Khronos Data Format Specification

Andrew Garrard

Version 1.0, Revision 5

March, 2019

COLLABORATORS

	TITLE : Khronos Data Format Specification		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Andrew Garrard	March, 2019	
	Frank Brill	March, 2019	
	Mark Callow	March, 2019	
	Sean Ellis	March, 2019	
	Jonas Gustavsson	March, 2019	
	Chris Hebert	March, 2019	
	Daniel Koch	March, 2019	
	Thierry Lepley	March, 2019	
	Tommaso Maestri	March, 2019	
	Kathleen Mattson	March, 2019	
	Hans-Peter Nilsson	March, 2019	
	Alon Or-bach	March, 2019	
	Erik Rainey	March, 2019	
	Daniel Rakos	March, 2019	
	Graham Sellers	March, 2019	
	David Sena	March, 2019	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	Jan 2015	Initial sharing	AG
0.2	Feb 2015	Added clarification, tables, examples	AG
0.3	Feb 2015	Further cleanup	AG
0.4	Apr 2015	Channel ordering standardized	AG
0.5	Apr 2015	Typos and clarification	AG
1.0	May 2015	Submission for 1.0 release	AG
1.0 rev 2	Jun 2015	Clarifications for 1.0 release	AG
1.0 rev 3	Jul 2015	Added KHR_DF_SAMPLE_DATATYPE_LINEAR	AG
1.0 rev 4	Jul 2015	Clarified KHR_DF_SAMPLE_DATATYPE_LINEAR	AG
1.0 rev 5	Mar 2019	Clarification and typography	AG

Contents

List of Figures

List of Tables

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.

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.

Overview

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 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 strides, tiling or dimensionality. An example pixel format is described in Figure 1: a single 5:6:5-bit pixel with blue in the low 5 bits, green in the next 6 bits, and red in the top 5 bits of a 16-bit word as laid out in memory on a little-endian machine (see Table 24).

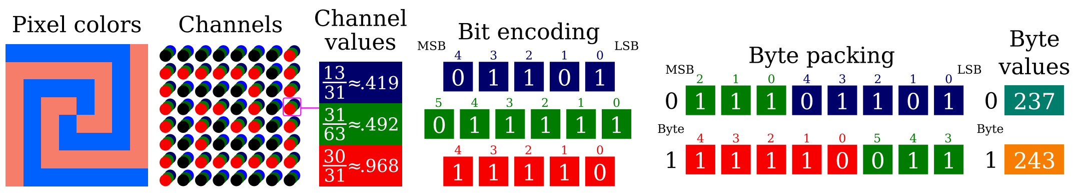

Figure 1: A simple one-texel texel block

In some cases, the elements of bulk texture data may not correspond to a conventional texel. For example, in a compressed texture it is common for the atomic element of the buffer to represent a rectangular block of texels. Alternatively the representation of the output of a camera may have a repeating pattern according to a Bayer or other layout. It is this repeating and self-contained atomic unit, termed a *texel block*, that is described by this standard.

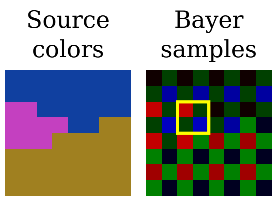

Figure 2: A Bayer-sampled image with a repeating 2×2 RG/GB texel block

The sampling or reconstruction of texel data is not a function of the data format. That is, a texture has the same format whether it is point sampled or a bicubic filter is used, and the manner of reconstructing full color data from a camera sensor is not defined. Where information making up the data format has a spatial aspect, this is part of the descriptor: it is part of the descriptor to define the spatial configuration of color samples in a Bayer sensor or whether the chroma difference channels in a $Y'C_BC_R$ format are considered to be centered or co-sited, but not how this information must be used to generate coordinate-aligned full color values.

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. 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 applications 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.

Data format descriptor
Descriptor block 1
Descriptor block 2
:

Table 1: Data format descriptor and descriptor blocks

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.

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 with subsampled channels, the texel block may describe 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.

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.

Plane: In some formats, a texel block is not contiguous in memory. In a two-dimensional texture, the texel block may be spread across multiple scan lines, or channels may be stored independently. 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.

In many existing APIs, planes may be “downsampled” differently. For example, in these APIs, a $Y'C_BC_R$ (colloquially *YUV*) 4:2:0 buffer as in Table 2 (with byte offsets shown for each channel/location) would typically be represented with three planes (Table 3), one for each channel, with the luma (Y') plane containing four times as many pixels as the chroma (C_B and C_R) planes, and with two horizontal lines of the luma held within the same plane for each horizontal line of the chroma planes.

Y' channel			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
C_B channel			
	16	17	
	18	19	
C_R channel			
	20	21	
	22	23	

Table 2: Possible memory representation of a 4×4 $Y'C_BC_R$ 4:2:0 buffer

Y' plane	offset 0	byte stride 4	downsample 1×1
C_B plane	offset 16	byte stride 2	downsample 2×2
C_R plane	offset 20	byte stride 2	downsample 2×2

Table 3: Plane descriptors for the above $Y'C_BC_R$-format buffer in a conventional API

This approach does not extend logically to more complex formats such as a Bayer grid. Therefore in this specification, we would instead define the luma channel as in Table 4, using two planes, vertically interleaved (in a linear mapping between addresses and samples) by the selection of a suitable offset and line stride, with each line of luma samples contiguous in memory. Only one plane is used for each of the chroma channels (or one plane collectively if the chroma samples are stored adjacently).

Y' plane 1	offset 0	byte stride 8	plane bytes 2
Y' plane 2	offset 4	byte stride 8	plane bytes 2
C_B plane	offset 16	byte stride 2	plane bytes 1
C_R plane	offset 20	byte stride 2	plane bytes 1

Table 4: Plane descriptors for the above $Y'C_BC_R$-format buffer using this standard

The same approach can be used to represent a static interlaced image, with a texel block consisting of two planes, one per field. This mechanism is all that is required to represent a static image without downsampled channels; however correct reconstruction of interlaced, downsampled color difference formats (such as $Y'C_BC_R$), which typically involves interpolation of the nearest chroma samples in a given *field* rather than the whole *frame*, is beyond the remit of this specification. There are many proprietary and often heuristic approaches to sample reconstruction, particularly for Bayer-like formats and for multi-frame images, and it is not practical to document them here.

There is no expectation that the internal format used by an API that wishes to make use of the Khronos Data Format Specification must use this specification’s representation internally: reconstructing downsampling information from this standard’s representation in order to revert to the more conventional representation should be trivial if required.

There is no requirement that the number of bytes occupied by the texel block be the same in each plane. The descriptor defines the number of bytes that the texel block occupies in each plane, which for most formats is sufficient to allow access to consecutive elements. For a two-dimensional data structure, it is up to the controlling interface to resolve byte stride between consecutive lines. For a three-dimensional structure, the controlling API may need to add a level stride.

Since these strides are determined by the data size and architecture alignment requirements, they are not considered to be part of the format.

Required concepts not contained in this format

This specification encodes how atomic data should be interpreted in a manner which is independent of the layout and dimensionality of the collective data. Collections of data may have a “compatible format” in that their format descriptor may be identical, yet be different sizes. Some additional information is therefore expected to be recorded alongside the “format description”.

The API which controls the bulk data is responsible for controlling which memory location corresponds to the indexing mechanism chosen. A texel block has the concept of a coordinate offset within the block, which implies that if the data is accessed in terms of spatial coordinates, a texel block has spatial locality as well as referring to contiguous memory (per plane). For texel blocks which represent only a single spatial location, this is irrelevant; for block-based compression, for formats with downsampled channels, or for Bayer-like formats, the texel block represents a finite extent in up to four dimensions. However, the mapping from coordinate system to the memory location containing a texel block is beyond the control of this API.

The minimum requirements for accessing a linearly-addressed buffer is to store the start address and a stride (typically in bytes) between texels in each dimension of the buffer, for each plane contributing to the texel block. For the first dimension, the memory stride between texels may simply be the byte size of texel block in that plane — this implies that there are no gaps between texel blocks. For other dimensions, the stride is a function of the size of the data structure being represented — for example, in a compact representation of a two-dimensional buffer, the texel block at coordinate $(x,y+1)$ might be found at the address of coordinate (x,y) plus the buffer width multiplied by the texel size in bytes. Similarly in a three-dimensional buffer, the address of the pixel at $(x,y,z+1)$ may be at the address of (x,y,z) plus the byte size of a two-dimensional slice of the texture. In practice, even linear layouts may have padding, and often more complex relationships between coordinates and memory location are used to encourage locality of reference. The details of all of these data structures are beyond the remit of this specification.

Most simple formats contain a single *plane* of data. Those formats which require additional planes compared with a conventional representation are typically downsampled $Y'C_BC_R$ formats, which already have the concept of separate storage for different color channels. While this specification uses multiple planes to describe texel blocks that span multiple scan lines if the data is disjoint, there is no expectation that the API using the data formats needs to maintain this representation — interleaved planes should be easy to identify and coalesce if the API requires a more conventional representation of downsampled formats.

Some image representations are composed of tiles of texels which are held contiguously in memory, with the texels within the tile stored in some order that improves locality of reference for multi-dimensional access. This is a common approach to improve memory efficiency when texturing. While it is possible to represent such a tile as a large texel block (up to the maximum representable texel block size in this specification), this is unlikely to be an efficient approach, since a large number of samples will be needed and the layout of a tile usually has a very limited number of possibilities. In most cases, the layout of texels within the tile should be described by whatever interface is aware of image-specific information such as size and stride, and only the format of the texels should be described by a format descriptor.

The complication to this is where texel blocks larger than a single pixel are themselves encoded using proprietary tiling. The spatial layout of samples within a texel block is required to be fixed in the basic format descriptor — for example, if the texel block size is 2×2 pixels, the top left pixel might always be expected to be in the first byte in that texel block. In some proprietary memory tiling formats, such as ones that store small rectangular blocks in raster order in consecutive bytes or in Morton order, this relationship may be preserved, and the only proprietary operation is finding the start of the texel block. In other proprietary layouts such as Hilbert curver order, or when the texel block size does not divide the tiling size, a direct representation of memory may be impossible. In these cases, it is likely that this data format standard would be used to describe the data as it would be seen in a linear format, and the mapping from coordinates to memory would have to be hidden in proprietary translation. As a logical format description, this is unlikely to be critical, since any software which accesses such a layout will necessarily need proprietary knowledge anyway.

Translation to API-specific data format representations

The data format container described here is too unwieldy to be expected to be used directly in most APIs. 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 returning 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.

Data format descriptor

The layout of the data structures described here are assumed to be little-endian for the purposes of data transfer, but may be implemented in the natural endianness of the platform for internal use.

The data format descriptor consists of a contiguous area of memory, 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.

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

Table 5: Data Format Descriptor layout

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.

Descriptor block

Each Descriptor Block has the same prefix:

`uint32_t`	***vendorId*** | (***descriptorType*** << 16)
`uint32_t`	***versionNumber*** | (***descriptorBlockSize*** << 16)
Format-specific data	

Table 6: Descriptor Block layout

The ***vendorId*** is a 16-bit value uniquely assigned to organisations, allocated by Khronos; ID 0 is used to identify Khronos itself. The ID 0xFFFF is reserved for internal use which is guaranteed not to clash with third-party implementations; this ID should not be shipped in libraries to avoid conflicts with development code.

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

The ***versionNumber*** is vendor-defined, and 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. 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 descriptor block. 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.

totalSize
vendorId | (***descriptorType*** << 16)
versionNumber | (***descriptorBlockSize*** << 16)
:
vendorId | (***descriptorType*** << 16)
versionNumber | (***descriptorBlockSize*** << 16)
:

Table 7: Data format descriptor header and descriptor block headers

Khronos Basic Data Format Descriptor Block

One *basic descriptor block* is intended to cover a large amount of metadata that is typically associated with common bulk data — most notably image or texture data. While this descriptor contains more information about the data interpretation than is needed by many applications, having a relatively comprehensive descriptor reduces the risk that metadata needed by different APIs will be 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, containing three samples, one of each color, in one plane. A $Y'C_BC_R$ 4:2:0 format may consist of a repeating 2×2 region consisting of four Y' samples and one sample each of C_B and C_R.

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		24 + 16 × *#samples* (***descriptorBlockSize***)	
colorModel	***colorPrimaries***	***transferFunction***	***flags***
texelBlockDimension0	***texelBlockDimension1***	***texelBlockDimension2***	***texelBlockDimension3***
bytesPlane0	***bytesPlane1***	***bytesPlane2***	***bytesPlane3***
bytesPlane4	***bytesPlane5***	***bytesPlane6***	***bytesPlane7***
Sample information for the first sample			
Sample information for the second sample (optional), etc.			

Table 8: Basic Data Format Descriptor layout

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

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`**.

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`**.

versionNumber

The ***versionNumber*** relating to the Basic Data Format Descriptor Block as described in this specification is 0.

descriptorBlockSize

The 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.

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.

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

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.

`KHR_DF_MODEL_UNSPECIFIED`

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.

KHR_DF_MODEL_RGBSDA

This color model represents additive colors of three channels, nominally red, green and blue, supplemented by channels for alpha, depth and stencil. 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.

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)

Table 9: Basic Data Format *RGBSDA* channels

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.

KHR_DF_MODEL_YUVSDA

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 10. 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.

Note
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.

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)

Table 10: Basic Data Format *YUVSDA* channels

KHR_DF_MODEL_YIQSDA

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 11. 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.

KHR_DF_MODEL_LABSDA

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 12.

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)

Table 11: Basic Data Format *YIQSDA* 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)

Table 12: Basic Data Format *LABSDA* channels

KHR_DF_MODEL_CMYKA

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

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)

Table 13: Basic Data Format *CMYKA* channels

KHR_DF_MODEL_XYZW

This “color model” represents channel data used for coordinate values — 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.

KHR_DF_MODEL_HSVA_ANG

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. In this model, the hue relates to the angular offset on a color wheel.

KHR_DF_MODEL_HSLA_ANG

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. In this model, the hue relates to the angular offset on a color wheel.

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*

Table 14: Basic Data Format *XYZW* 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)

Table 15: Basic Data Format angular *HSVA* channels

KHR_DF_MODEL_HSVA_HEX

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. In this model, the hue is generated by interpolation between extremes on a color hexagon.

KHR_DF_MODEL_HSLA_HEX

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. In this model, the hue is generated by interpolation between extremes on a color hexagon.

KHR_DF_MODEL_YCGCOA

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 19.

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 accurately.

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.

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)

Table 16: Basic Data Format angular *HSLA* 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)

Table 17: Basic Data Format hexagonal *HSVA* 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)

Table 18: Basic Data Format hexagonal *HSLA* channels

Floating-point compressed formats have lower and upper limits specified in floating point format. 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.

KHR_DF_MODEL_DXT1A/KHR_DF_MODEL_BC1A

This model represents the DXT1 or BC1 format. Channel 0 indicates color. If a second sample is present it should use channel 1 to indicate that the "special value" of the format should represent transparency — otherwise the "special value" represents opaque black.

KHR_DF_MODEL_DXT2/3/KHR_DF_MODEL_BC2

This model represents the DXT2/3 format, also described as BC2. The alpha premultiplication state (the distinction between DXT2 and DXT3) is recorded separately in the descriptor. This model has two channels: ID 0 contains the color information and ID 15 contains the alpha information. 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.

KHR_DF_MODEL_DXT4/5/KHR_DF_MODEL_BC3

This model represents the DXT4/5 format, also described as BC3. The alpha premultiplication state (the distinction between DXT4 and DXT5) is recorded separately in the descriptor. This model has two channels: ID 0 contains the color information and ID 15 contains the alpha information. The alpha channel is 64 bits and at offset 0; the color channel is 64 bits and at offset 64.

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)

Table 19: Basic Data Format *YCoCgA* channels

`KHR_DF_MODEL_BC4`

This model represents the Direct3D BC4 format for single-channel interpolated 8-bit data. The model has a single channel of id 0 with offset 0 and length 64 bits.

`KHR_DF_MODEL_BC5`

This model represents the Direct3D BC5 format for dual-channel interpolated 8-bit data. 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.

`KHR_DF_MODEL_BC6H`

This model represents the Direct3D BC6H format for *RGB* floating-point data. The model has a single channel 0, representing all three channels, and occupying 128 bits.

`KHR_DF_MODEL_BC7`

This model represents the Direct3D BC7 format for *RGBA* data. This model has a single channel 0 of 128 bits.

`KHR_DF_MODEL_ETC1`

This model represents the original Ericsson Texture Compression format, with a guarantee that the format does not rely on ETC2 extensions. It contains a single channel of *RGB* data.

`KHR_DF_MODEL_ETC2`

This model represents the updated Ericsson Texture Compression format, ETC2. Channel 0 represents red, and is used for the R11 EAC format. Channel 1 represents green, and both red and green should be present for the RG11 EAC format. Channel 2 represents *RGB* combined content. Channel 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. 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.

`KHR_DF_MODEL_ASTC`

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. The single sample has a size of 128 bits.

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 value.

KHR_DF_PRIMARIES_UNSPECIFIED

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.

KHR_DF_PRIMARIES_BT709

This value represents the Color Primaries defined by the ITU-R BT.709 specification, 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. 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**.

KHR_DF_PRIMARIES_BT601_EBU

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

KHR_DF_PRIMARIES_BT601_SMPTE

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

KHR_DF_PRIMARIES_BT2020

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

KHR_DF_PRIMARIES_CIEXYZ

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

KHR_DF_PRIMARIES_ACES

This value represents the Color Primaries defined for the Academy Color Encoding System.

transferFunction

Many color representations contain a nonlinear transfer function which maps between a linear (intensity-based) representation and a more perceptually-uniform encoding. Common Transfer Functions are encoded in the **khr_df_transfer_e** enumeration and represented as an unsigned 8-bit integer. 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.

KHR_DF_TRANSFER_UNSPECIFIED

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.

KHR_DF_TRANSFER_LINEAR

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.

KHR_DF_TRANSFER_SRGB

This value represents the nonlinear Transfer Function defined in the sRGB specification for mapping between numerical pixel values and intensity.

KHR_DF_TRANSFER_ITU

This value represents the nonlinear Transfer Function defined by the ITU and used in the BT.601, BT.709 and BT.2020 specifications.

KHR_DF_TRANSFER_NTSC

This value represents the nonlinear Transfer Function defined by the original NTSC television broadcast specification.

KHR_DF_TRANSFER_SLOG

This value represents a nonlinear Transfer Function used by some Sony video cameras to represent an increased dynamic range.

KHR_DF_TRANSFER_SLOG2

This value represents a nonlinear Transfer Function used by some Sony video cameras to represent a further increased dynamic range.

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.

In this version of the specification, the only flag defined is **`KHR_DF_FLAG_ALPHA_PREMULTIPIED`**. If this bit is set, any color information in the data should be interpreted as having been previously scaled by the alpha channel when performing blending operations. The value **`KHR_DF_FLAG_ALPHA_STRAIGHT`** is provided to represent this flag not being set, which indicates that the 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.

texelBlockDimension[0..3]

The ***texelBlockDimension*** fields define an integer bound on the range of coordinates covered by the repeating block described by the samples. 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 20. 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:

texelBlockDimension0	1
texelBlockDimension1	1
texelBlockDimension2	0
texelBlockDimension3	0

Table 20: Example Basic Data Format ***texelBlockDimension*** values for $Y'C_BC_R$ 4:2:0

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 texels (of up to 255 bytes). 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.

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 that plane contributes to the format. 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.

As an exception, if ***bytesPlane0*** has the value 0, the first plane is considered to hold indices into a color palette, which is described by one or more additional planes and samples in the normal way. The first sample in this case should describe a 1×1×1×1 texel holding an unsigned integer value. The number of bits used by the index should be encoded in this sample, with a maximum value of the largest palette entry held in ***sampleUpper***. Subsequent samples describe the entries in the palette, starting at an offset of bit 0. Note that the texel block in the index plane is not required to be byte-aligned in this case, and will not be for paletted formats which have small palettes. The channel type for the index is irrelevant.

For example, consider a 5-color paletted texture which describes each of these colors using 8 bits of red, green, blue and alpha. The color model would be *RGBSDA*, and the format would be described with two planes. ***bytesPlane0*** would be 0, indicating the special case of a palette, and ***bytesPlane1*** would be 4, representing the size of the palette entry. The first sample would then have a number of bits corresponding to the number of bits for the palette — in this case, three bits, corresponding to the requirements of a 5-color palette. The ***sampleUpper*** value for this sample is 4, indicating only 5 palette entries. Four subsequent samples represent the red, green, blue and alpha channels, starting from bit 0 as though the index value were not present, and describe the contents of the palette. The full data format descriptor for this example is provided as one of the example format descriptors.

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.

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 nonconsecutive 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. Note that 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 in the list of format descriptors provided. In this case, the ***sampleLower*** and ***sampleUpper*** fields for the combined sample are taken from the first sample to belong uniquely to this channel/position pair.

By convention, to avoid aliases for formats, samples should be listed in order starting with channels at the lowest bits of this bit stream.

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

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
bitOffset		***bitLength***	***channelType***
samplePosition0	***samplePosition1***	***samplePosition2***	***samplePosition3***
sampleLower			
sampleUpper			

Table 21: Basic Data Format Descriptor Sample Information

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.

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.

channelType

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.

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

If the **`KHR_DF_SAMPLE_DATATYPE_LINEAR`** bit is not set, the sample value is modified by the transfer function defined in the format's ***transferFunction*** field; if this bit is set, the sample is considered to contain a linearly-encoded value irrespective of the format's ***transferFunction***.

If the **`KHR_DF_SAMPLE_DATATYPE_EXPONENT`** bit is set, this sample holds an exponent (in integer form) for this channel. For example, this would be used to describe the shared exponent location in shared exponent formats (with the exponent bits listed separately under each channel). An exponent is applied to any integer sample of the same type. If this bit is not set, the sample is considered to contain mantissa information. If the **`KHR_DF_SAMPLE_DATATYPE_SIGNED`** bit 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 sample's lower and upper values. The presence or absence of an implicit leading digit in the mantissa of a format with an exponent can be determined by the upper value.

If the **`KHR_DF_SAMPLE_DATATYPE_SIGNED`** bit 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 is set, the sample holds floating point data in a conventional format of 10, 11, 16, 32, or 64-bits. **`KHR_DF_SAMPLE_DATATYPE_SIGNED`** should be set unless a genuine unsigned format is intended. Less common floating point representations can be generated with multiple samples and a combination of signed integer, unsigned integer and exponent fields.

samplePosition[0..3]

The sample has an associated location within the 4-dimensional space of the texel block. Each sample has an offset relative to the 0,0 position of the texel block, determined in units of half a coordinate. This allows the common situation of downsampled channels to have samples conceptually sited at the midpoint between full resolution samples. Support for offsets other than multiples of a half coordinates require an extension. 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.

Each ***samplePosition*** is an 8-bit unsigned integer quantity. ***samplePosition0*** is the X offset of the sample, ***samplePosition1*** is the Y offset of the sample, etc. Formats which use an offset larger than 127.5 in any dimension require an extension.

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

sampleLower

sampleLower, combined with ***sampleUpper***, is used to represent 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”.

If the channel encoding is an integer format, the ***sampleLower*** value is represented as a 32-bit integer — signed or unsigned according to whether the channel encoding is signed. If the channel encoding is a floating point value, the ***sampleLower*** value is also floating point. If the number of bits in the sample is greater than 32, the lowest representable value for ***sampleLower*** is interpreted as the smallest value representable in the channel format.

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. In this case, ***sampleLower*** should contain the value 16.

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 is equivalent to defining an “snorm” texture.

sampleUpper

sampleUpper, combined with ***sampleLower***, is used to represent the mapping between the numerical value stored in the format and the conceptual numerical interpretation. ***sampleUpper*** typically represents the value which should be interpreted as “1.0” (the “white point”).

If the channel encoding is an integer format, the ***sampleUpper*** value is represented as a 32-bit integer — signed or unsigned according to whether the channel encoding is signed. If the channel encoding is a floating point value, the ***sampleUpper*** value is also floating point. If the number of bits in the sample is greater than 32, the highest representable value for ***sampleUpper*** is interpreted as the largest value representable in the channel format.

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. In this case, ***sampleUpper*** should contain the value 235.

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 ***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.

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.

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.

As an example, consider a single-channel 32-bit depth buffer. 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 0xFFFF 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.

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.

56 (***totalSize***: total size of the two blocks plus one 32-bit value)			
Basic descriptor block			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		40 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`UNSPECIFIED` (*colorPrimaries*)	`UNSPECIFIED` (*transferFunction*)	0 (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
4 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the depth value			
0 (***bitOffset***)		31 (= “32”) (***bitLength***)	`SIGNED`|`FLOAT`|`DEPTH`
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0xbf800000 (***sampleLower***: -1.0f)			
0x3f800000U (***sampleUpper***: 1.0f)			
Extension descriptor block			
0xFFFF (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		12 (***descriptorBlockSize***)	
Data specific to the extension follows			
1 (buffer is “virtual”)			

Table 22: Example of a depth buffer with an extension to indicate a virtual allocation

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 nonlinear 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..

Frequently Asked Questions

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.

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.

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 behaviours. 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 behaviour based on format properties.

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.

Why should I use this format 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.

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.

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.

External references

Adaptive Scalable Texture Compression

https://www.khronos.org/registry/gles/extensions/KHR/texture_compression_astc_hdr.txt

ITU-T BT.601 specification for digital television

http://www.itu.int/rec/R-REC-BT.601/en

ITU-T BT.709 specification for HDTV

http://www.itu.int/rec/R-REC-BT.709-5-200204-I/en

ITU-T BT.2020 specification for UHDTV

http://www.itu.int/rec/R-REC-BT.2020/en

CIE 1931 *XYZ* tristimulus values

http://cie.co.at/index.php?i_ca_id=823

Academy Color Encoding System

http://www.oscars.org/science-technology/sci-tech-projects/aces

sRGB specification

http://www.w3.org/Graphics/Color/srgb

Example format descriptors

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		88 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	**`BT709`** (*colorPrimaries*)	**`SRGB`** (*transferFunction*)	**`PREMULTIPLIED`** (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
4 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample			
0 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (**`RED`**)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the second sample			
8 (***bitOffset***)		7 (= "8") (***bitLength***)	1 (***channelType***) (**`GREEN`**)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the third sample			
16 (***bitOffset***)		7 (= "8") (***bitLength***)	2 (***channelType***) (**`BLUE`**)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the fourth sample			
24 (***bitOffset***)		7 (= "8") (***bitLength***)	31 (***channelType***) (**`ALPHA`**|**`LINEAR`**)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			

Table 23: Four co-sited 8-bit sRGB channels, assuming premultiplied alpha

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
76 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		72 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`LINEAR` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
2 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample: 5 bits of blue			
0 (***bitOffset***)		4 (= "5") (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
31 (***sampleUpper***)			
Sample information for the second sample: 6 bits of green			
5 (***bitOffset***)		5 (= "6") (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
63 (***sampleUpper***)			
Sample information for the third sample: 5 bits of red			
11 (***bitOffset***)		4 (= "5") (***bitLength***)	0 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
31 (***sampleUpper***)			

Table 24: 565 *RGB* packed 16-bit format as written to memory by a little-endian architecture

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
44 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		40 (***descriptorBlockSize***)	
`YUVSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`ITU` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
4 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample			
0 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`Y`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			

Table 25: A single 8-bit monochrome channel

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
156 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		152 (***descriptorBlockSize***)	
`YUVSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`LINEAR` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
7 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
1 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample			
0 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the second sample			
1 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
2 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the third sample			
2 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
4 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the fourth sample			
3 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
6 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the fifth sample			
4 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
8 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the sixth sample			
5 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
10 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the seventh sample			
6 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
12 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			
Sample information for the eighth sample			
7 (***bitOffset***)		0 (= "1") (***bitLength***)	0 (***channelType***) (`Y`)
14 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
1 (***sampleUpper***)			

Table 26: A single 1-bit monochrome channel, as an 8×1 texel block to allow byte-alignment

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		88 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`SRGB` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
1 (***texelBlockDimension0***)	1 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
2 (***bytesPlane0***)	2 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample			
0 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the second sample			
8 (***bitOffset***)		7 (= "8") (***bitLength***)	1 (***channelType***) (`GREEN`)
2 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the third sample			
16 (***bitOffset***)		7 (= "8") (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	2 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the fourth sample			
24 (***bitOffset***)		7 (= "8") (***bitLength***)	2 (***channelType***) (`BLUE`)
2 (***samplePosition0***)	2 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			

Table 27: 2×2 Bayer pattern: four 8-bit distributed sRGB channels, spread across two lines (so two planes)

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
108 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		104 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`SRGB` (*transferFunction*)	`PREMULTIPLIED` (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
0 (***bytesPlane0***)	4 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the palette index			
0 (***bitOffset***)		2 (= “3”) (***bitLength***)	0 (***channelType***) (irrelevant)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
4 (***sampleUpper***) — this specifies that there are 5 palette entries			
Sample information for the first sample			
0 (***bitOffset***)		7 (= “8”) (***bitLength***)	0 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the second sample			
8 (***bitOffset***)		7 (= “8”) (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the third sample			
16 (***bitOffset***)		7 (= “8”) (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			
Sample information for the fourth sample			
24 (***bitOffset***)		7 (= “8”) (***bitLength***)	31 (***channelType***) (`ALPHA`|`LINEAR`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
255 (***sampleUpper***)			

Table 28: Four co-sited 8-bit channels in the sRGB color space described by an 5-entry, 3-bit palette

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
124 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		120 (***descriptorBlockSize***)	
`YUVSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`ITU` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
1 (***texelBlockDimension0***)	1 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
2 (***bytesPlane0***)	2 (***bytesPlane1***)	1 (***bytesPlane2***)	1 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first *Y* sample			
0 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`Y`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
235 (***sampleUpper***)			
Sample information for the second *Y* sample			
8 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`Y`)
2 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
235 (***sampleUpper***)			
Sample information for the third *Y* sample			
16 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`Y`)
0 (***samplePosition0***)	2 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
235 (***sampleUpper***)			
Sample information for the fourth *Y* sample			
24 (***bitOffset***)		7 (= "8") (***bitLength***)	0 (***channelType***) (`Y`)
2 (***samplePosition0***)	2 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
235 (***sampleUpper***)			
Sample information for the *U* sample			
32 (***bitOffset***)		7 (= "8") (***bitLength***)	1 (***channelType***) (`U`)
1 (***samplePosition0***)	1 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
240 (***sampleUpper***)			
Sample information for the *V* sample			
36 (***bitOffset***)		7 (= "8") (***bitLength***)	2 (***channelType***) (`V`)
1 (***samplePosition0***)	1 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
16 (***sampleLower***)			
240 (***sampleUpper***)			

Table 29: $Y'C_BC_R$ 4:2:0: BT.709 reduced-range data, with C_B and C_R aligned to the midpoint of the Y samples

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		88 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`SRGB` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
2 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the first sample: bit 0 belongs to green, bits 0..2 of channel in 13..15			
13 (***bitOffset***)		2 (= "3") (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
63 (***sampleUpper***)			
Sample information for the second sample: bits 3..5 of green in 0..2			
0 (***bitOffset***)		2 (= "3") (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***) — ignored, taken from first sample			
0 (***sampleUpper***) — ignored, taken from first sample			
Sample information for the third sample			
3 (***bitOffset***)		4 (= "5") (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
31 (***sampleUpper***)			
Sample information for the fourth sample			
8 (***bitOffset***)		4 (= "5") (***bitLength***)	1 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
31 (***sampleUpper***)			

Table 30: 565 *RGB* packed 16-bit format as written to memory by a big-endian architecture

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
124 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		120 (***descriptorBlockSize***)	
`RGBSDA` (***colorModel***)	`BT709` (***colorPrimaries***)	`LINEAR` (***transferFunction***)	`ALPHA_STRAIGHT` (***flags***)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
4 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the *R* mantissa			
0 (***bitOffset***)		8 (= "9") (***bitLength***)	0 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
256 (***sampleUpper***) — mantissa at 1.0			
Sample information for the *R* exponent			
27 (***bitOffset***)		4 (= "5") (***bitLength***)	32 (***channelType***) (`RED` | `EXPONENT`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — exponent bias			
Sample information for the *G* mantissa			
9 (***bitOffset***)		8 (= "9") (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
256 (***sampleUpper***) — mantissa at 1.0			
Sample information for the *G* exponent			
27 (***bitOffset***)		4 (= "5") (***bitLength***)	33 (***channelType***) (`GREEN` | `EXPONENT`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — exponent bias			
Sample information for the *B* mantissa			
18 (***bitOffset***)		8 (= "9") (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
256 (***sampleUpper***) — mantissa at 1.0			
Sample information for the *B* exponent			
27 (***bitOffset***)		4 (= "5") (***bitLength***)	34 (***channelType***) (`BLUE` | `EXPONENT`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — exponent bias			

Table 31: R9G9B9E5 shared-exponent format

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
108 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		104 (***descriptorBlockSize***)	
`RGBSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`LINEAR` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
0 (***texelBlockDimension0***)	0 (***texelBlockDimension1***)	0 (***texelBlockDimension2***)	0 (***texelBlockDimension3***)
1 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the *R* value and tint (shared low bits)			
0 (***bitOffset***)		3 (= “4”) (***bitLength***)	0 (***channelType***) (`RED`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — unique *R* upper value			
Sample information for the *G* tint (shared low bits)			
0 (***bitOffset***)		1 (= “2”) (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
0 (***sampleUpper***) — ignored, not unique			
Sample information for the *G* unique (high) bits			
4 (***bitOffset***)		1 (= “2”) (***bitLength***)	1 (***channelType***) (`GREEN`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — unique *G* upper value			
Sample information for the *B* tint (shared low bits)			
0 (***bitOffset***)		1 (= “2”) (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
0 (***sampleUpper***) — ignored, not unique			
Sample information for the *B* unique (high) bits			
6 (***bitOffset***)		1 (= “2”) (***bitLength***)	2 (***channelType***) (`BLUE`)
0 (***samplePosition0***)	0 (***samplePosition1***)	0 (***samplePosition2***)	0 (***samplePosition3***)
0 (***sampleLower***)			
15 (***sampleUpper***) — unique *B* upper value			

Table 32: Acorn 256-color format (2 bits each independent *RGB*, 2 bits shared “tint”)

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
220 (***totalSize***)			
0 (***vendorId***)		0 (***descriptorType***)	
0 (***versionNumber***)		216 (***descriptorBlockSize***) — 12 samples	
`YUVSDA` (*colorModel*)	`BT709` (*colorPrimaries*)	`ITU` (*transferFunction*)	`ALPHA_STRAIGHT` (*flags*)
5 (***dimension0***)	0 (***dimension1***)	0 (***dimension2***)	0 (***dimension3***)
16 (***bytesPlane0***)	0 (***bytesPlane1***)	0 (***bytesPlane2***)	0 (***bytesPlane3***)
0 (***bytesPlane4***)	0 (***bytesPlane5***)	0 (***bytesPlane6***)	0 (***bytesPlane7***)
Sample information for the shared *U0*/*U1* value			
0 (***bitOffset***)		9 (= “10”) (***bitLength***)	1 (***channelType***) (`U`)
1 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the *Y′0* value			
10 (***bitOffset***)		9 (= “10”) (***bitLength***)	0 (***channelType***) (`Y`)
0	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the shared *V0*/*V1* value			
20 (***bitOffset***)		9 (= “10”) (***bitLength***)	2 (***channelType***) (`V`)
1 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the *Y′1* value			
32 (***bitOffset***)		9 (= “10”) (***bitLength***)	0 (***channelType***) (`Y`)
2	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the shared *U2*/*U3* value			
42 (***bitOffset***)		9 (= “10”) (***bitLength***)	1 (***channelType***) (`U`)
5 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the *Y′2* value			
52 (***bitOffset***)		9 (= “10”) (***bitLength***)	0 (***channelType***) (`Y`)
4	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			

Table 33: V210 format (full-range $Y'C_BC_R$) part 1 of 2

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
Sample information for the shared *V2*/*V3* value			
64 (***bitOffset***)		9 (= "10") (***bitLength***)	2 (***channelType***) (`V`)
5 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the $Y'3$ value			
74 (***bitOffset***)		9 (= "10") (***bitLength***)	0 (***channelType***) (`Y`)
6	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the shared *U4*/*U5* value			
84 (***bitOffset***)		9 (= "10") (***bitLength***)	1 (***channelType***) (`U`)
9 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the $Y'4$ value			
96 (***bitOffset***)		9 (= "10") (***bitLength***)	0 (***channelType***) (`Y`)
8	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the shared *V4*/*V5* value			
106 (***bitOffset***)		9 (= "10") (***bitLength***)	2 (***channelType***) (`V`)
9 (assume mid-sited)	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			
Sample information for the $Y'4$ value			
116 (***bitOffset***)		9 (= "10") (***bitLength***)	0 (***channelType***) (`Y`)
10	0	0	0
0 (***sampleLower***)			
1023 (***sampleUpper***)			

Table 34: V210 format (full-range $Y'C_BC_R$) part 2 of 2