***S***	***E***	***L***	***channelType***				Sample information for the second sample																							
0	0	0	0	`Y`				***bitLength:*** 0 (= "1")								***bitOffset:*** 1															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x20							
sampleLower: 0																															
sampleUpper: 1																															
F	***S***	***E***	***L***	***channelType***				Sample information for the third sample																							
0	0	0	0	`Y`				***bitLength:*** 0 (= "1")								***bitOffset:*** 2															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x40							
sampleLower: 0																															
sampleUpper: 1																															
F	***S***	***E***	***L***	***channelType***				*Sample information for the fourth sample*																							
0	0	0	0	`Y`				***bitLength:*** 0 (= "1")								***bitOffset:*** 3															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x60							
sampleLower: 0																															
sampleUpper: 1																															
F	***S***	***E***	***L***	***channelType***				*Sample information for the fifth sample*																							
0	0	0	0	`Y`				***bitLength:*** 0 (= "1")								***bitOffset:*** 4															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							

0					0	0	0x80
sampleLower: 0							
sampleUpper: 1							
F	***S***	***E***	***L***	***channelType***	*Sample information for the sixth sample*		
0	0	0	0	**Y**	***bitLength:*** 0 (= "1")	***bitOffset:*** 5	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0xA0
sampleLower: 0							
sampleUpper: 1							
F	***S***	***E***	***L***	***channelType***	*Sample information for the seventh sample*		
0	0	0	0	**Y**	***bitLength:*** 0 (= "1")	***bitOffset:*** 6	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0xC0
sampleLower: 0							
sampleUpper: 1							
F	***S***	***E***	***L***	***channelType***	*Sample information for the eighth sample*		
0	0	0	0	**Y**	***bitLength:*** 0 (= "1")	***bitOffset:*** 7	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0xE0
sampleLower: 0							
sampleUpper: 1							

Table 93. 2×2 Bayer pattern: four 8-bit distributed sRGB channels, spread across two lines (as two planes)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 92																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 4) = 88																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `SRGB`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								1 (height = "2")								1 (width = "2")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 2								***bytesPlane0:*** 2							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information for the first sample																							
0	0	0	0	**RED**				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0x00 (*y* = "0")								0x00 (*x* = "0")							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Sample information for the second sample																							
0	0	0	0	**GREEN**				***bitLength:*** 7 (= "8")								***bitOffset:*** 8															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0x00 (*y* = "0")								0x80 (*x* = "1")							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Sample information for the third sample																							
0	0	0	0	**GREEN**				***bitLength:*** 7 (= "8")								***bitOffset:*** 16															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0x80 (*y* = "1")								0x00 (*x* = "0")							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Sample information for the fourth sample																							
0	0	0	0	**BLUE**				***bitLength:*** 7 (= "8")								***bitOffset:*** 24															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0x80 (*y* = "1")								0x80 (*x* = "1")							
sampleLower: 0																															
sampleUpper: 255																															

Table 94. Simple paletted format: 8-bit index to 240 entries of 4-bit *R*, *G*, *B* channels

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 92																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 4) = 88																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `SRGB`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 1							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information for the 8-bit index																							
0	0	0	0	0				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							

sampleLower: 0							
sampleUpper: 239							
F	***S***	***E***	***L***	***channelType***	Red channel of the palette entries		
0	0	0	0	`RED`	***bitLength:*** 3 (= "4")	***bitOffset:*** 8 (palette entry)	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0 (palette id)
sampleLower: 0							
sampleUpper: 15							
F	***S***	***E***	***L***	***channelType***	Green channel of the palette entries		
0	0	0	0	`GREEN`	***bitLength:*** 3 (= "4")	***bitOffset:*** 8 (palette entry)	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0 (palette id)
sampleLower: 0							
sampleUpper: 15							
F	***S***	***E***	***L***	***channelType***	Blue channel of the palette entries		
0	0	0	0	`BLUE`	***bitLength:*** 3 (= "4")	***bitOffset:*** 8 (palette entry)	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0 (palette id)
sampleLower: 0							
sampleUpper: 15							

Table 95. Paletted color look-up table format: three 8-bit indices into separate 256-entry 10-bit channels

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 124																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 6) = 120																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 4							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Red channel palette index																							
0	0	0	0	`RED`				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Green channel palette index																							
0	0	0	0	`GREEN`				***bitLength:*** 7 (= "8")								***bitOffset:*** 8															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Blue channel palette index																							
0	0	0	0	`BLUE`				***bitLength:*** 7 (= "8")								***bitOffset:*** 16															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	***S***	***E***	***L***	***channelType***				Red palette entry																							
0	0	0	0	`RED`				***bitLength:*** 9 (= "10")								***bitOffset:*** 32 (palette entry)															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0 (`RED` palette)							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Green palette entry																							
0	0	0	0	`RED`				***bitLength:*** 9 (= "10")								***bitOffset:*** 32 (palette entry)															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								1 (`GREEN` palette)							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Blue palette entry																							
0	0	0	0	`RED`				***bitLength:*** 9 (= "10")								***bitOffset:*** 32 (palette entry)															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								2 (`BLUE` palette)							
sampleLower: 0																															
sampleUpper: 1023																															

Table 96. $Y'C_BC_R$ 4:2:0: BT.709 reduced-range data, with C_B and C_R aligned to the midpoint of the Y' samples

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 124																															

descriptorType: 0						***vendorId:*** 0	
descriptorBlockSize: 24 + (16 × 6) = 120						***versionNumber:*** 2	
flags: `ALPHA_STRAIGHT`					***transferFunction:*** `ITU`	***colorPrimaries:*** `BT709`	***colorModel:*** `YUVSDA`
texelBlockDimension3					***texelBlockDimension2***	***texelBlockDimension1***	***texelBlockDimension0***
0					0	1 (height = "2")	1 (width = "2")
bytesPlane3: 1					***bytesPlane2:*** 1	***bytesPlane1:*** 2	***bytesPlane0:*** 2
bytesPlane7: 0					***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0
F	***S***	***E***	***L***	***channelType***	Sample information for the first *Y′* sample		
0	0	0	0	`Y`	***bitLength:*** 7 (= "8")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x00 (*y* = "0.0")	0x00 (*x* = "0.0")
sampleLower: 16							
sampleUpper: 235							
F	***S***	***E***	***L***	***channelType***	Sample information for the second *Y′* sample		
0	0	0	0	`Y`	***bitLength:*** 7 (= "8")	***bitOffset:*** 8	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x00 (*y* = "0.0")	0x80 (*x* = "1.0")
sampleLower: 16							
sampleUpper: 235							
F	***S***	***E***	***L***	***channelType***	Sample information for the third *Y′* sample		
0	0	0	0	`Y`	***bitLength:*** 7 (= "8")	***bitOffset:*** 16	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x80 (*y* = "1.0")	0x00 (*x* = "0.0")
sampleLower: 16							
sampleUpper: 235							
F	***S***	***E***	***L***	***channelType***	Sample information for the fourth *Y′* sample		
0	0	0	0	`Y`	***bitLength:*** 7 (= "8")	***bitOffset:*** 24	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x80 (*y* = "1.0")	0x80 (*x* = "1.0")
sampleLower: 16							
sampleUpper: 235							
F	***S***	***E***	***L***	***channelType***	Sample information for the *U* sample		
0	0	0	0	`U`	***bitLength:*** 7 (= "8")	***bitOffset:*** 32	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x40 (*y* = "0.5")	0x40 (*x* = "0.5")
sampleLower: 16							
sampleUpper: 240							
F	***S***	***E***	***L***	***channelType***	Sample information for the *V* sample		
0	0	0	0	`V`	***bitLength:*** 7 (= "8")	***bitOffset:*** 40	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0x40 (*y* = "0.5")	0x40 (*x* = "0.5")
sampleLower: 16							
sampleUpper: 240							

Table 97. 565 *RGB* packed 16-bit format as written to memory by a big-endian architecture

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 92																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 4) = 88																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`										***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`					
texelBlockDimension3										***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***					
0										0								0								0					
bytesPlane3: 0										***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 2					
bytesPlane7: 0										***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0					
F	***S***	***E***	***L***	***channelType***						Bit 0 is green, green bits 0..2 in bits 5..7 of the higher byte address (less-significant half of the big-endian word)																					
0	0	0	0	`GREEN`						***bitLength:*** 2 (= "3")								***bitOffset:*** 13													
samplePosition3										***samplePosition2***								***samplePosition1***								***samplePosition0***					
0										0								0								0					
sampleLower: 0 — lower 3 bits of 6-bit value 0																															
sampleUpper: 7 — lower 3 bits of 6-bit value 63																															
F	***S***	***E***	***L***	***channelType***						Continuation: green bits 3..5 in bits 0..2 of the lower byte address (more-significant half of the big-endian word)																					
0	0	0	0	`GREEN`						***bitLength:*** 2 (= "3")								***bitOffset:*** 0													
samplePosition3										***samplePosition2***								***samplePosition1***								***samplePosition0***					
0										0								0								0					
sampleLower: 0 — upper 3 bits of 6-bit value 0																															
sampleUpper: 7 — upper 3 bits of 6-bit value 63																															
F	***S***	***E***	***L***	***channelType***						Next undescribed bit: red in bits 3..7 of the lower byte address (more-significant half of the big-endian word)																					
0	0	0	0	`RED`						***bitLength:*** 4 (= "5")								***bitOffset:*** 3													
samplePosition3										***samplePosition2***								***samplePosition1***								***samplePosition0***					
0										0								0								0					
sampleLower: 0																															
sampleUpper: 31																															
F	***S***	***E***	***L***	***channelType***						Remaining bits: blue in bits 0..4 of the higher byte address (less-significant half of the big-endian word)																					
0	0	0	0	`BLUE`						***bitLength:*** 4 (= "5")								***bitOffset:*** 8													

samplePosition3	*samplePosition2*	*samplePosition1*	*samplePosition0*
0	0	0	0
sampleLower: 0			
sampleUpper: 31			

Table 98. R9G9B9E5 shared-exponent format

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 124																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 6) = 120																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								*texelBlockDimension2*								*texelBlockDimension1*								*texelBlockDimension0*							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 4							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	*S*	*E*	*L*	*channelType*				Sample information for the *R* mantissa																							
0	0	0	0	`RED`				***bitLength:*** 8 (= "9")								***bitOffset:*** 0															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 256 — mantissa at 1.0																															
F	*S*	*E*	*L*	*channelType*				Sample information for the *R* exponent																							
0	0	1	0	`RED`				***bitLength:*** 4 (= "5")								***bitOffset:*** 27															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 15 — exponent bias																															
F	*S*	*E*	*L*	*channelType*				Sample information for the *G* mantissa																							
0	0	0	0	`GREEN`				***bitLength:*** 8 (= "9")								***bitOffset:*** 9															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 256 — mantissa at 1.0																															
F	*S*	*E*	*L*	*channelType*				Sample information for the *G* exponent																							
0	0	1	0	`GREEN`				***bitLength:*** 4 (= "5")								***bitOffset:*** 27															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 15 — exponent bias																															
F	*S*	*E*	*L*	*channelType*				Sample information for the *B* mantissa																							
0	0	0	0	`BLUE`				***bitLength:*** 8 (= "9")								***bitOffset:*** 18															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 256 — mantissa at 1.0																															
F	*S*	*E*	*L*	*channelType*				Sample information for the *B* exponent																							
0	0	1	0	`BLUE`				***bitLength:*** 4 (= "5")								***bitOffset:*** 27															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 15 — exponent bias																															

Table 99. Acorn 256-color format (2 bits each independent *RGB*, 2 bits shared tint)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 108																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 5) = 104																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								*texelBlockDimension2*								*texelBlockDimension1*								*texelBlockDimension0*							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 1							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	*S*	*E*	*L*	*channelType*				*First sample: four bits of red starting at bit 0, including two shared bits 0..1*																							
0	0	0	0	`RED`				***bitLength:*** 3 (= "4")								***bitOffset:*** 0															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 15																															
F	*S*	*E*	*L*	*channelType*				Second sample: two bits of green "tint" shared with red																							
0	0	0	0	`GREEN`				***bitLength:*** 1 (= "2")								***bitOffset:*** 0															
samplePosition3								*samplePosition2*								*samplePosition1*								*samplePosition0*							

0					0	0	0
sampleLower: 0							
sampleUpper: 3							
F	***S***	***E***	***L***	***channelType***	Third sample: two bits unique to green		
0	0	0	0	`GREEN`	***bitLength:*** 1 (= "2")	***bitOffset:*** 4	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 3							
F	***S***	***E***	***L***	***channelType***	Fourth sample: two bits of blue "tint" shared with red		
0	0	0	0	`BLUE`	***bitLength:*** 1 (= "2")	***bitOffset:*** 0	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 3							
F	***S***	***E***	***L***	***channelType***	Fifth sample: two bits unique to blue		
0	0	0	0	`BLUE`	***bitLength:*** 1 (= "2")	***bitOffset:*** 6	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0
sampleLower: 0							
sampleUpper: 3							

Table 100. V210 format (full-range $Y'C_BC_R$)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 220																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 12) = 216																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `ITU`								***colorPrimaries:*** `BT709`								***colorModel:*** `YUVSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								5 (width = "6")							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 16							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				First sample: shared mid-sited *U0/U1*																							
0	0	0	0	`U`				***bitLength:*** 10								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x10 (*x* = "0.5")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Second sample: *Y'0*																							
0	0	0	0	`Y`				***bitLength:*** 9 (= "10")								***bitOffset:*** 10															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x00 (*x* = "0.0")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Third sample: shared mid-sited *V0/V1*																							
0	0	0	0	`V`				***bitLength:*** 9 (= "10")								***bitOffset:*** 20															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x10 (*x* = "0.5")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Fourth sample: *Y'1*																							
0	0	0	0	`Y`				***bitLength:*** 9 (= "10")								***bitOffset:*** 32															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x20 (*x* = "1.0")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Fifth sample: shared mid-sited *U2/U3*																							
0	0	0	0	`U`				***bitLength:*** 9 (= "10")								***bitOffset:*** 42															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x50 (*x* = "2.5")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Sixth sample: *Y'2*																							
0	0	0	0	`Y`				***bitLength:*** 9 (= "10")								***bitOffset:*** 52															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x40 (*x* = "2.0")							
sampleLower: 0																															
sampleUpper: 1023																															
F	***S***	***E***	***L***	***channelType***				Seventh sample: shared mid-sited *V2/V3*																							
0	0	0	0	`V`				***bitLength:*** 9 (= "10")								***bitOffset:*** 64															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0x50 (*x* = "2.5")							
sampleLower: 0																															
sampleUpper: 1023																															

F	S	E	L	channelType	Eighth sample: *Y'3*		
0	0	0	0	Y	***bitLength:*** 9 (= "10")	***bitOffset:*** 74	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0x60 (*x* = "3.0")
sampleLower: 0							
sampleUpper: 1023							
F	S	E	L	channelType	Ninth sample: shared mid-sited *U4/U5*		
0	0	0	0	U	***bitLength:*** 9 (= "10")	***bitOffset:*** 84	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0x90 (*x* = "4.5")
sampleLower: 0							
sampleUpper: 1023							
F	S	E	L	channelType	Tenth sample: *Y'4*		
0	0	0	0	Y	***bitLength:*** 9 (= "10")	***bitOffset:*** 96	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0x80 (*x* = "4.0")
sampleLower: 0							
sampleUpper: 1023							
F	S	E	L	channelType	Eleventh sample: shared mid-sited *V4/V5*		
0	0	0	0	V	***bitLength:*** 9 (= "10")	***bitOffset:*** 106	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0x90 (*x* = "4.5")
sampleLower: 0							
sampleUpper: 1023							
F	S	E	L	channelType	Twelfth sample: *Y'5*		
0	0	0	0	Y	***bitLength:*** 9 (= "10")	***bitOffset:*** 116	
samplePosition3					***samplePosition2***	***samplePosition1***	***samplePosition0***
0					0	0	0xA0 (*x* = "5.0")
sampleLower: 0							
sampleUpper: 1023							

Table 101. Intensity-alpha format showing aliased samples

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 92																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 4) = 88																***versionNumber:*** 2															
flags: ALPHA_STRAIGHT								***transferFunction:*** LINEAR								***colorPrimaries:*** BT709								***colorModel:*** RGBSDA							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 1							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	S	E	L	channelType				First sample: red																							
0	0	0	0	RED				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	S	E	L	channelType				Second sample: green																							
0	0	0	0	GREEN				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	S	E	L	channelType				Third sample: blue																							
0	0	0	0	BLUE				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															
F	S	E	L	channelType				Fourth sample: alpha																							
0	0	0	0	ALPHA				***bitLength:*** 7 (= "8")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 255																															

Table 102. A 48-bit signed middle-endian red channel: three co-sited 16-bit little-endian words, high word first

uint32_t bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 76																															
descriptorType: 0															***vendorId:*** 0																
descriptorBlockSize: 24 + (16 × 3) = 72																***versionNumber:*** 2															

flags: `ALPHA_STRAIGHT`				***transferFunction:*** `LINEAR`	***colorPrimaries:*** `BT709`	***colorModel:*** `RGBSDA`	
texelBlockDimension3				***texelBlockDimension2***	***texelBlockDimension1***	***texelBlockDimension0***	
0				0	0	0	
bytesPlane3: 0				***bytesPlane2:*** 0	***bytesPlane1:*** 0	***bytesPlane0:*** 6	
bytesPlane7: 0				***bytesPlane6:*** 0	***bytesPlane5:*** 0	***bytesPlane4:*** 0	
F	***S***	***E***	***L***	***channelType***	First sample: bits 0..15 of the 48-bit value, in memory bytes 4..5		
0	1	0	0	`RED`	***bitLength:*** 15 (= "16")	***bitOffset:*** 32	
samplePosition3				***samplePosition2***	***samplePosition1***	***samplePosition0***	
0				0	0	0	
sampleLower: 0xFFFF0000U — bottom 16 bits of 0x800000000000, sign-extended							
sampleUpper: 0x0000FFFFU — bottom 16 bits of 0x7FFFFFFFFFFF							
F	***S***	***E***	***L***	***channelType***	Second sample: bits 16..31 of the 48-bit value, in memory bytes 2..3		
0	1	0	0	`RED`	***bitLength:*** 15 (= "16")	***bitOffset:*** 16	
samplePosition3				***samplePosition2***	***samplePosition1***	***samplePosition0***	
0				0	0	0	
sampleLower: 0xFFFF0000U — middle 16 bits of 0x800000000000, sign-extended							
sampleUpper: 0x0000FFFFU — middle 16 bits of 0x7FFFFFFFFFFF							
F	***S***	***E***	***L***	***channelType***	Third sample: bits 32..47 of the 48-bit value, in memory bytes 0..1		
0	1	0	0	`RED`	***bitLength:*** 15 (= "16")	***bitOffset:*** 0	
samplePosition3				***samplePosition2***	***samplePosition1***	***samplePosition0***	
0				0	0	0	
sampleLower: 0xFFFF8000U — top 16 bits of 0x800000000000, sign-extended							
sampleUpper: 0x00007FFFU — top 16 bits of 0x7FFFFFFFFFFF							

Table 103. A single 16-bit floating-point red value, described explicitly (example only!)

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 76																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 3) = 72																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 2							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				First sample: mantissa																							
0	0	0	0	`RED`				***bitLength:*** 9 (= "10")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0																															
sampleUpper: 1024 — implicit 1.0 bit since 1024 > $2^{\textbf{bitLength}} - 1$																															
F	***S***	***E***	***L***	***channelType***				Second sample: sign of mantissa																							
0	1	0	0	`RED`				***bitLength:*** 0 (= "1")								***bitOffset:*** 15															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 1 — 1-bit "-1"																															
sampleUpper: 0 — 1-bit "0"																															
F	***S***	***E***	***L***	***channelType***				Third sample: exponent																							
0	0	1	0	`RED`				***bitLength:*** 4 (= "5")								***bitOffset:*** 10															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 15 — bias																															
sampleUpper: 30 — support for infinities because 30 < $2^{\textbf{bitLength}} - 1$																															

Table 104. A single 16-bit floating-point red value, described normally

`uint32_t` bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
totalSize: 44																															
descriptorType: 0																***vendorId:*** 0															
descriptorBlockSize: 24 + (16 × 3) = 40																***versionNumber:*** 2															
flags: `ALPHA_STRAIGHT`								***transferFunction:*** `LINEAR`								***colorPrimaries:*** `BT709`								***colorModel:*** `RGBSDA`							
texelBlockDimension3								***texelBlockDimension2***								***texelBlockDimension1***								***texelBlockDimension0***							
0								0								0								0							
bytesPlane3: 0								***bytesPlane2:*** 0								***bytesPlane1:*** 0								***bytesPlane0:*** 2							
bytesPlane7: 0								***bytesPlane6:*** 0								***bytesPlane5:*** 0								***bytesPlane4:*** 0							
F	***S***	***E***	***L***	***channelType***				Sample information																							
1	1	0	0	`RED`				***bitLength:*** 15 (= "16")								***bitOffset:*** 0															
samplePosition3								***samplePosition2***								***samplePosition1***								***samplePosition0***							
0								0								0								0							
sampleLower: 0xBF800000U — IEEE 754 floating-point representation for -1.0f																															
sampleUpper: 0x3F800000U — IEEE 754 floating-point representation for 1.0f																															

12. Introduction to color conversions

12.1. Color space composition

A "color space" determines the meaning of decoded numerical color values: that is, it is distinct from the bit patterns, compression schemes and locations in memory used to store the data.

A color space consists of three basic components:

- Transfer functions define the relationships between linear intensity and linear numbers in the encoding scheme. Since the human eye's sensitivity to changes in intensity is non-linear, a non-linear encoding scheme typically allows improved visual quality at reduced storage cost.
  - An opto-electrical transfer function (OETF) describes the conversion from "scene-referred" normalized linear light intensity to a (typically) non-linear electronic representation. The inverse function is written "OETF $^{-1}$".
  - An electro-optical transfer function (EOTF) describes the conversion from the electronic representation to "display-referred" normalized linear light intensity in the display system. The inverse function is written "EOTF $^{-1}$".
  - An opto-optical transfer function (OOTF) describes the relationship between the linear scene light intensity and linear display light intensity: OOTF(x) = EOTF(OETF(x)). OETF = EOTF $^{-1}$ and EOTF = OETF $^{-1}$ only if the OOTF is linear.
  - Historically, a non-linear transfer function has been implicit due to the non-linear relationship between voltage and intensity provided by a CRT display. In contrast, many computer graphics applications are best performed in a representation with a linear relationship to intensity.
  - Use of an incorrect transfer function can result in images which have too much or too little contrast or saturation, particularly in mid-tones.
- Color primaries define the spectral response of a "pure color" in an additive color model - typically, what is meant by "red", "green" and "blue" for a given system, and (allowing for the relative intensity of the primaries) consequently define the system's white balance.
  - These primary colors might refer to the wavelengths emitted by phosphors on a CRT, transmitted by filters on an LCD for a given back-light, or emitted by the LED sub-pixels of an OLED. The primaries are typically defined in terms of a reference display, and represent the most saturated colors the display can produce, since other colors are by definition created by combining the primaries. The definition usually describes a relationship to the responses of the human visual system rather than a full spectrum.
  - Use of incorrect primaries introduces a shift of hue, most visible in saturated colors.
- Color models describe the distinction between a color representation and additive colors. Since the human visual system treats differences in absolute intensity differently from differences in the spectrum composing a color, many formats benefit from transforming the color representation into one which can separate these aspects of color. Color models are frequently "named" by listing their component color channels.
  - For example, a color model might directly represent additive primaries (*RGB*), simple color difference values ($Y'C_BC_R$—colloquially *YUV*), or separate hue, saturation and intensity (*HSV*/*HSL*).
  - Interpreting an image with an incorrect color model typically results in wildly incorrect colors: a (0,0,0) triple in an *RGB* additive color model typically represents black, but may represent white in *CMYK*, or saturated green in color difference models.

12.2. Operations in a color conversion

Conversion between color representations may require a number of separate conversion operations:

- Conversion between representations with different color primaries can be performed directly. If the input and output of the conversion do not share the same color primaries, this transformation forms the "core" of the conversion.
- The color primary conversion operates on linear *RGB* additive color values; if the input or output are not defined in linear terms but with a non-linear transfer function, any color primary conversion must be "wrapped" with any transfer functions; conventionally, non-linear *RGB* values are written *R'G'B'*.
- If the input or output color model is not defined in terms of additive primaries (for example, $Y'C_BC_R$—colloquially known as *YUV*), the model conversion is applied to the non-linear *R'G'B'* values; the $Y'_CC'_{CB}C'_{CR}$ and IC_TC_P color models are created from both linear and non-linear *RGB*.
- Converting numerical values stored in memory to the representation of the color model may itself require additional operations - in order to remove dependence on bit depth, all the formulae described here work with continuous natural numbers, but some common in-memory quantization schemes must often be applied.

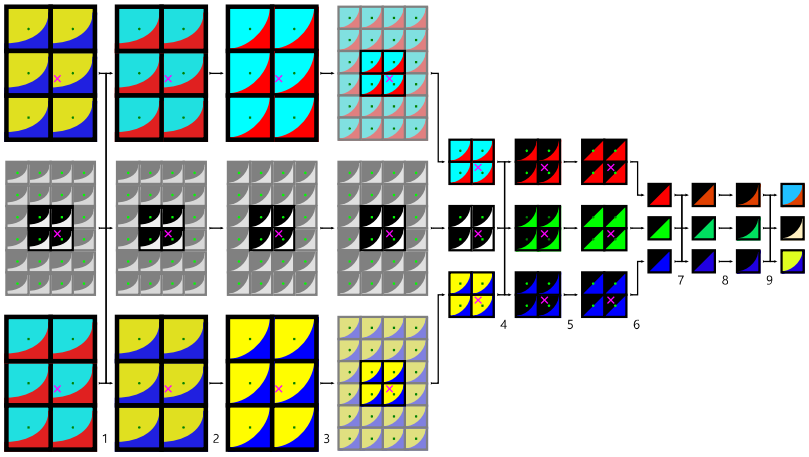
Details of these conversion operations are described in the following chapters.

> As described in the License Information at the start of this document, the Khronos Data Format Specification does not convey a right to implement the operations it describes. This is particularly true of the conversion formulae in the following sections, whose inclusion is purely informative. Please refer to the originating documents and the bodies responsible for the standards containing these formulae for the legal framework required for implementation.

Common cases such as converting a $Y'C_BC_R$ image encoded for 625-line BT.601 to a $Y'C_BC_R$ image encoded for BT.709 can involve multiple costly operations. An example is shown in the following diagram, which represents sampling from a $Y'C_BC_R$ texture in one color

space, and the operations needed to generate a different set of $Y'C_BC_R$ values representing the color of the sample position in a different color space:

Figure 22. Example sampling in one space and converting to a different space



In this diagram, non-linear luma Y' channels are shown in black and white, color difference C_B/C_R channels are shown with the colors at the extremes of their range, and color primary channels are shown as the primary color and black. Linear representations are shown diagonally divided by a straight line; non-linear representations are shown with a gamma curve. The luma and color difference representation is discussed in Section 15.1. The interpretation of color primaries is discussed in Section 14. Non-linear transfer functions are described in Section 13. As described below, the diagram shows a 2×3 grid of input chroma texel values, corresponding to a 4×6 grid of luma texel values, since the chroma channels are stored at half the horizontal and half the vertical resolution of the luma channel (i.e. in "4:2:0" representation). Grayed-out texel values do not contribute to the final output, and are shown only to indicate relative alignment of the coordinates.

The stages numbered in Figure 22 show the following operations:

1. Arranging the channels from the representation correctly for the conversion operations (a "swizzle"). In this example, the implementation requires that the C_B and C_R values be swapped.

2. Range expansion to the correct range for the values in the color model (handled differently, for example, for "full" and "narrow" ranges); in this example, the result is to increase the effective dynamic range of the encoding: contrast and saturation are increased.

   In this example, operations 1 and 2 can be combined into a single sparse matrix multiplication of the input channels, although actual implementations may wish to take advantage of the sparseness.

3. Reconstruction to full resolution of channels which are not at the full sampling resolution ("chroma reconstruction"), for example by replication or interpolation at the sites of the luma samples, allowing for the chroma sample positions; this example assumes that the chroma samples are being reconstructed through linear interpolation. In the diagram, sample positions for each channel are shown as green dots, and each channel corresponds to the same region of the image; in this example, the chroma samples are located at the horizontal and vertical midpoint of quads of luma samples, but different standards align the chroma samples differently. Note that interpolation for channel reconstruction necessarily happens in a non-linear representation for color difference representations such as $Y'C_BC_R$: creating a linear representation would require converting to *RGB*, which in turn requires a full set of $Y'C_BC_R$ samples for a given location.

4. Conversion between color models — in this example, from non-linear $Y'C_BC_R$ to non-linear $R'G'B'$. For example, the conversion might be that between BT.601 $Y'C_BC_R$ and BT.601 non-linear $R'G'B'$ described in Section 15.1.2. For $Y'C_BC_R$ to $R'G'B'$, this conversion is a sparse matrix multiplication.

5. Application of a transfer function to convert from non-linear $R'G'B'$ to linear *RGB*, using the color primaries of the input representation. In this case, the conversion might be the EOTF $^{-1}$ described in Section 13.2.

   The separation of stages 4 and 5 is specific to the $Y'C_BC_R$ to $R'G'B'$ color model conversion. Other representations such as $Y'_CC'_{BC}C'_{RC}$ and IC_TC_P have more complex interactions between the color model conversion and the transfer function.

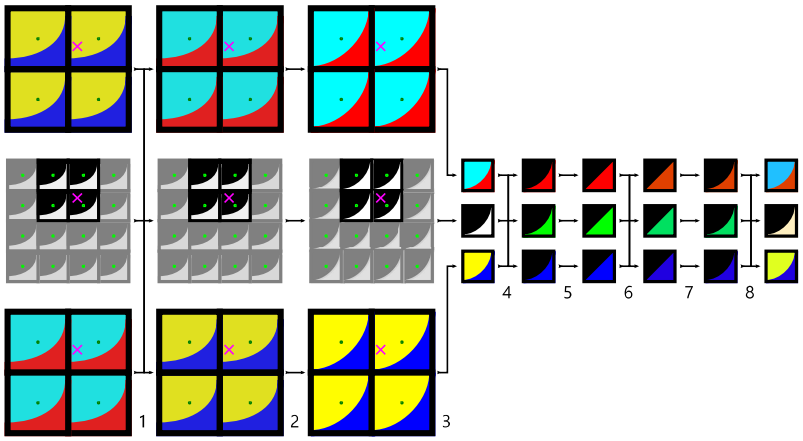
6. Interpolation of linear color values at the sampling position shown with a magenta cross according to the chosen sampling rules.
7. Convert from the color primaries of the input representation to the desired color primaries of the output representation, which is a matrix multiplication operation. Conversion from linear BT.601 EBU primaries to BT.709 primaries, as described in Section 14.2 and Section 14.1.
8. Convert from the linear *RGB* representation using the target primaries to a non-linear $R'G'B'$ representation, for example the OETF described in Section 13.2.
9. Conversion from non-linear $R'G'B'$ to the $Y'C_BC_R$ color model, for example as defined in as defined in Section 15.1.1 (a matrix multiplication).

If the output is to be written to a frame buffer with reduced-resolution chroma channels, chroma values for multiple samples need to be combined. Note that it is easy to introduce inadvertent chroma blurring in this operation if the source space chroma values are generated by interpolation.

In this example, generating the four linear *RGB* values required for linear interpolation at the magenta cross position requires *six* chroma samples. In the example shown, all four *Y′* values fall between the same two chroma sample centers on the horizontal axis, and therefore recreation of these samples by linear blending on the horizontal axis only requires two horizontally-adjacent samples. However, the upper pair of *Y′* values are sited above the sample position of the middle row of chroma sample centers, and therefore reconstruction of the corresponding chroma values requires interpolation between the upper four source chroma values. The lower pair of *Y′* values are sited below the sample position of the middle row of chroma sample centers, and therefore reconstruction of the corresponding chroma values requires interpolation between the lower four source chroma values. In general, reconstructing four chroma values by interpolation may require four, six or nine source chroma values, depending on which samples are required. The worst case is reduced if chroma samples are aligned ("co-sited") with the luma values, or if chroma channel reconstruction uses replication (nearest-neighbor filtering) rather than interpolation.

An approximation to the conversion described in Figure 22 is depicted in Figure 23:

Figure 23. Example approximated sampling in one space and converting to a different space



A performance-optimized approximation to our example conversion may use the following steps:

1. Channel rearrangement (as in the previous example)
2. Range expansion (as in the previous example)
3. Chroma reconstruction combined with sampling. In this case, the desired chroma reconstruction operation is approximated by adjusting the sample locations to compensate for the reduced resolution and sample positions of the chroma channels, resulting in a single set of non-linear $Y'C_BC_R$ values.
4. Model conversion from $Y'C_BC_R$ to $R'G'B'$ as described in Section 15.1.2, here performed *after* the sampling/filtering operation.
5. Conversion from non-linear $R'G'B'$ to linear *RGB*, using the EOTF $^{-1}$ described in Section 13.2.
6. Conversion of color primaries, corresponding to step 7 of the previous example.
7. Conversion to a non-linear representation, corresponding to step 8 of the previous example.
8. Conversion to the output color model, corresponding to step 9 of the previous example.

Since stages 1 and 2 represent an affine matrix transform, linear interpolation of input values may equivalently be performed before these operations. This observation allows stages 1..4 to be combined into a single matrix transformation.

Large areas of constant color will be correctly converted by this approximation. However, there are two sources of errors near color boundaries:

1. Interpolation takes place on values with a non-linear representation; the repercussions of this are discussed in Section 13, but can introduce both intensity and color shifts. Note that applying a non-linear transfer function as part of filtering does not improve accuracy for color models other than $R'G'B'$ since the non-linear additive values have been transformed as part of the color model representation.
2. When chroma reconstruction is bilinear and the final sample operation is bilinear, the interpolation operation now only access a maximum of four chroma samples, rather than up to nine for the precise series of operations. This has the potential to introduce a degree of aliasing in the output.

This approximation produces identical results to the more explicit sequence of operations in two cases:

1. If chroma reconstruction uses nearest-neighbor replication and the sampling operation is also a nearest-neighbor operation rather than a linear interpolation.
2. If the sampling operation is a nearest-neighbor operation and chroma reconstruction uses linear interpolation, *if* the sample coordinate position is adjusted to the nearest luma sample location.

As another example, the conversion from BT.709-encoded $Y'C_BC_R$ to sRGB $R'G'B'$ may be considered to be a simple model conversion (to BT.709 $R'G'B'$ non-linear primaries using the "ITU" OETF), since sRGB shares the BT.709 color primaries and is defined as a complementary EOTF intended to be combined with BT.709's OETF. This interpretation imposes a *[Math Processing Error]* 1.1 OOTF. Matching the OOTF of a BT.709-BT.1886 system, for which *[Math Processing Error]* 1.2, implies using the BT.1886 EOTF to convert to

linear light, then the sRGB EOTF $^{-1}$ to convert back to sRGB non-linear space. Encoding linear scene light with linear OOTF means applying the BT.709 OETF $^{-1}$; if the sRGB $R'G'B'$ target is itself intended to represent a linear OOTF, then the $\{R'_{sRGB}, G'_{sRGB}, B'_{sRGB}\}$ should be calculated as:

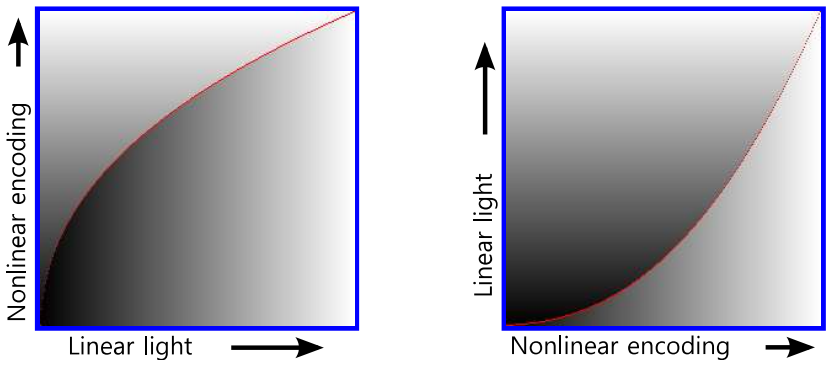
[Math Processing Error]

13. Transfer functions

13.1. About transfer functions (informative)

The **transfer function** describes the mapping between a linear numerical representation and a non-linear encoding. The eye is more sensitive to relative light levels than absolute light levels. That is, if one image region is twice as bright as another, this will be more visible than if one region is 10% brighter than another, even if the absolute difference in brightness is the same in both cases. To make use of the eye's non-linear response to light to provide better image quality with a limited number of quantization steps, it is common for color encodings to work with a non-linear representation which dedicates a disproportionate number of bits to darker colors compared with lighter colors. The typical effect of this encoding is that mid-tones are stored with a larger (nearer-to-white) numerical value than their actual brightness would suggest, and that mid-values in the non-linear encoding typically represent darker intensities than their fraction of the representation of white would suggest.

Figure 24. Conversion curves between linear light and encoded values (sRGB example)



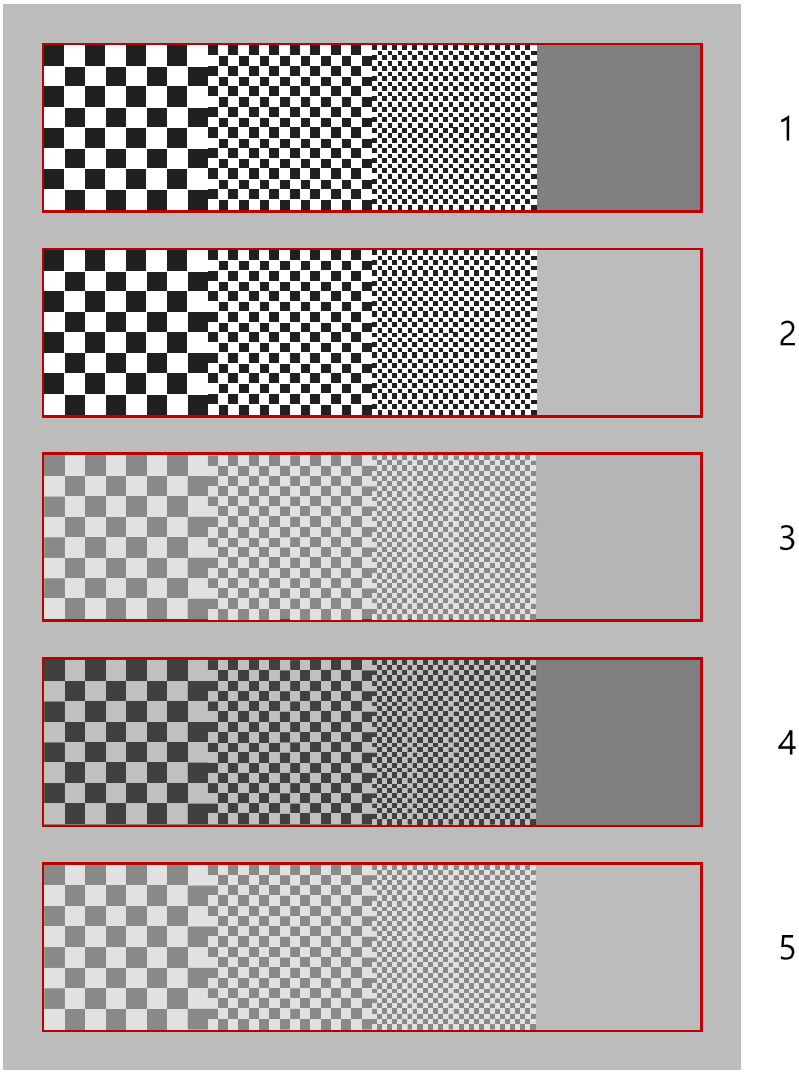
The behavior has historically been approximated by a power function with an exponent conventionally called *[Math Processing Error]* : $\{R,G,B\}_{\text{non-linear}} = \{R,G,B\}_{\text{linear}}$ *[Math Processing Error]* . Hence this conversion is colloquially known as *gamma correction*.

> Many practical transfer functions incorporate a small linear segment near 0, instead of being a pure power function. This linearity reduces the required resolution for representing the conversion, especially where results must be reversible, and also reduces the noise sensitivity of the function in an analog context. When combined with a linear segment, the power function has a different exponent from the pure power function that best approximates the resulting curve.

A consequence of this non-linear encoding is that many image processing operations should not be applied directly to the raw non-linearly-encoded numbers, but require conversion back to a linear representation. For example, linear color gradients will appear distorted unless the encoding is adjusted to compensate for the encoding; CGI lighting calculations need linear intensity values for operation, and filtering operations require texel intensities converted to linear form.

In the following example, the checker patterns are filtered in the right-most square of each row by averaging the checker colors, emulating a view of the pattern from a distance at which the individual squares are no longer distinct. The intended effect can be seen by viewing the diagram from a distance, or deliberately out of focus. The output is interpreted using the sRGB EOTF, approximating the behavior of a CRT with uncorrected signals. The background represents 50% gray.

Figure 25. Averaging checker values with different transfer functions



- **In row 1** black (0.0) and white (1.0) texels are averaged to calculate a 0.5 value in the frame buffer. Due to the sRGB non-linearity, the appearance of the value 0.5 is darker than the average value of the black and white texels, so the gray box appears darker than the average intensity of the checker pattern.
- **In row 2** the 0.5 average of the frame buffer is corrected using the sRGB (electro-optical) EOTF $^{-1}$ to ~0.74. The gray box accordingly appears a good match for the average intensity of the black and white squares on most media.
- **In row 3** the checker pattern instead represents values of 25% and 75% of the light intensity (the average of which should be the same as the correct average of the black and white squares in the first two rows). These checker values have been converted to their non-linear representations, as might be the case for a texture in this format: the darker squares are represented by ~0.54, and the lighter squares are represented by ~0.88. Averaging these two values to get a value of 0.71 results in the right-most square: this appears slightly too dark compared with the correct representation of mid-gray (~0.74) because, due to the non-linear encoding, the calculated value should not lie exactly half way between the two end points. Since the end points of the interpolation are less distant than the black and white case, the error is smaller than in the first example, and can more clearly be seen by comparing with the background gray.
- **In row 4** the checker values have been converted using the EOTF to a linear representation which can be correctly interpolated, but the resulting output represents linear light, which is therefore interpreted as too dark by the non-linear display.
- **In row 5** the results of row 4 have been converted back to the non-linear representation using the EOTF $^{-1}$. The checker colors are restored to their correct values, and the interpolated value is now the correct average intensity of the two colors.

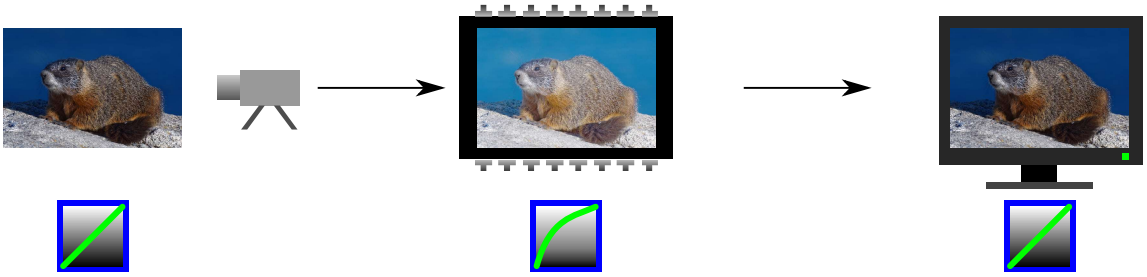
Incorrectly-applied transfer functions can also introduce color shifts, as demonstrated by the saturation change in the following examples:

Figure 26. *R, G, B* channels and combined color gradient with linear light intensity in each channel

Figure 27. *R′*, *G′*, *B′* channels and combined color gradient with non-linear sRGB encoding in each channel

A standard for image representation typically defines one or both of two types of transfer functions:

Figure 28. Opto-electronics and electro-optical transfer functions



- An opto-electronic transfer function (OETF) defines the conversion between a normalized linear light intensity as recorded in the scene, and a non-linear electronic representation. Note that there is no requirement that this directly correspond to actual captured light: the content creator or scene capture hardware may adjust the apparent intensity compared to reality for aesthetic reasons, as though the colors (or lighting) of objects in the scene were similarly different from reality. For example, a camera may implement a roll-off function for highlights, and a content creator may introduce tone mapping to preserve shadow detail in the created content, with these being logically recorded as if the scene was actually modified accordingly. The inverse of the OETF (the conversion from the non-linear electronic representation to linear scene light) is written $\text{OETF}^{-1}(n)$.
- An electro-optical transfer function (EOTF) converts between a non-linear electronic representation and a linear light normalized intensity as produced by the output display. The inverse of the EOTF (the conversion from linear display light to the non-linear electronic representation) is written $\text{EOTF}^{-1}(n)$. Typical CRT technology has implicitly applied a non-linear EOTF which coincidentally offered an approximately perceptually linear gradient when supplied with a linear voltage; other display technologies must implement the EOTF explicitly. As with the OETF, the EOTF describes a logical relationship that in reality will be modified by the viewer's aesthetic configuration choices and environment, and by the implementation choices of the display medium. Modern displays often incorporate proprietary variations from the reference intensity, particularly when mapping high dynamic range content to the capabilities of the hardware.

Some color models derive chroma (color difference) channels wholly or partly from non-linear intensities. It is common for image representations which use these color models to use a reduced-resolution representation of the chroma channels, since the human eye is less sensitive to high resolution chroma errors than to errors in brightness. Despite the color shift introduced by interpolating non-linear values, these chroma channels are typically resampled directly in their native, non-linear representation.

The opto-optical transfer function (OOTF) of a system is the relationship between linear input light intensity and displayed output light intensity: OOTF(*input*) = EOTF(OETF(*input*)). It is common for the OOTF to be non-linear. For example, a brightly-lit scene rendered on a display that is viewed in a dimly-lit room will typically appear washed-out and lacking contrast, despite mapping the full range of scene brightness to the full range supported by the display. A non-linear OOTF can compensate for this by reducing the intensity of mid-

tones, which is why television standards typically assume a non-linear OOTF: logical scene light intensity is not proportional to logical display intensity.

In the following diagram, the upper pair of images are identical, as are the lower pair of images (which have mid-tones darkened but the same maximum brightness). Adaptation to the surround means that the top left and lower right images look similar.

Figure 29. Simultaneous contrast

In the context of a non-linear OOTF, an application should be aware of whether operations on the image are intended to reflect the representation of colors in the scene or whether the intent is to represent the output color accurately, at least when it comes to the transfer function applied. For example, an application could choose to convert lighting calculations from a linear to non-linear representation using the OETF (to match the appearance of lighting in the scene), but to perform image scaling operations using the EOTF in order to avoid introducing intensity shifts due to filtering. Working solely with the EOTF or OETF results in ignoring the intended OOTF of the system.

In practice, the OOTF is usually near enough to linear that this distinction is subtle and rarely worth making for computer graphics, especially since computer-generated images may be designed to be viewed in brighter conditions which would merit a linear OOTF, and since a lot of graphical content is inherently not photo-realistic (or of limited realism, so that the transfer functions are not the most important factor in suspending disbelief). For video and photographic content viewed in darker conditions, the non-linearity of the OOTF is significant. The effect of a non-linear OOTF is usually secondary to the benefits of using non-linear encoding to reduce quantization.

By convention, non-linearly-encoded values are distinguished from linearly-encoded values by the addition of a prime (') symbol. For example, (R,G,B) may represent a linear set of red, green and blue components; (R',G',B') would represent the non-linear encoding of each value. Typically the non-linear encoding is applied to additive primary colors; derived color differences may or may not retain the prime symbol.

Charles Poynton provides a further discussion on "Gamma" in http://www.poynton.com/PDFs/TIDV/Gamma.pdf.

13.2. ITU transfer functions

"ITU" is used in this context as a shorthand for the OETF shared by the current BT.601, BT.709 and BT.2020 family of standard dynamic range digital television production standards. The same OETF is shared by SMPTE 170M. The ITU does define other transfer functions, for example the PQ and HLG transfer functions described below (originating in BT.2100) and the list of EOTFs listed in BT.470-6.

13.2.1. ITU OETF

The ITU-T BT.601, BT.709 and BT.2020 specifications (for standard definition television, HDTV and UHDTV respectively), and SMPTE 170M, which defines NTSC broadcasts, define an opto-electrical transfer function. The (OETF) conversion from linear (R,G,B) encoding to non-linear (R',G',B') encoding is:

[Math Processing Error]

Where α = 1.0993 and β = 0.0181 for 12-bit encoding in the BT.2020 specification, and *[Math Processing Error]* and *[Math Processing Error]* otherwise.

13.2.2. ITU OETF $^{-1}$

From this the inverse (OETF $^{-1}$) transformation can be deduced:

[Math Processing Error]

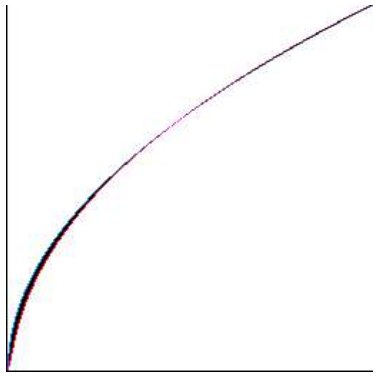
δ can be deduced from $\alpha \times \beta^{0.45} - (\alpha - 1) \approx 0.0812$. Note that this is subtly different from $4.5 \times \beta$ due to rounding. See the following section for the derivation of these values.

SMPTE 170M-2004, which defines the behavior of NTSC televisions, defines the EOTF of the "reference reproducer" as the OETF $^{-1}$ function above, with δ explicitly written as 0.0812. Therefore the SMPTE 170M-2004 EOTF $^{-1}$ equals the OETF given above. The "reference camera" of SMTPE 170M-2004 has an OETF function matching that of the ITU specifications. That is, the OOTF of the system described in SMPTE 170M-2004 provides a linear mapping of captured scene intensity to display intensity: the SMPTE 170M-2004 OETF is described as being chosen to result in a linear OOTF on a typical display. This is distinct from the current ITU specifications, which assume a non-linear OOTF. SMPTE 170M-2004 also represents a change from the "assumed gamma" of 2.2 associated with most NTSC display devices as described in ITU-T BT.470-6 and BT.2043, although these standards also define a linear OOTF.

This "ITU" OETF is closely approximated by a simple power function with an exponent of 0.5 (and therefore the OETF $^{-1}$ is quite closely approximated by a simple power function with an exponent of 2.0); the linear segment and offset mean that the best match is *not* the exponent of 0.45 that forms part of the exact equation. ITU standards deliberately chose a different transfer curve from that of a typical CRT in order to introduce a non-linear OOTF, as a means to compensate for the typically dim conditions in which a television is viewed. ITU BT.2087 refers to the approximation of the OETF with a square root *[Math Processing Error]* function.

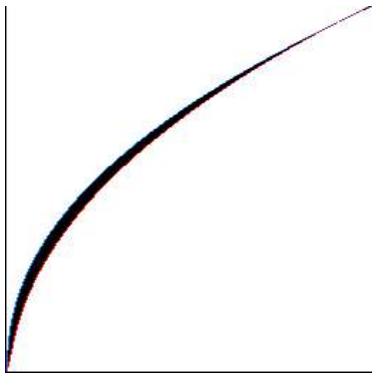
The following graph shows the close relationship between the ITU OETF (shown in red) and a pure power function with *[Math Processing Error]* (in blue). The difference between the curves is shown in black. The largest difference between the curve values at the same point when quantized to 8 bits is 15, mostly due to the sharp vertical gradient near 0.

Figure 30. ITU OETF vs pure gamma $^{1}/_{2}$



SMPTE 170M-2004 contains a note that the OETF is a more "technically correct" definition of the transfer function, and compares it to a "transfer gradient (gamma exponent) of 2.2" in previous specifications, and that the OETF in older documents is described as "1/2.2 (0.455...)". While both versions define a linear OOTF, there is no explicit mention that curve has substantially changed; this might be due to conflation of the 0.455 exponent in older specifications with the 0.45 exponent in the new formulae. The ITU OETF is actually a closer match to a gamma exponent of *[Math Processing Error]* , as shown above; it is a relatively poor match to a gamma exponent of *[Math Processing Error]* ; the following graph shows the difference between the ITU OETF (shown in red) and a pure power function with *[Math Processing Error]* (in blue). The difference between the curves is shown in black.

Figure 31. ITU OETF vs pure gamma $^{1}/_{2.2}$



13.2.3. Derivation of the ITU alpha and beta constants (informative)

Using the 12-bit encoding values for α and β from Rec.2020, there is an overlap around a non-linear value of 0.08145. In other cases, the conversion from linear to non-linear representation with encoding introduces a discontinuity between $(0.018 \times 4.500) = 0.081$ and $(1.099 \times 0.018^{0.45} - 0.099) \approx 0.0812$, corresponding to roughly a single level in a 12-bit range. SMPTE 170M-2004 provides formulae for both transformations and uses 0.0812 as a case selector for the non-linear-to-linear transformation.

The values of α and β in the ITU function were apparently chosen such that the linear segment and power segment meet at the same value and with the same derivative (that is, the linear segment meets the power segment at a tangent). The α and β values can be derived as follows:

At $\{R,G,B\} = \beta$, the linear and non-linear segments of the curve must calculate the same value:

[Math Processing Error]

Additionally, the derivatives of the linear and non-linear segments of the curve must match:

[Math Processing Error]

The derivative can be rearranged to give the equation:

[Math Processing Error]

Substituting this into the original equation results in the following:

[Math Processing Error]

This simplifies to:

[Math Processing Error]

This can be solved numerically (for example by Newton-Raphson iteration), and results in values of:

[Math Processing Error]

13.3. sRGB transfer functions

13.3.1. sRGB EOTF

The sRGB specification defines an electro-optical transfer function. The EOTF conversion from non-linear *[Math Processing Error]* encoding to linear *[Math Processing Error]* encoding is:

[Math Processing Error]

13.3.2. sRGB EOTF^{-1}

The corresponding sRGB EOTF $^{-1}$ conversion from linear *[Math Processing Error]* encoding to non-linear *[Math Processing Error]* encoding is:

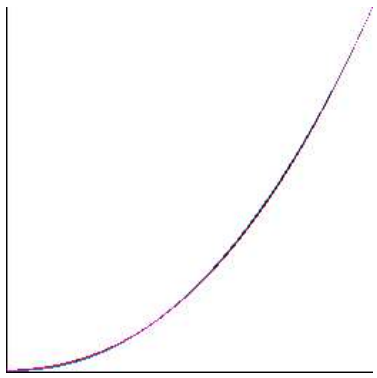
[Math Processing Error]

13.3.3. sRGB EOTF vs gamma 2.2

The sRGB EOTF approximates a simple power function with an exponent of 2.2, which is intended to be consistent with legacy CRT content, particularly for NTSC devices, and to approximate the expected EOTF for BT.709 content, given the implicit OOTF used in production video content. sRGB is distinct from ITU-T BT.1886, which offers a (different) reference EOTF for flat panels used for HDTV and is also intended to complement BT.709; in addition to the change in EOTF, sRGB specifies a reference display maximum luminance of 80cd/m^2, compared with 100cd/m^2 for BT.1886. sRGB is also distinct from SMPTE 170M, which defines its EOTF as the inverse of its (and BT.709's) OETF.

The following graph compares the sRGB EOTF (in red) and a pure power function with *[Math Processing Error]* (in blue); the area between the two curves is shown in black. The largest non-linear difference at the same linear value when quantized to 8 bits is 3.

Figure 32. sRGB EOTF vs pure gamma 2.2

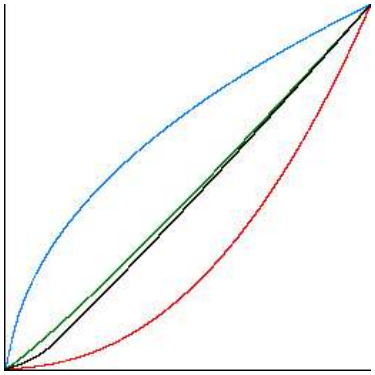


The sRGB standard assumes a quantization scheme in which 0.0 is represented by the value 0 and 1.0 is represented by 255. Despite the goal of complementing ITU-T Rec. BT.709, this is different from the ITU "full-range" encoding scheme defined in ITU-T Rec. BT.2100, which represents 1.0 as a power of two (not *[Math Processing Error]*) and therefore cannot exactly represent 1.0.

The following graph shows the relationship between the sRGB EOTF (shown in red) and the ITU OETF (shown in blue). The result of applying the two functions in turn, resulting in the OOTF of a combined ITU-sRGB system, is shown in black. Since the sRGB EOTF

approximates a power function with *[Math Processing Error]* and the ITU OETF approximates a power function with *[Math Processing Error]* , also shown in green is the resulting OOTF corresponding to a power function with *[Math Processing Error]* .

Figure 33. sRGB EOTF and ITU OETF



13.3.4. scRGB EOTF and EOTF^{-1}

The original sRGB specification was defined only in terms of positive values between 0 and 1. Subsequent standards, such as scRGB annex B, use the same transfer function but expand the range to incorporate values less than 0 and greater than 1.0. In these cases, when the input channel to the conversion is negative, the output should be the negative version of the conversion applied to the absolute value of the input. That is:

[Math Processing Error]

scRGB annex B changes the behavior of the *[Math Processing Error]* case compared with the sRGB specification. Since both calculations agree to seven decimal places, this is unlikely to be significant in most applications. scRGB annex B does not define the EOTF $^{-1}$, so the formulae below are derived by extending the sRGB formulae.

[Math Processing Error]

sYCC includes a hint that a 1cd/m^2 level of flare should be assumed for the reference 80cd/m^2 output, and that the black level should therefore be assumed to be *[Math Processing Error]* . It notes that the non-linear sRGB { *[Math Processing Error]* } values can be corrected as follows:
[Math Processing Error]
This is equivalent to applying *[Math Processing Error]* to linear *[Math Processing Error]* values. The resulting linear *[Math Processing Error]* values then need to be non-linearly encoded with the EOTF.

13.3.5. Derivation of the sRGB constants (informative)

Similar to the ITU transfer function, the EOTF $^{-1}$ of the sRGB function can be written as:

[Math Processing Error]

Like the ITU transfer function above, the values of *[Math Processing Error]* and *[Math Processing Error]* in the sRGB function appear to have been chosen such that the linear segment and power segment meet at the same value and with the same derivative (that is, the linear segment meets the power segment at a tangent). The *[Math Processing Error]* and *[Math Processing Error]* values can be derived as follows:

At *[Math Processing Error]* , the linear and non-linear segments of the function must calculate the same value:

[Math Processing Error]

Additionally, the derivatives of the linear and non-linear segments of the function must match:

[Math Processing Error]

This formula can be rearranged to give *[Math Processing Error]* in terms of *[Math Processing Error]* :

[Math Processing Error]

Substituting this into the formula for *[Math Processing Error]* :

[Math Processing Error]

This equation simplifies to:

[Math Processing Error]

This can be further simplified to:

[Math Processing Error]

The value of *[Math Processing Error]* can be found numerically (for example by Newton-Raphson iteration, with a derivative of *[Math Processing Error]*), and results in values of:

[Math Processing Error]

Where *[Math Processing Error]* is the value of the EOTF $^{-1}$ at *[Math Processing Error]* .

These deduced values are appreciably different from those in the sRGB specification, which does not state the origin of its constants. The intersection point of the sRGB EOTF has less numerical stability (and more nearby local minima in curves being optimized) that the corresponding ITU function - it is sensitive to the start value used for numerical approximations. This may explain how different values were reached for the sRGB specification. However, the errors both in value and derivative at the point of selection between the linear and exponent segments are small in practice.

The EOTF can be written with these derived values as:

[Math Processing Error]

Apple describes the Display P3 color space as using the sRGB transfer function. The profile viewer in Apple's ColorSync utility reports that the EOTF is of the following form:
[Math Processing Error]
The reported figures for *[Math Processing Error]* and *[Math Processing Error]* correspond to the equivalent values in the sRGB specification:
[Math Processing Error]
These values are correct to the reported precision both for the value *[Math Processing Error]* in the sRGB specification and for the more precise *[Math Processing Error]* derived above.

However, where the sRGB specification states that *[Math Processing Error]* , the profile viewer reports a corresponding d = 0.039. The disparity can be explained if the profile values have been derived as described in this section:
[Math Processing Error]
Note that this value assumes a correspondingly corrected version of *[Math Processing Error]* rather than *[Math Processing Error]* .

The extra precision may be needed over the constants in the sRGB specification due to the use of additional bits of accuracy in the Display P3 representation, which may expose a discontinuity due to rounding with the original numbers, particularly in the gradient of the curve. However, this distinction is subtle: when calculated over a [0..1] range, the derived EOTF and EOTF $^{-1}$ agree with the official sRGB formulae to greater than 16-bit precision.

Without allowing for adjusting the *[Math Processing Error]* constant in the sRGB formula, the power function cannot be made to intersect perfectly at a tangent to the linear segment with gradient of 12.92. However, the intersection point *[Math Processing Error]* can be found by solving:

[Math Processing Error]

This equation can give us a slightly more precise pair of values for the original sRGB equation:

[Math Processing Error]

In practice this makes no measurable difference, but does suggest that the values of *[Math Processing Error]* in the sRGB specification may have been incorrectly rounded.

13.4. BT.1886 transfer functions

The BT.1886 standard for the "Reference electro-optical transfer function for flat panel displays used in HDTV studio production" is intended to represent a typical EOTF for CRTs and to document this to ensure consistency between other display technologies:

[Math Processing Error]

L = screen luminance in cd/m^2
V = input signal normalized to [0..1]
a = user gain (legacy "contrast")
b = black level lift (legacy "brightness")
[Math Processing Error] = 2.4

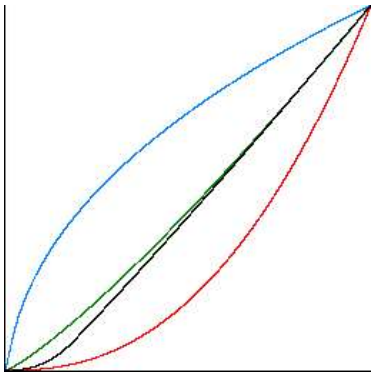
If L_W is the screen luminance of maximum white and L_B is the screen luminance of minimum black:

[Math Processing Error]

ITU BT.2087 proposes the use of a simple power function with a *[Math Processing Error]* as an approximation to this EOTF for the purposes of color conversion, effectively assuming b = 0 and L_B is pure black. The reference display described in BT.1886 has a maximum luminance level of 100cd/m^2 (brighter than the equivalent sRGB reference display).

The following graph shows the relationship between the BT.1886 EOTF (shown in red) and the ITU OETF such as used for BT.709 (shown in blue). The result of applying the two functions in turn, resulting in the OOTF of a combined BT.709-BT.1886 system, is shown in black. Since the ITU OETF approximates a power function with *[Math Processing Error]* , also shown in green is the resulting OOTF corresponding to a power function with *[Math Processing Error]* .

Figure 34. BT.1886 EOTF and BT.709 OETF



BT.1886 also offers an alternative EOTF which may provide a better match to CRT measured luminance than the standard formula listed above:
[Math Processing Error]

V_C = 0.35
[Math Processing Error] = 2.6
[Math Processing Error] = 3.0
k = coefficient of normalization (so that V = 1 gives white),
[Math Processing Error]
b = black level lift (legacy "brightness")

13.5. BT.2100 HLG transfer functions

HLG (and PQ, below) are intended to allow a better encoding of high-dynamic-range content compared with the standard ITU OETF.

13.5.1. HLG OETF (normalized)

The BT.2100-2 Hybrid Log Gamma description defines the following OETF for linear scene light:

[Math Processing Error]

[Math Processing Error] = the *[Math Processing Error]* , *[Math Processing Error]* or *[Math Processing Error]* color component of linear scene light, normalized to [0..1]
[Math Processing Error] = the resulting non-linear *[Math Processing Error]* , *[Math Processing Error]* or *[Math Processing Error]* non-linear scene light value in in the range [0..1]
a = 0.17883277
b = *[Math Processing Error]*
c = *[Math Processing Error]*

BT.2100-0, in note 5b, defines these formulae equivalently, but slightly differently:
[Math Processing Error]
This formulation in BT.2100-0 uses different constants for b and c (a is unmodified), as follows:

	BT.2100-2, BT.2100-1	BT.2100-0
b	b_1 = 0.28466892	b_0 = 0.02372241
c	c_1 = 0.55991073	c_0 = 1.00429347

The BT.2100-0 variations can be derived from the BT.2100-2 numbers as:
[Math Processing Error]

13.5.2. HLG OETF $^{-1}$ (normalized)

The OETF $^{-1}$ of normalized HLG is:

[Math Processing Error]

a, b and c are defined as for the normalized HLG OETF. BT.2100-0 again defines an equivalent formula without the *[Math Processing Error]* scale factor in the *[Math Processing Error]* < *[Math Processing Error]* term, using the modified b_0 and c_0 constants described in the note in the HLG OETF above.

13.5.3. Unnormalized HLG OETF

BT.2100-0 describes the HLG OETF formulae with E "normalized" to the range [0..12], with the variant with the range normalized to [0..1] as an alternative. Only the variant normalized to the range [0..1] is described in the updated versions of the specification, BT.2100-1 and BT.2100-2.

[Math Processing Error]

[Math Processing Error] = the *[Math Processing Error]* , *[Math Processing Error]* or *[Math Processing Error]* color component of linear scene light, normalized to [0..12]
[Math Processing Error] = the resulting non-linear *[Math Processing Error]* , *[Math Processing Error]* or *[Math Processing Error]* value in in the range [0..1]
a = 0.17883277

b = 0.28466892
c = 0.55991073

Note that these constants are the same as those used in the BT.2100-1 version of the normalized formulae.

13.5.4. Unnormalized HLG OETF $^{-1}$

The OETF $^{-1}$ of "unnormalized" HLG (producing E in the range [0..12]) is:

[Math Processing Error]

a, b and c are defined as for the unnormalized HLG OETF.

BT.2100-0 describes this "unnormalized" version of the formulae, with the variant with the E normalized to [0..1] as an alternative. Only the variant with E normalized to [0..1] is described in the updated versions, BT.2100-1 and BT.2100-2.

13.5.5. Derivation of the HLG constants (informative)

HLG constants appear to have chosen a, b and c to meet the following constraints, which are easiest to express in terms of the unnormalized OETF $^{-1}$:

- The derivative of the *[Math Processing Error]* term of the unnormalized OETF $^{-1}$ has the same value as the derivative of the *[Math Processing Error]* < *[Math Processing Error]* term of the unnormalized OETF $^{-1}$ at *[Math Processing Error]* :

  [Math Processing Error]

- The *[Math Processing Error]* term of the unnormalized OETF $^{-1}$ has the same value as the *[Math Processing Error]* < *[Math Processing Error]* term of the unnormalized OETF $^{-1}$ at *[Math Processing Error]* :

  [Math Processing Error]

- At E' = 1, the *[Math Processing Error]* < *[Math Processing Error]* term of the unnormalized OETF $^{-1}$ = 12:

  [Math Processing Error]

This last equation can be solved numerically to find:

[Math Processing Error]

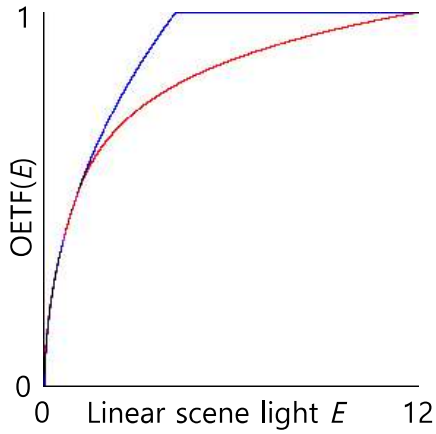
With this precision, more accurate values of the other constants are:

[Math Processing Error]

The b = 0.28466892 official figure assumes the rounded a = 0.17883277 value as an input to the b = *[Math Processing Error]* relation.

> No explanation for the choice of [0..12] range in the official version of the formula is explicitly offered in BT.2100-0 (it does *not*, for example, appear to relate to the BT.1886 OOTF *[Math Processing Error]* combined with the *[Math Processing Error]* ratio between the 1000cd/m^2 of a standard HLG HDR TV and the 100cd/m^2 of a standard dynamic range set). However, allowing for the difference in the maximum display brightness of HDR and SDR systems there is deliberate (scaled) compatibility between the HLG OETF and the BT.2020 OETF (which itself approximates a square root function) over much of the encodable dynamic range of a BT.2020 system. Since HDR content is intended to support accurate highlights more than to maintain a higher persistent screen brightness (many HDR displays can only support maximum brightness in a small area or over a small period without overheating), agreement over a significant chunk of the tone curve allows a simple adaptation between HDR and SDR devices: fed HLG-encoded content, an SDR display may represent darker tones accurately and simply under-represent highlights. The origins of both HLG and PQ are discussed in ITU-R BT.2390.
>
> As graphed in ITU-R BT.2390, the "unnormalized" HLG OETF (red) is a good approximation to the standard dynamic range ITU transfer function (blue, output scaled by 0.5) up to *[Math Processing Error]* and *[Math Processing Error]* , with a smooth curve up to the maximum HLG representable scene light value of "12":

Figure 35. HLG OETF (red) vs ITU OETF/2 (blue)



13.5.6. HLG OOTF

The OOTF of HLG is described as:

[Math Processing Error]

where R_D, G_D and B_D describe the luminance of the displayed linear component in cd/m^2 and R_S, G_S and B_S describe each color component in scene linear light, scaled by camera exposure and normalized to the representable range.

BT.2100 notes that some legacy displays apply the *[Math Processing Error]* function to each channel separately, rather than to the luminance component. That is, *[Math Processing Error]* . This is an approximation to the official OOTF.

Y_S is the normalized scene luminance, defined as:

[Math Processing Error]

[Math Processing Error] represents adjustable user gain (display "contrast") representing L_W, the nominal peak luminance of achromatic pixels.

Versions of BT.2100 prior to BT.2100-2 incorporated a *[Math Processing Error]* black level offset (display "brightness") representing the display luminance of black in cd/m^2:
[Math Processing Error]
[Math Processing Error] then represented the relative display "contrast":

	Scene light normalized to [0..1]	**Scene light normalized to [0..12]**
[Math Processing Error]	*[Math Processing Error]*	*[Math Processing Error]*

where L_W is the nominal peak luminance of the display in cd/m^2, and L_B is the display luminance of black in cd/m^2.
That is, in older versions of BT.2100, *[Math Processing Error]* represented the difference between minimum and maximum brightness, whereas in BT.2100-2 *[Math Processing Error]* is independent of black level.

[Math Processing Error] for a nominal peak display luminance of 1000cd/m^2. For displays with higher peak luminance or if peak luminance is reduced through a contrast control, *[Math Processing Error]* .

For the purposes of general conversion, L_W can be assumed to be 1000cd/m^2, and L_B can be approximated as 0, removing the constant offset from the above equations and meaning *[Math Processing Error]* .

13.5.7. HLG OOTF $^{-1}$

The inverse OOTF for HLG can be defined as:

[Math Processing Error]

For processing without reference to a specific display, *[Math Processing Error]* can be assumed to be 1.0cd/m^2.

Versions of BT.2100 prior to BT.2100-2 incorporated a *[Math Processing Error]* term into the OOTF. Using this formula from the OOTF leads to the following relationship between Y_D and Y_S:
[Math Processing Error]
From this, the following relations can be derived:
[Math Processing Error]
For processing without knowledge of the display, *[Math Processing Error]* can be treated as 1.0cd/m^2 and *[Math Processing Error]* can be considered to be 0.0cd/m^2. This simplifies the equations as follows:
[Math Processing Error]

13.5.8. HLG EOTF

The EOTF of BT.2100 HLG is defined in terms of the OETF and OOTF defined above:

[Math Processing Error]

where L_W is the nominal peak luminance of the display in cd/m^2, and L_B is the display luminance of black in cd/m^2.

> Versions of BT.2100 prior to BT.2100-2 incorporated the black level offset (display "brightness") *[Math Processing Error]* into the definition of the OOTF, such that:
> *[Math Processing Error]*

13.5.9. HLG EOTF $^{-1}$

The EOTF $^{-1}$ can be derived as:

[Math Processing Error]

> Versions of BT.2100 prior to BT.2100-2 incorporated the black level offset (display "brightness") *[Math Processing Error]* into the definition of the OOTF, such that:
> *[Math Processing Error]*

13.6. BT.2100 PQ transfer functions

> Unlike BT.2100 HLG and other ITU broadcast standards, PQ is defined in terms of an EOTF (mapping from the encoded values to the display output), not an OETF (mapping from captured scene content to the encoded values).

13.6.1. PQ EOTF

The BT.2100 Perceptual Quantization description defines the following EOTF:

[Math Processing Error]

[Math Processing Error] is a non-linear color channel *[Math Processing Error]* or *[Math Processing Error]* encoded as PQ in the range [0..1]. *[Math Processing Error]* is the luminance of the displayed component in cd/m^2 (where the luminance of an *[Math Processing Error]* or *[Math Processing Error]* or *[Math Processing Error]* component is considered to be the luminance of the color with all channels set to the same value as the component). When *[Math Processing Error]* the displayed pixel is monochromatic. *[Math Processing Error]* is a linear color value normalized to [0..1].

[Math Processing Error]

13.6.2. PQ EOTF $^{-1}$

The corresponding EOTF $^{-1}$ is:

[Math Processing Error]

13.6.3. PQ OOTF

The OOTF of PQ matches that of BT.1886's EOTF combined with BT.709's OETF:

[Math Processing Error]

where E is one of *[Math Processing Error]* , the linear representation of scene light scaled by camera exposure and in the range [0..1], G_{1886} is the EOTF described in BT.1886, and G_{709} is the OETF described in BT.709 with a scale factor of 59.5208 applied to E:

[Math Processing Error]

> ITU-R BT.2390 explains the derivation of the scale factor:
>
> PQ can encode 100 times the display brightness of a standard dynamic range ("SDR") encoding (10000cd/m^2 compared with the 100cd/m^2 SDR reference display described in BT.1886). High dynamic range (HDR) displays are intended to represent the majority of scene content within a "standard" dynamic range, and exposure of a normalized SDR signal is chosen to provide suitable exposure. HDR displays offer extra capability for representation of small or transient highlights (few HDR displays can actually reach the maximum 10000cd/m^2 encodable brightness, and few HDR displays can maintain their maximum intensity over a large area for an extended period without overheating). Therefore the behavior of HDR displays is intended to approximate a conventional standard dynamic range display for most of the image, while retaining the ability to encode extreme values.
>
> As described in BT.2390, the OOTF of SDR is roughly *[Math Processing Error]* (deviating from this curve more near a 0 value), so the maximum *scene* light intensity that can be represented is roughly *[Math Processing Error]* times that of a SDR encoding.
>
> Using exact equations from BT.709 and BT.1886 to create the OOTF, rather than the *[Math Processing Error]* approximation, the maximum representable scene brightness, if 1.0 is the maximum normalized SDR brightness is:

[Math Processing Error]
The other constants in the G_{709} formula are derived as follows:
[Math Processing Error]
Note that these constants differ slightly if the more accurate *[Math Processing Error]* figure from BT.2020 is used instead of 1.099.

13.6.4. PQ OETF

The OETF of PQ is described in terms of the above OOTF:

[Math Processing Error]

13.6.5. PQ OOTF $^{-1}$

The PQ OOTF $^{-1}$ is:

[Math Processing Error]

where F_D, display intensity, is one of *[Math Processing Error]* , and E is the corresponding normalized scene intensity.

[Math Processing Error]

13.6.6. PQ OETF $^{-1}$

The PQ OETF $^{-1}$ is described in terms of the OOTF $^{-1}$:

[Math Processing Error]

13.7. DCI P3 transfer functions

DCI P3 defines a simple power function with an exponent of 2.6 (applied to scaled CIE *XYZ* values):

EOTF $^{-1}$	EOTF
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*

This power function is applied directly to scaled CIE *XYZ* color coordinates: the "primaries" in DCI define the bounds of the gamut, but the actual color encoding uses *XYZ* coordinates. DCI scales the resulting non-linear values to the range [0..4095] prior to quantization, rounding to nearest.

"Display P3" uses the sRGB transfer function, modified in some implementations to have more accurate constants (see the section on the derivation of the sRGB constants).

13.8. Legacy NTSC transfer functions

ITU-R BT.470-6, which has now been deprecated, lists a number of regional TV standard variants; an updated list of variant codes used by country is defined in ITU-R BT.2043. This standard, along with e-CFR title 47 section 73.682, documents a simple EOTF power function with *[Math Processing Error]* for NTSC display devices.

EOTF $^{-1}$	EOTF
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*

This value of *[Math Processing Error]* is also used for N/PAL signals in the Eastern Republic of Uruguay, and was also adopted by ST-240.

Combined with the reference in SMPTE 170M to a *[Math Processing Error]* being used in "older documents", this suggests a linear design OOTF for NTSC systems.

ITU-R BT.1700, which partly replaced BT.470, also describes an "assumed gamma of display device" of 2.2 for PAL and SECAM systems; this is distinct from the *[Math Processing Error]* value listed in ITU-R BT.470-6. Combined with the ITU OETF which approximates *[Math Processing Error]* , the PAL OOTF retains a *[Math Processing Error]* when this value of *[Math Processing Error]* is used for the EOTF, similar to the figure described under the legacy PAL EOTF.

In contrast, ITU-R BT.1700 also includes SMPTE 170m, which defines the assumed EOTF of the display device as being the inverse of the current ITU OETF. Hence the new NTSC formulation also assumes a linear OOTF.

13.9. Legacy PAL OETF

ITU-R BT.472, "Video-frequency characteristics of a television system to be used for the international exchange of programmes between countries that have adopted 625-line colour or monochrome systems", defines that the "gamma of the picture signal" should be "approximately 0.4". The reciprocal of this value is 2.5.

That is, this standard defines an approximate OETF and OETF $^{-1}$ for PAL content:

OETF	OETF $^{-1}$
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*

13.10. Legacy PAL 625-line EOTF

ITU-R BT.470-6, which has now been deprecated in favor of BT.1700, lists a number of regional TV standard variants; an updated list of variant codes used by country is defined in ITU-R BT.2043.

This specification describes a simple EOTF power function with *[Math Processing Error]* for most PAL and SECAM display devices:

EOTF $^{-1}$	EOTF
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*
[Math Processing Error]	*[Math Processing Error]*

Poynton describes a *[Math Processing Error]* of 2.8 as being "unrealistically high" for actual CRT devices.

Combined with the corresponding legacy EOTF with *[Math Processing Error]* , the described system OOTF is:

[Math Processing Error]

Or *[Math Processing Error]* .

The value of *[Math Processing Error]* is described in BT.470-6 as being chosen for "an overall system gamma" (OOTF power function exponent) of "approximately 1.2"; this suggests that the "approximately 0.4" exponent in BT.472-6 should be interpreted as nearer to *[Math Processing Error]* , or at least that there was enough variation in early devices for precise formulae to be considered irrelevant.

The EOTF power function of *[Math Processing Error]* described in BT.1700 combines with the ITU OETF described in BT.601 (which approximates *[Math Processing Error]*) to give a similar system *[Math Processing Error]* . As described above, the ITU OETF combined with the BT.1886 EOTF results in a more strongly non-linear *[Math Processing Error]* .

13.11. ST240/SMPTE240M transfer functions

The ST-240, formerly SMPTE240M, interim standard for HDTV defines the following OETF:

[Math Processing Error] *[Math Processing Error]* *[Math Processing Error]*

Like SMPTE170m, ST-240 defines a linear OOTF. Therefore the above relationship also holds for the EOTF $^{-1}$.

The EOTF, and also OETF $^{-1}$, is:

[Math Processing Error] *[Math Processing Error]* *[Math Processing Error]*

13.12. Adobe RGB (1998) transfer functions

The Adobe RGB (1998) specification defines the following transfer function between nonlinear encoding and linear light intensity (notable for not including a linear component):

[Math Processing Error]

2.19921875 is obtained from *[Math Processing Error]* or hexadecimal 2.33. Therefore the inverse transfer function between linear light intensity and nonlinear encoding is:

[Math Processing Error]

13.13. Sony S-Log transfer functions

The Sony S-Log OETF is defined for each color channel as:

[Math Processing Error]

Linear camera input scaled by exposure *t* ranges from 0 to 10.0; *y* is the non-linear encoded value.

The OETF $^{-1}$ is:

[Math Processing Error]

The encoded non-linear value *t* ranges from 0 to 1.09; *Y* is the linear scene light.

13.14. Sony S-Log2 transfer functions

S-Log2 defines the following OETF:

[Math Processing Error]

x is the IRE in scene-linear space. y is the IRE in S-Log2 space.

The OETF $^{-1}$ is:

[Math Processing Error]

x is the IRE in S-Log2 space. y is the IRE in scene-linear space.

A reflection is calculated by multiplying an IRE by 0.9.

13.15. ACEScc transfer function

ACES is scene-referred; therefore ACEScc defines an OETF.

For each linear color channel lin_{AP1} transformed to the ACEScc primaries, the *ACEScc* non-linear encoding is:

[Math Processing Error]

13.16. ACEScct transfer function

ACES is scene-referred; therefore ACEScct defines an OETF.

For each linear color channel lin_{AP1} transformed to the ACEScc primaries, the *ACEScct* non-linear encoding is:

[Math Processing Error]

14. Color primaries

Color primaries define the interpretation of each channel of the color model, particularly with respect to the *RGB* model. In the context of a typical display, **color primaries** describe the color of the red, green and blue phosphors or filters.

Primaries are typically defined using the CIE 1931 *XYZ* color space, which is a color space which preserves the linearity of light intensity. Consequently, the transform from linear-intensity (R, G, B) to (X, Y, Z) is a simple matrix multiplication. Conversion between two sets of (R, G, B) color primaries can be performed by converting to the (X, Y, Z) space and back.

The (X, Y, Z) space describes absolute intensity. Since most standards do not make a requirement about the absolute intensity of the display, color primaries are typically defined using the x and y components of the xyY color space, in which the Y channel represents linear luminance. xyY is related to XYZ via the following formulae:

[Math Processing Error]

This is relevant because, although the brightness of the display in a color space definition is typically undefined, the **white point** is known: the x and y coordinates in xyY color space which corresponds to equal amounts of R, G and B. This makes it possible to determine the relative intensities of these color primaries.

> Many color standards use the CIE D65 standard illuminant as a white point. D65 is intended to represent average daylight, and has a color temperature of approximately 6500K. In CIE 1931 terms, this white point is defined in ITU standards as *[Math Processing Error]* , but elsewhere given as *[Math Processing Error]* . Different coordinates will affect the conversion matrices given below. The definition of the D65 white point is complicated by the constants in Planck's Law (which is a component in calculating the white point from the color temperature) having been revised since D65 was standardized, such that the standard formula for calculating CIE coordinates from the color temperature do not agree with the D65 standard. The actual color temperature of D65 is nearer to *[Math Processing Error]* .

Assuming an arbitrary white luminance (Y value) of 1.0, it is possible to express the following identity for the X, Y and Z coordinates of each color channel R, G and B, and of the white point W:

[Math Processing Error]

The identities *[Math Processing Error]* and *[Math Processing Error]* can be used to re-express the above terms in the xyY space:

[Math Processing Error]

This equation for W_Z can be simplified to:

[Math Processing Error]

Since *[Math Processing Error]* , this further simplifies to:

[Math Processing Error]

The *[Math Processing Error]* term for W_Y can be multiplied by *[Math Processing Error]* and subtracted from the equation for W_X:

[Math Processing Error]

Similarly, the *[Math Processing Error]* term can be multiplied by *[Math Processing Error]* and subtracted from the simplified W_Z line:

[Math Processing Error]

Finally, the G_Y term can be eliminated by multiplying the former of these two equations by *[Math Processing Error]* and subtracting it from the latter multiplied by *[Math Processing Error]* , giving:

[Math Processing Error] [Math Processing Error]

Thus:

[Math Processing Error]

This allows G_Y to be calculated by rearranging an earlier equation:

[Math Processing Error]

And finally:

[Math Processing Error]

These relative magnitudes allow the definition of vectors representing the color primaries in the *XYZ* space, which in turn provides a transformation between colors specified in terms of the color primaries and the *XYZ* space. Without an absolute magnitude the transformation to *XYZ* is incomplete, but sufficient to allow transformation to another set of color primaries.

The transform from the defined color primaries to *XYZ* space is:

[Math Processing Error]

The transform from *XYZ* space to the defined color primaries is therefore:

[Math Processing Error]

These transforms assume that the black point for the color space is at *[Math Processing Error]* . If the black point is non-zero, these transforms require a translational component. In some color spaces the black point has the same color as the white point, in which case it is also possible to adjust the *[Math Processing Error]* values outside the matrix.

Real-world objects and light sources typically emit a full spectrum of wavelengths in varying intensities. Describing the color of an object in terms of three wavelengths is necessarily an approximation, and effective only because the human eye typically sees the world mostly in terms of combinations of three color cone cell responses — "tristimulus values". The typical response of these "long", "medium" and "short" wavelength-sensitive cones is the basis of the CIE XYZ color space, so a linear combination of primaries described in this color space can be mapped to a linear combination of other primaries. Two colors with a different spectrum but an identical appearance are known as *metamers*, and the tristimulus model of color relies on "metameric matching": picking a linear combination of some set of primaries that will elicit the same cone cell response as the desired color.

Each of the cone cells can appear in mutated form which results in a change to its response to different frequencies of light. Since the genes for cone cells are encoded on the X chromosome, some women have more than one variant of the same cone cell, being "tetrachromats" — and there is evidence that such people can distinguish more colors than the general "trichromat" population, with only a subset of these colors describable with three color primaries. There is some evidence for people with two variants of more than one cone cell, thereby perceiving color in a five-dimensional space, although the processing performed by the human visual system is complex and it is not clear how independently these color dimensions are in practice. In the animal kingdom, it is not uncommon to have more than three types of cone cell, and mantis shrimp have up to sixteen kinds of photoreceptors. Note that rod cells, which dominate the human visual system in low light conditions, are typically ignored at higher illumination levels for the purposes of color matching.

The difference between a primary color representation and the full spectrum emitted by a real-world object is particularly visible for printed media. Printing colors are typically described assuming that they will reflect the spectrum of a theoretical illuminant; changing this light source may change the appearance of the printed object in a more complex manner than scaling the primary values. This can lead to "metameric failure", where two colors may appear identical under some lighting conditions and different under others. This effect is present in nature, notably in the mineral alexandrite, which may appear reddish purple under incandescent light but green in daylight. Camera sensors and film have a related problem that the spectrum of filters does not perfectly match that of cone cells, which can lead some colors (particularly with sharp peaks in their spectra) to appear different. Inkjet printing often uses many more than three inks in order to maximize the representable gamut and control metamerism.

Adaptation to a different white point is often performed by a linear scaling of primaries, known as the "von Kries transform". The Bradford color adaptation transform incorporates a slight nonlinear term to the blue component of colors to more accurately reflect visual behavior.

It is common for production rendering systems to represent each pixel with more than three channels. For example, the contributions from different lights may be recorded separately in order to allow later adjustment to color and relative intensity, specular and diffuse reflectance from a surface may be recorded separately (which is valuable, for example, in denoising algorithms), and virtual "channels" may be used to control shader effects. A larger number of wavelengths may also be used when rendering effects such as chromatic dispersion, which is important for rendering jewelry. Currently these are typically represented as multiple images or in a proprietary representation, although one could imagine a data format descriptor for such a pixel.

14.1. BT.709 color primaries

ITU-T BT.709 (HDTV) defines the following chromaticity coordinates:

[Math Processing Error]

These chromaticity coordinates are also shared by sRGB and scRGB.

Therefore to convert from linear color values defined in terms of BT.709 color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of BT.709 color primaries, is:

[Math Processing Error]

sYCC lists a slightly different version of this matrix, possibly due to rounding errors.

14.2. BT.601 625-line color primaries

ITU-T Rec.601 defines different color primaries for 625-line systems (as used in most PAL systems) and for 525-line systems (as used in the SMPTE 170M-2004 standard for NTSC).

The following chromaticity coordinates are defined for 625-line "EBU" systems:

[Math Processing Error]

BT.470-6, which also describes these constants in a legacy context, approximates D65 as *[Math Processing Error]* .

Therefore to convert from linear color values defined in terms of BT.601 color primaries for 625-line systems to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of BT.601 "EBU" 625-line color primaries, is:

[Math Processing Error]

14.3. BT.601 525-line color primaries

ITU-T Rec.601 defines different color primaries for 625-line systems (as used in most PAL systems) and for 525-line systems (as used in the SMPTE 170M-2004 standard for NTSC).

The following chromaticity coordinates are defined in BT.601 for 525-line digital systems and in SMPTE-170M:

[Math Processing Error]

Therefore to convert from linear color values defined in terms of BT.601 color primaries for 525-line systems to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of BT.601 525-line color primaries, is:

[Math Processing Error]

Analog 525-line PAL systems used a different white point, and therefore have a different conversion matrix.

14.4. BT.2020 color primaries

The following chromaticity coordinates are defined in BT.2020 for ultra-high-definition television:

[Math Processing Error]

The same primaries are used for BT.2100 for HDR TV.

Therefore to convert from linear color values defined in terms of BT.2020 color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of BT.2020 color primaries, is:

[Math Processing Error]

14.5. NTSC 1953 color primaries

The following chromaticity coordinates are defined in ITU-R BT.470-6 and SMPTE 170m as a reference to the legacy NTSC standard:

[Math Processing Error]

These primaries apply to the 1953 revision of the NTSC standard. Modern NTSC systems, which reflect displays that are optimized for brightness over saturation, use the color primaries as described in Section 14.3. The white point used in the original NTSC 1953 specification is CIE Standard Illuminant C, 6774K, as distinct from the CIE Illuminant D65 used by most modern standards. BT.470-6 notes that SECAM systems may use these NTSC primaries and white point. Japanese legacy NTSC systems used the same primaries but with the white point set to D-white at 9300K.

Therefore to convert from linear color values defined in terms of NTSC 1953 color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of NTSC 1953 color primaries, is:

[Math Processing Error]

14.6. PAL 525-line analog color primaries

ITU-R BT.1700 defines the following chromaticity coordinates for legacy 525-line PAL systems:

[Math Processing Error]

This matches the color primaries from SMPTE-170m analog NTSC and BT.601 525-line encoding, but the white point used is CIE Standard Illuminant C, 6774K, as distinct from the CIE Illuminant D65 white point used by most modern standards.

Therefore to convert from linear color values defined in terms of PAL 525-line color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of PAL 525-line 1953 color primaries, is:

[Math Processing Error]

14.7. ACES color primaries

The following chromaticity coordinates are defined in SMPTE ST 2065-1

[Math Processing Error]

Therefore to convert from linear color values defined in terms of ACES color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of ACES color primaries, is:

[Math Processing Error]

14.8. ACEScc color primaries

The following chromaticity coordinates are defined in Academy S-2016-001 (ACEScct) and S-2014-003 (ACEScc), which share the same primaries:

[Math Processing Error]

Therefore to convert from linear color values defined in terms of ACEScc/ACEScct color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of ACEScc/ACEScct color primaries, is:

[Math Processing Error]

14.9. Display P3 color primaries

The following chromaticity coordinates are defined in Display P3:

[Math Processing Error]

The DCI P3 color space defines the bounds of its gamut using these primaries, but actual color data in DCI P3 is encoded using CIE *XYZ* coordinates. Display P3, on the other hand, uses these values as primaries in an *RGB* color space, with a D65 white point.

Therefore to convert from linear color values defined in terms of Display P3 color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of DisplayP3 color primaries, is:

[Math Processing Error]

These matrices differ from those given in SMPTE EG 432-1 due to the choice of a D65 white point in Display P3. The matrices in 432-1 can be reproduced by applying a white point of *[Math Processing Error]* to the above primaries.

14.10. Adobe RGB (1998) color primaries

The following chromaticity coordinates are defined in Adobe RGB (1998):

[Math Processing Error]

Therefore to convert from linear color values defined in terms of Adobe RGB (1998) color primaries to *XYZ* space the formulae in Section 14 result in the following matrix:

[Math Processing Error]

The inverse transformation, from the *XYZ* space to a color defined in terms of Adobe RGB (1998) color primaries, is:

[Math Processing Error]

Adobe RGB (1998) defines a reference display white brightness of 160cd/m^2 and a black point 0.34731% of this brightness, or 0.5557cd/m^2, for a contrast ratio of 287.9. The black point has the same color temperature as the white point, and this does not affect the above matrices.

14.11. BT.709/BT.601 625-line primary conversion example

Conversion from BT.709 to BT.601 625-line primaries can be performed using the matrices in Section 14.1 and Section 14.2 as follows:

[Math Processing Error]
[Math Processing Error]

Conversion from BT.601 625-line to BT.709 primaries can be performed using these matrices:

[Math Processing Error]
[Math Processing Error]

14.12. BT.709/BT.2020 primary conversion example

Conversion from BT.709 to BT.2020 primaries can be performed using the matrices in Section 14.4 and Section 14.1 as follows:

[Math Processing Error]
[Math Processing Error]

Conversion from BT.2020 primaries to BT.709 primaries can be performed with the following matrices:

[Math Processing Error]
[Math Processing Error]

15. Color models

The human eye is more sensitive to high-frequency changes in intensity (absolute quantity of light) than to high-frequency changes in the dominant wavelength and saturation of a color. Additionally the eye does not exhibit equal sensitivity to all wavelengths. Many image representations take advantage of these facts to distribute the number of bits used to represent a texel in a more perceptually-uniform manner than is achieved by representing the color primaries independently - for example by encoding the chroma information at a reduced spatial resolution.

15.1. $Y'C_BC_R$ color model

Color models based on color differences are often referred to with incorrect or ambiguous terminology, the most common of which is *YUV*.

In the broadcast standards which define these models:

- A prime mark (') is used to refer to the "gamma pre-corrected" version of a value. That is, an approximation to a perceptually linear mapping between value and intensity. The absence of a prime mark indicates that the value is linear in intensity.
- *R'G'B'* is used to refer to the red, green and blue reference values in "gamma pre-corrected" form. That is, *R'*, *G'* and *B'* have a non-linear transfer function, whereas *R*, *G* and *B* are linear with respect to light intensity. The transfer function used resembles an exponentiation "gamma correction" operation, with a linear segment near zero for mathematical stability. See Section 13.2 for details of the transfer function typically used in these cases.
- IEEE standards BT.601 and BT.709 use a prefix of *E* to refer to a continuous signal value in the range [0..1], mirroring the terminology in analog standards such as BT.1700 and SMPTE-170M. For example, in these standards, the continuous encoding of *[Math Processing Error]* is written *[Math Processing Error]* . BT.2020 and BT.2100 no longer use the *E* convention, and refer to continuous values as, for example, *R'* directly. For brevity, this specification does not use the *E*-prefix convention for model conversions, and all values can be assumed to be continuous. BT.601 refers to the quantized digital version of *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* as *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* . In BT.709 the

quantized digital representation is instead *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* , in BT.2020 and BT.2100 written as DR', DG' and DB'.

- Y' is a weighted sum of R', G' and B' values, and represents non-physically-linear (but perceptually-linear) light intensity, as distinct from physically-linear light intensity. Note that the ITU broadcast standards use "luminance" for Y' despite some authorities reserving that term for a linear intensity representation. Since this is a weighted sum of non-linear values, *[Math Processing Error]* is not mathematically equivalent to applying the non-linear transfer function to a weighted sum of linear R, G and B values: *[Math Processing Error]* . The prime symbol is often omitted so that Y' is confusingly written Y. BT.601 and BT.709 refers to the continuous non-linear "luminance" signal as *[Math Processing Error]* ; in BT.2020 and BT.2100 this value is just Y'. The quantized digital representation is written as simply Y' in BT.601, as *[Math Processing Error]* in BT.709, and as DY' in BT.2020 and BT.2100. In this standard, Y' refers to a continuous value.

- For the purposes of this section, we will refer to the weighting factor applied to R' as K_R and the weighting factor applied to B' as K_B. The weighting factor of G' is therefore *[Math Processing Error]* . Thus *[Math Processing Error]* .

Color differences are calculated from the non-linear Y' and color components as:

[Math Processing Error]

Note that, for R', G', B' in the range [0..1]:

[Math Processing Error]

- *[Math Processing Error]* scaled appropriately for incorporation into a PAL sub-carrier signal is referred to in BT.1700 as U; note that the scale factor (0.493) is not the same as that used for digital encoding of this color difference. U is colloquially used for other representations of this value.
- *[Math Processing Error]* scaled appropriately for incorporation into a PAL sub-carrier signal is referred to in BT.1700 as V; note that the scale factor (0.877) is not the same as that used for digital encoding of this color difference. V is colloquially used for other representations of this value.
- *[Math Processing Error]* scaled to the range [*[Math Processing Error]*] is referred to in BT.601 and BT.709 as *[Math Processing Error]* , and in BT.2020 and BT.2100 as simply *[Math Processing Error]* . In ST-240 this value is referred to as *[Math Processing Error]* , and the analog signal is colloquially known as P_B. This standard uses the *[Math Processing Error]* terminology for brevity and consistency with *[Math Processing Error]* . It is common, especially in the name of a color model, to omit the prime symbol and write simply C_B.
- *[Math Processing Error]* scaled to the range [*[Math Processing Error]*] is referred to in BT.601 and BT.709 as *[Math Processing Error]* , and in BT.2020 and BT.2100 as simply *[Math Processing Error]* . In ST-240 this value is referred to as *[Math Processing Error]* , and the analog signal is colloquially known as P_R. This standard uses the *[Math Processing Error]* terminology for brevity and consistency with *[Math Processing Error]* . It is common, especially in the name of a color model, to omit the prime symbol and write simply C_R.
- *[Math Processing Error]* scaled and quantized for digital representation is known as simply *[Math Processing Error]* in BT.601, *[Math Processing Error]* in BT.709 and *[Math Processing Error]* in BT.2020 and BT.2100.
- *[Math Processing Error]* scaled and quantized for digital representation is known as simply *[Math Processing Error]* in BT.601, *[Math Processing Error]* in BT.709 and *[Math Processing Error]* in BT.2020 and BT.2100.
- This section considers the color channels in continuous terms; the terminology *[Math Processing Error]* and *[Math Processing Error]* is used in Section 16.

Using this terminology, the following conversion formulae can be derived:

[Math Processing Error]

For the inverse conversion:

[Math Processing Error]

The formula for G' can be derived by substituting the formulae for R' and B' into the derivation of Y':

[Math Processing Error]

The values chosen for K_R and K_B vary between standards.

The required color model conversion between *[Math Processing Error]* and *[Math Processing Error]* can typically be deduced from other color space parameters:

Primaries		OETF		Color model conversion	
Defined in	Described in	Defined in	Described in	Defined in	Described in
BT.709 sRGB	Section 14.1	BT.709	Section 13.2	BT.709	Section 15.1.1
BT.709 sRGB sYCC	Section 14.1	sRGB sYCC	Section 13.3	BT.601	Section 15.1.2
BT.601 (625-line)	Section 14.2	BT.601	Section 13.2	BT.601	Section 15.1.2
BT.601 (525-line) ST-240	Section 14.3	BT.601	Section 13.2	BT.601	Section 15.1.2
BT.601 (525-line) ST-240	Section 14.3	ST-240	Section 13.11	ST-240	Section 15.1.4
BT.2020 BT.2100	Section 14.4	BT.2020	Section 13.2	BT.2020	Section 15.1.3

15.1.1. BT.709 $Y'C_BC_R$ conversion

ITU Rec.709 defines K_R = 0.2126 and K_B = 0.0722.

That is, for conversion between (R', G', B') defined in BT.709 color primaries and using the ITU transfer function:

[Math Processing Error]

Alternatively:

[Math Processing Error]

For the inverse conversion:

[Math Processing Error]

15.1.2. BT.601 $Y'C_BC_R$ conversion

ITU Rec.601 defines K_R = 0.299 and K_B = 0.114.

That is, for conversion between (R', G', B') defined in BT.601 EBU color primaries or BT.601 SMPTE color primaries, and using the ITU transfer function:

[Math Processing Error]

Alternatively:

[Math Processing Error]

For the inverse conversion:

[Math Processing Error]

15.1.3. BT.2020 $Y'C_BC_R$ conversion

ITU Rec.2020 and ITU Rec.2100 define K_R = 0.2627 and K_B = 0.0593.

That is, for conversion between (R', G', B') defined in BT.2020 color primaries and using the ITU transfer function:

[Math Processing Error]

Alternatively:

[Math Processing Error]

For the inverse conversion:

[Math Processing Error]

15.1.4. ST-240/SMPTE 240M $Y'C_BC_R$ conversion

ST240, formerly SMPTE 240M, defines K_R = 0.212 and K_B = 0.087.

That is, for conversion using the ST240 transfer function:

[Math Processing Error]

Alternatively:

[Math Processing Error]

For the inverse conversion:

[Math Processing Error]

15.2. $Y'_CC'_{BC}C'_{CR}$ constant luminance color model

ITU-T Rec. BT.2020 introduced a "constant luminance" color representation as an alternative representation to $Y'C_BC_R$:

[Math Processing Error]

This terminology follow's BT.2020's convention of describing the continuous values as *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* ; BT.2020 uses *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* to represent the quantized integer representations of the same values.

[Math Processing Error] is derived from applying a non-linear transfer function to a combination of linear *[Math Processing Error]* components and applying a non-linear transfer function to the result, but the *[Math Processing Error]* and *[Math Processing Error]* color differences still encode differences between non-linear values.

The inverse transformation can be derived from the above:

[Math Processing Error]

Performing these calculations requires conversion between a linear representation and a non-linear transfer function during the transformation. This is distinct from the non-constant-luminance case, which is a simple matrix transform.

15.3. IC_TC_P constant intensity color model

ITU-T Rec. BT.2100 introduced a "constant intensity" color representation as an alternative representation to $Y'C_BC_R$:

[Math Processing Error]

Note that the suffix $_D$ indicates that PQ encoding is *display-referred* and the suffix $_S$ indicates that HLG encoding is *scene-referred*—that is, they refer to display and scene light respectively.

To invert this, it can be observed that:

[Math Processing Error] [Math Processing Error] [Math Processing Error] [Math Processing Error] [Math Processing Error] [Math Processing Error] [Math Processing Error] [Math Processing Error]

16. Quantization schemes

The formulae in the previous sections are described in terms of operations on continuous values. These values are typically represented by quantized integers. There are standard encodings for representing some color models within a given bit depth range.

16.1. "Narrow range" encoding

ITU broadcast standards typically reserve values at the ends of the representable integer range for rounding errors and for signal control data. The nominal range of representable values between these limits is represented by the following encodings, for bit depth n = {8, 10, 12}:

[Math Processing Error]

The dequantization formulae are therefore:

[Math Processing Error]

For consistency with *[Math Processing Error]* , these formulae use the BT.2020 and BT.2100 terminology of prefixing a D to represent the digital quantized encoding of a numerical value.

That is, in "narrow range" encoding:

Value	Continuous encoding value	Quantized encoding
Black	{R', G', B', Y', *[Math Processing Error]* , I} = 0.0	{DR', DG', DB', DY', *[Math Processing Error]* , DI} = *[Math Processing Error]*
Peak brightness	{R', G', B', Y', *[Math Processing Error]* , I} = 1.0	{DR', DG', DB', DY', *[Math Processing Error]* , DI} = *[Math Processing Error]*
Minimum color difference value	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = -0.5	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = *[Math Processing Error]*
Maximum color difference value	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = 0.5	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = *[Math Processing Error]*
Achromatic colors	$R' = G' = B'$ { *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = 0.0	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = *[Math Processing Error]*

If, instead of the quantized values, the input is interpreted as fixed-point values in the range 0.0..1.0, as might be the case if the values were treated as unsigned normalized quantities in a computer graphics API, the following conversions can be applied instead:

[Math Processing Error]

16.2. "Full range" encoding

ITU-T Rec. BT.2100-1 and the current Rec. T.871 JFIF specification define the following quantization scheme that does not incorporate any reserved head-room or foot-room, which is optional and described as "full range" in BT.2100, and integral to Rec. T.871.

Both these specifications modify a definition used in previous versions of their specifications, which is described below.

For bit depth n = {8 (JFIF),10,12 (Rec.2100)}:

[Math Processing Error]

BT.2100-1 defines Round() as:

[Math Processing Error]

Note that a chroma channel value of exactly 0.5 corresponds to a quantized encoding of *[Math Processing Error]* , and must therefore be clamped to the nominal peak value of *[Math Processing Error]* . Narrow-range encoding does not have this problem. A chroma channel value of -0.5 corresponds to a quantized encoding of 1, which is the nominal minimum peak value.

In Rec. T.871 (which defines only n = 8), the corresponding formula is:

[Math Processing Error]

Allowing for the clamping at a chroma value of 0.5, these formulae are equivalent across the expected -0.5..0.5 range for chroma and 0.0..1.0 range for luma values.

The dequantization formulae are therefore:

[Math Processing Error]

That is, in "full range" encoding:

Value	Continuous encoding value	Quantized encoding
Black	{R', G', B', Y', *[Math Processing Error]* , I} = 0.0	{DR', DG', DB', DY', *[Math Processing Error]* , DI} = 0
Peak brightness	{R', G', B', Y', *[Math Processing Error]* , I} = 1.0	{DR', DG', DB', DY', *[Math Processing Error]* , DI} = 2^n - 1
Minimum color difference value	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = -0.5	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = 1
Maximum color difference value	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = 0.5	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = *[Math Processing Error]* (clamped)
Achromatic colors	R' = G' = B' { *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , C_T, C_P} = 0.0	{ *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , DC_T, DC_P} = 2^{n-1}

If, instead of the quantized values, the input is interpreted as fixed-point values in the range 0.0..1.0, as might be the case if the values were treated as unsigned normalized quantities in a computer graphics API, the following conversions can be applied instead:

[Math Processing Error]

16.3. Legacy "full range" encoding.

ITU-T Rec. BT.2100-0 formalized an optional encoding scheme that does not incorporate any reserved head-room or foot-room. The legacy JFIF specification similarly used the full range of 8-bit channels to represent *[Math Processing Error]* color. For bit depth n = {8 (JFIF),10,12 (Rec.2100)}:

[Math Processing Error]

The dequantization formulae are therefore:

[Math Processing Error]

These formulae map luma values of 1.0 and chroma values of 0.5 to *[Math Processing Error]* , for bit depth *[Math Processing Error]* . This has the effect that the maximum value (e.g. pure white) cannot be represented directly. Out-of-bounds values must be clamped to the largest representable value.

ITU-R BT.2100-0 dictates that in 12-bit coding, the largest values encoded should be 4092 ("for consistency" with 10-bit encoding, with a maximum value of 1023). This slightly reduces the maximum intensity which can be expressed, and slightly reduces the saturation range. The achromatic color point is still 2048 in the 12-bit case, so no offset is applied in the transformation to compensate for this range reduction. BT.2100-1 removes this recommendation and lists 4095 as the nominal peak value.

If, instead of the quantized values, the input is interpreted as fixed-point values in the range 0.0..1.0, as might be the case if the values were treated as unsigned normalized quantities in a computer graphics API, the following conversions can be applied instead:

[Math Processing Error]
[Math Processing Error]

That is, to match the behavior described in these specifications, the inputs to color model conversion should be expanded such that the maximum representable value is that defined by the quantization of these encodings *[Math Processing Error]* , and the inverse operation should be applied to the result of the model conversion.

For example, a legacy shader-based JPEG decoder may read values in a normalized 0..1 range, where the in-memory value 0 represents 0.0 and the in-memory value 1 represents 1.0. The decoder should scale the Y' value by a factor of *[Math Processing Error]* to match the encoding in the JFIF3 document, and *[Math Processing Error]* and C_R should be scaled by *[Math Processing Error]* and offset by 0.5. After the model conversion matrix has been applied, the R', G' and B' values should be scaled by *[Math Processing Error]* , restoring the ability to represent pure white.

17. Compressed Texture Image Formats

For computer graphics, a number of texture compression schemes exist which reduce both the overall texture memory footprint and the bandwidth requirements of using the textures. In this context, "texture compression" is distinct from "image compression" in that texture compression schemes are designed to allow efficient random access as part of texture sampling: "image compression" can further reduce image redundancy by considering the image as a whole, but doing so is impractical for efficient texture access operations. The common texture compression schemes are "block-based", relying on similarities between nearby texel regions to describe "blocks" of nearby texels in a unit:

- "S3TC" describes a block of 4×4 *RGB* texels in terms of a low-precision pair of color "endpoints", and allow each texel to specify an interpolation point between these endpoints. Alpha channels, if present, may be described similarly or with an explict per-texel alpha value.
- "RGTC" provides one- and two-channel schemes for interpolating between two "endpoints" per 4×4 texel block, and are intended to provide efficient schemes for normal encoding, complementing the three-channel approach of S3TC.
- "BPTC" offers a number of ways of encoding and interpolating endpoints, and allows the 4×4 texel block to be divided into multiple "subsets" which can be encoded independently, which can be useful for managing different regions with sharp transitions.
- "ETC1" provides ways of encoding 4×4 texel blocks as two regions of 2×4 or 4×2 texels, each of which are specified as a base color; texels are then encoded as offsets relative to these bases, varying by a grayscale offset.
- "ETC2" is a superset of ETC1, adding schemes for color patterns that would fit poorly into ETC1 options.
- "ASTC" allows a wide range of ways of encoding each color block, and supports choosing different block sizes to encode the texture, providing a range of compression ratios; it also supports 3D and HDR textures.
- "PVRTC" describes several encoding schemes with two colors per block of 4×4 or 8×4 texels, interpolated between adjacent texel blocks, and means of modulating between them.

note Example data format descriptors for compressed formats can be found under the ***colorModel*** field in Section 5.6.

17.1. Terminology

As can be seen above, the compression schemes have a number of features in common — particularly in having a number of endpoints described encoded in some of the bits of the texel block. For consistency and to make the terms more concise, the following descriptions use some slightly unusual terminology: The value X_n^m refers to bit m (starting at 0) of the nth X value. For example, R_1^3 would refer to bit 3 of red value 1 — *R*, *G*, *B* and *A* (capitalized and italicized) are generally used to refer to color channels. Similarly, $R_1^{2..3}$ refers to bits 2..3 of red value 1. Although unusual, this terminology should be unambiguous (e.g. none of the formats require exponentiation of arguments).

18. S3TC Compressed Texture Image Formats

This description is derived from the EXT_texture_compression_s3tc extension.

Compressed texture images stored using the S3TC compressed image formats are represented as a collection of 4×4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If an S3TC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an S3TC image with a width of *w*, height of *h*, and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

[Math Processing Error]

When decoding an S3TC image, the block containing the texel at offset (*x*, *y*) begins at an offset (in bytes) relative to the base of the image of:

[Math Processing Error]

The data corresponding to a specific texel (*x*, *y*) are extracted from a 4×4 texel block using a relative (*x*, *y*) value of

[Math Processing Error]

There are four distinct S3TC image formats:

18.1. BC1 with no alpha

Each 4×4 block of texels consists of 64 bits of *RGB* image data.

Each *RGB* image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

[Math Processing Error]

The 8 bytes of the block are decoded into three quantities:

[Math Processing Error]

$color_0$ and $color_1$ are 16-bit unsigned integers that are unpacked to *RGB* colors RGB_0 and RGB_1 as though they were 16-bit unsigned packed pixels with the *R* channel in the high 5 bits, *G* in the next 6 bits and *B* in the low 5 bits:

[Math Processing Error]

bits is a 32-bit unsigned integer, from which a two-bit control code is extracted for a texel at location (*x*, *y*) in the block using:

[Math Processing Error]

where *bits*[31] is the most significant and *bits*[0] is the least significant bit.

The *RGB* color for a texel at location (*x*, *y*) in the block is given in Table 105.

Table 105. Block decoding for BC1

Texel value	Condition
RGB_0	$color_0 > color_1$ and $code(x, y) = 0$
RGB_1	$color_0 > color_1$ and $code(x, y) = 1$
[Math Processing Error]	$color_0 > color_1$ and $code(x, y) = 2$
[Math Processing Error]	$color_0 > color_1$ and $code(x, y) = 3$
RGB_0	$color_0 \leq color_1$ and $code(x, y) = 0$
RGB_1	$color_0 \leq color_1$ and $code(x, y) = 1$
[Math Processing Error]	$color_0 \leq color_1$ and $code(x, y) = 2$
BLACK	$color_0 \leq color_1$ and $code(x, y) = 3$

Arithmetic operations are done per component, and BLACK refers to an *RGB* color where red, green, and blue are all zero.

Since this image has an *RGB* format, there is no alpha component and the image is considered fully opaque.

18.2. BC1 with alpha

Each 4×4 block of texels consists of 64 bits of *RGB* image data and minimal alpha information. The *RGB* components of a texel are extracted in the same way as BC1 with no alpha.

The alpha component for a texel at location (*x*, *y*) in the block is given by Table 106.

Table 106. BC1 with alpha

Alpha value	Condition
0.0	$color_0 \leq color_1$ and $code(x, y) = 3$
1.0	otherwise

The red, green, and blue components of any texels with a final alpha of 0 should be encoded as zero (black).

Figure 36 shows an example BC1 texel block: $color_0$, encoded as *[Math Processing Error]* , and $color_1$, encoded as *[Math Processing Error]* , are shown as circles. The interpolated values are shown as small diamonds. Since 29 > 20, there are two interpolated values, accessed when $code(x, y) = 2$ and $code(x, y) = 3$.

Figure 36. BC1 two interpolated colors

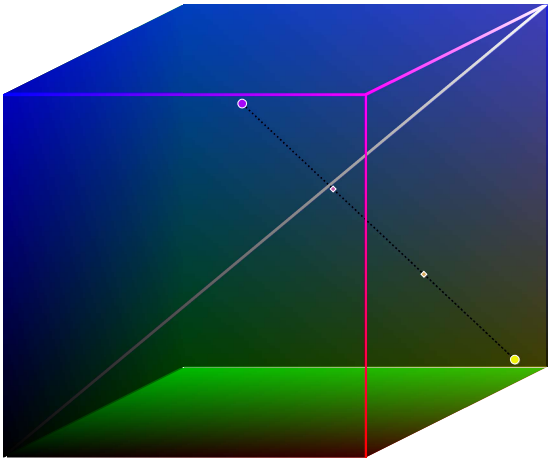
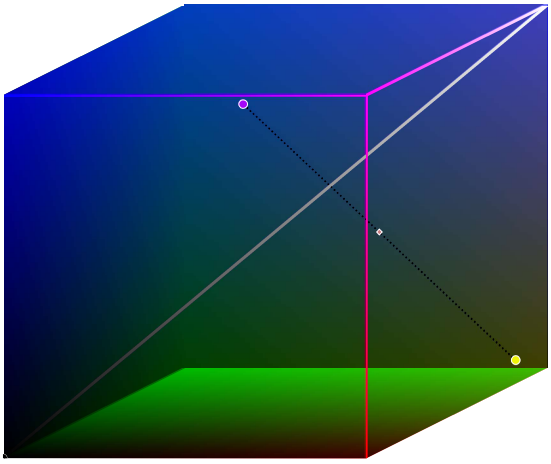


Figure 37 shows the example BC1 texel block with the colors swapped: $color_0$, encoded as *[Math Processing Error]* , and $color_1$, encoded as *[Math Processing Error]* , are shown as circles. The interpolated value is shown as a small diamonds. Since 20 ≤ 29, there is one interpolated value for $code(x, y) = 2$, and $code(x, y) = 3$ represents $(R, G, B) = (0, 0, 0)$.

Figure 37. BC1 one interpolated color + black



If the format is BC1 with alpha, $code(x, y) = 3$ is transparent (alpha = 0). If the format is BC1 with no alpha, $code(x, y) = 3$ represents opaque black.

18.3. BC2

Each 4×4 block of texels consists of 64 bits of uncompressed alpha image data followed by 64 bits of *RGB* image data.

Each *RGB* image data block is encoded according to the BC1 formats, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though $color_0 > color_1$, regardless of the actual values of $color_0$ and $color_1$.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

[Math Processing Error]

The 8 bytes of the block are decoded into one 64-bit integer:

[Math Processing Error]

alpha is a 64-bit unsigned integer, from which a four-bit alpha value is extracted for a texel at location (x, y) in the block using:

[Math Processing Error]

where *bits*[63] is the most significant and *bits*[0] is the least significant bit.

The alpha component for a texel at location (x, y) in the block is given by *[Math Processing Error]* .

18.4. BC3

Each 4×4 block of texels consists of 64 bits of compressed alpha image data followed by 64 bits of *RGB* image data.

Each *RGB* image data block is encoded according to the BC1 formats, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though $color_0 > color_1$, regardless of the actual values of $color_0$ and $color_1$.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

[Math Processing Error]

The $alpha_0$ and $alpha_1$ are 8-bit unsigned bytes converted to alpha components by multiplying by *[Math Processing Error]* .

The 6 *bits* bytes of the block are decoded into one 48-bit integer:

[Math Processing Error]

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x, y) in the block using:

[Math Processing Error]

where *bits*[47] is the most-significant and *bits*[0] is the least-significant bit.

The alpha component for a texel at location (x, y) in the block is given by Table 107.

Table 107. Alpha encoding for BC3 blocks

Alpha value	Condition
$alpha_0$	$code(x, y) = 0$
$alpha_1$	$code(x, y) = 1$
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 2$
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 3$
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 4$
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 5$

Alpha value	Condition
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 6$
[Math Processing Error]	$alpha_0 > alpha_1$ and $code(x, y) = 7$
[Math Processing Error]	$alpha_0 \le alpha_1$ and $code(x, y) = 2$
[Math Processing Error]	$alpha_0 \le alpha_1$ and $code(x, y) = 3$
[Math Processing Error]	$alpha_0 \le alpha_1$ and $code(x, y) = 4$
[Math Processing Error]	$alpha_0 \le alpha_1$ and $code(x, y) = 5$
0.0	$alpha_0 \le alpha_1$ and $code(x, y) = 6$
1.0	$alpha_0 \le alpha_1$ and $code(x, y) = 7$

19. RGTC Compressed Texture Image Formats

This description is derived from the "RGTC Compressed Texture Image Formats" section of the OpenGL 4.4 specification.

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of 4×4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If an RGTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of w, height of h, and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

[Math Processing Error]

When decoding an RGTC image, the block containing the texel at offset [Math Processing Error] begins at an offset (in bytes) relative to the base of the image of:

[Math Processing Error]

The data corresponding to a specific texel (x, y) are extracted from a 4×4 texel block using a relative (x, y) value of

[Math Processing Error]

There are four distinct RGTC image formats described in the following sections.

19.1. BC4 unsigned

Each 4×4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

[Math Processing Error]

The 6 $bits_{\{0..5\}}$ bytes of the block are decoded into a 48-bit bit vector:

[Math Processing Error]

red_0 and red_1 are 8-bit unsigned integers that are unpacked to red values RED_0 and RED_1 by multiplying by [Math Processing Error] .

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x, y) in the block using:

[Math Processing Error]

where *bits*[47] is the most-significant and *bits*[0] is the least-significant bit.

The red value R for a texel at location (x, y) in the block is given by Table 108.

Table 108. Block decoding for BC4

R value	Condition
RED_0	$red_0 > red_1$, $code(x, y) = 0$
RED_1	$red_0 > red_1$, $code(x, y) = 1$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 2$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 3$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 4$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 5$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 6$
[Math Processing Error]	$red_0 > red_1$, $code(x, y) = 7$
RED_0	$red_0 \le red_1$, $code(x, y) = 0$
RED_1	$red_0 \le red_1$, $code(x, y) = 1$
[Math Processing Error]	$red_0 \le red_1$, $code(x, y) = 2$
[Math Processing Error]	$red_0 \le red_1$, $code(x, y) = 3$
[Math Processing Error]	$red_0 \le red_1$, $code(x, y) = 4$

R value	Condition
[Math Processing Error]	$red_0 \le red_1$, $code(x, y) = 5$
RED_{min}	$red_0 \le red_1$, $code(x, y) = 6$
RED_{max}	$red_0 \le red_1$, $code(x, y) = 7$

RED_{min} and RED_{max} are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting *RGBA* value for the texel is (*R*, 0, 0, 1).

19.2. BC4 signed

Each 4×4 block of texels consists of 64 bits of signed red image data. The red values of a texel are extracted in the same way as BC4 unsigned except red_0, red_1, RED_0, RED_1, RED_{min}, and RED_{max} are signed values defined as follows:

[Math Processing Error]

red_0 and red_1 are 8-bit signed (two's complement) integers.

CAVEAT: For signed red_0 and red_1 values: the expressions $red_0 > red_1$ and $red_0 \le red_1$ above are considered undefined (read: may vary by implementation) when red_0 = -127 and red_1 = -128. This is because if red_0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed red-green formats should avoid encoding blocks where red_0 = -127 and red_1 = -128.

19.3. BC5 unsigned

Each 4×4 block of texels consists of 64 bits of compressed unsigned red image data followed by 64 bits of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like BC4 unsigned above. The second 64 bits of compressed green are decoded exactly like BC4 unsigned above except the decoded value *R* for this second block is considered the resulting green value *G*.

Since the decoded texel has a red-green format, the resulting *RGBA* value for the texel is (*R*, *G*, 0, 1).

19.4. BC5 signed

Each 4×4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like BC4 signed above. The second 64 bits of compressed green are decoded exactly like BC4 signed above except the decoded value *R* for this second block is considered the resulting green value *G*.

Since this image has a red-green format, the resulting *RGBA* value is (*R*, *G*, 0, 1).

20. BPTC Compressed Texture Image Formats

This description is derived from the "BPTC Compressed Texture Image Formats" section of the OpenGL 4.4 specification. More information on BC7, BC7 modes and BC6h can be found in Microsoft's online documentation.

Compressed texture images stored using the BPTC compressed image formats are represented as a collection of 4×4 texel blocks, each of which contains 128 bits of texel data stored in little-endian order. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If a BPTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined. When a BPTC image with width *w*, height *h*, and block size *blocksize* (16 bytes) is decoded, the corresponding image size (in bytes) is:

[Math Processing Error]

When decoding a BPTC image, the block containing the texel at offset (*x*, *y*) begins at an offset (in bytes) relative to the base of the image of:

[Math Processing Error]

The data corresponding to a specific texel (*x*, *y*) are extracted from a 4×4 texel block using a relative (*x*, *y*) value of:

[Math Processing Error]

There are two distinct BPTC image formats each of which has two variants. BC7 with or without an sRGB transform function used in the encoding of the *RGB* channels compresses 8-bit unsigned, normalized fixed-point data. BC6H in signed or unsigned form compresses high dynamic range floating-point values. The formats are similar, so the description of the BC6H format will reference significant sections of the BC7 description.

20.1. BC7

Each 4×4 block of texels consists of 128 bits of *RGBA* image data, of which the *RGB* channels may be encoded linearly or with the sRGB transfer function.

Each block contains enough information to select and decode a number of colors called endpoints, pairs of which forms subsets, then to interpolate between those endpoints in a variety of ways, and finally to remap the result into the final output by indexing into these interpolated values according to a partition layout which maps each relative coordinate to a subset.

Each block can contain data in one of eight modes. The mode is identified by the lowest bits of the lowest byte. It is encoded as zero or more zeros followed by a one. For example, using 'x' to indicate a bit not included in the mode number, mode 0 is encoded as xxxxxxx1 in the low byte in binary, mode 5 is xx100000, and mode 7 is 10000000. Encoding the low byte as zero is reserved and should not be used when encoding a BPTC texture; hardware decoders processing a texel block with a low byte of 0 should return 0 for all channels of all texels.

All further decoding is driven by the values derived from the mode listed in Table 109 and Table 110. The fields in the block are always in the same order for all modes. In increasing bit order after the mode, these fields are: partition pattern selection, rotation, index selection, color, alpha, per-endpoint P-bit, shared P-bit, primary indices, and secondary indices. The number of bits to be read in each field is determined directly from these tables, as shown in Table 111.

Per texel block, CB = 3(each of R, G, B)×2(endpoints)×NS(#subsets)×CB(bits/channel/endpoint). AB = 2(endpoints)×NS(#subsets)×AB(bits/endpoint). $\{IB, IB_2\}$ = 16(texels)×{IB,IB_2}(#index bits/texel) - NS(1bit/subset).

Table 109. Mode-dependent BPTC parameters

Mode	NS	PB	RB	ISB	CB	AB	EPB	SPB	IB	IB_2	M	*CB*	*AB*	*EPB*	*SPB*	*IB*	*IB_2*
Bits per...		...texel block			...channel/endpoint		...endpoint	...subset	...texel		Bits per texel block (total)						
0	3	4	0	0	4	0	1	0	3	0	1	72	0	6	0	45	0
1	2	6	0	0	6	0	0	1	3	0	2	72	0	0	2	46	0
2	3	6	0	0	5	0	0	0	2	0	3	90	0	0	0	29	0
3	2	6	0	0	7	0	1	0	2	0	4	84	0	4	0	30	0
4	1	0	2	1	5	6	0	0	2	3	5	30	12	0	0	31	47
5	1	0	2	0	7	8	0	0	2	2	6	42	16	0	0	31	31
6	1	0	0	0	7	7	1	0	4	0	7	42	14	2	0	63	0
7	2	6	0	0	5	5	1	0	2	0	8	60	20	4	0	30	0

Table 110. Full descriptions of the BPTC mode columns

M	Mode identifier bits
NS	Number of subsets
PB	Partition selection bits
RB	Rotation bits
ISB	Index selection bit
CB	Color bits
AB	Alpha bits
EPB	Endpoint P-bits (all channels)
SPB	Shared P-bits
IB	Index bits
IB_2	Secondary index bits

Each block can be divided into between 1 and 3 groups of pixels called *subsets*, which have different endpoints. There are two endpoint colors per subset, grouped first by endpoint, then by subset, then by channel. For example, mode 1, with two subsets and six color bits, would have six bits of red for endpoint 0 of the first subset, then six bits of red for endpoint 1, then the two ends of the second subset, then green and blue stored similarly. If a block has any alpha bits, the alpha data follows the color data with the same organization. If not, alpha is overridden to 255. These bits are treated as the high bits of a fixed-point value in a byte for each color channel of the endpoints: $\{E_R^{7..0}, E_G^{7..0}, E_B^{7..0}, E_A^{7..0}\}$ per endpoint. If the mode has shared P-bits, there are two endpoint bits, the lower of which applies to both endpoints of subset 0 and the upper of which applies to both endpoints of subset 1. If the mode has per-endpoint P-bits, then there are 2 × *subsets* P-bits stored in the same order as color and alpha. Both kinds of P-bits are added as a bit below the color data stored in the byte. So, for mode 1 with six red bits, the P-bit ends up in bit 1. For final scaling, the top bits of the value are replicated into any remaining bits in the byte. For the example of mode 1, bit 7 (which originated as bit 5 of the 6-bit encoded channel) would be replicated to bit 0. Table 112 and Table 113 show the origin of each endpoint color bit for each mode.

Table 111. Bit layout for BC7 modes (LSB..MSB)

Mode 0	*0:* M^0 = 1		*1..4:* $PB^{0..3}$			
	5..8: $R_0^{0..3}$	*9..12:* $R_1^{0..3}$	*13..16:* $R_2^{0..3}$	*17..20:* $R_3^{0..3}$	*21..24:* $R_4^{0..3}$	*25..28:* $R_5^{0..3}$
	29..32: $G_0^{0..3}$	*33..36:* $G_1^{0..3}$	*37..40:* $G_2^{0..3}$	*41..44:* $G_3^{0..3}$	*45..48:* $G_4^{0..3}$	*49..52:* $G_5^{0..3}$
	53..56: $B_0^{0..3}$	*57..60:* $B_1^{0..3}$	*61..64:* $B_2^{0..3}$	*65..68:* $B_3^{0..3}$	*69..72:* $B_4^{0..3}$	*73..76:* $B_5^{0..3}$
	77: EPB_0^0	*78:* EPB_1^0	*79:* EPB_2^0	*80:* EPB_3^0	*81:* EPB_4^0	*82:* EPB_5^0
	83..127: $IB^{0..44}$					
Mode 1	*0..1:* $M^{0..1}$ = 01		*2..7:* $PB^{0..5}$			
	8..13: $R_0^{0..5}$	*14..19:* $R_1^{0..5}$	*20..25:* $R_2^{0..5}$	*26..31:* $R_3^{0..5}$		
	32..37: $G_0^{0..5}$	*38..43:* $G_1^{0..5}$	*44..49:* $G_2^{0..5}$	*50..55:* $G_3^{0..5}$		
	56..61: $B_0^{0..5}$	*62..67:* $B_1^{0..5}$	*68..73:* $B_2^{0..5}$	*74..79:* $B_3^{0..5}$		

	80: SPB_0^0	*81:* SPB_1^0	*82..127:* $IB^{0..45}$			
Mode 2	*0..2:* $\mathbf{M^{0..2} = 001}$		*3..8:* $PB^{0..5}$			
	9..13: $R_0^{0..4}$	*14..18:* $R_1^{0..4}$	*19..23:* $R_2^{0..4}$	*24..28:* $R_4^{0..4}$	*29..33:* $R_4^{0..4}$	*34..38:* $R_5^{0..4}$
	39..43: $G_0^{0..4}$	*44..48:* $G_1^{0..4}$	*49..53:* $G_2^{0..4}$	*54..58:* $G_4^{0..4}$	*59..63:* $G_4^{0..4}$	*64..68:* $G_5^{0..4}$
	69..73: $B_0^{0..4}$	*74..78:* $B_1^{0..4}$	*79..83:* $B_2^{0..4}$	*84..88:* $B_4^{0..4}$	*89..93:* $B_4^{0..4}$	*94..98:* $B_5^{0..4}$
	99..127: $IB^{0..28}$					
Mode 3	*0..3:* $\mathbf{M^{0..3} = 0001}$		*4..9:* $PB^{0..5}$			
	10..16: $R_0^{0..6}$	*17..23:* $R_1^{0..6}$	*24..30:* $R_2^{0..6}$	*31..37:* $R_3^{0..6}$		
	38..44: $G_0^{0..6}$	*45..51:* $G_1^{0..6}$	*52..58:* $G_2^{0..6}$	*59..65:* $G_3^{0..6}$		
	66..72: $B_0^{0..6}$	*73..79:* $B_1^{0..6}$	*80..86:* $B_2^{0..6}$	*87..93:* $B_3^{0..6}$		
	94: EPB_0^0	*95:* EPB_1^0	*96:* EPB_2^0	*97:* EPB_3^0	*98..127:* $IB^{0..29}$	
Mode 4	*0..4:* $\mathbf{M^{0..4} = 00001}$		*5..6:* $RB^{0..1}$		*7:* ISB^0	
	8..12: $R_0^{0..4}$	*13..17:* $R_1^{0..4}$	*18..22:* $G_0^{0..4}$	*23..27:* $G_1^{0..4}$	*28..32:* $B_0^{0..4}$	*33..37:* $B_1^{0..4}$
	38..43: $A_0^{0..5}$	*44..49:* $A_1^{0..5}$	*50..80:* $IB^{0..30}$		*81..127:* $IB_2^{0..46}$	
Mode 5	*0..5:* $\mathbf{M^{0..5} = 000001}$		*6..7:* $RB^{0..1}$			
	8..14: $R_0^{0..6}$	*15..21:* $R_1^{0..6}$	*22..28:* $G_0^{0..6}$	*29..35:* $G_1^{0..6}$	*36..42:* $B_0^{0..6}$	*43..49:* $B_1^{0..6}$
	50..57: $A_0^{0..7}$	*58..65:* $A_1^{0..7}$	*66..96:* $IB^{0..30}$		*97..127:* $IB_2^{0..30}$	
Mode 6	*0..6:* $\mathbf{M^{0..6} = 0000001}$					
	7..13: $R_0^{0..6}$	*14..20:* $R_1^{0..6}$	*21..27:* $G_0^{0..6}$	*28..34:* $G_1^{0..6}$	*35..41:* $B_0^{0..6}$	*42..48:* $B_1^{0..6}$
	49..55: $A_0^{0..6}$	*56..62:* $A_1^{0..6}$	*63:* EPB_0^0	*64:* EPB_1^0	*65..127:* $IB^{0..62}$	
Mode 7	*0..7:* $\mathbf{M^{0..7} = 00000001}$		*8..13:* $PB^{0..5}$			
	14..18: $R_0^{0..4}$	*19..23:* $R_1^{0..4}$	*24..28:* $R_2^{0..4}$	*29..33:* $R_3^{0..4}$		
	34..38: $G_0^{0..4}$	*39..43:* $G_1^{0..4}$	*44..48:* $G_2^{0..4}$	*49..53:* $G_3^{0..4}$		
	54..58: $B_0^{0..4}$	*59..63:* $B_1^{0..4}$	*64..68:* $B_2^{0..4}$	*69..73:* $B_3^{0..4}$		
	74..78: $A_0^{0..4}$	*79..83:* $A_1^{0..4}$	*84..88:* $A_2^{0..4}$	*89..93:* $A_3^{0..4}$		
	94: EPB_0^0	*95:* EPB_1^0	*96:* EPB_2^0	*97:* EPB_3^0	*98..127:* $IB^{0..29}$	

Table 112. Bit sources for BC7 endpoints (modes 0..2, MSB..LSB per channel)

Mode 0																								
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$
8	7	6	5	77	8	7	6	32	31	30	29	77	32	31	30	56	55	54	53	77	56	55	54	*255*
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$
12	11	10	9	78	12	11	10	36	35	34	33	78	36	35	34	60	59	58	57	78	60	59	58	*255*
$E_{R2}^{7..0}$								$E_{G2}^{7..0}$								$E_{B2}^{7..0}$								$E_{A2}^{7..0}$
16	15	14	13	79	16	15	14	40	39	38	37	79	40	39	38	64	63	62	61	79	64	63	62	*255*
$E_{R3}^{7..0}$								$E_{G3}^{7..0}$								$E_{B3}^{7..0}$								$E_{A3}^{7..0}$
20	19	18	17	80	20	19	18	44	43	42	41	80	44	43	42	68	67	66	65	80	68	67	66	*255*
$E_{R4}^{7..0}$								$E_{G4}^{7..0}$								$E_{B4}^{7..0}$								$E_{A4}^{7..0}$
24	23	22	21	81	24	23	22	48	47	46	45	81	48	47	46	72	71	70	69	81	72	71	70	*255*
$E_{R5}^{7..0}$								$E_{G5}^{7..0}$								$E_{B5}^{7..0}$								$E_{A5}^{7..0}$
28	27	26	25	82	28	27	26	52	51	50	49	82	52	51	50	76	75	74	73	82	76	75	74	*255*
Mode 1																								
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$
13	12	11	10	9	8	80	13	37	36	35	34	33	32	80	37	61	60	59	58	57	56	80	61	*255*
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$
19	18	17	16	15	14	80	19	43	42	41	40	39	38	80	43	67	66	65	64	63	62	80	67	*255*
$E_{R2}^{7..0}$								$E_{G2}^{7..0}$								$E_{B2}^{7..0}$								$E_{A2}^{7..0}$
25	24	23	22	21	20	81	25	49	48	47	46	45	44	81	49	73	72	71	70	69	68	81	73	*255*
$E_{R3}^{7..0}$								$E_{G3}^{7..0}$								$E_{B3}^{7..0}$								$E_{A3}^{7..0}$
31	30	29	28	27	26	81	31	55	54	53	52	51	50	81	55	79	78	77	76	75	74	81	79	*255*
Mode 2																								
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$
13	12	11	10	9	13	12	11	43	42	41	40	39	43	42	41	73	72	71	70	69	73	72	71	*255*
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$
18	17	16	15	14	18	17	16	48	47	46	45	44	48	47	46	78	77	76	75	74	78	77	76	*255*
$E_{R2}^{7..0}$								$E_{G2}^{7..0}$								$E_{B2}^{7..0}$								$E_{A2}^{7..0}$
23	22	21	20	19	23	22	21	53	52	51	50	49	53	52	51	83	82	81	80	79	83	82	81	*255*

$E_{R3}^{7..0}$								$E_{G3}^{7..0}$								$E_{B3}^{7..0}$								$E_{A3}^{7..0}$
28	27	26	25	24	28	27	26	58	57	56	55	54	58	57	56	88	87	86	85	84	88	87	86	*255*
$E_{R4}^{7..0}$								$E_{G4}^{7..0}$								$E_{B4}^{7..0}$								$E_{A4}^{7..0}$
33	32	31	30	29	33	32	31	63	62	61	60	59	63	62	61	93	92	91	90	89	93	92	91	*255*
$E_{R5}^{7..0}$								$E_{G5}^{7..0}$								$E_{B5}^{7..0}$								$E_{A5}^{7..0}$
38	37	36	35	34	38	37	36	68	67	66	65	64	68	67	66	98	97	96	95	94	98	97	96	*255*

Table 113. Bit sources for BC7 endpoints (modes 3..7, MSB..LSB per channel)

Mode 3																															
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$							
16	15	14	13	12	11	10	94	44	43	42	41	40	39	38	94	72	71	70	69	68	67	66	94	*255*							
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$							
23	22	21	20	19	18	17	95	51	50	49	48	47	46	45	95	79	78	77	76	75	74	73	95	*255*							
$E_{R2}^{7..0}$								$E_{G2}^{7..0}$								$E_{B2}^{7..0}$								$E_{A2}^{7..0}$							
30	29	28	27	26	25	24	96	58	57	56	55	54	53	52	96	86	85	84	83	82	81	80	96	*255*							
$E_{R3}^{7..0}$								$E_{G3}^{7..0}$								$E_{B3}^{7..0}$								$E_{A3}^{7..0}$							
37	36	35	34	33	32	31	97	65	64	63	62	61	60	59	97	93	92	91	90	89	88	87	97	*255*							
Mode 4																															
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$							
12	11	10	9	8	12	11	10	22	21	20	19	18	22	21	20	32	31	30	29	28	32	31	30	43	42	41	40	39	38	43	42
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$							
17	16	15	14	13	17	16	15	27	26	25	24	23	27	26	25	37	36	35	34	33	37	36	35	49	48	47	46	45	44	49	48
Mode 5																															
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$							
14	13	12	11	10	9	8	14	28	27	26	25	24	23	22	28	42	41	40	39	38	37	36	42	57	56	55	54	53	52	51	50
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$							
21	20	19	18	17	16	15	21	35	34	33	32	31	30	29	35	49	48	47	46	45	44	43	49	65	64	63	62	61	60	59	58
Mode 6																															
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$							
13	12	11	10	9	8	7	63	27	26	25	24	23	22	21	63	41	40	39	38	37	36	35	63	55	54	53	52	51	50	49	63
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$							
20	19	18	17	16	15	14	64	34	33	32	31	30	29	28	64	48	47	46	45	44	43	42	64	62	61	60	59	58	57	56	64
Mode 7																															
$E_{R0}^{7..0}$								$E_{G0}^{7..0}$								$E_{B0}^{7..0}$								$E_{A0}^{7..0}$							
18	17	16	15	14	94	18	17	38	37	36	35	34	94	38	37	58	57	56	55	54	94	58	57	78	77	76	75	74	94	78	77
$E_{R1}^{7..0}$								$E_{G1}^{7..0}$								$E_{B1}^{7..0}$								$E_{A1}^{7..0}$							
23	22	21	20	19	95	23	22	43	42	41	40	39	95	43	42	63	62	61	60	59	95	63	62	83	82	81	80	79	95	83	82
$E_{R2}^{7..0}$								$E_{G2}^{7..0}$								$E_{B2}^{7..0}$								$E_{A2}^{7..0}$							
28	27	26	25	24	96	28	27	48	47	46	45	44	96	48	47	68	67	66	65	64	96	68	67	88	87	86	85	84	96	88	87
$E_{R3}^{7..0}$								$E_{G3}^{7..0}$								$E_{B3}^{7..0}$								$E_{A3}^{7..0}$							
33	32	31	30	29	97	33	32	53	52	51	50	49	97	53	52	73	72	71	70	69	97	73	72	93	92	91	90	89	97	93	92

A texel in a block with one subset is always considered to be in subset zero. Otherwise, a number encoded in the partition bits is used to look up a partition pattern in Table 114 or Table 115 for 2 subsets and 3 subsets respectively. This partition pattern is accessed by the relative *x* and *y* offsets within the block to determine the subset which defines the pixel at these coordinates.

The endpoint colors are interpolated using index values stored in the block. The index bits are stored in y-major order. That is, the bits for the index value corresponding to a relative (*x*, *y*) position of (0, 0) are stored in increasing order in the lowest index bits of the block (but see the next paragraph about anchor indices), the next bits of the block in increasing order store the index bits of (1, 0), followed by (2, 0) and (3, 0), then (0, 1) etc.

Each index has the number of bits indicated by the mode except for one special index per subset called the anchor index. Since the interpolation scheme between endpoints is symmetrical, we can save one bit on one index per subset by ordering the endpoints such that the highest bit for that index is guaranteed to be zero — and not storing that bit.

Each anchor index corresponds to an index in the corresponding partition number in Table 114 or Table 115, and are indicated in bold italics in those tables. In partition zero, the anchor index is always index zero — that is, at a relative position of (0,0) (as can be seen in Table 114 and Table 115, index 0 always corresponds to partition zero). In other partitions, the anchor index is specified by Table 118, Table 116, and Table 117.

In summary, the bit offset for index data with relative *x,y* coordinates within the texel block is:
[Math Processing Error]
where $anchor_2$ is Table 118, $anchor_{3,2}$ is Table 116, $anchor_{3,3}$ is Table 117, and *part* is encoded in the partition selection bits PB.

If secondary index bits are present, they follow the primary index bits and are read in the same manner. The anchor index information is only used to determine the number of bits each index has when read from the block data.

The endpoint color and alpha values used for final interpolation are the decoded values corresponding to the applicable subset as selected above. The index value for interpolating color comes from the secondary index bits for the texel if the mode has an index selection bit and its value is one, and from the primary index bits otherwise. The alpha index comes from the secondary index bits if the block has a secondary index and the block either doesn't have an index selection bit or that bit is zero, and from the primary index bits otherwise.

As an example of the texel decode process, consider a block encoded with mode 2—that is, $M^0 = 0$, $M^1 = 0$, $M^2 = 1$. This mode has three subsets, so Table 115 is used to determine which subset applies to each texel. Let us assume that this block has partition pattern 6 encoded in the partition selection bits, and that we wish to decode the texel at relative (*x*, *y*) offset (1, 2)—that is, index 9 in y-major order. We can see from Table 115 that this texel is partitioned into subset 1 (the second of three), and therefore by endpoints 2 and 3. Mode 2 stores two index bits per texel, except for index 0 (which is the anchor index for subset 0), index 15 (for subset 1, as indicated in Table 116) and index 3 (for subset 2, as indicated in Table 117). Index 9 is therefore stored in two bits starting at index bits offset 14 (for indices 1..2 and 4..8) plus 2 (for indices 0 and 3)—a total of 16 bit offset into the index bits or, as seen in Table 111, bits 115 and 116 of the block. These two bits are used to interpolate between endpoints 2 and 3 using Equation 2 with weights from the two-bit index row of Table 119, as described below.

Table 114. Partition table for 2-subset BPTC, with the 4×4 block of values for each partition number

0				1				2				3				4				5				6				7			
0	0	1	1	**0**	0	0	1	**0**	1	1	1	**0**	0	0	1	**0**	0	0	0	**0**	0	1	1	**0**	0	0	1	**0**	0	0	0
0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1
0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1	0	1	1	1	0	1	1	1	0	0	1	1
0	0	1	**1**	0	0	0	**1**	0	1	1	**1**	0	1	1	**1**	0	0	1	**1**	1	1	1	**1**	1	1	1	**1**	0	1	1	**1**

8				9				10				11				12				13				14				15			
0	0	0	0	**0**	0	1	1	**0**	0	0	0	**0**	0	0	0	**0**	0	0	1	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0
0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	**1**	1	1	1	**1**	1	1	1	**1**	0	1	1	**1**	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**	1	1	1	**1**

16				17				18				19				20				21				22				23			
0	0	0	0	**0**	1	**1**	1	**0**	0	0	0	**0**	1	**1**	1	**0**	0	**1**	1	**0**	0	0	0	**0**	0	0	0	**0**	1	**1**	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
1	1	1	0	0	0	0	0	**1**	0	0	0	0	0	0	1	0	0	0	0	**1**	1	0	0	**1**	0	0	0	0	0	1	1
1	1	1	**1**	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	**1**

24				25				26				27				28				29				30				31			
0	0	**1**	1	**0**	0	0	0	**0**	1	**1**	0	**0**	0	**1**	1	**0**	0	0	1	**0**	0	0	0	**0**	1	**1**	1	**0**	0	**1**	1
0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1
0	0	0	1	**1**	0	0	0	0	1	1	0	0	1	1	0	**1**	1	1	0	**1**	1	1	1	1	0	0	0	1	0	0	1
0	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0

32				33				34				35				36				37				38				39			
0	1	0	1	**0**	0	0	0	**0**	1	0	1	**0**	0	**1**	1	**0**	0	**1**	1	**0**	1	0	1	**0**	1	1	0	**0**	1	0	1
0	1	0	1	1	1	1	1	1	0	**1**	0	0	0	1	1	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0
0	1	0	1	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	**1**	0	1	0	0	1	1	0	1	0	1	0
0	1	0	**1**	1	1	1	**1**	1	0	1	0	1	1	0	0	1	1	0	0	1	0	1	0	1	0	0	**1**	0	1	0	**1**

40				41				42				43				44				45				46				47			
0	1	**1**	1	**0**	0	0	1	**0**	0	**1**	1	**0**	0	**1**	1	**0**	1	**1**	0	**0**	0	1	1	**0**	1	1	0	**0**	0	0	0
0	0	1	1	0	0	1	1	0	0	1	0	1	0	1	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	**1**	0
1	1	0	0	**1**	1	0	0	0	1	0	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0	0	1	0	1	1	0
1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	0	1	**1**	1	0	0	**1**	0	0	0	0

48				49				50				51				52				53				54				55			
0	1	0	0	**0**	0	**1**	0	**0**	0	0	0	**0**	0	0	0	**0**	1	1	0	**0**	0	1	1	**0**	1	**1**	0	**0**	0	**1**	1
1	1	**1**	0	0	1	1	1	0	0	**1**	0	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1
0	1	0	0	0	0	1	0	0	1	1	1	**1**	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	**1**	1	0	0	**1**	1	1	0	0	0	1	1	0

56				57				58				59				60				61				62				63			
0	1	1	0	**0**	1	1	0	**0**	1	1	1	**0**	0	0	1	**0**	0	0	0	**0**	0	**1**	1	**0**	0	**1**	0	**0**	1	0	0
1	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0	1	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0
1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1
1	0	0	**1**	1	0	0	**1**	0	0	0	**1**	0	1	1	**1**	0	0	1	**1**	0	0	0	0	1	1	1	0	0	1	1	**1**

Table 115. Partition table for 3-subset BPTC, with the 4×4 block of values for each partition number

0				1				2				3				4				5				6				7			
0	0	1	**1**	**0**	0	0	**1**	**0**	0	0	0	**0**	2	2	**2**	**0**	0	0	0	**0**	0	1	**1**	**0**	0	2	**2**	**0**	0	1	1
0	0	1	1	0	0	1	1	2	0	0	1	0	0	2	2	0	0	0	0	0	0	1	1	0	0	2	2	0	0	1	1
0	2	2	1	**2**	2	1	1	**2**	2	1	1	0	0	1	1	**1**	1	2	2	0	0	2	2	1	1	1	1	**2**	2	1	1
2	2	2	**2**	2	2	2	**1**	2	2	1	**1**	0	1	1	**1**	1	1	2	**2**	0	0	2	**2**	1	1	1	**1**	2	2	1	**1**

8				9				10				11				12				13				14				15			
0	0	0	0	**0**	0	0	0	**0**	0	0	0	**0**	0	1	2	**0**	1	1	2	**0**	1	2	2	**0**	0	1	**1**	**0**	0	1	**1**
0	0	0	0	1	1	1	1	1	1	**1**	1	0	0	**1**	2	0	1	**1**	2	0	**1**	2	2	0	1	1	2	2	0	0	1
1	1	1	1	**1**	1	1	1	2	2	2	2	0	0	1	2	0	1	1	2	0	1	2	2	1	1	2	2	**2**	2	0	0

2	2	2	2	2	2	2	2	2	2	2	2	0	0	1	2	0	1	1	2	0	1	2	2	1	2	2	2	2	2	2	0
16				17				18				19				20				21				22				23			
0	0	0	1	0	1	1	1	0	0	0	0	0	0	2	2	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	1	2	2	0	0	2	2	0	1	1	1	0	0	0	1	0	0	1	1	1	1	0	0
0	1	1	2	2	0	0	1	1	1	2	2	0	0	2	2	0	2	2	2	2	2	2	1	0	1	2	2	2	2	1	0
1	1	2	2	2	2	0	0	1	1	2	2	1	1	1	1	0	2	2	2	2	2	2	1	0	1	2	2	2	2	1	0
24				25				26				27				28				29				30				31			
0	1	2	2	0	0	1	2	0	1	1	0	0	0	0	0	0	0	2	2	0	1	1	0	0	0	1	1	0	0	0	0
0	1	2	2	0	0	1	2	1	2	2	1	0	1	1	0	1	1	0	2	0	1	1	0	0	1	2	2	2	0	0	0
0	0	1	1	1	1	2	2	1	2	2	1	1	2	2	1	1	1	0	2	2	0	0	2	0	1	2	2	2	2	1	1
0	0	0	0	2	2	2	2	0	1	1	0	1	2	2	1	0	0	2	2	2	2	2	2	0	0	1	1	2	2	2	1
32				33				34				35				36				37				38				39			
0	0	0	0	0	2	2	2	0	0	1	1	0	1	2	0	0	0	0	0	0	1	2	0	0	1	2	0	0	0	1	1
0	0	0	2	0	0	2	2	0	0	1	2	0	1	2	0	1	1	1	1	1	2	0	1	2	0	1	2	2	2	0	0
1	1	2	2	0	0	1	2	0	0	2	2	0	1	2	0	2	2	2	2	2	0	1	2	1	2	0	1	1	1	2	2
1	2	2	2	0	0	1	1	0	2	2	2	0	1	2	0	0	0	0	0	0	1	2	0	0	1	2	0	0	0	1	1
40				41				42				43				44				45				46				47			
0	0	1	1	0	1	0	1	0	0	0	0	0	0	2	2	0	0	2	2	0	2	2	0	0	1	0	1	0	0	0	0
1	1	2	2	0	1	0	1	0	0	0	0	1	1	2	2	0	0	1	1	1	2	2	1	2	2	2	2	2	1	2	1
2	2	0	0	2	2	2	2	2	1	2	1	0	0	2	2	0	0	2	2	0	2	2	0	2	2	2	2	2	1	2	1
0	0	1	1	2	2	2	2	2	1	2	1	1	1	2	2	0	0	1	1	1	2	2	1	0	1	0	1	2	1	2	1
48				49				50				51				52				53				54				55			
0	1	0	1	0	2	2	2	0	0	0	2	0	0	0	0	0	2	2	2	0	0	0	2	0	1	1	0	0	0	0	0
0	1	0	1	0	1	1	1	1	1	1	2	2	1	1	2	0	1	1	1	1	1	1	2	0	1	1	0	0	0	0	0
0	1	0	1	0	2	2	2	0	0	0	2	2	1	1	2	0	1	1	1	1	1	1	2	0	1	1	0	2	1	1	2
2	2	2	2	0	1	1	1	1	1	1	2	2	1	1	2	0	2	2	2	0	0	0	2	2	2	2	2	2	1	1	2
56				57				58				59				60				61				62				63			
0	1	1	0	0	0	2	2	0	0	2	2	0	0	0	0	0	0	0	2	0	2	2	2	0	1	0	1	0	1	1	1
0	1	1	0	0	0	1	1	1	1	2	2	0	0	0	0	0	0	0	1	1	2	2	2	2	2	2	2	2	0	1	1
2	2	2	2	0	0	1	1	1	1	2	2	0	0	0	0	0	0	0	2	0	2	2	2	2	2	2	2	2	0	1	1
2	2	2	2	0	0	2	2	0	0	2	2	2	1	1	2	0	0	0	1	1	2	2	2	2	2	2	2	2	2	2	0

Table 116. BPTC anchor index values for the second subset of three-subset partitioning, by partition number

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

Table 117. BPTC anchor index values for the third subset of three-subset partitioning, by partition number

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

Table 118. BPTC anchor index values for the second subset of two-subset partitioning, by partition number

0	1	2	3	4	5	6	7
15	15	15	15	15	15	15	15
8	9	10	11	12	13	14	15
15	15	15	15	15	15	15	15

16	17	18	19	20	21	22	23
15	2	8	2	2	8	8	15
24	25	26	27	28	29	30	31
2	8	2	2	8	8	2	2
32	33	34	35	36	37	38	39
15	15	6	8	2	8	15	15
40	41	42	43	44	45	46	47
2	8	2	2	2	15	15	6
48	49	50	51	52	53	54	55
6	2	6	8	15	15	2	2
56	57	58	59	60	61	62	63
15	15	15	15	15	2	2	15

Interpolation is always performed using a 6-bit interpolation factor. The effective interpolation factors for 2-, 3-, and 4-bit indices are given in Table 119.

Table 119. BPTC interpolation factors

2	Index	0				1				2				3			
	Weight	0				21				43				64			
3	Index	0		1		2		3		4		5		6		7	
	Weight	0		9		18		27		37		46		55		64	
4	Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

Given E_0 and E_1, unsigned integer endpoints [0 .. 255] for each channel and *weight* as an unsigned integer interpolation factor from Table 119:

Equation 2. BPTC endpoint interpolation formula

[Math Processing Error]

where *[Math Processing Error]* performs a (truncating) bitwise right-shift, and *interpolated value* is an (unsigned) integer in the range [0..255]. The interpolation results in an *RGBA* color. If rotation bits are present, this interpolated color is remapped according to Table 120.

Table 120. BPTC Rotation bits

0	no change
1	*swap*(*A*, *R*)
2	*swap*(*A*, *G*)
3	*swap*(*A*, *B*)

These 8-bit values should be interpreted as *RGBA* 8-bit normalized channels, either linearly encoded (by multiplying by *[Math Processing Error]*) or with the sRGB transfer function.

20.2. BC6H

Each 4×4 block of texels consists of 128 bits of *RGB* data. The signed and unsigned formats are very similar and will be described together. In the description and pseudocode below, *signed* will be used as a condition which is true for the signed version of the format and false for the unsigned version of the format. Both formats only contain *RGB* data, so the returned alpha value is 1.0. If a block uses a reserved or invalid encoding, the return value is (0.0, 0.0, 0.0, 1.0).

note Where BC7 encodes a fixed-point 8-bit value, BC6H encodes a 16-bit integer which will be interpreted as a 16-bit half float. Interpolation in BC6H is therefore nonlinear, but monotonic.

Each block can contain data in one of 14 modes. The mode number is encoded in either the low two bits or the low five bits. If the low two bits are less than two, that is the mode number, otherwise the low five bits is the mode number. Mode numbers not listed in Table 121 (19, 23, 27, and 31) are reserved.

Table 121. Endpoint and partition parameters for BPTC block modes

Mode number	Transformed endpoints	Partition bits (PB)	Endpoint bits (EPB)	Delta bits	Mode	Endpoint	Delta
		Bits per texel block	{*R*,*G*,*B*} bits per endpoint		Bits per texel block (total)		
0	✓	5	{10, 10, 10}	{5, 5, 5}	2	30	45
1	✓	5	{7, 7, 7}	{6, 6, 6}	2	21	54
2	✓	5	{11, 11, 11}	{5, 4, 4}	5	33	39
6	✓	5	{11, 11, 11}	{4, 5, 4}	5	33	39
10	✓	5	{11, 11, 11}	{4, 4, 5}	5	33	39
14	✓	5	{9, 9, 9}	{5, 5, 5}	5	27	45
18	✓	5	{8, 8, 8}	{6, 5, 5}	5	24	48
22	✓	5	{8, 8, 8}	{5, 6, 5}	5	24	48
26	✓	5	{8, 8, 8}	{5, 5, 6}	5	24	48
30		5	{6, 6, 6}	-	5	72	0

3		0	{10, 10, 10}	-	5	60	0
7	✓	0	{11, 11, 11}	{9, 9, 9}	5	33	27
11	✓	0	{12, 12, 12}	{8, 8, 8}	5	36	24
15	✓	0	{16, 16, 16}	{4, 4, 4}	5	48	12

The data for the compressed blocks is stored in a different manner for each mode. The interpretation of bits for each mode are specified in Table 122. The descriptions are intended to be read from left to right with the LSB on the left. Each element is of the form $v^{a..b}$. If $a \geq b$, this indicates extracting $b - a + 1$ bits from the block at that location and put them in the corresponding bits of the variable v. If $a < b$, then the bits are reversed. v^a is used as a shorthand for the one bit $v^{a..a}$. As an example, $M^{1..0}$, G_2^4 would move the low two bits from the block into the low two bits of mode number M, then the next bit of the block into bit 4 of G_2. The resultant bit interpretations are shown explicitly in Table 123 and Table 124. The variable names given in the table will be referred to in the language below.

Subsets and indices work in much the same way as described for the BC7 formats above. If a float block has no partition bits, then it is a single-subset block. If it has partition bits, then it is a two-subset block. The partition number references the first half of Table 114.

Table 122. Block descriptions for BC6H block modes (LSB..MSB)

Mode Number	Block description
0	$M^{1..0}$, G_2^4, B_2^4, B_3^4, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{4..0}$, G_3^4, $G_2^{3..0}$, $G_1^{4..0}$, B_3^0, $G_3^{3..0}$, $B_1^{4..0}$, B_3^1, $B_2^{3..0}$, $R_2^{4..0}$, B_3^2, $R_3^{4..0}$, B_3^3, $PB^{4..0}$
1	$M^{1..0}$, G_2^5, G_3^4, G_3^5, $R_0^{6..0}$, B_3^0, B_3^1, B_2^4, $G_0^{6..0}$, B_2^5, B_3^2, G_2^4, $B_0^{6..0}$, B_3^3, B_3^5, B_3^4, $R_1^{5..0}$, $G_2^{3..0}$, $G_1^{5..0}$, $G_3^{3..0}$, $B_1^{5..0}$, $B_2^{3..0}$, $R_2^{5..0}$, $R_3^{5..0}$, $PB^{4..0}$
2	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{4..0}$, R_0^{10}, $G_2^{3..0}$, $G_1^{3..0}$, G_0^{10}, B_3^0, $G_3^{3..0}$, $B_1^{3..0}$, B_0^{10}, B_3^1, $B_2^{3..0}$, $R_2^{4..0}$, B_3^2, $R_3^{4..0}$, B_3^3, $PB^{4..0}$
6	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{3..0}$, R_0^{10}, G_3^4, $G_2^{3..0}$, $G_1^{4..0}$, G_0^{10}, $G_3^{3..0}$, $B_1^{3..0}$, B_0^{10}, B_3^1, $B_2^{3..0}$, $R_2^{3..0}$, B_3^0, B_3^2, $R_3^{3..0}$, G_2^4, B_3^3, $PB^{4..0}$
10	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{3..0}$, R_0^{10}, B_2^4, $G_2^{3..0}$, $G_1^{3..0}$, G_0^{10}, B_3^0, $G_3^{3..0}$, $B_1^{4..0}$, B_0^{10}, $B_2^{3..0}$, $R_2^{3..0}$, B_3^1, B_3^2, $R_3^{3..0}$, B_3^4, B_3^3, $PB^{4..0}$
14	$M^{4..0}$, $R_0^{8..0}$, B_2^4, $G_0^{8..0}$, G_2^4, $B_0^{8..0}$, B_3^4, $R_1^{4..0}$, G_3^4, $G_2^{3..0}$, $G_1^{4..0}$, B_3^0, $G_3^{3..0}$, $B_1^{4..0}$, B_3^1, $B_2^{3..0}$, $R_2^{4..0}$, B_3^2, $R_3^{4..0}$, B_3^3, $PB^{4..0}$
18	$M^{4..0}$, $R_0^{7..0}$, G_3^4, B_2^4, $G_0^{7..0}$, B_3^2, G_2^4, $B_0^{7..0}$, B_3^3, B_3^4, $R_1^{5..0}$, $G_2^{3..0}$, $G_1^{4..0}$, B_3^0, $G_3^{3..0}$, $B_1^{4..0}$, B_3^1, $B_2^{3..0}$, $R_2^{5..0}$, $R_3^{5..0}$, $PB^{4..0}$
22	$M^{4..0}$, $R_0^{7..0}$, B_3^0, B_2^4, $G_0^{7..0}$, G_2^5, G_2^4, $B_0^{7..0}$, G_3^5, B_3^4, $R_1^{4..0}$, G_3^4, $G_2^{3..0}$, $G_1^{5..0}$, $G_3^{3..0}$, $B_1^{4..0}$, B_3^1, $B_2^{3..0}$, $R_2^{4..0}$, B_3^2, $R_3^{4..0}$, B_3^3, $PB^{4..0}$
26	$M^{4..0}$, $R_0^{7..0}$, B_3^1, B_2^4, $G_0^{7..0}$, B_2^5, G_2^4, $B_0^{7..0}$, B_3^5, B_3^4, $R_1^{4..0}$, G_3^4, $G_2^{3..0}$, $G_1^{4..0}$, B_3^0, $G_3^{3..0}$, $B_1^{5..0}$, $B_2^{3..0}$, $R_2^{4..0}$, B_3^2, $R_3^{4..0}$, B_3^3, $PB^{4..0}$
30	$M^{4..0}$, $R_0^{5..0}$, G_3^4, B_3^0, B_3^1, B_2^4, $G_0^{5..0}$, G_2^5, B_2^5, B_3^2, G_2^4, $B_0^{5..0}$, G_3^5, B_3^3, B_3^5, B_3^4, $R_1^{5..0}$, $G_2^{3..0}$, $G_1^{5..0}$, $G_3^{3..0}$, $B_1^{5..0}$, $B_2^{3..0}$, $R_2^{5..0}$, $R_3^{5..0}$, $PB^{4..0}$
3	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{9..0}$, $G_1^{9..0}$, $B_1^{9..0}$
7	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{8..0}$, R_0^{10}, $G_1^{8..0}$, G_0^{10}, $B_1^{8..0}$, B_0^{10}
11	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{7..0}$, $R_0^{10..11}$, $G_1^{7..0}$, $G_0^{10..11}$, $B_1^{7..0}$, $B_0^{10..11}$
15	$M^{4..0}$, $R_0^{9..0}$, $G_0^{9..0}$, $B_0^{9..0}$, $R_1^{3..0}$, $R_0^{10..15}$, $G_1^{3..0}$, $G_0^{10..15}$, $B_1^{3..0}$, $B_0^{10..15}$

Indices are read in the same way as the BC7 formats including obeying the anchor values for index 0 and as needed by Table 118. That is, for modes with only one partition, the mode and endpoint data are followed by 63 bits of index data (four index bits $IB_{x,y}^{0..3}$ per texel, with one implicit bit for $IB_{x,y}^3$) starting at bit 65 with $IB_{0,0}^0$. For modes with two partitions, the mode, endpoint and partition data are followed by 46 bits of index data (three per texel $IB_{x,y}^{0..2}$, with two implicit bits, one for partition 0 at $IB_{0,0}^2$ and one $IB_{x,y}^2$ bit for partition 1 at an offset determined by the partition pattern selected) starting at bit 82 with $IB_{0,0}^0$. In both cases, index bits are stored in y-major offset order by increasing little-endian bit number, with the bits for each index stored consecutively:

[Math Processing Error]

Table 123. Interpretation of lower bits for BC6H block modes

	Mode													
Bit	**0**	**1**	**2**	**6**	**10**	**14**	**18**	**22**	**26**	**30**	**3**	**7**	**11**	**15**
0	M^0: **0**	M^0: **1**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **0**	M^0: **1**	M^0: **1**	M^0: **1**	M^0: **1**
1	M^1: **0**	M^1: **0**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**	M^1: **1**
2	G_2^4	G_2^5	M^2: **0**	M^2: **1**	M^2: **0**	M^2: **1**	M^2: **0**	M^2: **1**	M^2: **0**	M^2: **1**	M^2: **0**	M^2: **1**	M^2: **0**	M^2: **1**
3	B_2^4	G_3^4	M^3: **0**	M^3: **0**	M^3: **1**	M^3: **1**	M^3: **0**	M^3: **0**	M^3: **1**	M^3: **1**	M^3: **0**	M^3: **0**	M^3: **1**	M^3: **1**

4	B_3^4	G_3^5	M^4: **0**	M^4: **0**	M^4: **0**	M^4: **0**	M^4: **1**	M^4: **1**	M^4: **1**	M^4: **1**	M^4: **0**	M^4: **0**	M^4: **0**	M^4: **0**
5	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0	R_0^0
6	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1	R_0^1
7	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2	R_0^2
8	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3	R_0^3
9	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4	R_0^4
10	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5	R_0^5
11	R_0^6	R_0^6	R_0^6	R_0^6	R_0^6	R_0^6	R_0^6	R_0^6	R_0^6	G_3^4	R_0^6	R_0^6	R_0^6	R_0^6
12	R_0^7	B_3^0	R_0^7	R_0^7	R_0^7	R_0^7	R_0^7	R_0^7	R_0^7	B_3^0	R_0^7	R_0^7	R_0^7	R_0^7
13	R_0^8	B_3^1	R_0^8	R_0^8	R_0^8	R_0^8	G_3^4	B_3^0	B_3^1	B_3^1	R_0^8	R_0^8	R_0^8	R_0^8
14	R_0^9	B_2^4	R_0^9	R_0^9	R_0^9	B_2^4	B_2^4	B_2^4	B_2^4	B_2^4	R_0^9	R_0^9	R_0^9	R_0^9
15	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0	G_0^0
16	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1	G_0^1
17	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2	G_0^2
18	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3	G_0^3
19	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4	G_0^4
20	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5	G_0^5
21	G_0^6	G_0^6	G_0^6	G_0^6	G_0^6	G_0^6	G_0^6	G_0^6	G_0^6	G_2^5	G_0^6	G_0^6	G_0^6	G_0^6
22	G_0^7	B_2^5	G_0^7	G_0^7	G_0^7	G_0^7	G_0^7	G_0^7	G_0^7	B_2^5	G_0^7	G_0^7	G_0^7	G_0^7
23	G_0^8	B_3^2	G_0^8	G_0^8	G_0^8	G_0^8	B_3^2	G_2^5	B_2^5	B_3^2	G_0^8	G_0^8	G_0^8	G_0^8
24	G_0^9	G_2^4	G_0^9	G_0^9	G_0^9	G_2^4	G_2^4	G_2^4	G_2^4	G_2^4	G_0^9	G_0^9	G_0^9	G_0^9
25	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0	B_0^0
26	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1	B_0^1
27	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2	B_0^2
28	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3	B_0^3
29	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4	B_0^4
30	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5	B_0^5
31	B_0^6	B_0^6	B_0^6	B_0^6	B_0^6	B_0^6	B_0^6	B_0^6	B_0^6	G_3^5	B_0^6	B_0^6	B_0^6	B_0^6
32	B_0^7	B_3^3	B_0^7	B_0^7	B_0^7	B_0^7	B_0^7	B_0^7	B_0^7	B_3^3	B_0^7	B_0^7	B_0^7	B_0^7
33	B_0^8	B_3^5	B_0^8	B_0^8	B_0^8	B_0^8	B_3^3	G_3^5	B_3^5	B_3^5	B_0^8	B_0^8	B_0^8	B_0^8
34	B_0^9	B_3^4	B_0^9	B_0^9	B_0^9	B_3^4	B_3^4	B_3^4	B_3^4	B_3^4	B_0^9	B_0^9	B_0^9	B_0^9
35	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0	R_1^0
36	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1	R_1^1
37	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2	R_1^2
38	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3	R_1^3
39	R_1^4	R_1^4	R_1^4	R_0^{10}	R_0^{10}	R_1^4	R_1^4	R_1^4	R_1^4	R_1^4	R_1^4	R_1^4	R_1^4	R_0^{15}
40	G_3^4	R_1^5	R_0^{10}	G_3^4	B_2^4	G_3^4	R_1^5	G_3^4	G_3^4	R_1^5	R_1^5	R_1^5	R_1^5	R_0^{14}

Table 124. Interpretation of upper bits for BC6H block modes

	Mode													
Bit	**0**	**1**	**2**	**6**	**10**	**14**	**18**	**22**	**26**	**30**	**3**	**7**	**11**	**15**
41	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	G_2^0	R_1^6	R_1^6	R_1^6	R_0^{13}
42	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	G_2^1	R_1^7	R_1^7	R_1^7	R_0^{12}
43	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	G_2^2	R_1^8	R_1^8	R_0^{11}	R_0^{11}
44	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	G_2^3	R_1^9	R_0^{10}	R_0^{10}	R_0^{10}
45	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0	G_1^0
46	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1	G_1^1
47	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2	G_1^2
48	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3	G_1^3
49	G_1^4	G_1^4	G_0^{10}	G_1^4	G_0^{10}	G_1^4	G_1^4	G_1^4	G_1^4	G_1^4	G_1^4	G_1^4	G_1^4	G_0^{15}
50	B_3^0	G_1^5	B_3^0	G_0^{10}	B_3^0	B_3^0	B_3^0	G_1^5	B_3^0	G_1^5	G_1^5	G_1^5	G_1^5	G_0^{14}
51	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_3^0	G_1^6	G_1^6	G_1^6	G_0^{13}
52	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_3^1	G_1^7	G_1^7	G_1^7	G_0^{12}

53	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_3^2	G_1^8	G_1^8	G_0^{11}	G_0^{11}
54	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_3^3	G_1^9	G_0^{10}	G_0^{10}	G_0^{10}
55	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0	B_1^0
56	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1	B_1^1
57	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2	B_1^2
58	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3	B_1^3
59	B_1^4	B_1^4	B_0^{10}	B_0^{10}	B_1^4	B_1^4	B_1^4	B_1^4	B_1^4	B_1^4	B_1^4	B_1^4	B_1^4	B_0^{15}
60	B_3^1	B_1^5	B_3^1	B_3^1	B_0^{10}	B_3^1	B_3^1	B_3^1	B_1^5	B_1^5	B_1^5	B_1^5	B_1^5	B_0^{14}
61	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_2^0	B_1^6	B_1^6	B_1^6	B_0^{13}
62	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_2^1	B_1^7	B_1^7	B_1^7	B_0^{12}
63	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_2^2	B_1^8	B_1^8	B_0^{11}	B_0^{11}
64	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_2^3	B_1^9	B_0^{10}	B_0^{10}	B_0^{10}
65	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	R_2^0	$IB_{0,0}^0$	$IB_{0,0}^0$	$IB_{0,0}^0$	$IB_{0,0}^0$
66	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	R_2^1	$IB_{0,0}^1$	$IB_{0,0}^1$	$IB_{0,0}^1$	$IB_{0,0}^1$
67	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	R_2^2	$IB_{0,0}^2$	$IB_{0,0}^2$	$IB_{0,0}^2$	$IB_{0,0}^2$
68	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	R_2^3	$IB_{1,0}^0$	$IB_{1,0}^0$	$IB_{1,0}^0$	$IB_{1,0}^0$
69	R_2^4	R_2^4	R_2^4	B_3^0	B_3^1	R_2^4	R_2^4	R_2^4	R_2^4	R_2^4	$IB_{1,0}^1$	$IB_{1,0}^1$	$IB_{1,0}^1$	$IB_{1,0}^1$
70	B_3^2	R_2^5	B_3^2	B_3^2	B_3^2	B_3^2	R_2^5	B_3^2	B_3^2	R_2^5	$IB_{1,0}^2$	$IB_{1,0}^2$	$IB_{1,0}^2$	$IB_{1,0}^2$
71	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	R_3^0	$IB_{1,0}^3$	$IB_{1,0}^3$	$IB_{1,0}^3$	$IB_{1,0}^3$
72	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	R_3^1	$IB_{2,0}^0$	$IB_{2,0}^0$	$IB_{2,0}^0$	$IB_{2,0}^0$
73	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	R_3^2	$IB_{2,0}^1$	$IB_{2,0}^1$	$IB_{2,0}^1$	$IB_{2,0}^1$
74	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	R_3^3	$IB_{2,0}^2$	$IB_{2,0}^2$	$IB_{2,0}^2$	$IB_{2,0}^2$
75	R_3^4	R_3^4	R_3^4	G_2^4	B_3^4	R_3^4	R_3^4	R_3^4	R_3^4	R_3^4	$IB_{2,0}^3$	$IB_{2,0}^3$	$IB_{2,0}^3$	$IB_{2,0}^3$
76	B_3^3	R_3^5	B_3^3	B_3^3	B_3^3	B_3^3	R_3^5	B_3^3	B_3^3	R_3^5	$IB_{3,0}^0$	$IB_{3,0}^0$	$IB_{3,0}^0$	$IB_{3,0}^0$
77	PB^0	PB^0	PB^0	PB^0	PB^0	PB^0	PB^0	PB^0	PB^0	PB^0	$IB_{3,0}^1$	$IB_{3,0}^1$	$IB_{3,0}^1$	$IB_{3,0}^1$
78	PB^1	PB^1	PB^1	PB^1	PB^1	PB^1	PB^1	PB^1	PB^1	PB^1	$IB_{3,0}^2$	$IB_{3,0}^2$	$IB_{3,0}^2$	$IB_{3,0}^2$
79	PB^2	PB^2	PB^2	PB^2	PB^2	PB^2	PB^2	PB^2	PB^2	PB^2	$IB_{3,0}^3$	$IB_{3,0}^3$	$IB_{3,0}^3$	$IB_{3,0}^3$
80	PB^3	PB^3	PB^3	PB^3	PB^3	PB^3	PB^3	PB^3	PB^3	PB^3	$IB_{0,1}^0$	$IB_{0,1}^0$	$IB_{0,1}^0$	$IB_{0,1}^0$
81	PB^4	PB^4	PB^4	PB^4	PB^4	PB^4	PB^4	PB^4	PB^4	PB^4	$IB_{0,1}^1$	$IB_{0,1}^1$	$IB_{0,1}^1$	$IB_{0,1}^1$

Table 123 and Table 124 show bits 0..81 for each mode. Since modes 3, 7, 11 and 15 each have only one partition, only the first index is an anchor index, and there is a fixed mapping between texels and index bits. These modes also have four index bits $IB_{x,y}^{0..3}$ per texel (except for the anchor index), and these pixel indices start at bit 65 with $IB_{0,0}^0$. The interpretation of bits 82 and later is not tabulated. For modes with two partitions, the mapping from index bits $IB_{x,y}$ to coordinates depends on the choice of anchor index for the secondary partition (determined by the pattern selected by the partition bits $PB^{4..0}$), and is therefore not uniquely defined by the mode — and not useful to tabulate in this form.

In a single-subset blocks, the two endpoints are contained in R_0, G_0, B_0 (collectively referred to as E_0) and R_1, G_1, B_1 (collectively E_1). In a two-subset block, the endpoints for the second subset are in R_2, G_2, B_2 and R_3, G_3, B_3 (collectively E_2 and E_3 respectively). The values in E_0 are sign-extended to the implementation's internal integer representation if the format of the texture is signed. The values in E_1 (and E_2 and E_3 if the block has two subsets) are sign-extended if the format of the texture is signed or if the block mode has transformed endpoints. If the mode has transformed endpoints, the values from E_0 are used as a base to offset all other endpoints, wrapped at the number of endpoint bits. For example, R_1 = (R_0 + R_1) & *[Math Processing Error]* .

In BC7, all modes represent endpoint values independently. This means it is always possible to represent the endpoints nearest to the anchor indices by choosing the endpoint order appropriately. Since in BC6H transformed endpoints are represented as two's complement offsets relative to the first endpoint, there is an asymmetry: it is possible to represent larger negative values in two's complement than positive values, so E_1, E_2 and E_3 can be more distant from E_0 in a negative direction than positive in modes with transformed endpoints. This means that endpoints cannot necessarily be chosen independently of the anchor index in BC6H, since the order of endpoints cannot necessarily be reversed. In addition, E_2 and E_3 always depends on E_0, so swapping E_0 and E_1 to suit the anchor bit for the first subset may make make the relative offsets of E_2 and E_3 unrepresentable in a given mode if they fall out of range.

Next, the endpoints are unquantized to maximize the usage of the bits and to ensure that the negative ranges are oriented properly to interpolate as a two's complement value. The following pseudocode assumes the computation uses sufficiently large intermediate values to avoid overflow. For the unsigned float format, we unquantize a value *x* to *unq* by:

```
if (EPB >= 15)
    unq = x;
```

```
else if (x == 0)
    unq = 0;
else if (x == ((1 << EPB)-1))
    unq = 0xFFFF;
else
    unq = ((x << 15) + 0x4000) >> (EPB-1);
```

The signed float unquantization is similar, but needs to worry about orienting the negative range:

```
s = 0;
if (EPB >= 16) {
    unq = x;
} else {
    if (x < 0) {
        s = 1;
        x = -x;
    }

    if (x == 0)
        unq = 0;
    else if (x >= ((1 << (EPB-1))-1))
        unq = 0x7FFF;
    else
        unq = ((x << 15) + 0x4000) >> (EPB-1);

    if (s)
        unq = -unq;
}
```

After the endpoints are unquantized, interpolation proceeds as in the fixed-point formats above using Equation 2, including the interpolation weight table, Table 119.

The interpolated values are passed through a final unquantization step. For the unsigned format, this limits the range of the integer representation to those bit sequences which, when interpreted as a 16-bit half float, represent [0.0..65504.0], where 65504.0 is the largest finite value representable in a half float. The bit pattern that represents 65504.0 is integer 0x7BFF, so the integer input range 0..0xFFFF can be mapped to this range by scaling the interpolated integer i by *[Math Processing Error]* :

```
out = (i * 31) >> 6;
```

For the signed format, the final unquantization step limits the range of the integer representation to the bit sequences which, when interpreted as a 16-bit half float, represent the range [*[Math Processing Error]* ..65504.0], where *[Math Processing Error]* is represented in half float as the bit pattern 0xFC00. The signed 16-bit integer range [-0x8000..0x7FFF] is remapped to this float representation by taking the absolute value of the interpolated value i, scaling it by *[Math Processing Error]* , and restoring the sign bit:

```
out = i < 0 ? (((-i) * 31) >> 5) | 0x8000 : (i * 31) >> 5;
```

The resultant bit pattern should be interpreted as a 16-bit half float.

> The ability to support *[Math Processing Error]* is considered "accidental" due to the asymmetry of two's complement representation: in order to map integer 0x7FFF to 65504.0 and 0x0000 to 0.0, -0x7FFF maps to the largest finite negative value, -65504.0, represented as 0xFBFF. A two's complement signed integer can also represent -0x8000; it happens that the same unquantization formula maps 0x8000 to 0xFC00, which is the half float bit pattern for *[Math Processing Error]* . Although decoders for BC6H should be bit-exact, encoders for this format are encouraged to map *[Math Processing Error]* to -65504.0 (and to map *[Math Processing Error]* to 65504.0 and NaN values to 0.0) prior to encoding.

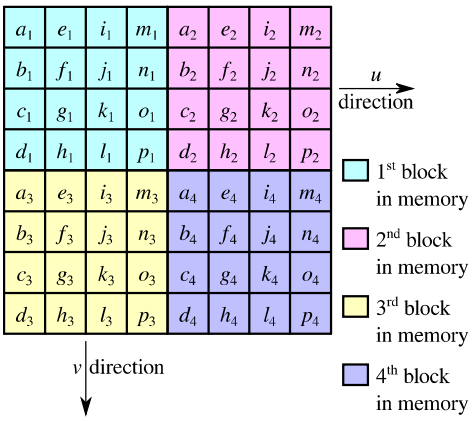
21. ETC1 Compressed Texture Image Formats

This description is derived from the OES_compressed_ETC1_RGB8_texture OpenGL extension.

The texture is described as a number of 4×4 pixel blocks. If the texture (or a particular mip-level) is smaller than 4 pixels in any dimension (such as a 2×2 or a 8×1 texture), the texture is found in the upper left part of the block(s), and the rest of the pixels are not used. For instance, a texture of size 4×2 will be placed in the upper half of a 4×4 block, and the lower half of the pixels in the block will not be accessed.

Pixel a_1 (see Figure 38) of the first block in memory will represent the texture coordinate (u=0, v=0). Pixel a_2 in the second block in memory will be adjacent to pixel m_1 in the first block, etc. until the width of the texture. Then pixel a_3 in the following block (third block in memory for a 8×8 texture) will be adjacent to pixel d_1 in the first block, etc. until the height of the texture. The data storage for an 8×8 texture using the first, second, third and fourth block if stored in that order in memory would have the texels encoded in the same order as a simple linear format as if the bytes describing the pixels came in the following memory order: a_1 e_1 i_1 m_1 a_2 e_2 i_2 m_2 b_1 f_1 i_1 n_1 b_2 f_2 i_2 n_2 c_1 g_1 k_1 o_1 c_2 g_2 k_2 o_2 d_1 h_1 l_1 p_1 d_2 h_2 l_2 p_2 a_3 e_3 i_3 m_3 a_4 e_4 i_4 m_4 b_3 f_3 i_3 n_3 b_4 f_4 i_4 n_4 c_3 g_3 k_3 o_3 c_4 g_4 k_4 o_4 d_3 h_3 l_3 p_3 d_4 h_4 l_4 p_4.

Figure 38. Pixel layout for an 8×8 texture using four ETC1 compressed blocks



Note how pixel a_2 in the second block is adjacent to pixel m_1 in the first block.

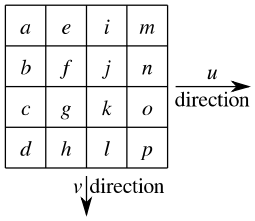
The number of bits that represent a 4×4 texel block is 64 bits.

The data for a block is stored as a number of bytes, q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, where byte q_0 is located at the lowest memory address and q_7 at the highest. The 64 bits specifying the block are then represented by the following 64 bit integer:

[Math Processing Error]

Each 64-bit word contains information about a 4×4 pixel block as shown in Figure 39.

Figure 39. Pixel layout for an ETC1 compressed block



There are two modes in ETC1: the 'individual' mode and the 'differential' mode. Which mode is active for a particular 4×4 block is controlled by bit 33, which we call *diff bit*. If *diff bit* = 0, the 'individual' mode is chosen, and if *diff bit* = 1, then the 'differential' mode is chosen. The bit layout for the two modes are different: The bit layout for the individual mode is shown in Table 125 part a and part c, and the bit layout for the differential mode is laid out in Table 125 part b and part c.

Table 125. Texel Data format for ETC1 compressed textures

a) Bit layout in bits 63 through 32 if *diff bit* = 0															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
Base color 1 R (4 bits)				Base color 2 R_2 (4 bits)				Base color 1 G (4 bits)				Base color 2 G_2 (4 bits)			
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Base color 1 B (4 bits)				Base color 2 B_2 (4 bits)				*Table codeword* 1			*Table codeword* 2			*diff bit*	*flip bit*
b) Bit layout in bits 63 through 32 if *diff bit* = 1															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
Base color R (5 bits)					Color delta R_d			Base color G (5 bits)					Color delta G_d		
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Base color B (5 bits)					Color delta B_d			*Table codeword* 1			*Table codeword* 2			*diff bit*	*flip bit*
c) Bit layout in bits 31 through 0 (in both cases)															
More significant pixel index bits															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
p^1	o^1	n^1	m^1	l^1	k^1	j^1	i^1	h^1	g^1	f^1	e^1	d^1	c^1	b^1	a^1
Less significant pixel index bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p^0	o^0	n^0	m^0	l^0	k^0	j^0	i^0	h^0	g^0	f^0	e^0	d^0	c^0	b^0	a^0

In both modes, the 4×4 block is divided into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we call *flip bit*. If *flip bit* = 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Figure 40. If *flip bit* = 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Figure 41.

Figure 40. Two 2×4-pixel ETC1 subblocks side-by-side

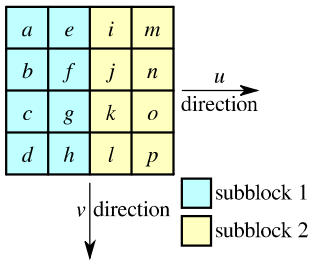
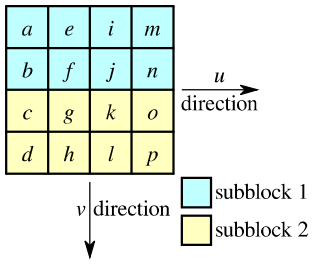


Figure 41. Two 4×2-pixel ETC1 subblocks on top of each other



In both individual and differential mode, a *base color* for each subblock is stored, but the way they are stored is different in the two modes:

In the 'individual' mode (*diff bit* = 0), the *base color* for subblock 1 is derived from the codewords *R* (bits 63..60), *G* (bits 55..52) and *B* (bits 47..44), see section a of Table 125. These four bit values are extended to *RGB*:888 by replicating the four higher order bits in the four lower order bits. For instance, if *R* = 14 = 1110b, *G* = 3 = 0011b and *B* = 8 = 1000b, then the red component of the *base color* of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The *base color* for subblock 2 is decoded the same way, but using the 4-bit codewords R_2 (bits 59..56), G_2 (bits 51..48) and B_2 (bits 43..40) instead. In summary, the *base colors* for the subblocks in the individual mode are:

[Math Processing Error]

In the 'differential' mode (*diff bit* = 1), the *base color* for subblock 1 is derived from the five-bit codewords *R*, *G* and *B*. These five-bit codewords are extended to eight bits by replicating the top three highest-order bits to the three lowest order bits. For instance, if *R* = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if *G* = 4 = 00100b and *B* = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the *base color* for subblock 1 is (231, 33, 24). The five-bit representation for the *base color* of subblock 2 is obtained by modifying the five-bit codewords *R*, *G* and *B* by the codewords R_d, G_d and B_d. Each of R_d, G_d and B_d is a three-bit two's-complement number that can hold values between -4 and +3. For instance, if *R*= 28 as above, an R_d = 100b = -4, then the five-bit representation for the red color component is 28+(-4) = 24 = 11000b, which is then extended to eight bits, to 11000110b = 198. Likewise, if *G* = 4, G_d = 2, *B* = 3 and B_d = 0, the *base color* of subblock 2 will be *RGB* = (198, 49, 24). In summary, the *base colors* for the subblocks in the differential mode are:

[Math Processing Error]

Note that these additions are not allowed to under- or overflow (go below zero or above 31). (The compression scheme can easily make sure they don't.) For over- or underflowing values, the behavior is undefined for all pixels in the 4×4 block. Note also that the extension to eight bits is performed *after* the addition.

After obtaining the base color, the operations are the same for the two modes 'individual' and 'differential'. First a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39..37), and for subblock 2, table codeword 2 is used (bits 36..34), see Table 125. The table codeword is used to select one of eight modifier tables, see Table 126. For instance, if the table code word is 010b = 2, then the modifier table [-29, -9, 9, 29] is selected. Note that the values in Table 126 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which *modifier* value to use from the modifier table using the two 'pixel index' bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel *d* (see Figure 39) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see section c of Table 125. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits *diff bit* and *flip bit*. The pixel index bits are decoded using Table 127. If, for instance, the pixel index bits are 01b = 1, and the modifier table [-29, -9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see Table 127). This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: (231+29, 8+29, 16+29) resulting in (260, 37, 45). These values are then clamped to [0..255], resulting in the color (255, 37, 45), and we are finished decoding the texel.

Table 126. Intensity modifier sets for ETC1 compressed textures

Table codeword	Modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60

5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table 127. Mapping from pixel index values to modifier values for ETC1 compressed textures

Pixel index value		Resulting modifier value
MSB	LSB	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	+a (small positive value)
0	1	+b (large positive value)

ETC1 is a proper subset of ETC2. There are examples of "individual" and "differential" decoding in Section 22.

21.1. ETC1S

The ETC1S format is a subset the ETC1, simplified to facilitate image compression. The blocks use differential encoding (*diff bit* = 1); Color deltas $R_d = G_d = B_d = 0$, so the two subblocks share base colors. The *Table codeword* for each subblock is identical. Finally, the *flip bit* is encoded as 0 — the subsets are identical anyway.

22. ETC2 Compressed Texture Image Formats

This description is derived from the "ETC Compressed Texture Image Formats" section of the OpenGL 4.4 specification.

The ETC formats form a family of related compressed texture image formats. They are designed to do different tasks, but also to be similar enough that hardware can be reused between them. Each one is described in detail below, but we will first give an overview of each format and describe how it is similar to others and the main differences.

RGB ETC2 is a format for compressing *RGB* data. It is a superset of the older ETC1 format. This means that an older ETC1 texture can be decoded using an ETC2-compliant decoder. The main difference is that the newer version contains three new modes; the 'T-mode' and the 'H-mode' which are good for sharp chrominance blocks and the 'Planar' mode which is good for smooth blocks.

RGB ETC2 with sRGB encoding is the same as linear RGB ETC2 with the difference that the values should be interpreted as being encoded with the sRGB transfer function instead of linear *RGB*-values.

RGBA ETC2 encodes *RGBA* 8-bit data. The *RGB* part is encoded exactly the same way as RGB ETC2. The alpha part is encoded separately.

RGBA ETC2 with sRGB encoding is the same as RGBA ETC2 but here the *RGB* values (but not the alpha value) should be interpreted as being encoded with the sRGB transfer function.

Unsigned R11 EAC is a one-channel unsigned format. It is similar to the alpha part of RGBA ETC2 but not exactly the same; it delivers higher precision. It is possible to make hardware that can decode both formats with minimal overhead.

Unsigned RG11 EAC is a two-channel unsigned format. Each channel is decoded exactly as Unsigned R11 EAC.

Signed R11 EAC is a one-channel signed format. This is good in situations when it is important to be able to preserve zero exactly, and still use both positive and negative values. It is designed to be similar enough to Unsigned R11 EAC so that hardware can decode both with minimal overhead, but it is not exactly the same. For example; the signed version does not add 0.5 to the *base codeword*, and the extension from 11 bits differ. For all details, see the corresponding sections.

Signed RG11 EAC is a two-channel signed format. Each channel is decoded exactly as signed R11 EAC.

RGB ETC2 with "punchthrough" alpha is very similar to RGB ETC2, but has the ability to represent "punchthrough" alpha (completely opaque or transparent). Each block can select to be completely opaque using one bit. To fit this bit, there is no individual mode in RGB ETC2 with punchthrough alpha. In other respects, the opaque blocks are decoded as in RGB ETC2. For the transparent blocks, one index is reserved to represent transparency, and the decoding of the *RGB* channels are also affected. For details, see the corresponding sections.

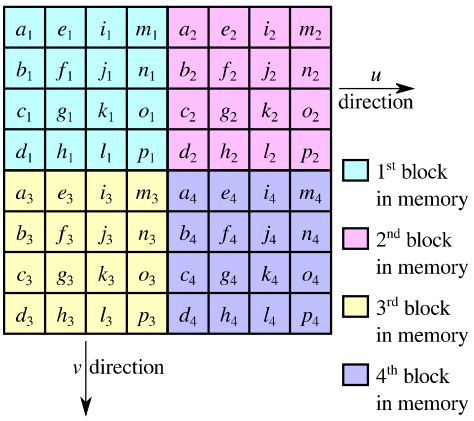
RGB ETC2 with punchthrough alpha and sRGB encoding is the same as linear RGB ETC2 with punchthrough alpha but the *RGB* channel values should be interpreted as being encoded with the sRGB transfer function.

A texture compressed using any of the ETC texture image formats is described as a number of 4×4 pixel blocks.

Pixel a_1 (see Figure 42) of the first block in memory will represent the texture coordinate (u=0, v=0). Pixel a_2 in the second block in memory will be adjacent to pixel m_1 in the first block, etc. until the width of the texture. Then pixel a_3 in the following block (third block in memory for an 8×8 texture) will be adjacent to pixel d_1 in the first block, etc. until the height of the texture.

The data storage for an 8×8 texture using the first, second, third and fourth block if stored in that order in memory would have the texels encoded in the same order as a simple linear format as if the bytes describing the pixels came in the following memory order: a_1 e_1 i_1 m_1 a_2 e_2 i_2 m_2 b_1 f_1 i_1 n_1 b_2 f_2 i_2 n_2 c_1 g_1 k_1 o_1 c_2 g_2 k_2 o_2 d_1 h_1 l_1 p_1 d_2 h_2 l_2 p_2 a_3 e_3 i_3 m_3 a_4 e_4 i_4 m_4 b_3 f_3 i_3 n_3 b_4 f_4 i_4 n_4 c_3 g_3 k_3 o_3 c_4 g_4 k_4 o_4 d_3 h_3 l_3 p_3 d_4 h_4 l_4 p_4.

Figure 42. Pixel layout for an 8×8 texture using four ETC2 compressed blocks



Note how pixel a_3 in the third block is adjacent to pixel d_1 in the first block.

If the width or height of the texture (or a particular mip-level) is not a multiple of four, then padding is added to ensure that the texture contains a whole number of 4×4 blocks in each dimension. The padding does not affect the texel coordinates. For example, the texel shown as a_1 in Figure 42 always has coordinates (i=0, j=0). The values of padding texels are irrelevant, e.g., in a 3×3 texture, the texels marked as m_1, n_1, o_1, d_1, h_1, l_1 and p_1 form padding and have no effect on the final texture image.

The number of bits that represent a 4×4 texel block is 64 bits if the format is RGB ETC2, RGB ETC2 with sRGB encoding, RGBA ETC2 with punchthrough alpha, or RGB ETC2 with punchthrough alpha and sRGB encoding.

In those cases the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The 64 bits specifying the block are then represented by the following 64 bit integer:

[Math Processing Error]

The number of bits that represent a 4×4 texel block is 128 bits if the format is RGBA ETC2 with a linear or sRGB transfer function. In those cases the data for a block is stored as a number of bytes: $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}\}$, where byte q_0 is located at the lowest memory address and q_{15} at the highest.

This is split into two 64-bit integers, one used for color channel decompression and one for alpha channel decompression:

[Math Processing Error]

22.1. Format RGB ETC2

For RGB ETC2, each 64-bit word contains information about a three-channel 4×4 pixel block as shown in Figure 43.

Figure 43. Pixel layout for an ETC2 compressed block

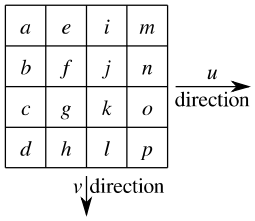


Table 128. Texel Data format for ETC2 compressed texture formats

a) Location of bits for mode selection																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					R_d			G					G_d			B					B_d			.....						D	.
b) Bit layout for bits 63 through 32 for 'individual' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R				R_2				G				G_2				B				B_2				$table_1$			$table_2$			0	F_B
c) Bit layout for bits 63 through 32 for 'differential' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					R_d			G					G_d			B					B_d			$table_1$			$table_2$			1	F_B
d) Bit layout for bits 63 through 32 for 'T' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
...			$R^{3..2}$	.	$R^{1..0}$			G				B				R_2				G_2				B_2				d_a		1	d_b
e) Bit layout for bits 63 through 32 for 'H' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R				$G^{3..1}$			...			G^0	B^3	.	$B^{2..0}$			R_2				G_2				B_2				d_a	1	d_b
f) Bit layout for bits 31 through 0 for 'individual', 'differential', 'T' and 'H' modes																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p^1	o^1	n^1	m^1	l^1	k^1	j^1	i^1	h^1	g^1	f^1	e^1	d^1	c^1	b^1	a^1	p^0	o^0	n^0	m^0	l^0	k^0	j^0	i^0	h^0	g^0	f^0	e^0	d^0	c^0	b^0	a^0

g) Bit layout for bits 63 through 0 for 'planar' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R						G^6	.	$G^{5..0}$						B^5	...			$B^{4..3}$		.	$B^{2..0}$			$R_h^{5..1}$					1	R_h^0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
G_h							B_h						R_v						G_v							B_v					

The blocks are compressed using one of five different 'modes'. Section a of Table 128 shows the bits used for determining the mode used in a given block. First, if the 'differential bit' marked D is set to 0, the 'individual' mode is used. Otherwise, the three 5-bit values R, G and B, and the three 3-bit values R_d, G_d and B_d are examined. R, G and B are treated as integers between 0 and 31 and R_d, G_d and B_d as two's-complement integers between -4 and +3. First, R and R_d are added, and if the sum is not within the interval [0..31], the 'T' mode is selected. Otherwise, if the sum of G and G_d is outside the interval [0..31], the 'H' mode is selected. Otherwise, if the sum of B and B_d is outside of the interval [0..31], the 'planar' mode is selected. Finally, if the D bit is set to 1 and all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

The layout of the bits used to decode the 'individual' and 'differential' modes are shown in section b and section c of Table 128, respectively. Both of these modes share several characteristics. In both modes, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the *flip bit* (F_B in Table 128 (b) and (c)). If the *flip bit* is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Figure 44. If the *flip bit* is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Figure 45. In both modes, a *base color* for each subblock is stored, but the way they are stored is different in the two modes:

Figure 44. Two 2×4-pixel ETC2 subblocks side-by-side

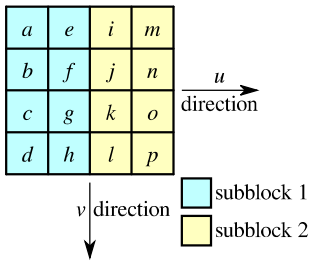
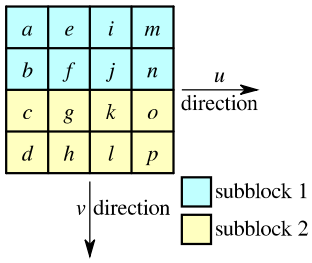


Figure 45. Two 4×2-pixel ETC2 subblocks on top of each other



In the 'individual' mode, following the layout shown in section b of Table 128, the *base color* for subblock 1 is derived from the codewords R (bits 63..60), G (bits 55..52) and B (bits 47..44). These four bit values are extended to *RGB*:888 by replicating the four higher order bits in the four lower order bits. For instance, if R = 14 = 1110 binary (1110b for short), G = 3 = 0011b and B = 8 = 1000b, then the red component of the *base color* of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The *base color* for subblock 2 is decoded the same way, but using the 4-bit codewords R_2 (bits 59..56), G_2 (bits 51..48) and B_2 (bits 43..40) instead. In summary, the *base colors* for the subblocks in the individual mode are:

[Math Processing Error]

In the 'differential' mode, following the layout shown in section c of Table 128, the *base color* for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest-order bits to the three lowest-order bits. For instance, if R = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the *base color* for subblock 1 is (231, 33, 24). The five-bit representation for the *base color* of subblock 2 is obtained by modifying the five-bit codewords R, G and B by the codewords R_d, G_d and B_d. Each of R_d, G_d and B_d is a 3-bit two's-complement number that can hold values between -4 and +3. For instance, if R = 28 as above, and R_d = 100b = y - 4, then the five bit representation for the red color component is 28+(-4) = 24 = 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, G_d = 2, B = 3 and B_d = 0, the *base color* of subblock 2 will be *RGB* = 198, 49, 24. In summary, the *base colors* for the subblocks in the 'differential' mode are:

[Math Processing Error]

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the 'differential' mode.

After obtaining the *base color*, the operations are the same for the two modes 'individual' and 'differential'. First a table is chosen using the *table codewords*: For subblock 1, *table codeword 1* is used (bits 39..37), and for subblock 2, *table codeword 2* is used (bits 36..34),

see section b or section c of Table 128. The *table codeword* is used to select one of eight modifier tables, see Table 129. For instance, if the *table codeword* is 010 binary = 2, then the modifier table [-29, -9, 9, 29] is selected for the corresponding sub-block. Note that the values in Table 129 are valid for all textures and can therefore be hardcoded into the decompression unit.

Table 129. ETC2 intensity modifier sets for 'individual' and 'differential' modes

Table codeword	Modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

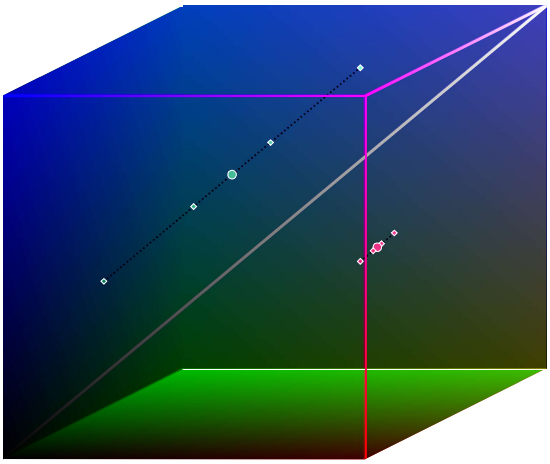
Table 130. Mapping from pixel index values to modifier values for RGB ETC2 compressed textures

***Pixel index* value**		**Resulting *modifier* value**
MSB	**LSB**	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	+a (small positive value)
0	1	+b (large positive value)

Next, we identify which *modifier* value to use from the modifier table using the two *pixel index* bits. The *pixel index* bits are unique for each pixel. For instance, the *pixel index* for pixel *d* (see Figure 43) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see section f of Table 128. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits *diff bit* and *flip bit*. The *pixel index* bits are decoded using Table 130. If, for instance, the *pixel index* bits are 01 binary = 1, and the modifier table [-29, -9, 9, 29] is used, then the *modifier* value selected for that pixel is 29 (see Table 130). This *modifier* value is now used to additively modify the *base color*. For example, if we have the *base color* (231, 8, 16), we should add the *modifier* value 29 to all three components: (231+29, 8+29, 16+29) resulting in (260, 37, 45). These values are then clamped to [0..255], resulting in the color (255, 37, 45), and we are finished decoding the texel.

Figure 46 shows an example 'individual mode' ETC2 block. The two *base colors* are shown as circles, and *modifiers* are applied to each channel to give the 'paint colors' selectable by each *pixel index*, shown as small diamonds. Since the same *modifier* is applied to each channel, each *paint color* for a subblock falls on a line (shown dashed) parallel to the grayscale (0, 0, 0) to (255, 255, 255) axis, unless the channels are modified by clamping to the range [0..255].

Figure 46. ETC2 'individual' mode



In this example, one *base color* is encoded as the 4-bit triple (4, 11, 9), which is expanded by *extend4to8bits* to (68, 187, 153). Modifier table 4 [-60, -18, 18, 60] is selected for this subblock, giving the following *paint colors*:

Modifier	***R***	***G***	***B***
-60	8	127	93
-18	58	169	135
18	86	205	171
60	128	247	213

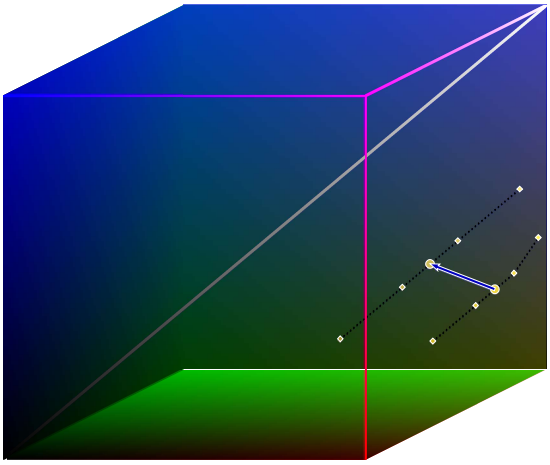
The other *base color* is encoded as the 4-bit triple (14, 3, 8), which is expanded by *extend4to8bits* to (238, 51, 136). Modifier table 0 [-8, -2, 2, 8] is selected for this subblock, giving the following *paint colors* for the subblock:

Modifier	*R*	*G*	*B*
-8	230	43	128
-2	236	49	134
2	240	53	138
8	246	59	144

In this example, none of the *paint colors* are modified by the process of clipping the channels to the range [0..255]. Since there is no difference in the way the *base colors* are encoded in 'individual mode', either *base color* could correspond to either subblock.

Figure 47 shows an example 'differential mode' ETC2 block. The two *base colors* are shown as circles; an arrow shows the *base color* of the second subblock (the upper left circle) derived from the first subblock's *base color* (lower right circle). *Modifiers* to the *base colors* give 'paint colors' selectable by each *pixel index*, shown as small diamonds. Since the same *modifier* is applied to each channel, each *paint color* for a subblock falls on a line (shown dashed) parallel to the grayscale (0, 0, 0) to (255, 255, 255) axis, unless channels are modified by clamping to [0..255].

Figure 47. ETC2 'differential' mode



Here the first subblock's *base color* is encoded as the 5-bit triple (29, 26, 8), and expanded by *extend5to8bits* to (239, 214, 66). Note that not every color representable in 'individual mode', exists in 'differential mode', or vice-versa.

The *base color* of subblock 2 is the five-bit representation of the *base color* of subblock 1 (29, 26, 8) plus a (R_d, G_d, B_d) offset of (-4, -3, +3), for a new *base color* of (25, 23, 11) - expanded by *extend5to8bits* to (206, 189, 90). The offset cannot exceed the range [0..31] (expanded to [0..255]): this would select the 'T', 'H' or 'planar' modes. For 'differential mode', the *base colors* must be similar in each channel. The two's complement offset gives an asymmetry: we could not swap the subblocks of this example, since a R_d offset of +4 is unrepresentable.

In this example, modifier table 2 [-29, -9, 9, 29] is applied to subblock 1's *base color* of (239, 214, 66):

Modifier	*R*	*G*	*B*
-29	210	185	37
-9	230	205	57
9	248	223	75
29	268	243	95

The last row is clamped to (255, 243, 95), so subblock 1's *paint colors* are not colinear in this example. With *modifiers*, all grays [0..255] are representable. Similarly, modifier table 3 [-42, -13, 13, 42] is applied to the *base color* of subblock 2, (206, 189, 90):

Modifier	*R*	*G*	*B*
-42	164	147	48
-13	193	176	77
13	219	202	103
42	248	231	132

The 'T' and 'H' compression modes also share some characteristics: both use two *base colors* stored using 4 bits per channel decoded as in the individual mode. Unlike the 'individual' mode however, these bits are not stored sequentially, but in the layout shown in section d and section e of Table 128. To clarify, in the 'T' mode, the two colors are constructed as follows:

[Math Processing Error]

Here, *[Math Processing Error]* denotes bit-wise left shift and *[Math Processing Error]* denotes bit-wise OR. In the 'H' mode, the two colors are constructed as follows:

[Math Processing Error]

Both the 'T' and 'H' modes have four *paint colors* which are the colors that will be used in the decompressed block, but they are assigned in a different manner. In the 'T' mode, *paint color 0* is simply the first *base color*, and *paint color 2* is the second *base color*. To obtain the other *paint colors*, a 'distance' is first determined, which will be used to modify the luminance of one of the *base colors*. This is done by combining the values d_a and d_b shown in section d of Table 128 by (d_a *[Math Processing Error]* 1) | d_b, and then using this value as an index into the small look-up table shown in Table 131. For example, if d_a is 10 binary and d_b is 1 binary, the *distance index* is 101 binary and the selected 'distance' *d* will be 32. *Paint color 1* is then equal to the second *base color* with the 'distance' *d* added to each channel, and *paint color 3* is the second *base color* with the 'distance' *d* subtracted.

Table 131. Distance table for ETC2 'T' and 'H' modes

Distance index	Distance *d*
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

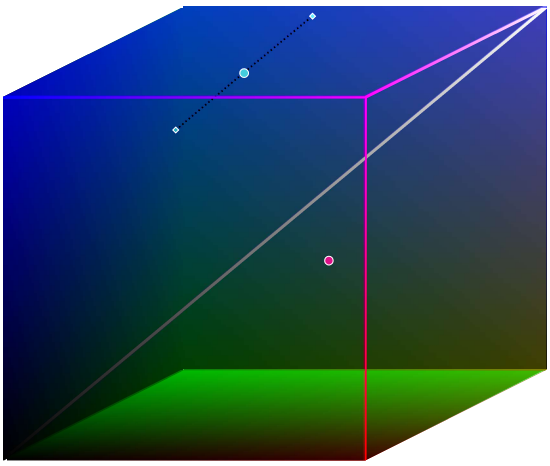
In summary, to determine the four *paint colors* for a 'T' block:

[Math Processing Error]

In both cases, the value of each channel is clamped to within [0..255].

Figure 48 shows an example 'T-mode' ETC2 block. The two *base colors* are shown as circles, and *modifiers* are applied to *base color 2* to give the other two 'paint colors', shown as small diamonds. Since the same *modifier* is applied to each channel, *base color 2* and the two *paint colors* derived from it fall on a line (shown dashed) parallel to the grayscale (0, 0, 0) to (255, 255, 255) axis, unless channels are modified by clamping to [0..255].

Figure 48. ETC2 'T' mode



In this example, the first *base color* is defined as the triple of 4-bit *RGB* values (13, 1, 8), which is expanded by *extend4to8bits* to (221, 17, 136). This becomes *paint color 0*.

The second *base color* is encoded as the triple of 4-bit *RGB* values (4, 12, 13), which is expanded by *extend4to8bits* to (68, 204, 221).

Distance index 5 is used to select a distance value *d* of 32, which is added to and subtracted from the second base color, giving (100, 236, 253) as *paint color 1* and (36, 172, 189) as *paint color 3*. On this occasion, the channels of these *paint colors* are not modified by the process of clamping them to [0..255].

A 'distance' value is computed for the 'H' mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table 131, d_a and d_b shown in section e of Table 128 are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (*base color 1* value *[Math Processing Error]* *base color 2* value), the 'value' of a color for the comparison being equal to (*R* *[Math Processing Error]* 16) + (*G* *[Math Processing Error]* 8) + *B*. Once the 'distance' *d* has been determined for an 'H' block, the four *paint colors* will be:

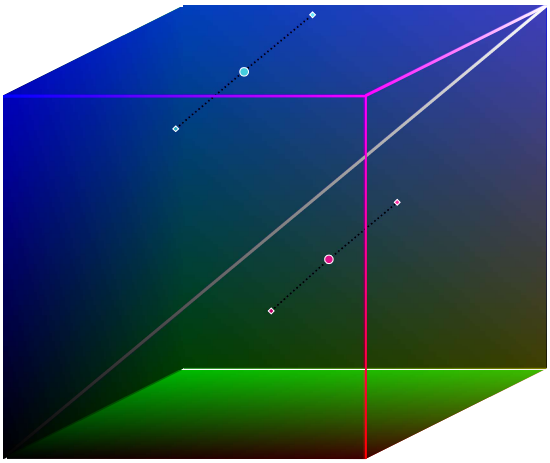
[Math Processing Error]

Again, all color components are clamped to within [0..255].

Figure 49 shows an example 'H mode' ETC2 block. The two *base colors* are shown as circles, and *modifiers* are applied to each channel to give the 'paint colors' selectable by each *pixel index*, shown as small diamonds. Since the same *modifier* is applied to each channel, each *paint color* falls on a line through the *base color* from which

it is derived (shown dashed) parallel to the grayscale (0, 0, 0) to (255, 255, 255) axis, unless the channels are modified by clamping to the range [0..255].

Figure 49. ETC2 'H' mode



In this example, the first *base color* is defined as the triple of 4-bit *RGB* values (13, 1, 8), as in the 'T mode' case above. This is expanded by *extend4to8bits* to (221, 17, 136).

The second *base color* is defined as the 4-bit triple (4, 12, 13), which expands to (68, 204, 221).

The block encodes a *distance index* of 5 (this means that *base color 1* must be greater than *base color 2*), corresponding to a distance *d* of 32. This leads to the following *paint colors*:

Paint color	***Base color***			**Distance**	***Paint color***		
id	***R***	***G***	***B***	***d***	***R***	***G***	***B***
0	221	17	136	+32	253	49	168
1				-32	189	-15	104
2	68	204	221	+32	100	236	253
3				-32	36	172	189

The *G* channel of *paint color 1* is clamped to 0, giving (189, 0, 104). This stops *paint color* 1 being colinear with *paint color 0* and *base color 1*.

Finally, in both the 'T' and 'H' modes, every pixel is assigned one of the four *paint colors* in the same way the four *modifier* values are distributed in 'individual' or 'differential' blocks. For example, to choose a *paint color* for pixel *d*, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned *paint color 2*.

The final mode possible in an RGB ETC2-compressed block is the 'planar' mode. Here, three *base colors* are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three *base colors* are stored in *RGB*:676 format, and stored in the manner shown in section g of Table 128. The two secondary colors are given the suffix 'h' and 'v', so that the red component of the three colors are R, R_h and R_v, for example. Some color channels are split into non-consecutive bit-ranges; for example B is reconstructed using B^5 as the most-significant bit, $B^{4..3}$ as the two following bits, and $B^{2..0}$ as the three least-significant bits.

Once the bits for the *base colors* have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the *base colors* in other modes. For example, the 6-bit blue and red channels are extended by replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three *base colors* in *RGB*:888 format, the color of each pixel can then be determined as:

[Math Processing Error]

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Figure 43 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

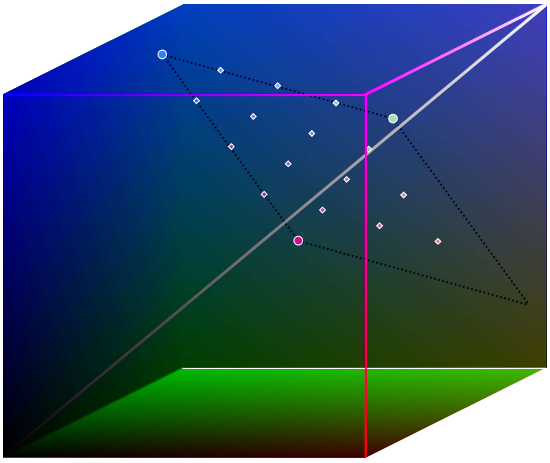
[Math Processing Error]

where *clamp255*(·) clamps the value to a number in the range [0..255] and where *[Math Processing Error]* performs bit-wise right shift.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

Figure 50 shows an example 'planar mode' ETC2 block. The three *base colors* are shown as circles, and the interpolated values are shown as small diamonds.

Figure 50. ETC2 'planar' mode



In this example, the origin (*R*, *G*, *B*) is encoded as the 6-7-6-bit value (12, 64, 62), which is expanded to (48, 129, 251). The 'horizontal' (interpolated by *x*) *base color* (R_h, G_h, B_h) = (50, 5, 37) and 'vertical' (interpolated by *y*) *base color* (R_v, G_v, B_v) = (40, 112, 45) expand to (203, 10, 150) and (162, 225, 182) respectively.

The resulting texel colors are then:

x	*y*	*R*	*G*	*B*
0	0	48	129	251
1	0	87	99	226
2	0	126	70	201
3	0	164	40	175
0	1	77	153	234
1	1	115	123	209
2	1	154	94	183
3	1	193	64	158
0	2	105	177	217
1	2	144	147	191
2	2	183	118	166
3	2	221	88	141
0	3	134	201	199
1	3	172	171	174
2	3	211	142	149
3	3	250	112	124

22.2. Format RGB ETC2 with sRGB encoding

Decompression of floating point sRGB values in RGB ETC2 with sRGB encoding follows that of floating point *RGB* values of linear RGB ETC2. The result is sRGB-encoded values between 0.0 and 1.0. The further conversion from an sRGB encoded component *cs* to a linear component *cl* is done according to the formulae in Section 13.3. Assume *cs* is the sRGB component in the range [0, 1].

22.3. Format RGBA ETC2

Each 4×4 block of *RGBA*:8888 information is compressed to 128 bits. To decode a block, the two 64-bit integers $int64bit_{Alpha}$ and $int64bit_{Color}$ are calculated as described in Section 22.1. The *RGB* component is then decoded the same way as for RGB ETC2 (see Section 22.1), using $int64bit_{Color}$ as the *int64bit* codeword.

Table 132. Texel Data format for alpha part of RGBA ETC2 compressed textures

a) Bit layout in bits 63 through 48															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
base codeword								*multiplier*				*table index*			
b) Bit layout in bits 47 through 0, with pixels as name in Figure 43, bits labeled from 0 being the LSB to 47 being the MSB															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
a_α^2	a_α^1	a_α^0	b_α^2	b_α^1	b_α^0	c_α^2	c_α^1	c_α^0	d_α^2	d_α^1	d_α^0	e_α^2	e_α^1	e_α^0	f_α^2
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
f_α^1	f_α^0	g_α^2	g_α^1	g_α^0	h_α^2	h_α^1	h_α^0	i_α^2	i_α^1	i_α^0	j_α^2	j_α^1	j_α^0	k_α^2	k_α^1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
k_α^0	l_α^2	l_α^1	l_α^0	m_α^2	m_α^1	m_α^0	n_α^2	n_α^1	n_α^0	o_α^2	o_α^1	o_α^0	p_α^2	p_α^1	p_α^0

The 64-bits in $int64bit_{Alpha}$ used to decompress the alpha channel are laid out as shown in Table 132. The information is split into two parts. The first 16 bits comprise a *base codeword*, a *table codeword* and a *multiplier*, which are used together to compute 8 pixel values to be used in the block. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of these 8 possible values for each pixel in the block.

> The color pixel indices are stored in *a..p* order in increasing bit order in a big-endian word representation, with the low bit stored separately from the high bit. However, the alpha indices are stored in *p..a* order in increasing bit order in a big-endian word representation, with each bit of each alpha index stored consecutively.

The decoded value of a pixel is a value between 0 and 255 and is calculated the following way:

Equation 3. ETC2 base

[Math Processing Error]

where *clamp255*(·) maps values outside the range [0..255] to 0.0 or 255.0.

The *base codeword* is stored in the first 8 bits (bits 63..56) as shown in Table 132 part (a). This is the first term in Equation 3.

Next, we want to obtain the *modifier*. Bits 51..48 in Table 132 part (a) form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 133.

Table 133. Intensity modifier sets for RGBA ETC2 alpha component

Table index	Modifier table							
0	-3	-6	-9	-15	2	5	8	14
1	-3	-7	-10	-13	2	6	9	12
2	-2	-5	-8	-13	1	4	7	12
3	-2	-4	-6	-13	1	3	5	12
4	-3	-6	-8	-12	2	5	7	11
5	-3	-7	-9	-11	2	6	8	10
6	-4	-7	-8	-11	3	6	7	10
7	-3	-5	-8	-11	2	4	7	10
8	-2	-6	-8	-10	1	5	7	9
9	-2	-5	-8	-10	1	4	7	9
10	-2	-4	-8	-10	1	3	7	9
11	-2	-5	-7	-10	1	4	6	9
12	-3	-4	-7	-10	2	3	6	9
13	-1	-2	-3	-10	0	1	2	9
14	-4	-6	-8	-9	3	5	7	8
15	-3	-5	-7	-9	2	4	6	8

For example, a *table index* of 13 (1101 binary) means that we should use table [-1, -2 -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the *pixel index* of the pixel we want to decode. As shown in Table 132 part (b), bits 47..0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel index is stored in bits 44..42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the *pixel index* is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our *modifier*, which is the starting point of our second term in the addition.

In the next step we obtain the *multiplier* value; bits 55..52 form a four-bit *multiplier* between 0 and 15. This value should be multiplied with the *modifier*. An encoder is not allowed to produce a *multiplier* of zero, but the decoder should still be able to handle this case (and produce 0 × *modifier* = 0 in that case).

The *modifier* times the *multiplier* now provides the third and final term in the sum in Equation 3. The sum is calculated and the value is clamped to the interval [0..255]. The resulting value is the 8-bit output value.

For example, assume a *base codeword* of 103, a *table index* of 13, a *pixel index* of 3 and a *multiplier* of 2. We will then start with the *base codeword* 103 (01100111 binary). Next, a *table index* of 13 selects table [-1, -2, -3, -10, 0, 1, 2, 9], and using a *pixel index* of 3 will result in a *modifier* of -10. The *multiplier* is 2, forming -10 × 2 = -20. We now add this to the base value and get 103 - 20 = 83. After clamping we still get 83 = 01010011 binary. This is our 8-bit output value.

This specification gives the output for each channel in 8-bit integer values between 0 and 255, and these values all need to be divided by 255 to obtain the final floating point representation.

Note that hardware can be effectively shared between the alpha decoding part of this format and that of R11 EAC texture. For details on how to reuse hardware, see Section 22.5.

22.4. Format RGBA ETC2 with sRGB encoding

Decompression of floating point sRGB values in RGBA ETC2 with sRGB encoding follows that of floating point *RGB* values of linear RGBA ETC2. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component *cs* to a linear component *cl* is according to the formula in Section 13.3. Assume *cs* is the sRGB component in the range [0, 1].

The alpha component of RGBA ETC2 with sRGB encoding is done in the same way as for linear RGBA ETC2.

22.5. Format Unsigned R11 EAC

The number of bits to represent a 4×4 texel block is 64 bits. if format is R11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64-bit integer:

[Math Processing Error]

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Figure 43. The 64-bit word is split into two parts. The first 16 bits comprise a *base codeword*, a *table codeword* and a *multiplier*. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table 132.

The decoded value is calculated as:

Equation 4. Unsigned R11 EAC start

[Math Processing Error]

where *clamp1*(·) maps values outside the range [0.0, 1.0] to 0.0 or 1.0.

We will now go into detail how the decoding is done. The result will be an 11-bit fixed point number where 0 represents 0.0 and 2047 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between 0 and 2047 we must multiply Equation 4 by 2047.0:

[Math Processing Error]

where *clamp2*(·) clamps to the range [0.0, 2047.0].

Since *[Math Processing Error]* is exactly 8.0, the above equation can be written as

Equation 5. Unsigned R11 EAC simple

[Math Processing Error]

The *base codeword* is stored in the first 8 bits as shown in Table 132 part (a). Bits 63..56 in each block represent an eight-bit integer (*base codeword*) which is multiplied by 8 by shifting three steps to the left. We can add 4 to this value without addition logic by just inserting 100 binary in the last three bits after the shift. For example, if *base codeword* is 129 = 10000001 binary (or 10000001b for short), the shifted value is 10000001000b and the shifted value including the +4 term is 10000001100b = 1036 = 129×8+4. Hence we have summed together the first two terms of the sum in Equation 5.

Next, we want to obtain the *modifier*. Bits 51..48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 133. For example, a *table index* of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the *pixel index* of the pixel we want to decode. Bits 47..0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel indices are stored in bit 44..42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the *pixel index* is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our *modifier*, which is the starting point of our second term in the sum.

In the next step we obtain the *multiplier* value; bits 55..52 form a four-bit *multiplier* between 0 and 15. We will later treat what happens if the *multiplier* value is zero, but if it is nonzero, it should be multiplied with the *modifier*. This product should then be shifted three steps to the left to implement the ×8 multiplication. The result now provides the third and final term in the sum in Equation 5. The sum is calculated and the result is clamped to a value in the interval [0..2047]. The resulting value is the 11-bit output value.

For example, assume a *base codeword* of 103, a *table index* of 13, a *pixel index* of 3 and a *multiplier* of 2. We will then first multiply the *base codeword* 103 (01100111b) by 8 by left-shifting it (0110111000b) and then add 4 resulting in 0110111100b = 828 = 103 × 8 + 4. Next, a *table index* of 13 selects table [-1, -2, -3, -10, 0, 1, 2, 9], and using a *pixel index* of 3 will result in a *modifier* of -10. The *multiplier* is nonzero, which means that we should multiply it with the *modifier*, forming -10 × 2 = -20 = 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b = -160. We now add this to the base value and get 828 - 160 = 668. After clamping we still get 668 = 01010011100b. This is our 11-bit output value, which represents the value *[Math Processing Error]*

If the *multiplier* value is zero (i.e., the *multiplier* bits 55..52 are all zero), we should set the *multiplier* to *[Math Processing Error]* . Equation 5 can then be simplified to

Equation 6. Unsigned R11 EAC simpler

[Math Processing Error]

As an example, assume a *base codeword* of 103, a *table index* of 13, a *pixel index* of 3 and a *multiplier* value of 0. We treat the *base codeword* the same way, getting 828 = 103×8+4. The *modifier* is still -10. But the *multiplier* should now be *[Math Processing Error]* , which means that third term becomes *[Math Processing Error]* . The sum therefore becomes 828-10 = 818. After clamping we still get 818 = 01100110010b, and this is our 11-bit output value, and it represents *[Math Processing Error]*

Some OpenGL ES implementations may find it convenient to use 16-bit values for further processing. In this case, the 11-bit value should be extended using bit replication. An 11-bit value *x* is extended to 16 bits through *[Math Processing Error]* . For example, the value 668 = 01010011100b should be extended to 0101001110001010b = 21386.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that the method does not have the same reconstruction levels as the alpha part in the RGBA ETC2 format. For instance, for a *base codeword* of 255 and a *table value* of 0, the alpha part of the RGBA ETC2 format will represent a value of *[Math Processing Error]* exactly. In R11 EAC the same *base codeword* and *table value* will instead represent *[Math Processing Error]* That said, it is still possible to decode the alpha part of the RGBA ETC2-format using R11 EAC hardware. This is done by truncating the 11-bit number to 8 bits. As an example, if *base codeword* = 255 and *table value* = 0, we get the 11-bit value (255×8+4+0) = 2044 = 1111111100b, which after truncation becomes the 8-bit value 11111111b = 255 which is exactly the correct value according to RGBA ETC2. Clamping has to be done to [0, 255] after truncation for RGBA ETC2 decoding. Care must also be taken to handle the case when the *multiplier* value is zero. In the 11-bit version, this means multiplying by *[Math Processing Error]* , but in the 8-bit version, it really means multiplication by 0. Thus, the decoder will have to know if it is an RGBA ETC2 texture or an R11 EAC texture to decode correctly, but the hardware can be 100% shared.

As stated above, a *base codeword* of 255 and a *table value* of 0 will represent a value of *[Math Processing Error]* , and this does not reach 1.0 even though 255 is the highest possible *base codeword*. However, it is still possible to reach a pixel value of 1.0 since a *modifier* other than 0 can be used. Indeed, half of the *modifiers* will often produce a value of 1.0. As an example, assume we choose the *base codeword* 255, a *multiplier* of 1 and the modifier table [-3, -5, -7, -9, 2, 4, 6, 8]. Starting with Equation 5,

[Math Processing Error]

we get

[Math Processing Error]

which equals

[Math Processing Error]

or after clamping

[Math Processing Error]

which shows that several values can be 1.0, even though the base value does not reach 1.0. The same reasoning goes for 0.0.

22.6. Format Unsigned RG11 EAC

The number of bits to represent a 4×4 texel block is 128 bits if the format is RG11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

[Math Processing Error]

The 64-bit word $int64bit_0$ contains information about the red component of a two-channel 4×4 pixel block as shown in Figure 43, and the word $int64bit_1$ contains information about the green component. Both 64-bit integers are decoded in the same way as R11 EAC described in Section 22.5.

22.7. Format Signed R11 EAC

The number of bits to represent a 4×4 texel block is 64 bits if the format is signed R11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64 bit integer:

[Math Processing Error]

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Figure 43. The 64-bit word is split into two parts. The first 16 bits comprise a *base codeword*, a *table codeword* and a *multiplier*. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table 132.

The decoded value is calculated as

Equation 7. Signed R11 EAC start

[Math Processing Error]

where *clamp1*(·) maps values outside the range [-1.0, 1.0] to -1.0 or 1.0. We will now go into detail how the decoding is done. The result will be an 11-bit two's-complement fixed point number where -1023 represents -1.0 and 1023 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between -1023 and 1023 we must multiply Equation 7 by 1023.0:

[Math Processing Error]

where *clamp2*(·) clamps to the range [-1023.0, 1023.0]. Since *[Math Processing Error]* is exactly 8, the above formula can be written as:

Equation 8. Signed R11 EAC simple

[Math Processing Error]

The *base codeword* is stored in the first 8 bits as shown in Table 132 part (a). It is a two's-complement value in the range [-127, 127], and where the value -128 is not allowed; however, if it should occur anyway it must be treated as -127. The *base codeword* is then multiplied by 8 by shifting it left three steps. For example the value 65 = 01000001 binary (or 01000001b for short) is shifted to 01000001000b = 520 = 65×8.

Next, we want to obtain the *modifier*. Bits 51..48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 133. For example, a *table index* of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the *pixel index* of the pixel we want to decode. Bits 47..0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel indices are stored in bit 44..42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the *pixel index* is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our *modifier*, which is the starting point of our second term in the sum.

In the next step we obtain the *multiplier* value; bits 55..52 form a four-bit *multiplier* between 0 and 15. We will later treat what happens if the *multiplier* value is zero, but if it is nonzero, it should be multiplied with the *modifier*. This product should then be shifted three steps to the left to implement the ×8 multiplication. The result now provides the third and final term in the sum in Equation 8. The sum is calculated and the result is clamped to a value in the interval [-1023..1023]. The resulting value is the 11-bit output value.

For example, assume a a *base codeword* of 60, a *table index* of 13, a *pixel index* of 3 and a *multiplier* of 2. We start by multiplying the *base codeword* (00111100b) by 8 using bit shift, resulting in (00111100000b) = 480 = 60 × 8. Next, a *table index* of 13 selects table [-1, -2, -3, -10, 0, 1, 2, 9], and using a *pixel index* of 3 will result in a *modifier* of -10. The *multiplier* is nonzero, which means that we should multiply it with the *modifier*, forming -10×2 = -20 = 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b = -160. We now add this to the base value and get 480-160 = 320. After clamping we still get 320 = 00101000000b. This is our 11-bit output value, which represents the value *[Math Processing Error]* .

If the *multiplier* value is zero (i.e., the *multiplier* bits 55..52 are all zero), we should set the *multiplier* to *[Math Processing Error]* . Equation 8 can then be simplified to:

Equation 9. Signed R11 EAC simpler

[Math Processing Error]

As an example, assume a *base codeword* of 65, a *table index* of 13, a *pixel index* of 3 and a *multiplier* value of 0. We treat the *base codeword* the same way, getting 480 = 60×8. The *modifier* is still -10. But the *multiplier* should now be *[Math Processing Error]* , which means that third term becomes *[Math Processing Error]* . The sum therefore becomes 480-10 = 470. Clamping does not affect the value since it is already in the range [-1023, 1023], and the 11-bit output value is therefore 470 = 00111010110b. This represents *[Math Processing Error]*

Some OpenGL ES implementations may find it convenient to use two's-complement 16-bit values for further processing. In this case, a positive 11-bit value should be extended using bit replication on all the bits except the sign bit. An 11-bit value x is extended to 16 bits through (*x* *[Math Processing Error]* 5) + (*x* *[Math Processing Error]* 5). Since the sign bit is zero for a positive value, no addition logic is needed for the bit replication in this case. For example, the value 470 = 00111010110b in the above example should be expanded to 0011101011001110b = 15054. A negative 11-bit value must first be made positive before bit replication, and then made negative again:

```
if (result11bit >= 0) {
  result16bit = (result11bit << 5) + (result11bit >> 5);
} else {
  result11bit = -result11bit;
  result16bit = (result11bit << 5) + (result11bit >> 5);
  result16bit = -result16bit;
}
```

Simply bit replicating a negative number without first making it positive will not give a correct result.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication according to the above should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that it is not possible to specify a base value of 1.0 or -1.0. The largest possible *base codeword* is +127, which represents *[Math Processing Error]* . However, it is still possible to reach a pixel value of 1.0 or -1.0, since the base value is modified by the table before the pixel value is calculated. Indeed, half of the *modifiers* will often produce a value of 1.0. As an example, assume the *base codeword* is +127, the modifier table is [-3, -5, -7, -9, 2, 4, 6, 8] and the *multiplier* is one. Starting with Equation 7,

[Math Processing Error]

we get

[Math Processing Error]

which equals

[Math Processing Error]

or after clamping

[Math Processing Error]

This shows that it is indeed possible to arrive at the value 1.0. The same reasoning goes for -1.0.

Note also that Equation 8/Equation 9 are very similar to Equation 5/Equation 6 in the unsigned version EAC_R11. Apart from the +4, the clamping and the extension to bit sizes other than 11, the same decoding hardware can be shared between the two codecs.

22.8. Format Signed RG11 EAC

The number of bits to represent a 4×4 texel block is 128 bits if the format is signed RG11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

[Math Processing Error]

The 64-bit word $int64bit_0$ contains information about the red component of a two-channel 4×4 pixel block as shown in Figure 43, and the word $int64bit_1$ contains information about the green component. Both 64-bit integers are decoded in the same way as signed R11 EAC described in Section 22.8.

22.9. Format RGB ETC2 with punchthrough alpha

For RGB ETC2 with punchthrough alpha, each 64-bit word contains information about a four-channel 4×4 pixel block as shown in Figure 43.

The blocks are compressed using one of four different 'modes'. Table 134 part (a) shows the bits used for determining the mode used in a given block.

Table 134. Texel Data format for punchthrough alpha ETC2 compressed texture formats

a) Location of bits for mode selection																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					R_d			G					G_d			B					B_d			.....						Op	.
b) Bit layout for bits 63 through 32 for 'differential' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					R_d			G					G_d			B					B_d			$table_1$			$table_2$			Op	F_B
c) Bit layout for bits 63 through 32 for 'T' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
...			$R^{3..2}$		.	$R^{1..0}$		G					B			R_2					G_2			B_2				d_a		Op	d_b
d) Bit layout for bits 63 through 32 for 'H' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R				$G^{3..1}$			...			G^0	B^3	.	$B^{2..0}$		R_2					G_2				B_2				d_a	Op	d_b
e) Bit layout for bits 31 through 0 for 'differential', 'T' and 'H' modes																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p^1	o^1	n^1	m^1	l^1	k^1	j^1	i^1	h^1	g^1	f^1	e^1	d^1	c^1	b^1	a^1	p^0	o^0	n^0	m^0	l^0	k^0	j^0	i^0	h^0	g^0	f^0	e^0	d^0	c^0	b^0	a^0
f) Bit layout for bits 63 through 0 for 'planar' mode																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R						G^6	.	$G^{5..0}$						B^5	...			$B^{4..3}$		.	$B^{2..0}$				$R_h^{5..1}$				1	R_h^0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
G_h							B_h						R_v						G_v							B_v					

To determine the mode, the three 5-bit values R, G and B, and the three 3-bit values R_d, G_d and B_d are examined. R, G and B are treated as integers between 0 and 31 and R_d, G_d and B_d as two's-complement integers between -4 and +3. First, R and R_d are added, and if the sum is not within the interval [0..31], the 'T' mode is selected. Otherwise, if the sum of G and G_d is outside the interval [0..31], the 'H' mode is selected. Otherwise, if the sum of B and B_d is outside of the interval [0..31], the 'planar' mode is selected. Finally, if all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

The layout of the bits used to decode the 'differential' mode is shown in Table 134 part (b). In this mode, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the *flip bit* (F_B in Table 134 (b) and (c)). If the *flip bit* is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Figure 44. If the *flip bit* is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Figure 45. For each subblock, a *base color* is stored.

In the 'differential' mode, following the layout shown in Table 134 part (b), the *base color* for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest-order bits to the three lowest-order bits. For instance, if R = 28 = 11100 binary (11100b for short), the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the *base color* for subblock 1 is (231, 33, 24). The five bit representation for the *base color* of subblock 2 is obtained by modifying the 5-bit codewords R, G and B by the codewords R_d, G_d and B_d. Each of R_d, G_d and B_d is a 3-bit two's-complement number that can hold values between -4 and +3. For instance, if R = 28 as above, and R_d = 100b = -4, then the five bit representation for the red color component is 28+(-4)=24 = 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, G_d = 2, B = 3 and B_d = 0, the *base color* of subblock 2 will be RGB = (198, 49, 24). In summary, the *base colors* for the subblocks in the differential mode are:

[Math Processing Error]

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the 'differential' mode.

Table 135. ETC2 intensity modifier sets for the 'differential' if 'opaque' (*Op*) is set

Table codeword	Modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table 136. ETC2 intensity modifier sets for the 'differential' if 'opaque' (*Op*) is unset

Table codeword	Modifier table			
0	-8	0	0	8
1	-17	0	0	17
2	-29	0	0	29
3	-42	0	0	42
4	-60	0	0	60
5	-80	0	0	80
6	-106	0	0	106
7	-183	0	0	183

After obtaining the *base color*, a table is chosen using the *table codewords*: For subblock 1, *table codeword 1* is used (bits 39..37), and for subblock 2, *table codeword 2* is used (bits 36..34), see Table 134 part (b). The *table codeword* is used to select one of eight modifier tables. If the 'opaque'-bit (bit 33) is set, Table 135 is used. If it is unset, Table 136 is used. For instance, if the 'opaque'-bit is 1 and the *table codeword* is 010 binary = 2, then the modifier table [-29, -9, 9, 29] is selected for the corresponding sub-block. Note that the values in Table 135 and Table 136 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which *modifier* value to use from the modifier table using the two *pixel index* bits. The *pixel index* bits are unique for each pixel. For instance, the *pixel index* for pixel *d* (see Figure 43) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table 134 part (e). Note that the *pixel index* for a particular texel is always stored in the same bit position, irrespectively of the *flip bit*.

If the 'opaque'-bit (bit 33) is set, the *pixel index* bits are decoded using Table 137. If the 'opaque'-bit is unset, Table 138 will be used instead. If, for instance, the 'opaque'-bit is 1, and the *pixel index* bits are 01 binary = 1, and the modifier table [-29, -9, 9, 29] is used, then the *modifier* value selected for that pixel is 29 (see Table 137). This *modifier* value is now used to additively modify the *base color*. For example, if we have the *base color* (231, 8, 16), we should add the *modifier* value 29 to all three components: (231+29, 8+29, 16+29) resulting in (260, 37, 45). These values are then clamped to [0..255], resulting in the color (255, 37, 45).

Table 137. ETC2 mapping from pixel index values to modifier values when 'opaque' bit is set

Pixel index value		Resulting *modifier* value
msb	**lsb**	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	+a (small positive value)
0	1	+b (large positive value)

Table 138. ETC2 mapping from pixel index values to modifier values when 'opaque' bit is unset

Pixel index value		Resulting *modifier* value
msb	**lsb**	
1	1	-b (large negative value)
1	0	0 (zero)
0	0	0 (zero)
0	1	+b (large positive value)

The alpha component is decoded using the 'opaque'-bit, which is positioned in bit 33 (see Table 134 part (b)). If the 'opaque'-bit is set, alpha is always 255. However, if the 'opaque'-bit is zero, the alpha-value depends on the pixel indices; if MSB==1 and LSB==0, the alpha value will be zero, otherwise it will be 255. Finally, if the alpha value equals 0, the red, green and blue components will also be zero.

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

Hence *paint color 2* will equal *RGBA* = (0, 0, 0, 0) if opaque = 0.

In the example above, assume that the 'opaque'-bit was instead 0. Then, since the MSB = 0 and LSB 1, alpha will be 255, and the final decoded *RGBA*-tuple will be (255, 37, 45, 255).

The 'T' and 'H' compression modes share some characteristics: both use two *base colors* stored using 4 bits per channel. These bits are not stored sequentially, but in the layout shown in Table 134 part (c) and Table 134 part (d). To clarify, in the 'T' mode, the two colors are constructed as follows:

[Math Processing Error]

In the 'H' mode, the two colors are constructed as follows:

[Math Processing Error]

The function *extend4to8bits*(·) just replicates the four bits twice. This is equivalent to multiplying by 17. As an example, *extend4to8bits*(1101b) equals 11011101b = 221.

Both the 'T' and 'H' modes have four *paint colors* which are the colors that will be used in the decompressed block, but they are assigned in a different manner. In the 'T' mode, *paint color 0* is simply the first *base color*, and *paint color 2* is the second *base color*. To obtain the other *paint colors*, a 'distance' is first determined, which will be used to modify the luminance of one of the *base colors*. This is done by combining the values d_a and d_b shown in Table 134 part (c) by (d_a *[Math Processing Error]* 1) | d_b, and then using this value as an index into the small look-up table shown in Table 131. For example, if d_a is 10 binary and d_b is 1 binary, the index is 101 binary and the selected distance *d* will be 32. *Paint color 1* is then equal to the second *base color* with the 'distance' *d* added to each channel, and *paint color 3* is the second *base color* with the 'distance' *d* subtracted. In summary, to determine the four *paint colors* for a 'T' block:

[Math Processing Error]

In both cases, the value of each channel is clamped to within [0..255].

Just as for the differential mode, the *RGB* channels are set to zero if alpha is zero, and the alpha component is calculated the same way:

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

A 'distance' value is computed for the 'H' mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table 131, d_a and d_b shown in Table 134 part (d) are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (*base color 1* value *[Math Processing Error]* *base color 2* value), the 'value' of a color for the comparison being equal to (*R* *[Math Processing Error]* 16) + (*G* *[Math Processing Error]* 8) + *B*. Once the 'distance' *d* has been determined for an 'H' block, the four *paint colors* will be:

[Math Processing Error]

Yet again, *RGB* is zeroed if alpha is 0 and the alpha component is determined the same way:

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

Hence *paint color 2* will have *R* = *G* = *B* = alpha = 0 if opaque = 0.

Again, all color components are clamped to within [0..255]. Finally, in both the 'T' and 'H' modes, every pixel is assigned one of the four *paint colors* in the same way the four *modifier* values are distributed in 'individual' or 'differential' blocks. For example, to choose a *paint color* for pixel *d*, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned *paint color 2*.

The final mode possible in an RGB ETC2 with punchthrough alpha — compressed block is the 'planar' mode. In this mode, the 'opaque'-bit must be 1 (a valid encoder should not produce an 'opaque'-bit equal to 0 in the planar mode), but should the 'opaque'-bit anyway be 0 the decoder should treat it as if it were 1. In the 'planar' mode, three *base colors* are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three *base colors* are stored in *RGB*:676 format, and stored in the manner shown in Table 134 part (f). The two secondary colors are given the suffix 'h' and 'v', so that the red component of the three colors are *R*, R_h and R_v, for example. Some color channels are split into non-consecutive bit-ranges; for example *B* is reconstructed using B^5 as the most-significant bit, $B^{4..3}$ as the two following bits, and $B^{2..0}$ as the three least-significant bits.

Once the bits for the *base colors* have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the *base colors* in other modes. For example, the 6-bit blue and red channels are extended by replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three *base colors* in *RGB*:888 format, the color of each pixel can then be determined as:

[Math Processing Error]

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Figure 43 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

[Math Processing Error]

where *clamp255*(·) clamps the value to a number in the range [0..255].

Note that the alpha component is always 255 in the planar mode.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

22.10. Format RGB ETC2 with punchthrough alpha and sRGB encoding

Decompression of floating point sRGB values in RGB ETC2 with sRGB encoding and punchthrough alpha follows that of floating point *RGB* values of RGB ETC2 with punchthrough alpha. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, *cs*, to a linear component, *cl*, is according to the formula in Section 13.3. Assume *cs* is the sRGB component in the range [0, 1]. Note that the alpha component is not gamma corrected, and hence does not use this formula.

23. ASTC Compressed Texture Image Formats

This description is derived from the Khronos OES_texture_compression_astc OpenGL extension.

23.1. What is ASTC?

ASTC stands for Adaptive Scalable Texture Compression. The ASTC formats form a family of related compressed texture image formats. They are all derived from a common set of definitions.

ASTC textures may be either 2D or 3D.

ASTC textures may be encoded using either high or low dynamic range. Low dynamic range images may optionally be specified using the sRGB transfer function for the *RGB* channels.

Two sub-profiles ("LDR Profile" and "HDR Profile") may be implemented, which support only 2D images at low or high dynamic range respectively.

ASTC textures may be encoded as 1, 2, 3 or 4 components, but they are all decoded into *RGBA*. ASTC has a variable block size.

23.2. Design Goals

The design goals for the format are as follows:

- Random access. This is a must for any texture compression format.
- Bit exact decode. This is a must for conformance testing and reproducibility.
- Suitable for mobile use. The format should be suitable for both desktop and mobile GPU environments. It should be low bandwidth and low in area.
- Flexible choice of bit rate. Current formats only offer a few bit rates, leaving content developers with only coarse control over the size/quality tradeoff.
- Scalable and long-lived. The format should support existing *R*, *RG*, *RGB* and *RGBA* image types, and also have high "headroom", allowing continuing use for several years and the ability to innovate in encoders. Part of this is the choice to include HDR and 3D.
- Feature orthogonality. The choices for the various features of the format are all orthogonal to each other. This has three effects: first, it allows a large, flexible configuration space; second, it makes that space easier to understand; and third, it makes verification easier.
- Best in class at given bit rate. It should beat or match the current best in class for peak signal-to-noise ratio (PSNR) at all bit rates.
- Fast decode. Texel throughput for a cached texture should be one texel decode per clock cycle per decoder. Parallel decoding of several texels from the same block should be possible at incremental cost.
- Low bandwidth. The encoding scheme should ensure that memory access is kept to a minimum, cache reuse is high and memory bandwidth for the format is low.
- Low area. It must occupy comparable die size to competing formats.

23.3. Basic Concepts

ASTC is a block-based lossy compression format. The compressed image is divided into a number of blocks of uniform size, which makes it possible to quickly determine which block a given texel resides in.

Each block has a fixed memory footprint of 128 bits, but these bits can represent varying numbers of texels (the block "footprint").

The term "block footprint" in ASTC refers to the same concept as "compressed texel block dimensions" elsewhere in the Data Format Specification.

Block footprint sizes are not confined to powers-of-two, and are also not confined to be square. They may be 2D, in which case the block dimensions range from 4 to 12 texels, or 3D, in which case the block dimensions range from 3 to 6 texels.

Decoding one texel requires only the data from a single block. This simplifies cache design, reduces bandwidth and improves encoder throughput.

23.4. Block Encoding

To understand how the blocks are stored and decoded, it is useful to start with a simple example, and then introduce additional features.

The simplest block encoding starts by defining two color "endpoints". The endpoints define two colors, and a number of additional colors are generated by interpolating between them. We can define these colors using 1, 2, 3, or 4 components (usually corresponding to *R*, *RG*, *RGB* and *RGBA* textures), and using low or high dynamic range.

We then store a color interpolant weight for each texel in the image, which specifies how to calculate the color to use. From this, a weighted average of the two endpoint colors is used to generate the intermediate color, which is the returned color for this texel.

There are several different ways of specifying the endpoint colors, and the weights, but once they have been defined, calculation of the texel colors proceeds identically for all of them. Each block is free to choose whichever encoding scheme best represents its color endpoints, within the constraint that all the data fits within the 128 bit block.

For blocks which have a large number of texels (e.g. a 12×12 block), there is not enough space to explicitly store a weight for every texel. In this case, a sparser grid with fewer weights is stored, and interpolation is used to determine the effective weight to be used for each texel position. This allows very low bit rates to be used with acceptable quality. This can also be used to more efficiently encode blocks with low detail, or with strong vertical or horizontal features.

For blocks which have a mixture of disparate colors, a single line in the color space is not a good fit to the colors of the pixels in the original image. It is therefore possible to partition the texels into multiple sets, the pixels within each set having similar colors. For each of these "partitions", we specify separate endpoint pairs, and choose which pair of endpoints to use for a particular texel by looking up the partition index from a partitioning pattern table. In ASTC, this partition table is actually implemented as a function.

The endpoint encoding for each partition is independent.

For blocks which have uncorrelated channels — for example an image with a transparency mask, or an image used as a normal map — it may be necessary to specify two weights for each texel. Interpolation between the components of the endpoint colors can then proceed independently for each "plane" of the image. The assignment of channels to planes is selectable.

Since each of the above options is independent, it is possible to specify any combination of channels, endpoint color encoding, weight encoding, interpolation, multiple partitions and single or dual planes.

Since these values are specified per block, it is important that they are represented with the minimum possible number of bits. As a result, these values are packed together in ways which can be difficult to read, but which are nevertheless highly amenable to hardware decode.

All of the values used as weights and color endpoint values can be specified with a variable number of bits. The encoding scheme used allows a fine-grained tradeoff between weight bits and color endpoint bits using "integer sequence encoding". This can pack adjacent values together, allowing us to use fractional numbers of bits per value.

Finally, a block may be just a single color. This is a so-called "void extent block" and has a special coding which also allows it to identify nearby regions of single color. This may be used to short-circuit fetching of what would be identical blocks, and further reduce memory bandwidth.

23.5. LDR and HDR Modes

The decoding process for LDR content can be simplified if it is known in advance that sRGB output is required. This selection is therefore included as part of the global configuration.

The two modes differ in various ways, as shown in Table 139.

Table 139. ASTC differences between LDR and HDR modes

Operation	LDR mode	HDR mode
Returned Value	Determined by decoding mode	
sRGB compatible	Yes	No
LDR endpoint decoding precision	16 bits, or 8 bits for sRGB	16 bits
HDR endpoint mode results	Error color	As decoded
Error results	Error color	Vector of NaNs (0xFFFF)

The type of the values returned by the decoding process is determined by the decoding mode as shown in Table 140.

Table 140. ASTC decoding modes

Decode mode	LDR mode	HDR mode
decode_float16	Vector of FP16 values	

Decode mode	LDR mode	HDR mode
decode_unorm8	Vector of 8-bit unsigned normalized values	invalid
decode_rgb9e5	Vector using a shared exponent format	

Using the decode_unorm8 decoding mode in HDR mode gives undefined results.

For sRGB, the decoding mode is ignored, ad the decoding always returns a vector of 8-bit unsigned normalized values.

The error color is opaque fully-saturated magenta (*R,G,B,A*) = (0xFF, 0x00, 0xFF, 0xFF). This has been chosen as it is much more noticeable than black or white, and occurs far less often in valid images.

For linear *RGB* decode, the error color may be either opaque fully-saturated magenta (*R,G,B,A*) = (1.0, 0.0, 1.0, 1.0) or a vector of four *NaNs* (*R,G,B,A*) = (*NaN*, *NaN*, *NaN*, *NaN*). In the latter case, the recommended *NaN* value returned is 0xFFFF.

When using the decode_rgb9e5 decoding mode in HDR mode, error results will return the error color because NaN cannot be represented.

The error color is returned as an informative response to invalid conditions, including invalid block encodings or use of reserved endpoint modes.

Future, forward-compatible extensions to ASTC may define valid interpretations of these conditions, which will decode to some other color. Therefore, encoders and applications must not rely on invalid encodings as a way of generating the error color.

23.6. Configuration Summary

The global configuration data for the format are as follows:

- Block dimension (2D or 3D)
- Block footprint size
- sRGB output enabled or not

The data specified per block are as follows:

- Texel weight grid size
- Texel weight range
- Texel weight values
- Number of partitions
- Partition pattern index
- Color endpoint modes (includes LDR or HDR selection)
- Color endpoint data
- Number of planes
- Plane-to-channel assignment

23.7. Decode Procedure

To decode one texel:

```
(Optimization: If within known void-extent, immediately return single color)

Find block containing texel
Read block mode
If void-extent block, store void extent and immediately return single color

For each plane in image
  If block mode requires infill
    Find and decode stored weights adjacent to texel, unquantize and interpolate
  Else
    Find and decode weight for texel, and unquantize

Read number of partitions
If number of partitions > 1
  Read partition table pattern index
  Look up partition number from pattern

Read color endpoint mode and endpoint data for selected partition
Unquantize color endpoints
Interpolate color endpoints using weight (or weights in dual-plane mode)
Return interpolated color
```

23.8. Block Determination and Bit Rates

The block footprint is a global setting for any given texture, and is therefore not encoded in the individual blocks.

For 2D textures, the block footprint's width and height are selectable from a number of predefined sizes, namely 4, 5, 6, 8, 10 and 12 pixels.

For square and nearly-square blocks, this gives the bit rates in Table 141.

Table 141. ASTC 2D footprint and bit rates

Footprint		Bit Rate	Increment
Width	Height		
4	4	8.00	125%
5	4	6.40	125%
5	5	5.12	120%
6	5	4.27	120%
6	6	3.56	114%
8	5	3.20	120%
8	6	2.67	105%
10	5	2.56	120%
10	6	2.13	107%
8	8	2.00	125%
10	8	1.60	125%
10	10	1.28	120%
12	10	1.07	120%
12	12	0.89	

The "Increment" column indicates the ratio of bit rate against the next lower available rate. A consistent value in this column indicates an even spread of bit rates.

For 3D textures, the block footprint's width, height and depth are selectable from a number of predefined sizes, namely 3, 4, 5, and 6 pixels.

For cubic and near-cubic blocks, this gives the bit rates in Table 142.

Table 142. ASTC 3D footprint and bit rates

Block Footprint			Bit Rate	Increment
Width	Height	Depth		
3	3	3	4.74	133%
4	3	3	3.56	133%
4	4	3	2.67	133%
4	4	4	2.00	125%
5	4	4	1.60	125%
5	5	4	1.28	125%
5	5	5	1.02	120%
6	5	5	0.85	120%
6	6	5	0.71	120%
6	6	6	0.59	

The full profile supports only those block footprints listed in Table 141 and Table 142. Other block sizes are not supported.

For images which are not an integer multiple of the block size, additional texels are added to the edges with maximum X and Y (and Z for 3D textures). These texels may be any color, as they will not be accessed.

Although these are not all powers of two, it is possible to calculate block addresses and pixel addresses within the block, for legal image sizes, without undue complexity.

Given an image which is $W \times H \times D$ pixels in size, with block size $w \times h \times d$, the size of the image in blocks is:

[Math Processing Error]

For a 3D image built from 2D slices, each 2D slice is a single texel thick, so that for an image which is $W \times H \times D$ pixels in size, with block size $w \times h$, the size of the image in blocks is:

[Math Processing Error]

23.9. Block Layout

Each block in the image is stored as a single 128-bit block in memory. These blocks are laid out in raster order, starting with the block at (0, 0, 0), then ordered sequentially by X, Y and finally Z (if present). They are aligned to 128-bit boundaries in memory.

The bits in the block are labeled in little-endian order — the byte at the lowest address contains bits 0..7. Bit 0 is the least significant bit in the byte.

Each block has the same basic layout, shown in Table 143.

Table 143. ASTC block layout

127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112
Texel weight data (variable width)												Fill direction *[Math Processing Error]*			
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
Texel weight data															

95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
Texel weight data															
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
Texel weight data															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
					More config data										
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
[Math Processing Error] Fill direction								Color endpoint data							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
				Extra configuration data											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Extra			Part		Block mode										

Since the size of the "texel weight data" field is variable, the positions shown for the "more config data" field and "color endpoint data" field are only representative and not fixed.

The "Block mode" field specifies how the Texel Weight Data is encoded.

The "Part" field specifies the number of partitions, minus one. If dual plane mode is enabled, the number of partitions must be 3 or fewer. If 4 partitions are specified, the error value is returned for all texels in the block.

The size and layout of the extra configuration data depends on the number of partitions, and the number of planes in the image, as shown in Table 144 (only the bottom 32 bits are shown).

Table 144. ASTC single-partition block layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Color endpoint data															CEM
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CEM			0	0	Block mode										

CEM is the color endpoint mode field, which determines how the Color Endpoint Data is encoded.

If dual-plane mode is active, the color component selector bits appear directly below the weight bits, as shown in Table 145.

Table 145. ASTC multi-partition block layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			CEM						Partition index						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Partition index			Part		Block mode										

The Partition Index field specifies which partition layout to use. CEM is the first 6 bits of color endpoint mode information for the various partitions. For modes which require more than 6 bits of CEM data, the additional bits appear at a variable position directly beneath the texel weight data.

If dual-plane mode is active, the color component selector bits then appear directly below the additional CEM bits.

The final special case is that if bits [8..0] of the block are "111111100", then the block is a void-extent block, which has a separate encoding described in Section 23.23.

23.10. Block mode

The *block mode* field specifies the width, height and depth of the grid of weights, what range of values they use, and whether dual weight planes are present. Since some these are not represented using powers of two (there are 12 possible weight widths, for example), and not all combinations are allowed, this is not a simple bit packing. However, it can be unpacked quickly in hardware.

The weight ranges are encoded using a 3-bit range value ρ, which is interpreted together with a low/high-precision bit *P*, as shown in Table 146. Each weight value is encoded using the specified number of Trits, Quints and Bits. The details of this encoding can be found in Section 23.12.

Table 146. ASTC weight range encodings

$\rho^{2..0}$	Low-precision range (*P*=0)				High-precision range (*P*=1)			
	Weight range	Trits	Quints	Bits	Weight range	Trits	Quints	Bits
000	Invalid				Invalid			
001	Invalid				Invalid			
010	0..1			1	0..9		1	1
011	0..2	1			0..11	1		2
100	0..3			2	0..15			4
101	0..4		1		0..19		1	2
110	0..5	1		1	0..23	1		3
111	0..7			3	0..31			5

For 2D blocks, the Block Mode field is laid out as shown in Table 147.

Table 147. ASTC 2D block mode layout, weight grid width and height

10	9	8	7	6	5	4	3	2	1	0	W_{width}	W_{height}	Notes
D_P	P	W		H		ρ^0	0	0	ρ^2	ρ^1	W+4	H+2	
D_P	P	W		H		ρ^0	0	1	ρ^2	ρ^1	W+8	H+2	
D_P	P	H		W		ρ^0	1	0	ρ^2	ρ^1	W+2	H+8	
D_P	P	0	H	W		ρ^0	1	1	ρ^2	ρ^1	W+2	H+6	
D_P	P	1	W	H		ρ^0	1	1	ρ^2	ρ^1	W+2	H+2	
D_P	P	0	0	H		ρ^0	ρ^2	ρ^1	0	0	12	H+2	
D_P	P	0	1	W		ρ^0	ρ^2	ρ^1	0	0	W+2	12	
D_P	P	1	1	0	0	ρ^0	ρ^2	ρ^1	0	0	6	10	
D_P	P	1	1	0	1	ρ^0	ρ^2	ρ^1	0	0	10	6	
H		1	0	W		ρ^0	ρ^2	ρ^1	0	0	W+6	H+6	D_P=0, P=0
x	x	1	1	1	1	1	1	1	0	0	-	-	Void-extent
x	x	1	1	1	x	x	x	x	0	0	-	-	Reserved*
x	x	x	x	x	x	x	0	0	0	0	-	-	Reserved

Note that, due to the encoding of the ρ field, as described in the previous page, bits ρ^2 and ρ^1 cannot both be zero, which disambiguates the first five rows from the rest of the table.

Bit positions with a value of x are ignored for purposes of determining if a block is a void-extent block or reserved, but may have defined encodings for specific void-extent blocks.

The penultimate row of Table 147 is reserved only if bits [5..2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

For 3D blocks, the Block Mode field is laid out as shown in Table 148.

Table 148. ASTC 3D block mode layout, weight grid width, height and depth

10	9	8	7	6	5	4	3	2	1	0	W_{width}	W_{height}	W_{depth}	Notes
D_P	P	H		W		ρ^0	D		ρ^2	ρ^1	W+2	H+2	D+2	
H		0	0	D		ρ^0	ρ^2	ρ^1	0	0	6	H+2	D+2	D_P=0, P=0
D		0	1	W		ρ^0	ρ^2	ρ^1	0	0	W+2	6	D+2	D_P=0, P=0
H		1	0	W		ρ^0	ρ^2	ρ^1	0	0	W+2	H+2	6	D_P=0, P=0
D_P	P	1	1	0	0	ρ^0	ρ^2	ρ^1	0	0	6	2	2	
D_P	P	1	1	0	1	ρ^0	ρ^2	ρ^1	0	0	2	6	2	
D_P	P	1	1	1	0	ρ^0	ρ^2	ρ^1	0	0	2	2	6	
x	x	1	1	1	1	1	1	1	0	0	-	-	-	Void-extent
x	x	1	1	1	1	x	x	x	0	0	-	-	-	Reserved*
x	x	x	x	x	x	x	0	0	0	0	-	-	-	Reserved

The D_P bit is set to indicate dual-plane mode. In this mode, the maximum allowed number of partitions is 3.

The penultimate row of Table 148 is reserved only if bits [4..2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

The size of the weight grid in each dimension must be less than or equal to the corresponding dimension of the block footprint. If the grid size is greater than the footprint dimension in any axis, then this is an illegal block encoding and all texels will decode to the error color.

23.11. Color Endpoint Mode

In single-partition mode, the Color Endpoint Mode (CEM) field stores one of 16 possible values. Each of these specifies how many raw data values are encoded, and how to convert these raw values into two *RGBA* color endpoints. They can be summarized as shown in Table 149.

Table 149. ASTC color endpoint modes

CEM	Description	Class
0	LDR Luminance, direct	0
1	LDR Luminance, base+offset	0
2	HDR Luminance, large range	0
3	HDR Luminance, small range	0
4	LDR Luminance+Alpha, direct	1
5	LDR Luminance+Alpha, base+offset	1
6	LDR *RGB*, base+scale	1
7	HDR *RGB*, base+scale	1
8	LDR *RGB*, direct	2
9	LDR *RGB*, base+offset	2
10	LDR *RGB*, base+scale plus two *A*	2

CEM	Description	Class
11	HDR *RGB*, direct	2
12	LDR *RGBA*, direct	3
13	LDR *RGBA*, base+offset	3
14	HDR *RGB*, direct + LDR Alpha	3
15	HDR *RGB*, direct + HDR Alpha	3

In multi-partition mode, the CEM field is of variable width, from 6 to 14 bits. The lowest 2 bits of the CEM field specify how the endpoint mode for each partition is calculated as shown in Table 150.

Table 150. ASTC Multi-Partition Color Endpoint Modes

Value	Meaning
00	All color endpoint pairs are of the same type; a full 4-bit CEM is stored in block bits [28..25] and is used for all partitions
01	All endpoint pairs are of class 0 or 1
10	All endpoint pairs are of class 1 or 2
11	All endpoint pairs are of class 2 or 3

If the CEM selector value in bits [24..23] is not 00, then data layout is as shown in Table 151 and Table 152.

Table 151. ASTC multi-partition color endpoint mode layout

Part			n	m	l	k	j	i	h	g	
2	...	Weight	M_1								...
3	...	Weight	M_2		M_1		M_0				...
4	...	Weight	M_3		M_2		M_1		M_0		...

Table 152. ASTC multi-partition color endpoint mode layout (2)

Part	28	27	26	25	24	23
2	M_0		C_1	C_0	CEM	
3	M_0	C_2	C_1	C_0	CEM	
4	C_3	C_2	C_1	C_0	CEM	

In this view, each partition *i* has two fields. C_i is the class selector bit, choosing between the two possible CEM classes (0 indicates the lower of the two classes), and M_i is a two-bit field specifying the low bits of the color endpoint mode within that class. The additional bits appear at a variable bit position, immediately below the texel weight data.

The ranges used for the data values are not explicitly specified. Instead, they are derived from the number of available bits remaining after the configuration data and weight data have been specified.

Details of the decoding procedure for Color Endpoints can be found in Section 23.13.

23.12. Integer Sequence Encoding

Both the weight data and the endpoint color data are variable width, and are specified using a sequence of integer values. The range of each value in a sequence (e.g. a color weight) is constrained.

Since it is often the case that the most efficient range for these values is not a power of two, each value sequence is encoded using a technique known as "integer sequence encoding". This allows efficient, hardware-friendly packing and unpacking of values with non-power-of-two ranges.

In a sequence, each value has an identical range. The range is specified in one of the forms shown in Table 153 and Table 154.

Table 153. ASTC range forms

Value range	MSB encoding	LSB encoding
[Math Processing Error]	-	*n*-bit value *m* ($n \le 8$)
[Math Processing Error]	Base-3 "trit" value *t*	*n*-bit value *m* ($n \le 6$)
[Math Processing Error]	Base-5 "quint" value *q*	*n*-bit value *m* ($n \le 5$)

Table 154. ASTC encoding for different ranges

Value range	Value	Block	Packed block size
[Math Processing Error]	*m*	1	*n*
[Math Processing Error]	[Math Processing Error]	5	$8 + 5 \times n$
[Math Processing Error]	[Math Processing Error]	3	$7 + 3 \times n$

Since 3^5 is 243, it is possible to pack five trits into 8 bits (which has 256 possible values), so a trit can effectively be encoded as 1.6 bits. Similarly, since 5^3 is 125, it is possible to pack three quints into 7 bits (which has 128 possible values), so a quint can be encoded as 2.33 bits.

The encoding scheme packs the trits or quints, and then interleaves the *n* additional bits in positions that satisfy the requirements of an arbitrary-length stream. This makes it possible to correctly specify lists of values whose length is not an integer multiple of 3 or 5 values. It also makes it possible to easily select a value at random within the stream.

If there are insufficient bits in the stream to fill the final block, then unused (higher-order) bits are assumed to be 0 when decoding.

To decode the bits for value number *i* in a sequence of bits *b*, both indexed from 0, perform the following:

If the range is encoded as *n* bits per value, then the value is bits *[Math Processing Error]* — a simple multiplexing operation.

If the range is encoded using a trit, then each block contains 5 values (v_0 to v_4), each of which contains a trit (t_0 to t_4) and a corresponding LSB value (m_0 to m_4). The first bit of the packed block is bit *[Math Processing Error]* . The bits in the block are packed as shown in Table 155 (in this example, *n* is 4).

Table 155. ASTC trit-based packing

				27	26	25	24	23	22	21	20	19	18	17	16
				T^7	m_4				T^6	T^5	m_3				T^4
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m_2				T^3	T^2	m_1				T^1	T^0	m_0			

The five trits t_0 to t_4 are obtained by bit manipulations of the 8 bits $T^{7..0}$ as follows:

```
if T[4:2] = 111
    C = { T[7:5], T[1:0] }; t4 = t3 = 2
else
    C = T[4:0]
    if T[6:5] = 11
        t4 = 2; t3 = T[7]
    else
        t4 = T[7]; t3 = T[6:5]

if C[1:0] = 11
    t2 = 2; t1 = C[4]; t0 = { C[3], C[2]&~C[3] }
else if C[3:2] = 11
    t2 = 2; t1 = 2; t0 = C[1:0]
else
    t2 = C[4]; t1 = C[3:2]; t0 = { C[1], C[0]&~C[1] }
```

If the range is encoded using a quint, then each block contains 3 values (v_0 to v_2), each of which contains a quint (q_0 to q_2) and a corresponding LSB value (m_0 to m_2). The first bit of the packed block is bit *[Math Processing Error]* .

The bits in the block are packed as described in Table 156 and Table 157 (in this example, *n* is 4).

Table 156. ASTC quint-based packing

18	17	16
Q^6	Q^5	m_2

Table 157. ASTC quint-based packing (2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m_2			Q^4	Q^3	m_1				Q^2	Q^1	Q^0	m_0			

The three quints q_0 to q_2 are obtained by bit manipulations of the 7 bits $Q^{6..0}$ as follows:

```
if Q[2:1] = 11 and Q[6:5] = 00
    q2 = { Q[0], Q[4]&~Q[0], Q[3]&~Q[0] }; q1 = q0 = 4
else
    if Q[2:1] = 11
        q2 = 4; C = { Q[4:3], ~Q[6:5], Q[0] }
    else
        q2 = Q[6:5]; C = Q[4:0]

    if C[2:0] = 101
        q1 = 4; q0 = C[4:3]
    else
        q1 = C[4:3]; q0 = C[2:0]
```

Both these procedures ensure a valid decoding for all 128 possible values (even though a few are duplicates). They can also be implemented efficiently in software using small tables.

Encoding methods are not specified here, although table-based mechanisms work well.

23.13. Endpoint Unquantization

Each color endpoint is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding, as a stream of bits stored from just above the configuration data, and growing upwards.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..255.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers. This procedure ensures correct scaling, but scrambles the order of the decoded values relative to the encoded values. This must be compensated for using a table in the encoder.

The initial inputs to the procedure are denoted `A` (9 bits), `B` (9 bits), `C` (9 bits) and `D` (3 bits), and are decoded using the range as described in Table 158.

Table 158. ASTC color unquantization parameters

Range	#Trits	#Quints	#Bits	Bit layout	A	B	C	D
0..5	1		1	a	aaaaaaaaa	000000000	204	Trit value
0..9		1	1	a	aaaaaaaaa	000000000	113	Quint value
0..11	1		2	ba	aaaaaaaaa	b000b0bb0	93	Trit value
0..19		1	2	ba	aaaaaaaaa	b0000bb00	54	Quint value
0..23	1		3	cba	aaaaaaaaa	cb000cbcb	44	Trit value
0..39		1	3	cba	aaaaaaaaa	cb0000cbc	26	Quint value
0..47	1		4	dcba	aaaaaaaaa	dcb000dcb	22	Trit value
0..79		1	4	dcba	aaaaaaaaa	dcb0000dc	13	Quint value
0..95	1		5	edcba	aaaaaaaaa	edcb000ed	11	Trit value
0..159		1	5	edcba	aaaaaaaaa	edcb0000e	6	Quint value
0..191	1		6	fedcba	aaaaaaaaa	fedcb000f	5	Trit value

These are then processed as follows:

```
unq = D * C + B;
unq = unq ^ A;
unq = (A & 0x80) | (unq >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

23.14. LDR Endpoint Decoding

The decoding method used depends on the Color Endpoint Mode (CEM) field, which specifies how many values are used to represent the endpoint.

The CEM field also specifies how to take the *n* unquantized color endpoint values v_0 to v_{n-1} and convert them into two *RGBA* color endpoints e_0 and e_1.

The HDR Modes are more complex and do not fit neatly into this section. They are documented in following section.

The methods can be summarized as shown in Table 159.

Table 159. ASTC LDR color endpoint modes

CEM	Range	Description	*n*
0	LDR	Luminance, direct	2
1	LDR	Luminance, base+offset	2
2	HDR	Luminance, large range	2
3	HDR	Luminance, small range	2
4	LDR	Luminance+Alpha, direct	4
5	LDR	Luminance+Alpha, base+offset	4
6	LDR	*RGB*, base+scale	4
7	HDR	*RGB*, base+scale	4
8	LDR	*RGB*, direct	6
9	LDR	*RGB*, base+offset	6
10	LDR	*RGB*, base+scale plus two *A*	6
11	HDR	*RGB*	6
12	LDR	*RGBA*, direct	8
13	LDR	*RGBA*, base+offset	8
14	HDR	*RGB* + LDR Alpha	8
15	HDR	*RGB* + HDR Alpha	8

Mode 14 is special in that the alpha values are interpolated linearly, but the color components are interpolated logarithmically. This is the only endpoint format with mixed-mode operation, and will return the error value if encountered in LDR mode.

Decode the different LDR endpoint modes as follows:

23.14.1. Mode 0 LDR Luminance, direct

```
e0=(v0,v0,v0,0xFF); e1=(v1,v1,v1,0xFF);
```

23.14.2. Mode 1 LDR Luminance, base+offset

```
L0 = (v0>>2)|(v1&0xC0); L1=L0+(v1&0x3F);
if (L1>0xFF) { L1=0xFF; }
e0=(L0,L0,L0,0xFF); e1=(L1,L1,L1,0xFF);
```

23.14.3. Mode 4 LDR Luminance+Alpha,direct

```
e0=(v0,v0,v0,v2);
e1=(v1,v1,v1,v3);
```

23.14.4. Mode 5 LDR Luminance+Alpha, base+offset

```
bit_transfer_signed(v1,v0); bit_transfer_signed(v3,v2);
e0=(v0,v0,v0,v2); e1=(v0+v1,v0+v1,v0+v1,v2+v3);
clamp_unorm8(e0); clamp_unorm8(e1);
```

23.14.5. Mode 6 LDR *RGB*, base+scale

```
e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, 0xFF);
e1=(v0,v1,v2,0xFF);
```

23.14.6. Mode 8 LDR *RGB*, Direct

```
s0= v0+v2+v4; s1= v1+v3+v5;
if (s1>=s0){e0=(v0,v2,v4,0xFF);
            e1=(v1,v3,v5,0xFF); }
else { e0=blue_contract(v1,v3,v5,0xFF);
       e1=blue_contract(v0,v2,v4,0xFF); }
```

23.14.7. Mode 9 LDR *RGB*, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
if(v1+v3+v5 >= 0)
{ e0=(v0,v2,v4,0xFF); e1=(v0+v1,v2+v3,v4+v5,0xFF); }
else
{ e0=blue_contract(v0+v1,v2+v3,v4+v5,0xFF);
  e1=blue_contract(v0,v2,v4,0xFF); }
clamp_unorm8(e0); clamp_unorm8(e1);
```

23.14.8. Mode 10 LDR *RGB*, base+scale plus two *A*

```
e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, v4);
e1=(v0,v1,v2, v5);
```

23.14.9. Mode 12 LDR *RGBA*, direct

```
s0= v0+v2+v4; s1= v1+v3+v5;
if (s1>=s0){e0=(v0,v2,v4,v6);
            e1=(v1,v3,v5,v7); }
else { e0=blue_contract(v1,v3,v5,v7);
       e1=blue_contract(v0,v2,v4,v6); }
```

23.14.10. Mode 13 LDR *RGBA*, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
bit_transfer_signed(v7,v6);
if(v1+v3+v5>=0) { e0=(v0,v2,v4,v6);
       e1=(v0+v1,v2+v3,v4+v5,v6+v7); }
else { e0=blue_contract(v0+v1,v2+v3,v4+v5,v6+v7);
       e1=blue_contract(v0,v2,v4,v6); }
clamp_unorm8(e0); clamp_unorm8(e1);
```

The `bit_transfer_signed()` procedure transfers a bit from one value (*a*) to another (*b*). Initially, both *a* and *b* are in the range 0..255. After calling this procedure, *a*'s range becomes -32..31, and *b* remains in the range 0..255. Note that, as is often the case, this is easier to express in hardware than in C:

```
bit_transfer_signed(int& a, int& b)
{
    b >>= 1;
    b |= a & 0x80;
    a >>= 1;
    a &= 0x3F;
```

```
    if( (a&0x20)!=0 ) a-=0x40;
}
```

The `blue_contract()` procedure is used to give additional precision to *RGB* colors near gray:

```
color blue_contract( int r, int g, int b, int a )
{
    color c;
    c.r = (r+b) >> 1;
    c.g = (g+b) >> 1;
    c.b = b;
    c.a = a;
    return c;
}
```

The `clamp_unorm8()` procedure is used to clamp a color into 8-bit unsigned normalized fixed-point range:

```
void clamp_unorm8(color c)
{
    if(c.r < 0) {c.r=0;} else if(c.r > 255) {c.r=255;}
    if(c.g < 0) {c.g=0;} else if(c.g > 255) {c.g=255;}
    if(c.b < 0) {c.b=0;} else if(c.b > 255) {c.b=255;}
    if(c.a < 0) {c.a=0;} else if(c.a > 255) {c.a=255;}
}
```

23.15. HDR Endpoint Decoding

For HDR endpoint modes, color values are represented in a 12-bit pseudo-logarithmic representation.

23.15.1. HDR Endpoint Mode 2

Mode 2 represents luminance-only data with a large range. It encodes using two values (v_0, v_1). The complete decoding procedure is as follows:

```
if(v1 >= v0)
{
    y0 = (v0 << 4);
    y1 = (v1 << 4);
}
else
{
    y0 = (v1 << 4) + 8;
    y1 = (v0 << 4) - 8;
}
// Construct RGBA result (0x780 is 1.0f)
e0 = (y0, y0, y0, 0x780);
e1 = (y1, y1, y1, 0x780);
```

23.15.2. HDR Endpoint Mode 3

Mode 3 represents luminance-only data with a small range. It packs the bits for a base luminance value, together with an offset, into two values (v_0, v_1), according to Table 160.

Table 160. ASTC HDR mode 3 value layout

Value	7	6	5	4	3	2	1	0
v_0	M	$L^{6..0}$						
v_1	$X^{3..0}$				$d^{3..0}$			

The bit field marked as X allocates different bits to L or d depending on the value of the mode bit M.

The complete decoding procedure is as follows:

```
// Check mode bit and extract.
if((v0&0x80) !=0)
{
    y0 = ((v1 & 0xE0) << 4) | ((v0 & 0x7F) << 2);
    d  =  (v1 & 0x1F) << 2;
}
else
{
    y0 = ((v1 & 0xF0) << 4) | ((v0 & 0x7F) << 1);
    d  =  (v1 & 0x0F) << 1;
}

// Add delta and clamp
y1 = y0 + d;
if(y1 > 0xFFF) { y1 = 0xFFF; }

// Construct RGBA result (0x780 is 1.0f)
```

```
e0 = (y0, y0, y0, 0x780);
e1 = (y1, y1, y1, 0x780);
```

23.15.3. HDR Endpoint Mode 7

Mode 7 packs the bits for a base *RGB* value, a scale factor, and some mode bits into the four values (v_0, v_1, v_2, v_3), as shown in Table 161.

Table 161. ASTC HDR mode 7 value layout

Value	7	6	5	4	3	2	1	0
v_0	$M^{3..2}$		$R^{5..0}$					
v_1	M^1	X^0	X^1	$G^{4..0}$				
v_2	M^0	X^2	X^3	$B^{4..0}$				
v_3	X^4	X^5	X^6	$S^{4..0}$				

The mode bits M^0 to M^3 are a packed representation of an endpoint bit mode, together with the major component index. For modes 0 to 4, the component (red, green, or blue) with the largest magnitude is identified, and the values swizzled to ensure that it is decoded from the red channel.

The endpoint bit mode is used to determine the number of bits assigned to each component of the endpoint, and the destination of each of the extra bits X^0 to X^6, as shown in Table 162.

Table 162. ASTC HDR mode 7 endpoint bit mode

	Number of bits					Destination of extra bits						
Mode	***R***	***G***	***B***	**S**		X^0	X^1	X^2	X^3	X^4	X^5	X^6
0	11	5	5	7		R^9	R^8	R^7	R^{10}	R^6	S^6	S^5
1	11	6	6	5		R^8	G^5	R^7	B^5	R^6	R^{10}	R^9
2	10	5	5	8		R^9	R^8	R^7	R^6	S^7	S^6	S^5
3	9	6	6	7		R^8	G^5	R^7	B^5	R^6	S^6	S^5
4	8	7	7	6		G^6	G^5	B^6	B^5	R^6	R^7	S^5
5	7	7	7	7		G^6	G^5	B^6	B^5	R^6	S^6	S^5

As noted before, this appears complex when expressed in C, but much easier to achieve in hardware: bit masking, extraction, shifting and assignment usually ends up as a single wire or multiplexer.

The complete decoding procedure is as follows:

```
// Extract mode bits and unpack to major component and mode.
int majcomp; int mode; int modeval = ((v0&0xC0)>>6) | ((v1&0x80)>>5) | ((v2&0x80)>>4);

if( (modeval & 0xC ) != 0xC ) {
    majcomp = modeval >> 2; mode = modeval & 3;
} else if( modeval != 0xF ) {
    majcomp = modeval & 3;  mode = 4;
} else {
    majcomp = 0; mode = 5;
}

// Extract low-order bits of r, g, b, and s.
int red   = v0 & 0x3f; int green = v1 & 0x1f;
int blue  = v2 & 0x1f; int scale = v3 & 0x1f;

// Extract high-order bits, which may be assigned depending on mode
int x0 = (v1 >> 6) & 1; int x1 = (v1 >> 5) & 1; int x2 = (v2 >> 6) & 1;
int x3 = (v2 >> 5) & 1; int x4 = (v3 >> 7) & 1; int x5 = (v3 >> 6) & 1;
int x6 = (v3 >> 5) & 1;

// Now move the high-order xs into the right place.
int ohm = 1 << mode;
if( ohm & 0x30 ) green |= x0 << 6;
if( ohm & 0x3A ) green |= x1 << 5;
if( ohm & 0x30 ) blue |= x2 << 6;
if( ohm & 0x3A ) blue |= x3 << 5;
if( ohm & 0x3D ) scale |= x6 << 5;
if( ohm & 0x2D ) scale |= x5 << 6;
if( ohm & 0x04 ) scale |= x4 << 7;
if( ohm & 0x3B ) red |= x4 << 6;
if( ohm & 0x04 ) red |= x3 << 6;
if( ohm & 0x10 ) red |= x5 << 7;
if( ohm & 0x0F ) red |= x2 << 7;
if( ohm & 0x05 ) red |= x1 << 8;
if( ohm & 0x0A ) red |= x0 << 8;
if( ohm & 0x05 ) red |= x0 << 9;
if( ohm & 0x02 ) red |= x6 << 9;
if( ohm & 0x01 ) red |= x3 << 10;
if( ohm & 0x02 ) red |= x5 << 10;
```

```
// Shift the bits to the top of the 12-bit result.
static const int shamts[6] = { 1,1,2,3,4,5 };
int shamt = shamts[mode];
red <<= shamt; green <<= shamt; blue <<= shamt; scale <<= shamt;

// Minor components are stored as differences
if( mode != 5 ) { green = red - green; blue = red - blue; }

// Swizzle major component into place
if( majcomp == 1 ) swap( red, green );
if( majcomp == 2 ) swap( red, blue );

// Clamp output values, set alpha to 1.0
e1.r = clamp( red, 0, 0xFFF );
e1.g = clamp( green, 0, 0xFFF );
e1.b = clamp( blue, 0, 0xFFF );
e1.alpha = 0x780;
e0.r = clamp( red - scale, 0, 0xFFF );
e0.g = clamp( green - scale, 0, 0xFFF );
e0.b = clamp( blue - scale, 0, 0xFFF );
e0.alpha = 0x780;
```

23.15.4. HDR Endpoint Mode 11

Mode 11 specifies two *RGB* values, which it calculates from a number of bitfields (a, b_0, b_1, c, d_0 and d_1) which are packed together with some mode bits into the six values (v_0, v_1, v_2, v_3, v_4, v_5) as shown in Table 163.

Table 163. ASTC HDR mode 11 value layout

Value	7	6	5	4	3	2	1	0
v_0	$a^{7..0}$							
v_1	m_0	a^8	$c^{5..0}$					
v_2	m_1	X^0	$b_0^{5..0}$					
v_3	m_2	X^1	$b_1^{5..0}$					
v_4	mj_0	X^2	X^4	$d_0^{4..0}$				
v_5	mj_1	X^3	X^5	$d_1^{4..0}$				

If the major component bits $mj^{1..0}$ are both 1, then the *RGB* values are specified directly by Table 164.

Table 164. ASTC HDR mode 11 direct value layout

Value	7	6	5	4	3	2	1	0
v_0	$R_0^{11..4}$							
v_1	$R_1^{11..4}$							
v_2	$G_0^{11..4}$							
v_3	$G_1^{11..4}$							
v_4	1	$B_0^{11..5}$						
v_5	1	$B_1^{11..5}$						

The mode bits $m^{2..0}$ specify the bit allocation for the different values, and the destinations of the extra bits X^0 to X^5 as shown in Table 165.

Table 165. ASTC HDR mode 11 endpoint bit mode

	Number of bits					Destination of extra bits					
Mode	**a**	**b**	**c**	**d**		X^0	X^1	X^2	X^3	X^4	X^5
0	9	7	6	7		b_0^6	b_1^6	d_0^6	d_1^6	d_0^5	d_1^5
1	9	8	6	6		b_0^6	b_1^6	b_0^7	b_1^7	d_0^5	d_1^5
2	10	6	7	7		a^9	c^6	d_0^6	d_1^6	d_0^5	d_1^5
3	10	7	7	6		b_0^6	b_1^6	a^9	c^6	d_0^5	d_1^5
4	11	8	6	5		b_0^6	b_1^6	b_0^7	b_1^7	a^9	a^{10}
5	11	6	7	6		a^9	a^{10}	c^7	c^6	d_0^5	d_1^5
6	12	7	7	5		b_0^6	b_1^6	a^{11}	c^6	a^9	a^{10}
7	12	6	7	6		a^9	a^{10}	a^{11}	c^6	d_0^5	d_1^5

The complete decoding procedure is as follows:

```
// Find major component
int majcomp = ((v4 & 0x80) >> 7) | ((v5 & 0x80) >> 6);
```

```
// Deal with simple case first
if( majcomp == 3 ) {
    e0 = (v0 << 4, v2 << 4, (v4 & 0x7f) << 5, 0x780);
    e1 = (v1 << 4, v3 << 4, (v5 & 0x7f) << 5, 0x780);
    return;
}

// Decode mode, parameters.
int mode = ((v1&0x80)>>7) | ((v2&0x80)>>6) | ((v3&0x80)>>5);
int va  = v0 | ((v1 & 0x40) << 2);
int vb0 = v2 & 0x3f; int vb1 = v3 & 0x3f;
int vc  = v1 & 0x3f;
int vd0 = v4 & 0x7f; int vd1 = v5 & 0x7f;

// Assign top bits of vd0, vd1.
static const int dbitstab[8] = {7,6,7,6,5,6,5,6};
vd0 = signextend( vd0, dbitstab[mode] );
vd1 = signextend( vd1, dbitstab[mode] );

// Extract and place extra bits
int x0 = (v2 >> 6) & 1;
int x1 = (v3 >> 6) & 1;
int x2 = (v4 >> 6) & 1;
int x3 = (v5 >> 6) & 1;
int x4 = (v4 >> 5) & 1;
int x5 = (v5 >> 5) & 1;

int ohm = 1 << mode;
if( ohm & 0xA4 ) va |= x0 << 9;
if( ohm & 0x08 ) va |= x2 << 9;
if( ohm & 0x50 ) va |= x4 << 9;
if( ohm & 0x50 ) va |= x5 << 10;
if( ohm & 0xA0 ) va |= x1 << 10;
if( ohm & 0xC0 ) va |= x2 << 11;
if( ohm & 0x04 ) vc |= x1 << 6;
if( ohm & 0xE8 ) vc |= x3 << 6;
if( ohm & 0x20 ) vc |= x2 << 7;
if( ohm & 0x5B ) vb0 |= x0 << 6;
if( ohm & 0x5B ) vb1 |= x1 << 6;
if( ohm & 0x12 ) vb0 |= x2 << 7;
if( ohm & 0x12 ) vb1 |= x3 << 7;

// Now shift up so that major component is at top of 12-bit value
int shamt = (modeval >> 1) ^ 3;
va <<= shamt; vb0 <<= shamt; vb1 <<= shamt;
vc <<= shamt; vd0 <<= shamt; vd1 <<= shamt;

e1.r = clamp( va, 0, 0xFFF );
e1.g = clamp( va - vb0, 0, 0xFFF );
e1.b = clamp( va - vb1, 0, 0xFFF );
e1.alpha = 0x780;
e0.r = clamp( va - vc, 0, 0xFFF );
e0.g = clamp( va - vb0 - vc - vd0, 0, 0xFFF );
e0.b = clamp( va - vb1 - vc - vd1, 0, 0xFFF );
e0.alpha = 0x780;

if( majcomp == 1 )      { swap( e0.r, e0.g ); swap( e1.r, e1.g ); }
else if( majcomp == 2 ) { swap( e0.r, e0.b ); swap( e1.r, e1.b ); }
```

23.15.5. HDR Endpoint Mode 14

Mode 14 specifies two *RGBA* values, using the eight values (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7). First, the *RGB* values are decoded from (v_0..v_5) using the method from Mode 11, then the alpha values are filled in from v_6 and v_7:

```
// Decode RGB as for mode 11
(e0,e1) = decode_mode_11(v0,v1,v2,v3,v4,v5)

// Now fill in the alphas
e0.alpha = v6;
e1.alpha = v7;
```

Note that in this mode, the alpha values are interpreted (and interpolated) as 8-bit unsigned normalized values, as in the LDR modes. This is the only mode that exhibits this behavior.

23.15.6. HDR Endpoint Mode 15

Mode 15 specifies two *RGBA* values, using the eight values (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7). First, the *RGB* values are decoded from (v_0..v_5) using the method from Mode 11. The alpha values are stored in values v_6 and v_7 as a mode and two values which are interpreted according to the mode *M*, as shown in Table 166.

Table 166. ASTC HDR mode 15 alpha value layout

Value	7	6	5	4	3	2	1	0
v_6	M^0	$A^{6..0}$						
v_7	M^1	$B^{6..0}$						

The alpha values are decoded from v_6 and v_7 as follows:

```
// Decode RGB as for mode 11
(e0,e1) = decode_mode_11(v0,v1,v2,v3,v4,v5)

// Extract mode bits
mode = ((v6 >> 7) & 1) | ((v7 >> 6) & 2);
v6 &= 0x7F;
v7 &= 0x7F;

if(mode==3)
{
    // Directly specify alphas
    e0.alpha = v6 << 5; e1.alpha = v7 << 5;
}
else
{
    // Transfer bits from v7 to v6 and sign extend v7.
    v6 |= (v7 << (mode+1))) & 0x780;
    v7 &= (0x3F >> mode);
    v7 ^= 0x20 >> mode;
    v7 -= 0x20 >> mode;
    v6 <<= (4-mode); v7 <<= (4-mode);

    // Add delta and clamp
    v7 += v6;
    v7 = clamp(v7, 0, 0xFFF);
    e0.alpha = v6; e1.alpha = v7;
}
```

Note that in this mode, the alpha values are interpreted (and interpolated) as 12-bit HDR values, and are interpolated as for any other HDR component.

23.16. Weight Decoding

The weight information is stored as a stream of bits, growing downwards from the most significant bit in the block. Bit *n* in the stream is thus bit 127-*n* in the block.

For each location in the weight grid, a value (in the specified range) is packed into the stream. These are ordered in a raster pattern starting from location (0,0,0), with the X dimension increasing fastest, and the Z dimension increasing slowest. If dual-plane mode is selected, both weights are emitted together for each location, plane 0 first, then plane 1.

23.17. Weight Unquantization

Each weight plane is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..64. The procedure for doing so is similar to the color endpoint unquantization.

First, we unquantize the actual stored weight values to the range 0..63.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers.

For representations with no additional bits, the results are as shown in Table 167.

Table 167. ASTC weight unquantization values

Range	0	1	2	3	4
0..2	0	32	63	-	-
0..4	0	16	32	47	63

For other values, we calculate the initial inputs to a bit manipulation procedure. These are denoted `A` (7 bits), `B` (7 bits), `C` (7 bits), and `D` (3 bits) and are decoded using the range as shown in Table 168.

Table 168. ASTC weight unquantization parameters

Range	#Trits	#Quints	#Bits	Bit layout	A	B	C	D
0..5	1		1	a	aaaaaaa	0000000	50	Trit value
0..9		1	1	a	aaaaaaa	0000000	28	Quint value
0..11	1		2	ba	aaaaaaa	b000b0b	23	Trit value
0..19		1	2	ba	aaaaaaa	b0000b0	13	Quint value
0..23	1		3	cba	aaaaaaa	cb000cb	11	Trit value

These are then processed as follows:

```
unq = D * C + B;
unq = unq ^ A;
```

```
unq = (A & 0x20) | (unq >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

As a final step, for all types of value, the range is expanded from 0..63 up to 0..64 as follows:

```
if (unq > 32) { unq += 1; }
```

This allows the implementation to use 64 as a divisor during interpolation, which is much easier than using 63.

23.18. Weight Infill

After unquantization, the weights are subject to weight selection and infill. The infill method is used to calculate the weight for a texel position, based on the weights in the stored weight grid array (which may be a different size). The procedure below must be followed exactly, to ensure bit exact results.

The block size is specified as three dimensions along the *s*, *t* and *r* axes (B_s, B_t, B_r). Texel coordinates within the block (b_s, b_t, b_r) can have values from 0 to one less than the block dimension in that axis. For each block dimension, we compute scale factors (D_s, D_t, D_r):

[Math Processing Error]

Since the block dimensions are constrained, these are easily looked up in a table. These scale factors are then used to scale the (b_s, b_t, b_r) coordinates to a homogeneous coordinate (c_s, c_t, c_r):

```
cs = Ds * bs;
ct = Dt * bt;
cr = Dr * br;
```

This homogeneous coordinate (c_s, c_t, c_r) is then scaled again to give a coordinate (g_s, g_t, g_r) in the weight-grid space. The weight-grid is of size (W_{width}, W_{height}, W_{depth}), as specified in the block mode field (Table 147 and Table 148):

```
gs = (cs*(Wwidth-1)+32) >> 6;
gt = (ct*(Wheight-1)+32) >> 6;
gr = (cr*(Wdepth-1)+32) >> 6;
```

The resulting coordinates may be in the range 0..176. These are interpreted as 4:4 unsigned fixed point numbers in the range 0.0 .. 11.0.

If we label the integral parts of these (j_s, j_t, j_r) and the fractional parts (f_s, f_t, f_r), then:

```
js = gs >> 4; fs = gs & 0x0F;
jt = gt >> 4; ft = gt & 0x0F;
jr = gr >> 4; fr = gr & 0x0F;
```

These values are then used to interpolate between the stored weights. This process differs for 2D and 3D.

For 2D, bilinear interpolation is used:

```
v0 = js + jt*Wwidth;
p00 = decode_weight(v0);
p01 = decode_weight(v0 + 1);
p10 = decode_weight(v0 + Wwidth);
p11 = decode_weight(v0 + Wwidth + 1);
```

The function `decode_weight(n)` decodes the nth weight in the stored weight stream. The values p_{00} to p_{11} are the weights at the corner of the square in which the texel position resides. These are then weighted using the fractional position to produce the effective weight *i* as follows:

```
w11 = (fs*ft+8) >> 4;
w10 = ft - w11;
w01 = fs - w11;
w00 = 16 - fs - ft + w11;
i = (p00*w00 + p01*w01 + p10*w10 + p11*w11 + 8) >> 4;
```

For 3D, simplex interpolation is used as it is cheaper than a naïve trilinear interpolation. First, we pick some parameters for the interpolation based on comparisons of the fractional parts of the texel position as shown in Table 169.

Table 169. ASTC simplex interpolation parameters

$f_s > f_t$	$f_t > f_r$	$f_s > f_r$	s_1	s_2	w_0	w_1	w_2	w_3
True	True	*True*	1	W_{width}	$16 - f_s$	$f_s - f_t$	$f_t - f_r$	f_r
False	*True*	True	W_{width}	1	$16 - f_t$	$f_t - f_s$	$f_s - f_r$	f_r
True	False	True	1	$W_{width} \times W_{height}$	$16 - f_s$	$f_s - f_r$	$f_r - f_t$	f_t
True	*False*	False	$W_{width} \times W_{height}$	1	$16 - f_r$	$f_r - f_s$	$f_s - f_t$	f_t
False	True	False	W_{width}	$W_{width} \times W_{height}$	$16 - f_t$	$f_t - f_r$	$f_r - f_s$	f_s

$f_s > f_t$	$f_t > f_r$	$f_s > f_r$	s_1	s_2	w_0	w_1	w_2	w_3
False	False	*False*	$W_{width} \times W_{height}$	W_{width}	$16 - f_r$	$f_r - f_t$	$f_t - f_s$	f_s

Italicized test results are implied by the others. The effective weight *i* is then calculated as:

```
v0 = js + jt*Wwidth + jr*Wwidth*Wheight;
p0 = decode_index(v0);
p1 = decode_index(v0 + s1);
p2 = decode_index(v0 + s1 + s2);
p3 = decode_index(v0 + Wwidth*Wheight + Wwidth + 1);
i = (p0*w0 + p1*w1 + p2*w2 + p3*w3 + 8) >> 4;
```

23.19. Weight Application

Once the effective weight *i* for the texel has been calculated, the color endpoints are interpolated and expanded.

For LDR endpoint modes, each color component C is calculated from the corresponding 8-bit endpoint components C_0 and C_1 as follows:

If sRGB conversion is not enabled, or for the alpha channel in any case, C_0 and C_1 are first expanded to 16 bits by bit replication:

```
C0 = (C0 << 8) | C0;     C1 = (C1 << 8) | C1;
```

If sRGB conversion is enabled, C_0 and C_1 for the *R*, *G*, and *B* channels are expanded to 16 bits differently, as follows:

```
C0 = (C0 << 8) | 0x80;  C1 = (C1 << 8) | 0x80;
```

C_0 and C_1 are then interpolated to produce a UNORM16 result C:

```
C = floor( (C0*(64-i) + C1*i + 32)/64 )
```

If sRGB conversion is not enabled and the decoding mode is decode_float16, then if C = 65535 the final result is 1.0 (0x3C00); otherwise C is divided by 65536 and the infinite-precision result of the division is converted to FP16 with round-to-zero semantics.

If sRGB conversion is not enabled and the decoding mode is decode_unorm8, then the top 8 bits of the interpolation result for the *R*, *G*, *B* and *A* channels are used as the final result.

If sRGB conversion is not enabled and the decoding mode is decode_rgb9e5, then the final result is a combination of the (UNORM16) values of C for the three color components (C_r, C_g and C_b) computed as follows:

```
int lz = clz17(Cr | Cg | Cb | 1);
if (Cr == 65535) { Cr = 65536; lz = 0; }
if (Cg == 65535) { Cg = 65536; lz = 0; }
if (Cb == 65535) { Cb = 65536; lz = 0; }
Cr <<= lz; Cg <<= lz; Cb <<= lz;
Cr = (Cr >> 8) & 0x1FF;
Cg = (Cg >> 8) & 0x1FF;
Cb = (Cb >> 8) & 0x1FF;
uint32_t exponent = 16 - lz;
uint32_t texel = (exponent << 27) | (Cb << 18) | Cg << 9) | Cr;
```

The `clz17()` function counts leading zeroes in a 17-bit value.

if sRGB conversion is enabled, then the decoding mode is ignored and the top 8 bits of the interpolation result for the *R*, *G* and *B* channels are passed to the external sRGB conversion block and used as the final result. The *A* channel uses the decode_float16 decoding mode.

For HDR endpoint modes, color values are represented in a 12-bit pseudo-logarithmic representation, and interpolation occurs in a piecewise-approximate logarithmic manner as follows:

In LDR mode, the error result is returned.

In HDR mode, the color components from each endpoint, C_0 and C_1, are initially shifted left 4 bits to become 16-bit integer values and these are interpolated in the same way as LDR. The 16-bit value C is then decomposed into the top five bits, E, and the bottom 11 bits M, which are then processed and recombined with E to form the final value C_f:

```
C = floor( (C0*(64-i) + C1*i + 32)/64 )
E = (C & 0xF800) >> 11; M = C & 0x7FF;
if (M < 512) { Mt = 3*M; }
else if (M >= 1536) { Mt = 5*M - 2048; }
else { Mt = 4*M - 512; }
Cf = (E<<10) + (Mt>>3);
```

This interpolation is a considerably closer approximation to a logarithmic space than simple 16-bit interpolation.

This final value C_f is interpreted as an IEEE FP16 value. If the result is +Inf or NaN, it is converted to the bit pattern 0x7BFF, which is the largest representable finite value.

If the decoding mode is decode_rgb9e5, then the final result is a combination fo the (IEEE FP16) values of C_f for the three color components (C_r, C_g and C_b) computed as follows:

```
if( Cr > 0x7c00 ) Cr = 0; else if( Cr == 0x7c00 ) Cr = 0x7bff;
if( Cg > 0x7c00 ) Cg = 0; else if( Cg == 0x7c00 ) Cg = 0x7bff;
if( Cb > 0x7c00 ) Cb = 0; else if( Cb == 0x7c00 ) Cb = 0x7bff;
int Re = (Cr >> 10) & 0x1F;
int Ge = (Cg >> 10) & 0x1F;
int Be = (Cb >> 10) & 0x1F;
int Rex = Re == 0 ? 1 : Re;
int Gex = Ge == 0 ? 1 : Ge;
int Bex = Be == 0 ? 1 : Be;
int Xm = ((Cr | Cg | Cb) & 0x200) >> 9;
int Xe = Re | Ge | Be;
uint32_t rshift, gshift, bshift, expo;

if (Xe == 0)
{
    expo = rshift = gshift = bshift = Xm;
}
else if (Re >= Ge && Re >= Be)
{
    expo = Rex + 1;
    rshift = 2;
    gshift = Rex - Gex + 2;
    bshift = Rex - Bex + 2;
}
else if (Ge >= Be)
{
    expo = Gex + 1;
    rshift = Gex - Rex + 2;
    gshift = 2;
    bshift = Gex - Bex + 2;
}
else
{
    expo = Bex + 1;
    rshift = Bex - Rex + 2;
    gshift = Bex - Gex + 2;
    bshift = 2;
}

int Rm = (Cr & 0x3FF) | (Re == 0 ? 0 : 0x400);
int Gm = (Cg & 0x3FF) | (Ge == 0 ? 0 : 0x400);
int Bm = (Cb & 0x3FF) | (Be == 0 ? 0 : 0x400);
Rm = (Rm >> rshift) & 0x1FF;
Gm = (Gm >> gshift) & 0x1FF;
Bm = (Bm >> bshift) & 0x1FF;

uint32_t texel = (expo << 27) | (Bm << 18) | (Gm << 9) | (Rm << 0);
```

23.20. Dual-Plane Decoding

If dual-plane mode is disabled, all of the endpoint components are interpolated using the same weight value.

If dual-plane mode is enabled, two weights are stored with each texel. One component is then selected to use the second weight for interpolation, instead of the first weight. The first weight is then used for all other components.

The component to treat specially is indicated using the 2-bit Color Component Selector (CCS) field as shown in Table 170.

Table 170. ASTC dual plane color component selector values

Value	Weight 0	Weight 1
0	*GBA*	*R*
1	*RBA*	*G*
2	*RGA*	*B*
3	*RGB*	*A*

The CCS bits are stored at a variable position directly below the weight bits and any additional CEM bits.

23.21. Partition Pattern Generation

When multiple partitions are active, each texel position is assigned a partition index. This partition index is calculated using a seed (the partition pattern index), the texel's *x*, *y*, *z* position within the block, and the number of partitions. An additional argument, `small_block`, is set to 1 if the number of texels in the block is less than 31, otherwise it is set to 0.

This function is specified in terms of *x*, *y* and *z* in order to support 3D textures. For 2D textures and texture slices, *z* will always be 0.

The full partition selection algorithm is as follows:

```
int select_partition(int seed, int x, int y, int z,
                     int partitioncount, int small_block)
{
    if( small_block ){ x <<= 1; y <<= 1; z <<= 1; }
    seed += (partitioncount-1) * 1024;
    uint32_t rnum = hash52(seed);
    uint8_t seed1  =  rnum        & 0xF;
    uint8_t seed2  = (rnum >>  4) & 0xF;
    uint8_t seed3  = (rnum >>  8) & 0xF;
    uint8_t seed4  = (rnum >> 12) & 0xF;
    uint8_t seed5  = (rnum >> 16) & 0xF;
    uint8_t seed6  = (rnum >> 20) & 0xF;
    uint8_t seed7  = (rnum >> 24) & 0xF;
    uint8_t seed8  = (rnum >> 28) & 0xF;
    uint8_t seed9  = (rnum >> 18) & 0xF;
    uint8_t seed10 = (rnum >> 22) & 0xF;
    uint8_t seed11 = (rnum >> 26) & 0xF;
    uint8_t seed12 = ((rnum >> 30) | (rnum << 2)) & 0xF;

    seed1  *= seed1;    seed2  *= seed2;
    seed3  *= seed3;    seed4  *= seed4;
    seed5  *= seed5;    seed6  *= seed6;
    seed7  *= seed7;    seed8  *= seed8;
    seed9  *= seed9;    seed10 *= seed10;
    seed11 *= seed11;   seed12 *= seed12;

    int sh1, sh2, sh3;
    if( seed & 1 )
        { sh1 = (seed&2 ? 4:5); sh2 = (partitioncount==3 ? 6:5); }
    else
        { sh1 = (partitioncount==3 ? 6:5); sh2 = (seed&2 ? 4:5); }
    sh3 = (seed & 0x10) ? sh1 : sh2:

    seed1 >>= sh1; seed2  >>= sh2; seed3  >>= sh1; seed4  >>= sh2;
    seed5 >>= sh1; seed6  >>= sh2; seed7  >>= sh1; seed8  >>= sh2;
    seed9 >>= sh3; seed10 >>= sh3; seed11 >>= sh3; seed12 >>= sh3;

    int a = seed1*x + seed2*y + seed11*z + (rnum >> 14);
    int b = seed3*x + seed4*y + seed12*z + (rnum >> 10);
    int c = seed5*x + seed6*y + seed9 *z + (rnum >>  6);
    int d = seed7*x + seed8*y + seed10*z + (rnum >>  2);

    a &= 0x3F; b &= 0x3F; c &= 0x3F; d &= 0x3F;

    if( partitioncount < 4 ) d = 0;
    if( partitioncount < 3 ) c = 0;

    if( a >= b && a >= c && a >= d ) return 0;
    else if( b >= c && b >= d ) return 1;
    else if( c >= d ) return 2;
    else return 3;
}
```

As has been observed before, the bit selections are much easier to express in hardware than in C.

The seed is expanded using a hash function `hash52()`, which is defined as follows:

```
uint32_t hash52( uint32_t p )
{
    p ^= p >> 15;  p -= p << 17;  p += p << 7; p += p <<  4;
    p ^= p >>  5;  p += p << 16;  p ^= p >> 7; p ^= p >> 3;
    p ^= p <<  6;  p ^= p >> 17;
    return p;
}
```

This assumes that all operations act on 32-bit values

23.22. Data Size Determination

The size of the data used to represent color endpoints is not explicitly specified. Instead, it is determined from the block mode and number of partitions as follows:

```
config_bits = 17;
if(num_partitions>1)
    if(single_CEM)
        config_bits = 29;
    else
        config_bits = 25 + 3*num_partitions;

num_weights = Wwidth * Wheight * Wdepth; // size of weight grid

if(dual_plane)
    config_bits += 2;
    num_weights *= 2;

weight_bits = ceil(num_weights*8*trits_in_weight_range/5) +
              ceil(num_weights*7*quints_in_weight_range/3) +
```

```
                num_weights*bits_in_weight_range;

remaining_bits = 128 - config_bits - weight_bits;

num_CEM_pairs = base_CEM_class+1 + count_bits(extra_CEM_bits);
```

The CEM value range is then looked up from a table indexed by remaining bits and `num_CEM_pairs`. This table is initialized such that the range is as large as possible, consistent with the constraint that the number of bits required to encode `num_CEM_pairs` pairs of values is not more than the number of remaining bits.

An equivalent iterative algorithm would be:

```
num_CEM_values = num_CEM_pairs*2;

for(range = each possible CEM range in descending order of size)
{
    CEM_bits = ceil(num_CEM_values*8*trits_in_CEM_range/5) +
               ceil(num_CEM_values*7*quints_in_CEM_range/3) +
               num_CEM_values*bits_in_CEM_range;

    if(CEM_bits <= remaining_bits)
        break;
}
return range;
```

In cases where this procedure results in unallocated bits, these bits are not read by the decoding process and can have any value.

23.23. Void-Extent Blocks

A void-extent block is a block encoded with a single color. It also specifies some additional information about the extent of the single-color area beyond this block, which can optionally be used by a decoder to reduce or prevent redundant block fetches.

In the HDR case, if the decoding mode is decode_rgb9e5, then any negative color component values are set to 0 before conversion to the shared exponent format (as described in Section 23.19).

The layout of a 2D Void-Extent block is as shown in Table 171.

Table 171. ASTC 2D void-extent block layout overview

127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112
Block color *A* component$^{15..0}$															
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
Block color *B* component$^{15..0}$															
95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
Block color *G* component$^{15..0}$															
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
Block color *R* component$^{15..0}$															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
Void-extent maximum *t* coordinate$^{12..0}$													Min *t* coord$^{12..10}$		
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Void-extent minimum *t* coordinate$^{9..0}$										Void-extent maximum *s* coordinate$^{12..7}$					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Void-extent maximum *s* coordinate$^{5..0}$							Void-extent minimum *s* coordinate$^{12..4}$								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Minimum *s* coordinate$^{3..0}$				1	1	D	1	1	1	1	1	1	1	0	0

The layout of a 3D Void-Extent block is as shown in Table 172.

Table 172. ASTC 3D void-extent block layout overview

127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112
Block color *A* component$^{15..0}$															
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
Block color *B* component$^{15..0}$															
95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
Block color *G* component$^{15..0}$															
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
Block color *R* component$^{15..0}$															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
Void-extent maximum *r* coordinate$^{8..0}$									Void-extent minimum *r* coordinate$^{8..2}$						
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Min *r* coord$^{1..0}$		Void-extent maximum *t* coordinate$^{8..0}$									Void-extent min *t* coordinate$^{8..4}$				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Minimum *t* coordinate$^{3..0}$				Void-extent minimum *s* coordinate$^{8..0}$									Min *s* coord$^{8..6}$		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Void-extent minimum *s* coordinate$^{5..0}$						D	1	1	1	1	1	1	1	0	0

Bit 9 is the Dynamic Range flag, which indicates the format in which colors are stored. A 0 value indicates LDR, in which case the color components are stored as UNORM16 values. A 1 indicates HDR, in which case the color components are stored as FP16 values.

The reason for the storage of UNORM16 values in the LDR case is due to the possibility that the value will need to be passed on to sRGB conversion. By storing the color value in the format which comes out of the interpolator, before the conversion to FP16, we avoid having to have separate versions for sRGB and linear modes.

If a void-extent block with HDR values is decoded in LDR mode, then the result will be the error color, opaque magenta, for all texels within the block.

In the HDR case, if the color component values are infinity or NaN, this will result in undefined behavior. As usual, this must not lead to an API's interruption or termination.

Bits 10 and 11 are reserved and must be 1.

The minimum and maximum coordinate values are treated as unsigned integers and then normalized into the range 0..1 (by dividing by 2^{13}-1 or 2^{9}-1, for 2D and 3D respectively). The maximum values for each dimension must be greater than the corresponding minimum values, unless they are all all-1s.

If all the coordinates are all-1s, then the void extent is ignored, and the block is simply a constant-color block.

The existence of single-color blocks with void extents must not produce results different from those obtained if these single-color blocks are defined without void-extents. Any situation in which the results would differ is invalid. Results from invalid void extents are undefined.

If a void-extent appears in a MIPmap level other than the most detailed one, then the extent will apply to all of the more detailed levels too. This allows decoders to avoid sampling more detailed MIPmaps.

If the more detailed MIPmap level is not a constant color in this region, then the block may be marked as constant color, but without a void extent, as detailed above.

If a void-extent extends to the edge of a texture, then filtered texture colors may not be the same color as that specified in the block, due to texture border colors, wrapping, or cube face wrapping.

Care must be taken when updating or extracting partial image data that void-extents in the image do not become invalid.

23.24. Illegal Encodings

In ASTC, there is a variety of ways to encode an illegal block. Decoders are required to recognize all illegal blocks and emit the standard error color value upon encountering an illegal block.

Here is a comprehensive list of situations that represent illegal block encodings:

- The block mode specified is one of the modes explicitly listed as Reserved.
- A 2D void-extent block that has any of the reserved bits not set to 1.
- A block mode has been specified that would require more than 64 weights total.
- A block mode has been specified that would require more than 96 bits for integer sequence encoding of the weight grid.
- A block mode has been specified that would require fewer than 24 bits for integer sequence encoding of the weight grid.
- The size of the weight grid exceeds the size of the block footprint in any dimension.
- Color endpoint modes have been specified such that the color integer sequence encoding would require more than 18 integers.
- The number of bits available for color endpoint encoding after all the other fields have been counted is less than *[Math Processing Error]* where C is the number of color endpoint integers (this would restrict color integers to a range smaller than 0..5, which is not supported).
- Dual weight mode is enabled for a block with 4 partitions.
- Void-Extent blocks where the low coordinate for some texture axis is greater than or equal to the high coordinate.

Note also that, in LDR mode, a block which has both HDR and LDR endpoint modes assigned to different partitions is not an error block. Only those texels which belong to the HDR partition will result in the error color. Texels belonging to a LDR partition will be decoded as normal.

23.25. LDR PROFILE SUPPORT

In order to ease verification and accelerate adoption, an LDR-only subset of the full ASTC specification has been made available.

Implementations of this LDR Profile must satisfy the following requirements:

- All textures with valid encodings for LDR Profile must decode identically using either a LDR Profile, HDR Profile, or Full Profile decoder.
- All features included only in the HDR Profile or Full Profile must be treated as reserved in the LDR Profile, and return the error color on decoding.
- Any sequence of API calls valid for the LDR Profile must also be valid for the HDR Profile or Full Profile and return identical results when given a texture encoded for the LDR Profile.

The feature subset for the LDR profile is:

- 2D textures only.
- Only those block sizes listed in Table 141 are supported.
- LDR operation mode only.
- Only LDR endpoint formats must be supported, namely formats 0, 1, 4, 5, 6, 8, 9, 10, 12, 13.
- Decoding from a HDR endpoint results in the error color.
- Interpolation returns UNORM8 results when used in conjunction with sRGB.
- LDR void extent blocks must be supported, but void extents may not be checked.

23.26. HDR PROFILE SUPPORT

In order to ease verification and accelerate adoption, a second subset of the full ASTC specification has been made available, known as the HDR profile.

Implementations of the HDR Profile must satisfy the following requirements:

- The HDR profile is a superset of the LDR profile and therefore all valid LDR encodings must decode identically using a HDR profile decoder.
- All textures with valid encodings for HDR Profile must decode identically using either a HDR Profile or Full Profile decoder.
- All features included only in the Full Profile must be treated as reserved in the HDR Profile, and return the error color on decoding.
- Any sequence of API calls valid for the HDR Profile must also be valid for the Full Profile and return identical results when given a texture encoded for the HDR Profile.

The feature subset for the HDR profile is:

- 2D textures only.
- Only those block sizes listed in Table 141 are supported.
- All endpoint formats must be supported.
- 2D void extent blocks must be supported, but void extents may not be checked.

24. PVRTC Compressed Texture Image Formats

This description is derived from the PVRTC Texture Compression User Guide, part of the PowerVR Documentation.

24.1. PVRTC Overview

PVRTC is PowerVR's family of proprietary texture compression schemes, providing compression of color data at 4 or 2 bits per pixel (4/2bpp).

There are two generations of PVRTC: PVRTC1 and PVRTC2. Both support 4bpp and 2bpp compression ratios. They are broadly similar, but PVRTC2 adds additional features to the format. Both primarily use an interpolation and modulation scheme to compress texture data wherein texel values are encoded as two low-resolution images, $\mathbf{A_{Low}}$ and $\mathbf{B_{Low}}$, along with a full-resolution, low bit-precision modulation signal, $\mathbf{M_{Sig}}$, to combine those images.

More information on the specifics of the PVRTC1 compression technique can be found in the "Graphics Hardware 2003" paper Texture Compression using Low-Frequency Signal Modulation.

In PVRTC, images are described in terms of 64-bit little-endian words, each of which contains a pixel from each of the low-resolution images, $\mathbf{A_{Low}}$ and $\mathbf{B_{Low}}$, and a subset of the modulation data, $\mathbf{M_{Sig}}$, corresponding to either a 4×4 or 8×4 set of pixels. Unlike traditional block-based formats, PVRTC uses adjacent data-words to reconstruct the original image.

If combined with this encoding scheme, the sRGB EOTF is applied to the unquantized version of the R', G' and B' channels (that is, *[Math Processing Error]* , etc.) at the end of the texel decode process, but the A channel is interpreted linearly.

24.2. Format PVRTC1 4bpp

For PVRTC1 4bpp, each 64-bit word $W_{X,Y}$ at block coordinates (X,Y) contains the modulation information for a 4×4 block of texels beginning at image pixels *[Math Processing Error]* , and a color sample for each low-resolution image which influences an overlapping 7×7 texel region of the final output, with each sample centered on output pixel *[Math Processing Error]* . Nearly every texel location requires data from a set of 2×2 data words in order to be decoded: the low-resolution encoded images are bilinearly interpolated to, in effect, generate a pair of full-resolution images prior to modulation, and this interpolation requires examining multiple data words in the compressed image data. At least one word of PVRTC data must exist in each mip level of a texture. If a texture (or a particular mip-level) is smaller than the size of the block of texels described by one word in any dimension, reconstruction of an image occurs as if fetching the upper-left most texels from the region covered by those words. PVRTC1 images must have dimensions that are power of two values.

24.2.1. PVRTC1 4bpp word encoding

Each data word contains two colors and a set of modulation data describing how to interpolate between them for each pixel location in the original image. A flag M describes how the modulation data should be interpreted, and each color contains a bit to describe whether it contains alpha data.

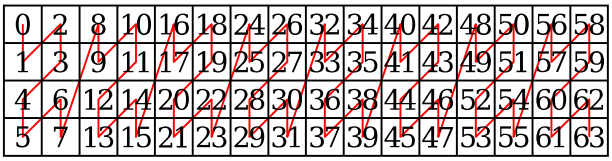
Table 173. Texel Data format for PVRTC1 4bpp compressed texture formats

Bits 64..32: Color data and flags																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Color B																**Color A**															*M*
Bits 31..0: Modulation data bits [1..0] for pixel offset (*x*, *y*)																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3,3		2,3		1,3		0,3		3,2		2,2		1,2		0,2		3,1		2,1		1,1		0,1		3,0		2,0		1,0		0,0	

24.2.2. PVRTC1 4bpp word offset calculation

Data words in PVRTC1 formats are stored in a reflected Morton order, as shown in the figure below, where each cell corresponds to the index of a 64-bit word $W_{X,Y}$:

Figure 51. Reflected Morton Order 16×4-block image



Expressing each of the *X* and *Y* indices as an array of bits, the index of a particular PVRTC word can be found by interleaving the bits of the *X* and *Y* indices as follows:

Let *Xb* be the number of bits used to express *X*—i.e. *[Math Processing Error]* .

Let *Yb* be the number of bits used to express *Y*—i.e. *[Math Processing Error]* .

Then:

[Math Processing Error]

That is, to form the word offset, bits of *X* and *Y* are interleaved, with bits from *Y* in the lower of each interleaved pair. These bits are interleaved up to the number of bits needed to represent the largest possible *X* or *Y*, for whichever of *width* and *height* is smaller. Any remaining bits from *X* or *Y*, for whichever of *width* and *height* is larger, are appended to the offset bit pattern.

For example, Figure 51 represents a 64×16-texel image represented by 16×4 = 64 words of 64 bits. The largest possible *X* value in this example is *[Math Processing Error]* = 15; the largest possible *Y* value in this example is *[Math Processing Error]* = 3. The bottom four bits of the word offset are composed by interleaving the bottom two bits of *Y* and the bottom two bits of *X* (with *Y* in the lowest bit). Bits 5..4 of the word offset are derived from bits 3..2 of *X*, since no further bits are required to represent *Y*.

In this case, the word at *X*=13, *Y*=2, $W_{13,2}$=54 is constructed as follows:

Table 174. Calculation of reflected Morton word offset for 13,2

X	**1**	**1**	**0**		**1**		= 13
$W_{13,2}$	**1**	**1**	**0**	*1*	**1**	*0*	= 54
Y			*1*		*0*		= 2

Where *wordWidth* and *wordHeight* are the image width and height in units of 4×4 areas encoded by words:

[Math Processing Error]

the word offset for *[Math Processing Error]* and *[Math Processing Error]* can be calculated iteratively as follows:

```
uint32_t reflectedMortonOffset(const uint32_t X,
                               const uint32_t Y,
                               const uint32_t wordWidth,
                               const uint32_t wordHeight)
{
    const uint32_t minDim = (wordWidth <= wordHeight) ? wordWidth : wordHeight;
    uint32_t offset = 0, shift = 0, mask;

    // Tests XY bounds AND that Width and Height != 0
    assert(X < wordWidth && Y < wordHeight);

    // Must be (non-zero) powers of 2
    assert((wordWidth  & (wordWidth  - 1)) == 0);
    assert((wordHeight & (wordHeight - 1)) == 0);

    for (mask = 1; mask < minDim; mask <<= 1)
    {
        offset |= (((Y & mask) | ((X & mask) <<  1))) << shift;
        shift++;
    }

    // At least one of X or Y will have run out of MSBs
    offset |= ((X | Y) >> shift) << (shift * 2);
```

```
    return offset;
}
```

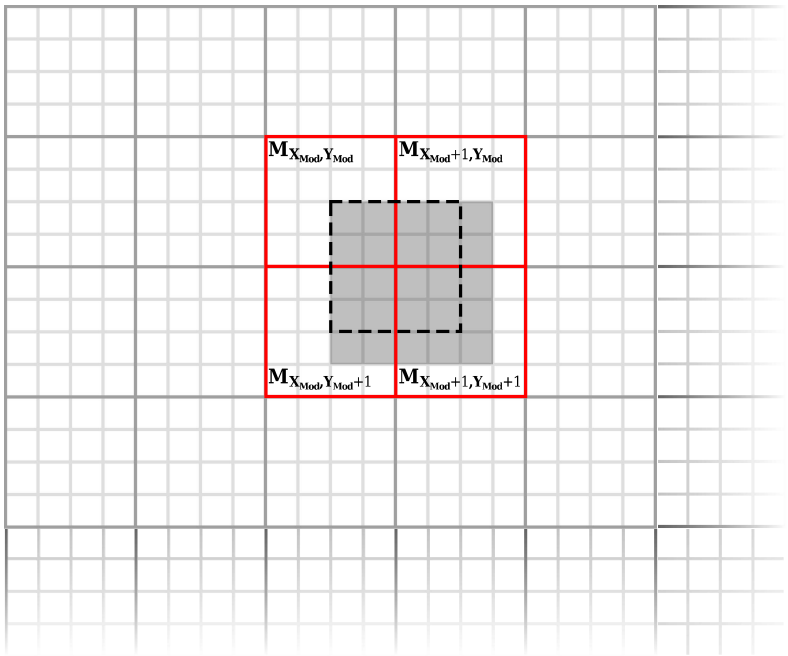
24.2.3. PVRTC1 4bpp color reconstruction samples

Each data word encodes a color sample value from each of the two low-resolution images, with the (X, Y) location of the block corresponding to the (X_{Low}, Y_{Low}) location of the colors in the low-resolution images. The image colors for a given pixel location (x, y) are reconstructed using the words containing the four nearest color samples: *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* , where X_{Low} and Y_{Low} are derived as follows:

[Math Processing Error]

Figure 52 shows a grid of pixels with (x = 0, y = 0) at top left. Each word *[Math Processing Error]* holds modulation values for a 4×4 texel region *[Math Processing Error]* , as described in Section 24.2.5, where *[Math Processing Error]* and *[Math Processing Error]* .
For *[Math Processing Error]* and *[Math Processing Error]* , color reconstruction for the pixels shaded in Figure 52 requires data from the words *[Math Processing Error]* through *[Math Processing Error]* ; the pixels for which these these words hold modulation values are shown as *[Math Processing Error]* through *[Math Processing Error]* , outlined in red.
All pixels within the region contained by the dashed outline have the same values for *[Math Processing Error]* and *[Math Processing Error]* . The remaining shaded pixels have different calculated *[Math Processing Error]* and/or *[Math Processing Error]* values, but due to Equation 10, no contribution is required from additional words.

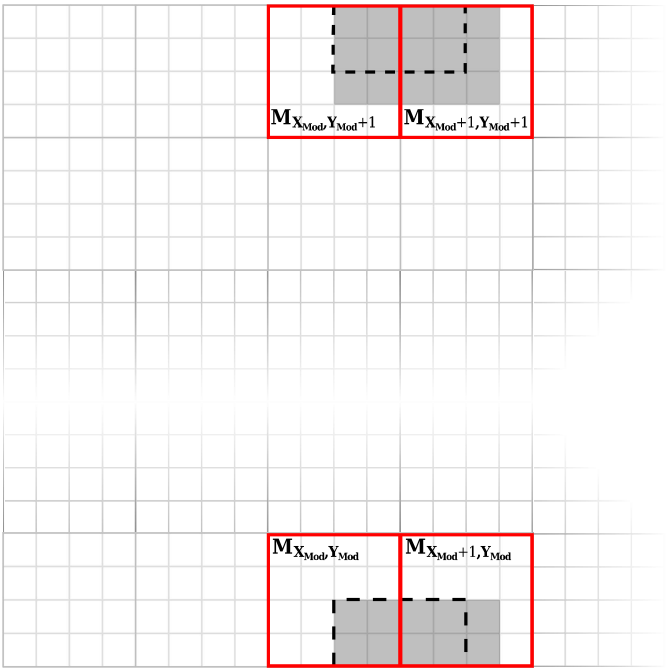
Figure 52. PVRTC1 image reconstruction



The texture data words are wrapped toroidally, such that the "nearest" sample may exist on the opposite side of the image.

For example, sampling a pixel in any corner of the image results in the words in all four corners being examined — or sampling a pixel at the bottom of the image will result in words from the top of the image being examined, as shown in Figure 53. In this example, the nearest samples "below" the shaded pixels in regions *[Math Processing Error]* and *[Math Processing Error]* the row of words at the top of the image, and the nearest samples "above" the shaded pixels in regions *[Math Processing Error]* and *[Math Processing Error]* are in words *[Math Processing Error]* and *[Math Processing Error]* at the bottom of the image.

Figure 53. PVRTC1 image reconstruction (wrapping)



24.2.4. PVRTC1 4bpp image reconstruction

The layout of color data in PVRTC word depends on the value of the opacity flag *Op*, stored in the most-significant bit of the color. If *Op* is 1, then the color is treated as opaque (no alpha data), and if *Op* is 0, the color has alpha data and may be translucent. The exact data layout of each color is described below:

Table 175. Data layout of color segments in a PVRTC1 word

Color B — opaque color mode (opacity flag *Op* = 1)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
1	Red					Green					Blue				
Color B — translucent color mode (opacity flag *Op* = 0)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
0	Alpha			Red				Green				Blue			
Color A — opaque color mode (opacity flag *Op* = 1)															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	Red					Green					Blue				*M*
Color A — translucent color mode (opacity flag *Op* = 0)															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	Alpha			Red				Green				Blue			*M*

The **Color A** values and **Color B** values for each word are bilinearly interpolated by a factor of 4 in both dimensions, resulting in a pair of color values from two virtual images, **Image A** and **Image B**. This upscale operation is performed by treating each color as an *ARGB*:4555 format, and generating results in an *ARGB*:8888 format. Any *R*, *G*, or *B* channel in each color with fewer than 5 bits is initially expanded via bit replication:

- a 3-bit channel, $C^2C^1C^0$ becomes $C^2C^1C^0C^2C^1$
- a 4-bit channel, $C^3C^2C^1C^0$ becomes $C^3C^2C^1C^0C^3$

The 3-bit alpha channel values are expanded to 4 bits by zero padding:

- $A^2A^1A^0$ becomes $A^2A^1A^0 0$

If the color is *opaque*, then the alpha channel is expanded to the 4-bit value, 0b1111.

For each channel *C* of each color (**Color A** and **Color B**), the interpolation proceeds as follows:

- For low-resolution image color channel *[Math Processing Error]* stored in word *[Math Processing Error]* :
  - X_{Low} = *[Math Processing Error]* , as described above.
  - Y_{Low} = *[Math Processing Error]* , as described above.
- Using relative coordinates:
  - *[Math Processing Error]*
  - *[Math Processing Error]*
- Coordinates wrap at the edges of the image.
- An interpolation is performed which is mathematically equivalent to Equation 10.

Equation 10. PVRTC1 4bpp color channel interpolation

[Math Processing Error]

> The colors of **Image A** and **Image B** at (*[Math Processing Error]* , *[Math Processing Error]*) are exactly the corresponding colors that word *[Math Processing Error]* encodes. Texels in the same column as a texel block such that *[Math Processing Error]* are not influenced by the color samples from words *[Math Processing Error]* and *[Math Processing Error]* . Texels in the same row as a texel block such that *[Math Processing Error]* are not influenced by the color samples from words *[Math Processing Error]* and *[Math Processing Error]* . Therefore a single color sample at *[Math Processing Error]* influences the interpolated color of all texels in the 7×7 region from *[Math Processing Error]* to *[Math Processing Error]* centered on the color sample.
>
> Any 2×2 quad of texel values can be evaluated from a single set of four adjacent texel blocks. This means that the number of texel blocks accessed during bilinear filtering is no worse than the worst case of the self-contained texel blocks of other schemes.

For the red, green and blue channels, $C_{x,y}$ is a 5.4-bit fixed-point value whose bit pattern can be converted to an 8-bit normalized value, i.e. UNORM, as *[Math Processing Error]* .

For the alpha channel, $C_{x,y}$ is a 4.4-bit fixed-point value whose bit pattern can be converted to an 8-bit normalized value as *[Math Processing Error]* .

24.2.5. PVRTC1 4bpp color modulation

The final image is created by linearly interpolating between the **Image A** and **Image B** texels, using the modulation data for each pixel to determine the weighting. The modulation information is retrieved from word *[Math Processing Error]* where *[Math Processing Error]* and *[Math Processing Error]* . The weight for the interpolation is derived from the 2 bits of the modulation data corresponding to the relevant pixel offset ($x_{offset} = x - (4 \times X_{Mod})$, $y_{offset} = y - (4 \times Y_{Mod})$), depending on the value of modulation flag M.

Table 176. Modulation weights for PVRTC1 4bpp

Standard bilinear (M = 0)		**Punch-through** (M = 1)		
Modulation bits	**Weight**	**Modulation bits**	**Weight**	**Alpha**
00	0	00	0	Normal
01	3	01	4	
10	5	10		"Punch-through"
11	8	11	8	Normal

Equation 11. PVRTC image modulation

[Math Processing Error]

If punch-through mode is selected, and the modulation bits for a given pixel have a value of 0b10, the alpha value of the resulting color is 0x00. This is irrespective of the presence or values of any alpha channel in the input colors.

> For punch-through pixels, the *RGB* components are 50:50 blends of the corresponding pixels in the upscaled images. For this reason, with PVRTC1 4bpp, it is advised to not use pre-multiplied alpha textures, and to change the color of fully transparent areas to the average of the local neighborhood. PVRTexTool provides "alpha bleed" functionality to modify fully-transparent areas appropriately.

24.3. Format PVRTC1 2bpp

PVRTC1 2bpp has the same broad data layout as PVRTC1 4bpp, but instead uses an 8×4 bilinear upscale.

24.3.1. PVRTC1 2bpp word offset calculation

The inputs to the Morton order encoding for 2bpp mode are:

Let Xb be the number of bits used to express X—i.e. *[Math Processing Error]* .

Let Yb be the number of bits used to express Y—i.e. *[Math Processing Error]* .

24.3.2. PVRTC1 2bpp image reconstruction

For each channel C of each color (**Color A** and **Color B**), the interpolation proceeds as follows:

- For low-resolution image color channel *[Math Processing Error]* stored in word *[Math Processing Error]* :
  - X_{Low} = *[Math Processing Error]*
  - Y_{Low} = *[Math Processing Error]*
- Using relative coordinates:
  - *[Math Processing Error]*
  - *[Math Processing Error]*

- Coordinates wrap at the edges of the image.
- An interpolation is performed which is mathematically equivalent to Equation 12.

Equation 12. PVRTC1 2bpp color channel interpolation

[Math Processing Error]

For the red, green and blue channels, $C_{x,y}$ is a 5.5-bit fixed-point value whose bit pattern can be converted to an 8-bit normalized value as *[Math Processing Error]* .

For the alpha channel, $C_{x,y}$ is a 4.5-bit fixed-point value whose bit pattern can be converted to an 8-bit normalized value as *[Math Processing Error]* .

24.3.3. PVRTC1 2bpp color modulation

The modulation data, retrieved from word $W_{X,Y}$ where *[Math Processing Error]* and *[Math Processing Error]* , are interpreted differently in order to accommodate the additional pixels. Each word holds the modulation data which corresponds to pixels that have offsets ($x_{offset} = x - 8 \times X_{Mod}$, $y_{offset} = y - 4 \times Y_{Mod}$).

In PVRTC1 2bpp, rather than changing the weight values as in PVRTC1 4bpp, the modulation flag M in bit 32 affects the modulation data layout. Optional flags in bit 0 and bit 20 of the modulation data may further affect the layout:

Table 177. Texel Data format for PVRTC1 2bpp compressed texture formats

Bits 64..32: Color data and flags																															
Identical to PVRTC1 4bpp																															
Bits 31..0: Modulation data: direct encoding, 1 bit per pixel (modulation flag M = 0) for pixel offset (x, y)																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7,3	6,3	5,3	4,3	3,3	2,3	1,3	0,3	7,2	6,2	5,2	4,2	3,2	2,2	1,2	0,2	7,1	6,1	5,1	4,1	3,1	2,1	1,1	0,1	7,0	6,0	5,0	4,0	3,0	2,0	1,0	0,0
Bits 31..0: Modulation data, checkerboard-interpolated encoding, samples for pixel offset (x, y), bits[1..0] (modulation flag M = 1, bit 0 flag = 0)																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7,3		5,3		3,3		1,3		6,2		4,2		2,2		0,2		7,1		5,1		3,1		1,1		6,0		4,0		2,0		0,0	0
Bits 31..0: Modulation data, horizontally- or vertically-interpolated encoding, samples for pixel offset (x, y) (modulation flag M = 1, bit 0 flag = 1)																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7,3		5,3		3,3		1,3		6,2		4,2	F	2,2		0,2		7,1		5,1		3,1		1,1		6,0		4,0		2,0		0,0	1

If the modulation flag M is set to 0, each pixel only has a single bit of modulation data. The selected color is **Image A** for modulation data bit = 0, and **Image B** for modulation data bit = 1.

Table 178. Modulation modes for PVRTC1 2bpp

Modulation flag value M	**Mode**
0	Standard Bilinear, 1bpp modulation
1	Punch-through, interpolated modulation

If the modulation flag M is set to 1, the pixels with 2-bit stored values have modulation weights equal to those of PVRTC1 4bpp for modulation mode 1, as shown in Table 176.

The modulation data for the pixel at 0,0 (and when bit_0 = 1, the pixel at 2,4) is treated as if the value was duplicated — so the single-bit 0 encoding becomes 0b00, and 1 becomes 0b11, such that the texels always correspond to **Image A** or **Image B**.

For pixels without stored modulation data, bit_0 and flag F in bit 20 determine how they are reconstructed:

- If bit_0 is 0, the value is the mean of the weights of the four horizontally- and vertically-adjacent pixels, rounded to the nearest integer: *[Math Processing Error]* .
- If bit_0 is 1, and flag F is 1, the value is the mean of the weights of the two vertically-adjacent pixels, rounded to the nearest integer: *[Math Processing Error]* .
- If bit_0 is 1, and flag F is 0, the value is the mean of the weights from the horizontally-adjacent pixels, rounded to the nearest integer: *[Math Processing Error]* .

where $md(x,y)$ is the modulation data for texel offset (x, y) and $w()$ is the weighting described in Table 176 for M = 1.

If an adjacent pixel's modulation value is not present in the current word, the value is obtained from the adjacent PVRTC word which does contain that pixel's modulation data, with the location wrapping to the other side of the image if necessary.

This weight is then applied to Equation 11.

24.4. Format PVRTC2 4bpp

PVRTC2 is the second revision of the PVRTC compression scheme, and shares most of its layout and interpretation. The PVRTC2 sections document the differences between the formats.

PVRTC2 images may have logical dimensions of any size greater than zero. If the logical size of the image is not a multiple of the interpolation size, the stored format uses the next-larger multiple. Any padding texels added by this resize are not accessed when

reconstructing the image.

PVRTC2 words are not laid out in a reflected Morton order, instead they are laid out in a standard linear order, as with all other formats.

The layout of PVRTC2 data words is very similar to that of PVRTC1, though there are two main differences:

- There is only a single opacity flag *Op* that affects both colors, rather than per-color flags.
- A *hard transition* flag *H* is included to aid representing color discontinuities, or diverse color distributions.

Table 179. Texel Data format for PVRTC2 4bpp compressed texture formats

Bits 64..32: Color data and flags																																
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Op	**Color B**															*H*	**Color A**														*M*
Bits 31..0: Modulation data bits [1..0] for pixel offset (*x*, *y*) — identical to PVRTC1 4bpp																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3,3		2,3		1,3		0,3		3,2		2,2		1,2		0,2		3,1		2,1		1,1		0,1		3,0		2,0		1,0		0,0	

Table 180. Data layout of color segments in a PVRTC2 word

Color B — opaque color mode (opacity flag *Op* = 1)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
1	Red					Green					Blue				
Color B — translucent color mode (opacity flag *Op* = 0)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
0	Alpha			Red				Green				Blue			
Color A — opaque color mode (opacity flag *Op* = 1)															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
H	Red					Green					Blue				*M*
Color A — translucent color mode (opacity flag *Op* = 0)															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
H	Alpha			Red				Green				Blue			*M*

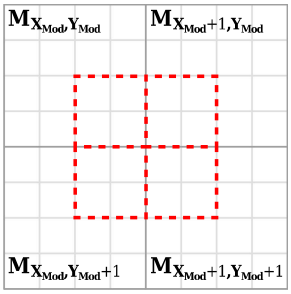
There is one change to the interpretation of the color data relative to PVRTC1: as there is only one opacity flag for each PVRTC2 data word, to allow a local area to span the full alpha gamut when in translucent mode, **Color B**'s 3-bit alpha channel is expanded to four bits as $A^2A^1A^01$, instead of $A^2A^1A^00$.

24.4.1. Hard transition flag

The bilinear interpolation scheme of PVRTC1 usually works well as most areas of natural image textures are reasonably 'continuous'. However at the boundaries of non-tiling textures, around sub-textures in texture atlases, or in some areas of hand-drawn graphics, this assumption can break down. To address these issues, PVRTC2 includes the *hard transition flag*, *H*.

Since it can be assumed that such discontinuities are more likely to be centered on boundaries of multiples of 4×4 texel regions, the hard transition flag *H* changes the behavior of the entire red-dotted region shown in Figure 54. This is a subset of the logical 4×4 pixel regions *[Math Processing Error]* through *[Math Processing Error]* that correspond to the modulation data stored in 64-bit data words *[Math Processing Error]* through *[Math Processing Error]* . The flag *H* for this hard transition region is stored in *[Math Processing Error]* .

Figure 54. PVRTC2 hard transition subsets



The hard transition region is further subdivided into four smaller subregions, shown with the dotted 2×2 texel outlines in Figure 54, where it intersects the pixel regions *[Math Processing Error]* through *[Math Processing Error]* . The hard transition flag *H*, coupled with the relevant modulation flag *M* for the texel subregion, determines how the colors for each reconstructed pixel in the subregion are evaluated, as summarized in the table below.

Table 181. Modulation modes for PVRTC2 4bpp

Modulation flag *M*	Hard transition flag *H*	Mode
0	0	Standard bilinear
1	0	Punch-through alpha
0	1	Non-interpolated
1	1	Local palette

In 'standard bilinear' the modulation behaves as described for PVRTC1 4bpp.

In 'punch-through alpha', the modulation behaves in almost the same manner as the equivalent mode as for PVRTC1 4bpp except that for texels marked as 'punch-through'; i.e. using the 2-bit encoding 0b10, the output texel is set to transparent black, which may be better suited to pre-multiplied texture formats.

The remaining two modes are described below.

24.4.2. Non-interpolated

In the non-interpolated mode, the **A** and **B** base colors are not bilinearly interpolated in the affected regions. Instead, the colors for the word encapsulating each particular pixel are used directly, in the sense that the stored colors are expanded, where necessary, via bit replication to an *ARGB*:4555 format and then, again, by bit replication to *ARGB*:8888. The modulation encodings are interpreted in the same manner as for the "standard bilinear" weights, and the colors blended, as before, with Equation 11.

24.4.3. Local palette mode

In local palette mode, the hard transition region is no longer reconstructed by interpolating the upscaled images. Instead, the eight distinct colors from each surrounding word make up a local palette from which colors are selected.

Denoting **Color B** and **Color A** from words *[Math Processing Error]* through *[Math Processing Error]* as described above, the following colors are available: *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* , *[Math Processing Error]* and *[Math Processing Error]* .

Whilst 8 distinct colors exist in each of those four words, only two bits of modulation data are available for each pixel. Subsequently, each pixel at offset *[Math Processing Error]* relative to start of the hard transition region has access to a subset of the palette as follows:

Table 182. Color mappings in local palette mode for PVRTC2 4bpp

Modulation bits	0,0	1,0	2,0	3,0
0	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
1	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
2	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
3	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
Modulation bits	**0,1**	**1,1**	**2,1**	**3,1**
0	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
1	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
2	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
3	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
Modulation bits	**0,2**	**1,2**	**2,2**	**3,2**
0	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
1	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
2	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
3	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
Modulation bits	**0,3**	**1,3**	**2,3**	**3,3**
0	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
1	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
2	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*
3	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*	*[Math Processing Error]*

The entry for offset 0,0 is interpolated as per PVRTC's standard bilinear filtering mode. It will thus only use colors from word *[Math Processing Error]* .

The local palette mode shares with the other modes of PVRTC2 the property that the column *[Math Processing Error]* has no contribution from words *[Math Processing Error]* and *[Math Processing Error]* , and that row *[Math Processing Error]* has no contribution from words *[Math Processing Error]* and *[Math Processing Error]* . Therefore any 2×2 quad of texel values, required for example by bilinear filtering, can be evaluated from a single set of four adjacent texel blocks.

- The modulation values correspond to the same entry in the list for each pixel above. For instance, a modulation value of 3 (bit pattern 11) for pixel location 0,1 (which is offset 2,3 relative to the top left of block P) would correspond to color *[Math Processing Error]* being selected.
- The stored color values are first expanded, *where necessary*, to *ARGB*:4555 (via bit replication for *R*, *G*, or *B* and via padding for Alpha) and then from 4555 to *ARGB*:8888, again by bit replication. For example,
  - A three-bit alpha value for **Color A**, $A^2A^1A^0$, is initially mapped to $A^2A^1A^0$0, and then to $A^2A^1A^0$0 $A^2A^1A^0$0
  - A four-bit color value, $C^3C^2C^1C^0$, is first mapped to $C^3C^2C^1C^0C^3$ and then subsequently to $C^3C^2C^1C^0C^3C^3C^2C^1$

24.5. Format PVRTC2 2bpp

PVRTC2 2bpp data layout has color data laid out identically to PVRTC2 4bpp, and modulation data equivalent to PVRTC1 2bpp. The only difference between the PVRTC1 and PVRTC2 variants is the addition of the hard transition flag *H* and the single opacity flag *Op*, as specified in the PVRTC2 4bpp format.

Table 183. Texel Data format for PVRTC2 2bpp compressed texture formats

Bits 63-32: Color data and flags
Identical to PVRTC2 4bpp
Bits 31-0: Modulation data

Identical to PVRTC1 2bpp

Color values are interpreted in the same manner as for the PVRTC2 4bpp format.

The 2bpp variation of PVRTC2 is slightly simpler than the 4bpp, in that it only uses the non-interpolated mode — no local palette mode exists.

Table 184. Modulation modes for PVRTC2 2bpp

Modulation flag *M*	**Hard transition** *H*	Mode
0	0	Standard bilinear, 1bpp modulation
1	0	Standard bilinear, interpolated modulation
0	1	Non-interpolated, 1bpp modulation
1	1	Non-interpolated, interpolated modulation

If the hard transition flag *H* for PVRTC2 2bpp is equal to 0, the format is interpreted in the same manner as the PVRTC1 2bpp format.

When the hard transition flag *H* for PVRTC2 2bpp is equal to 1, the initial bilinear interpolation of block colors across the hard transition region is skipped as for PVRTC2 4bpp. Subsequently, the format is interpreted in the same way as the PVRTC1 2bpp format.

When interpreting modulation data in the interpolated modulation mode, the hard transition flag has no effect on this — the modulation values are always interpolated across PVRTC words.

25. External references

IEEE754-2008 - IEEE standard for floating-point arithmetic

IEEE Std 754-2008 http://dx.doi.org/10.1109/IEEESTD.2008.4610935, August, 2008.

CIE Colorimetry - Part 3: CIE tristimulus values

http://cie.co.at/index.php?i_ca_id=823

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

http://www.itu.int/rec/R-REC-BT.601/en

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

https://www.itu.int/rec/R-REC-BT.709/en

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

http://www.itu.int/rec/R-REC-BT.2020/en

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

http://www.itu.int/rec/R-REC-BT.2100/en

JPEG File Interchange Format (JFIF)

https://www.itu.int/rec/T-REC-T.871/en

Legacy version:

https://www.w3.org/Graphics/JPEG/jfif3.pdf

ITU-R BT.1886: Reference electro-optical transfer function for flat panel displays used in HDTV studio production

https://www.itu.int/rec/R-REC-BT.1886/en

ITU-R BT.2087 Colour conversion from Recommendation ITU-R BT.709 to Recommendation ITU-R BT.2020

http://www.itu.int/rec/R-REC-BT.2087/en

ITU-R BT.2390-1 High dynamic range television for production and international programme exchange

https://www.itu.int/pub/R-REP-BT.2390-5-2018

ITU-R BT.470 Conventional analogue television systems

https://www.itu.int/rec/R-REC-BT.470/en

BT.470-6 contains descriptions of analog broadcast systems. BT.470-7 deprecates this description in favor of BT.1700.

Although this specification is written in English, the countries in Appendix 1 appear to be listed in alphabetical order as they would have been written in French.

ITU-R BT.472-3: Video-frequency characteristics of a television system to be used for the international exchange of programmes between countries that have adopted 625-line colour or monochrome systems

http://www.itu.int/rec/R-REC-BT.472/en

ITU-R BT.1700: Characteristics of composite video signals for conventional analogue television systems

https://www.itu.int/rec/R-REC-BT.1700/en

This specification includes SMTPE170M-2004 (which describes NTSC) along with PAL and NTSC.

ITU-R BT.2043: Analogue television systems currently in use throughout the world

https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-BT.2043-2004-PDF-E.pdf

Although this specification is written in English, the countries appear to be listed in alphabetical order as they would have been written in French.

SMPTE 170m Composite analog video signal — NTSC for studio applications

The latest version is available from https://www.smpte.org/ and from the IEEE at:

http://ieeexplore.ieee.org/document/7291416/.

SMPTE 170m-2004 is freely available as part of ITU-R BT.1700 at:

https://www.itu.int/rec/R-REC-BT.1700/en

FCC 73.682 - TV transmission standards

https://www.gpo.gov/fdsys/search/pagedetails.action?collectionCode=CFR&browsePath=Title+47%2FChapter+I%2FSubchapter+C%2FPart+73%2FSubpart+E%2FSection+73.682&granuleId=CFR2001-title47-vol4-sec73-682&packageId=CFR-2001-title47-vol4&collapse=true&fromBrowse=true

ST 240:1999 - SMPTE Standard - For Television — 1125-Line High-Definition Production Systems — Signal Parameters

Formerly known as SMPTE240M - interim HDTV standard prior to international agreement on BT.709.

Available from the SMPTE and via IEEE:

http://ieeexplore.ieee.org/document/7291461/

IEC/4WD 61966-2-1: Colour measurement and management in multimedia systems and equipment - part 2-1: default RGB colour space - sRGB

https://webstore.iec.ch/publication/6169 (specification)

http://www.w3.org/Graphics/Color/srgb

IEC 61966-2-2:2003: Multimedia systems and equipment — Colour measurement and management — Part 2-2: Colour management — Extended RGB colour space — scRGB

https://www.iso.org/standard/35876.html

http://webstore.iec.ch/webstore/webstore.nsf/artnum/029678

http://www.color.org/chardata/rgb/scrgb.xalter

https://webstore.iec.ch/preview/info_iec61966-2-2%7Bed1.0%7Den.pdf

A working draft is freely available at https://web.archive.org/web/20110725185444/http://www.colour.org/tc8-05/Docs/colorspace/61966-2-2NPa.pdf.

http://www.color.org/sycc.pdf - sYCC (the *[Math Processing Error]* variant of sRGB)

https://webstore.iec.ch/corrigenda/iec61966-2-2-cor1%7Bed1.0%7Den.pdf - Annex B: Non-linear encoding for scRGB : scRGB-nl and its YCC Transformation: scYCC-nl

DCI P3 color space

- SMPTE 428-1: D-Cinema Distribution Master — Image Characteristics http://ieeexplore.ieee.org/document/7290876/
- SMPTE EG 432-1: Digital Source Processing — Color Processing for D-Cinema http://ieeexplore.ieee.org/document/7289763/
- SMPTE RP 431-2: D-Cinema Quality — Reference Projector and Environment http://ieeexplore.ieee.org/document/7290729/

The latest version is available from https://www.smpte.org/

https://developer.apple.com/reference/coregraphics/cgcolorspace/1408916-displayp3 describes Apple's Display P3 color space.

Academy Color Encoding System

http://www.oscars.org/science-technology/sci-tech-projects/aces/aces-documentation

The international standard for ACES, SMPTE ST 2065-1:2012 - Academy Color Encoding Specification (ACES), is available from the SMPTE, and also from the IEEE.

TB-2014-004: Informative Notes on SMPTE ST 2065-1 – Academy Color Encoding Specification (ACES) is freely available and contains a draft of the international standard.

ACEScc — A Logarithmic Encoding of ACES Data for use within Color Grading Systems

ACEScct — A Quasi-Logarithmic Encoding of ACES Data for use within Color Grading Systems

Sony S-Log

https://pro.sony.com/bbsccms/assets/files/mkt/cinema/solutions/slog_manual.pdf - S-Log description

https://pro.sony.com/bbsccms/assets/files/micro/dmpc/training/S-Log2_Technical_PaperV1_0.pdf - S-Log2 description

http://www.sony.co.uk/pro/support/attachment/1237494271390/1237494271406/technical-summary-for-s-gamut3-cine-s-log3-and-s-gamut3-s-log3.pdf - S-Log3 description

Adobe RGB (1998)

https://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf

https://www.adobe.com/digitalimag/adobergb.html

26. Contributors

Frank Brill

Mark Callow

Sean Ellis

Simon Fenney

Jan-Harald Fredriksen

Andrew Garrard

Fabien Giesen

Courtney Goeltzenleuchter

Jonas Gustavsson

Chris Hebert

Tobias Hector

Alexey Knyazev

Daniel Koch

Jon Leech

Thierry Lepley

Tommaso Maestri

Kathleen Mattson

Hans-Peter Nilsson

XianQuan Ooi

Alon Or-bach

Jan Outters

Erik Rainey

Daniel Rakos

Donald Scorgie

Graham Sellers

David Sena

Stuart Smith

Alex Walters

Eric Werness

David Wilkinson