Khronos Data Format Specification

Andrew Garrard

Version 1.1, Revision 8

June 14, 2017

Collaborator	*Contribution*
Andrew Garrard	WRITTEN BY
Frank Brill	
Sean Ellis	
Jonas Gustavsson	
Chris Hebert	
Daniel Koch	
Thierry Lepley	
Tommaso Maestri	
Kathleen Mattson	
Hans-Peter Nilsson	
Alon Or-bach	
Erik Rainey	
Daniel Rakos	
Graham Sellers	
David Sena	
Mark Callow	
Jon Leech	

Revision number	*Date*	*Release Info*	*Author*
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 standardised	AG
0.5	Apr 2015	Typos and clarification	AG
1.0	May 2015	Submission for 1.0 release	AG
1.0 rev 2	June 2015	Clarifications for 1.0 release	AG
1.0 rev 3	July 2015	Added KHR_DF_SAMPLE_DATATYPE_LINEAR	AG
1.0 rev 4	July 2015	Clarified KHR_DF_SAMPLE_DATATYPE_LINEAR	AG
1.1	November 2015	Added definitions of compressed texture formats	AG
1.1 rev 2	January 2016	Added definitions of floating point formats	AG
1.1 rev 3	February 2016	Fixed typo in sRGB conversion (thank you, Tom Grim!)	AG
1.1 rev 4	March 2016	Fixed typo/clarified sRGB in ASTC, typographical improvements	AG
1.1 rev 5	March 2016	Switch to official Khronos logo, removed scripts, restored title	AG
1.1 rev 6	June 2016	ASTC "block footprint" note, fixed credits/changelog/contents	AG
1.1 rev 7	September 2016	ASTC multi-point "part" and quint decode typo fixes	AG
1.1 rev 8	June 2017	ETC2 legibility and table typo fix	AG

Contents

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. This document also acts as a reference for the memory layout of a number of common compressed texture formats.

Chapter 1

Introduction

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

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

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

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

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

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

Later sections provide a description of the memory layout of a number of common texture compression formats.

Chapter 2

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, 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 Table 2.1.

Red	Green	Blue

Table 2.1: A simple texel block — three channels representing one pixel

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, as shown in Table 2.2. It is this repeating and self-contained atomic unit, termed a *texel block*, that is described by this standard.

Red	Green
Green	Blue

Table 2.2: A simple Bayer-pattern texel block — three channels spread across 2×2 pixels

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 YUV format are considered to be centred 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, as shown in Table 2.3. 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 2.3: 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.

Later sections of this specification provide a description of the in-memory representation of a number of common compressed texture formats.

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 YUV buffer with downsampled chroma may have more luma sample 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 YUV4:2:0 buffer as in Table 2.4 would typically be represented with three planes (Table 2.5), one for each channel, with the luma plane four times larger than the chroma planes, and with two lines of the luma held within the same plane. 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 2.6, using two planes, vertically interleaved, with each line contiguous (the chroma channels remain one each). 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.

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

Table 2.4: Possible memory representation of a 4×4 YUV4:2:0 buffer (numbers are byte offsets)

Y plane	offset 0	byte stride 4	downsample 1×1
U plane	offset 16	byte stride 2	downsample 2×2
V plane	offset 20	byte stride 2	downsample 2×2

Table 2.5: Plane descriptors for the above YUV format buffer in a conventional API

Y plane 1	offset 0	byte stride 8	plane bytes 2
Y plane 2	offset 4	byte stride 8	plane bytes 2
U plane	offset 16	byte stride 2	plane bytes 1
V plane	offset 20	byte stride 2	plane bytes 1

Table 2.6: Plane descriptors for the above YUV format buffer using this standard

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.

Chapter 3

Required concepts not in the “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 YUV 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.

Chapter 4

Translation to API-specific 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.

Chapter 5

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, as shown in Table 5.1, divided into one or more *descriptor blocks*, as in Table 5.2 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	total_size
Descriptor block	first descriptor
Descriptor block	second descriptor (optional) etc.

Table 5.1: Data Format Descriptor layout

The **total_size** field, measured in bytes, allows the full format descriptor to be copied without need for details of the descriptor to be interpreted.

total_size
Descriptor block 1
Descriptor block 2
:

Table 5.2: Data format descriptor header and descriptor blocks

Chapter 6

Descriptor block

Each Descriptor Block has the same prefix, shown in Table 6.1.

uint32_t	vendor_id | (descriptor_type << 16)
uint32_t	version_number | (descriptor_block_size << 16)
Format-specific data	

Table 6.1: Descriptor Block layout

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

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

The **descriptor_block_size** indicates the size in bytes of this Descriptor Block, remembering that there may be multiple Descriptor Blocks within one container, as shown in Table 6.2. The **descriptor_block_size** therefore gives the offset between the start of the current Descriptor Block and the start of the next — so the size includes the **vendor_id**, **descriptor_type**, **version_number** and **descriptor_block_size** fields, which collectively contribute 8 bytes.

Having an explicit **descriptor_block_size** 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.

total_size	
	vendor_id | (descriptor_type << 16)
	version_number | (descriptor_block_size << 16)
	:
	vendor_id | (descriptor_type << 16)
	version_number | (descriptor_block_size << 16)
	:

Table 6.2: Data format descriptor header and descriptor block headers

Chapter 7

Khronos Basic Data Format Descriptor Block

One *basic descriptor block*, shown in Table 7.1 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 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 YUV 4:2:0 format may consist of a repeating 2×2 region consisting of four Y samples and one sample each of U and V.

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		24 + 16 times number of samples (descriptor_block_size)	
color_model	color_primaries	transfer_function	flags
texel_block_dimension_0	texel_block_dimension_1	texel_block_dimension_2	texel_block_dimension_3
bytes_plane_0	bytes_plane_1	bytes_plane_2	bytes_plane_3
bytes_plane_4	bytes_plane_5	bytes_plane_6	bytes_plane_7
Sample information for first sample			
Sample information for second sample (optional), etc.			

Table 7.1: Basic Data Format Descriptor layout

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

vendor_id

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

descriptor_type

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

version_number

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

descriptor_block_size

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 **descriptor_block_size** is measured in bytes.

color_model

The **color_model** 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 **color_model** 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, as shown in Table 7.2. 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.

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.

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 (transparency)

Table 7.2: Basic Data Format RGBSDA channels

KHR_DF_MODEL_YUVSDA

This color model represents color differences with three channels, nominally luma (Y) and two color-difference chroma channels, U (Cb) and V (Cr), supplemented by channels for alpha, depth and stencil, as shown in Table 7.3. These formats are distinguished by U and V being a delta between the Y channel and the blue and red channels respectively, rather than requiring a full color matrix. The conversion between YUV and RGB color spaces is defined in this case by the choice of value in the **color_primaries** field.

Note: Most single-channel luma 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 (luma/luminance)
1	KHR_DF_CHANNEL_YUVSDA_CB	U (Cb)
2	KHR_DF_CHANNEL_YUVSDA_CR	V (Cr)
13	KHR_DF_CHANNEL_YUVSDA_STENCIL	Stencil
14	KHR_DF_CHANNEL_YUVSDA_DEPTH	Depth
15	KHR_DF_CHANNEL_YUVSDA_ALPHA	Alpha (transparency)

Table 7.3: 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 7.4. This format is distinguished by I and Q each requiring all three additive channels to evaluate.

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 (transparency)

Table 7.4: Basic Data Format YIQSDA channels

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 7.5.

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 (transparency)

Table 7.5: 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, as shown in Table 7.6.

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 (transparency)

Table 7.6: Basic Data Format CMYKA channels

KHR_DF_MODEL_XYZW

This "color model" represents channel data used for coordinate values, as shown in Table 7.7 — 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 **channel_type** field.

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 7.7: Basic Data Format XYZW channels

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, as shown in Table 7.8. 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, as shown in Table 7.9. In this model, the hue relates to the angular offset on a color wheel.

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 (transparency)

Table 7.8: Basic Data Format HSVA_ANG channels

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

Table 7.9: Basic Data Format HSLA_ANG 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, as shown in Table 7.10. In this model, the hue is generated by interpolation between extremes on a color hexagon.

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 (transparency)

Table 7.10: Basic Data Format HSVA_HEX channels

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, as shown in Table 7.11. 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 7.12.

Compressed formats

A number of compressed formats are supported as part of **khr_df_model_e**. In general, these formats will have a texel block dimension 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 **texel_block_dimensions** field determines the compression ratio for ASTC.

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 (transparency)

Table 7.11: Basic Data Format HSLA_HEX channels

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

Table 7.12: Basic Data Format YCoCgA 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 the 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.

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 **channel_id** 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 **channel_id KHR_DF_SAMPLE_DATATYPE_FLOAT** bit clear, and an ASTC texture that may include HDR-encoded blocks should have the **channel_id 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.

color_primaries

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 is also shared by sRGB. RGB data is distinguished between BT.709 and sRGB by the Transfer Function. Conversion to and from BT.709 Y'CbCr (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 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'CbCr (YUV) 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'CbCr (YUV) 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'CbCr (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.

transfer_function

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 **transfer_function**.

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.

That is, the conversion from linear (R,G,B) encoding to nonlinear (R',G',B') encoding is:

$$R' = \begin{cases} R \times 12.92, & R \leq 0.0031308 \\ 1.055 \times R^{\frac{1}{2.4}} - 0.055, & R > 0.0031308 \end{cases}$$

$$G' = \begin{cases} G \times 12.92, & G \leq 0.0031308 \\ 1.055 \times G^{\frac{1}{2.4}} - 0.055, & G > 0.0031308 \end{cases}$$

$$B' = \begin{cases} B \times 12.92, & B \leq 0.0031308 \\ 1.055 \times B^{\frac{1}{2.4}} - 0.055, & B > 0.0031308 \end{cases}$$

The corresponding conversion from nonlinear (R',G',B') encoding to linear (R,G,B) encoding is:

$$R = \begin{cases} \frac{R'}{12.92}, & R' \leq 0.04045 \\ \left(\frac{R'+0.055}{1.055}\right)^{2.4}, & R' > 0.04045 \end{cases}$$

$$G = \begin{cases} \frac{G'}{12.92}, & G' \leq 0.04045 \\ \left(\frac{G'+0.055}{1.055}\right)^{2.4}, & G' > 0.04045 \end{cases}$$

$$B = \begin{cases} \frac{B'}{12.92}, & B' \leq 0.04045 \\ \left(\frac{B'+0.055}{1.055}\right)^{2.4}, & B' > 0.04045 \end{cases}$$

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 **khr_df_flags_e** enumeration 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.

texel_block_dimensions_[0..3]

The **texel_block_dimensions** define the number 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 YUV 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 7.13. 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 **texel_block_dimensions_[0..3]** values would therefore be:

texel_block_dimensions_0	1
texel_block_dimensions_1	1
texel_block_dimensions_2	0
texel_block_dimensions_3	0

Table 7.13: Example Basic Data Format **texel_block_dimensions** for YUV 4:2:0

bytes_plane_[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 YUV 4:2:0 — may require multiple planes to describe a channel. A typical YUV 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, contained within a.

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 **bytes_plane_[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 **bytes_plane** value of 0, and any **bytes_plane** values after the first 0 are ignored. If no **bytes_plane** value is zero, 8 planes are considered to exist.

As an exception, if **bytes_plane_0** 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 $1\times1\times1\times1$ 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 **sample_upper**. 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. bytes_plane_0 would be 0, indicating the special case of a palette, and bytes_plane_1 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 the requirements of a 5-color palette. The **sample_upper** 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 in Table 18.5 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, as shown in Table 7.14.

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

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

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, Table 18.7, in the list of format descriptors provided. In this case, the **sample_lower** and **sample_upper** 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. Ties should be broken by increasing channel type id, as shown in Table 18.11.

The number of samples present in the format is determined by the **descriptor_block_size** 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)**
bit_offset		bit_length	channel_type
sample_position_0	sample_position_1	sample_position_2	sample_position_3
sample_lower			
sample_upper			

Table 7.14: Basic Data Format Descriptor Sample Information

bit_offset

The **bit_offset** 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 **bit_offset** 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 Bit Offset of 8. The Bit Offset is an unsigned 16-bit integer quantity.

bit_length

The **bit_length** 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 **bit_length** field value of 7. If a **bit_length** 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.

channel_type

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

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

The top four bits of the **channel_type** 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 **transfer_function** field; if this bit is set, the sample is considered to contain a linearly-encoded value irrespective of the format's **transfer_function**.

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 exponent's **sample_lower** value. The presence or absence of an implicit leading digit in the mantissa of a format with an exponent can be determined by the **sample_upper** value of the mantissa.

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 or 16 bits, as described in Chapter 17, or of 32, or 64 bits as described in [IEEE 754]. Unless a genuine unsigned format is intended, **KHR_DF_SAMPLE_DATATYPE_SIGNED** should be set. Less common floating point representations can be generated with multiple samples and a combination of signed integer, unsigned integer and exponent fields, as described above and in Section 17.4.

sample_position_[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 **sample_position** is an 8-bit unsigned integer quantity. **sample_position_0** is the X offset of the sample, **sample_position_1** 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.

sample_lower

Sample_lower, combined with **sample_upper**, is used to represent the mapping between the numerical value stored in the format and the conceptual numerical interpretation. For unsigned formats, **sample_lower** typically represents the value which should be interpreted as zero (the black point). For signed formats, **sample_lower** typically represents "-1".

If the channel encoding is an integer format, the **sample_lower** value is represented as a 32-bit integer — signed or unsigned according to whether the channel encoding is signed. Signed negative values should be sign-extended if the channel has fewer than 32-bit, such that the value encoded in **sample_lower** is itself negative, If the channel encoding is a floating point value, the **sample_lower** value is also floating point. If the number of bits in the sample is greater than 32, the lowest representable value for **sample_lower** is interpreted as the smallest value representable in the channel format.

If the channel consists of multiple co-sited integer samples, for example because the channel bits are non-contiguous, there are two possible behaviors. If the total number of bits in the channel is less than or equal to 32, the **sample_lower** values in the samples corresponding to the least-significant bits of the sample are ignored, and only the **sample_lower** from the most-significant sample is considered. If the number of bits in the channel exceeds 32, the **sample_lower** values from the sample corresponding to the most-significant bits within any 32-bit subset of the total number are concatenated to generate the final **sample_lower** value. For example, a 48-bit signed integer may be encoded in three 16-bit samples. The first sample, corresponding to the least-significant 16 bits, will have its **sample_lower** value ignored. The next sample of 16 bits takes the total to 32, and so the **sample_lower** value of this sample should represent the lowest 32 bits of the desired 48-bit virtual **sample_lower** value. Finally, the third sample indicates the top 16 bits of the 48-bit channel, and its **sample_lower** contains the top 16 bits of the 48-bit virtual **sample_lower** value.

The **sample_lower** value for an exponent should represent the exponent bias — the value that should be subtracted from the encoded exponent to indicate that the mantissa's **sample_upper** value will represent 1.0. See Section 17.4 for more detail on this.

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, **sample_lower** 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 **sample_lower** to the minimum signed integer value representable in the channel is equivalent to defining an "snorm" texture.

sample_upper

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

If the channel encoding is an integer format, the **sample_upper** 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 **sample_upper** value is also floating point. If the number of bits in the sample is greater than 32, the highest representable value for **sample_upper** is interpreted as the largest value representable in the channel format. If the channel encoding is the mantissa of a custom floating point format (that is, the encoding is integer but the same sample location and channel is shared by a sample that encodes an exponent), the presence of an implicit "1" digit can be represented by setting the **sample_upper** value to a value one larger than can be encoded in the available bits for the mantissa, as described in Section 17.4.

The **sample_upper** value for an exponent should represent the largest conventional legal exponent value. If the encoded exponent exceeds this value, the encoded floating point value encodes either an infinity or a *NaN* value, depending on the mantissa. See Section 17.4 for more detail on this.

If the channel consists of multiple co-sited integer samples, for example because the channel bits are non-contiguous, there are two possible behaviors. If the total number of bits in the channel is less than or equal to 32, the **sample_upper** values in the samples corresponding to the least-significant bits of the sample are ignored, and only the **sample_upper** from the most-significant sample is considered. If the number of bits in the channel exceeds 32, the **sample_upper** values from the sample corresponding to the most-significant bits within any 32-bit subset of the total number are concatenated to generate the final **sample_upper** value. For example, a 48-bit signed integer may be encoded in three 16-bit samples. The first sample, corresponding to the least-significant 16 bits, will have its **sample_upper** value ignored. The next sample of 16 bits takes the total to 32, and so the **sample_upper** value of this sample should represent the lowest 32 bits of the desired 48-bit virtual **sample_upper** value. Finally, the third sample indicates the top 16 bits of the 48-bit channel, and its **sample_upper** contains the top 16 bits of the 48-bit virtual **sample_upper** value.

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, **sample_upper** 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 **sample_upper** to the maximum signed integer value representable in the channel for a signed channel type is equivalent to defining an "snorm" texture. Setting **sample_upper** to the maximum unsigned value representable in the channel for an unsigned channel type is equivalent to defining a "normalized" texture. Setting **sample_upper** 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. **Sample_upper** 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 **sample_lower** and **sample_upper** unless the users of a format agree that convention.

Chapter 8

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 **total_size** 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 **version_id** field.

As an example, consider a single-channel 32-bit depth buffer, as shown in Table 8.1. 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 **total_size** that includes both of these descriptor blocks.

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..

56 (total_size: total size of the two blocks plus one 32-bit value)			
Basic descriptor block			
0 (vendor_id)		0 (descriptor_type)	
0 (version_number)		40 (descriptor_block_size)	
RGBSDA (color_model)	UNSPECIFIED (color_primaries)	UNSPECIFIED (transfer_function)	0 (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
4 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for the depth value			
0 (bit_offset)		32 (bit_length)	SIGNED | FLOAT | DEPTH
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0xbf800000 (sample_lower: -1.0f)			
0x3f800000U (sample_upper: 1.0f)			
Extension descriptor block			
0xFFFF (vendor_id)		0 (descriptor_type)	
0 (version_number)		12 (descriptor_block_size)	
Data specific to the extension follows			
1 (buffer is “virtual”)			

Table 8.1: Example of a depth buffer with an extension to indicate a virtual allocation

Chapter 9

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, YUV 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.

Chapter 10

External references

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

IEEE Standard for Floating-Point Arithmetic

IEEE Std 754-2008 http://dx.doi.org/10.1109/IEEESTD.2008.4610935, August, 2008.

Chapter 11

S3TC Compressed Texture Image Formats

This description is derived from the EXT_texture_compression_s3tc extension.

Compressed texture images stored using the S3TC compressed image formats are represented as a collection of 4×4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If an S3TC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an S3TC image with a width of w, height of h, and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times blocksize$$

When decoding an S3TC image, the block containing the texel at offset (x,y) begins at an offset (in bytes) relative to the base of the image of:

$$blocksize \times \left(\left\lceil \frac{w}{4} \right\rceil \times \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{x}{4} \right\rfloor \right)$$

The data corresponding to a specific texel (x,y) are extracted from a 4×4 texel block using a relative (x,y) value of

$$(x \bmod 4, y \bmod 4)$$

There are four distinct S3TC image formats:

BC1 with no alpha

Each 4×4 block of texels consists of 64 bits of RGB image data.

Each RGB image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$c0_{lo}, c0_{hi}, c1_{lo}, c1_{hi}, bits_0, bits_1, bits_2, bits_3$$

The 8 bytes of the block are decoded into three quantities:

$$\begin{aligned} color_0 &= c0_{lo} + c0_{hi} \times 256 \\ color_1 &= c1_{lo} + c1_{hi} \times 256 \\ bits &= bits_0 + 256 \times (bits_1 + 256 \times (bits_2 + 256 \times bits_3)) \end{aligned}$$

color0 and *color1* are 16-bit unsigned integers that are unpacked to RGB colors RGB0 and RGB1 as though they were 16-bit packed pixels with the R channel in the high 5 bits, G in the next 6 bits and B in the low 5 bits.

Block value	Condition
RGB_0	$color_0 > color_1$ and $code(x,y) = 0$
RGB_1	$color_0 > color_1$ and $code(x,y) = 1$
$\frac{(2\times RGB_9+RGB_1)}{3}$	$color_0 > color_1$ and $code(x,y) = 2$
$\frac{(RGB_0+2\times RGB_1)}{3}$	$color_0 > color_1$ and $code(x,y) = 3$
RGB_0	$color_0 \leq color_1$ and $code(x,y) = 0$
RGB_1	$color_0 \leq color_1$ and $code(x,y) = 1$
$\frac{(RGB_0+RGB_1)}{2}$	$color_0 \leq color_1$ and $code(x,y) = 2$
BLACK	$color_0 \leq color_1$ and $code(x,y) = 3$

Table 11.1: Block decoding for BC1

bits is a 32-bit unsigned integer, from which a two-bit control code is extracted for a texel at location (x,y) in the block using:

$$code(x,y) = bits[2\times(4\times y+x)+1 \ \ldots \ 2\times(4\times y+x)+0]$$

where bit 31 is the most significant and bit 0 is the least significant bit.

The RGB color for a texel at location (x,y) in the block is given in Table 11.1.

Arithmetic operations are done per component, and BLACK refers to an RGB color where red, green, and blue are all zero.

Since this image has an RGB format, there is no alpha component and the image is considered fully opaque.

BC1 with alpha

Each 4×4 block of texels consists of 64 bits of RGB image data and minimal alpha information. The RGB components of a texel are extracted in the same way as BC1 with no alpha.

The alpha component for a texel at location (x,y) in the block is given by Table 11.2.

Alpha value	Condition
0.0	$color_0 \leq color1$ and $code(x,y) = 3$
1.0	otherwise

Table 11.2: BC1 with alpha

The red, green, and blue components of any texels with a final alpha of 0 should be encoded as zero (black).

BC2

Each 4×4 block of texels consists of 64 bits of uncompressed alpha image data followed by 64 bits of RGB image data.

Each RGB image data block is encoded according to the BC1 formats, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though color0 > color1, regardless of the actual values of color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$$

The 8 bytes of the block are decoded into one 64-bit integer:

$$alpha = a_0 + 256\times(a_1 + 256\times(a_2 + 256\times(a_3 + 256\times(a_4 + 256\times(a_5 + 256\times(a_6 + 256\times a_7))))))$$

alpha is a 64-bit unsigned integer, from which a four-bit alpha value is extracted for a texel at location (x,y) in the block using:

$$alpha(x,y) = bits[4\times(4\times y+x)+3\ldots4\times(4\times y+x)+0]$$

where bit 63 is the most significant and bit 0 is the least significant bit.

The alpha component for a texel at location (x,y) in the block is given by $\frac{alpha(x,y)}{15}$.

BC3

Each 4×4 block of texels consists of 64 bits of compressed alpha image data followed by 64 bits of RGB image data.

Each RGB image data block is encoded according to the BC1 formats, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though color0 > color1, regardless of the actual values of color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$alpha_0, alpha_1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5$$

The $alpha_0$ and $alpha_1$ are 8-bit unsigned bytes converted to alpha components by multiplying by $\frac{1}{255}$.

The 6 *bits* bytes of the block are decoded into one 48-bit integer:

$$bits = bits_0+256\times(bits_1+256\times(bits_2+256\times(bits_3+256\times(bits_4+256\times bits_5))))$$

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x,y) in the block using:

$$code(x,y) = bits[3\times(4\times y+x)+2\ldots3\times(4\times y+x)+0]$$

where bit 47 is the most-significant and bit 0 is the least-significant bit.

The alpha component for a texel at location (x,y) in the block is given by Table 11.3.

Alpha value	Condition
$alpha0$	$code(x,y)=0$
$alpha1$	$code(x,y)=1$
$\frac{(6*alpha0+1*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=2$
$\frac{(5*alpha0+2*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=3$
$\frac{(4*alpha0+3*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=4$
$\frac{(3*alpha0+4*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=5$
$\frac{(2*alpha0+5*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=6$
$\frac{(1*alpha0+6*alpha1)}{7}$	$alpha0 > alpha1$ and $code(x,y)=7$
$\frac{(4*alpha0+1*alpha1)}{5}$	$alpha0 \leq alpha1$ and $code(x,y)=2$
$\frac{(3*alpha0+2*alpha1)}{5}$	$alpha0 \leq alpha1$ and $code(x,y)=3$
$\frac{(2*alpha0+3*alpha1)}{5}$	$alpha0 \leq alpha1$ and $code(x,y)=4$
$\frac{(1*alpha0+4*alpha1)}{5}$	$alpha0 \leq alpha1$ and $code(x,y)=5$
0.0	$alpha0 \leq alpha1$ and $code(x,y)=6$
1.0	$alpha0 \leq alpha1$ and $code(x,y)=7$

Table 11.3: Alpha encoding for BC3 blocks

Chapter 12

RGTC Compressed Texture Image Formats

This description is derived from the "RGTC Compressed Texture Image Formats" section of the OpenGL 4.5 specification.

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of 4×4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If an RGTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of w, height of h, and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times blocksize$$

When decoding an RGTC image, the block containing the texel at offset (x,y) begins at an offset (in bytes) relative to the base of the image of:

$$blocksize \times \left(\left\lceil \frac{w}{4} \right\rceil \times \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{x}{4} \right\rfloor \right)$$

The data corresponding to a specific texel (x,y) are extracted from a 4×4 texel block using a relative (x,y) value of

$$(x \bmod 4, y \bmod 4).$$

There are four distinct RGTC image formats:

BC4 unsigned

Each 4×4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$red_0, red_1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5$$

The 6 $bits_*$ bytes of the block are decoded into a 48-bit bit vector:

$$bits = bits_0 + 256 \times (bits_1 + 256 \times (bits_2 + 256 \times (bits_3 + 256 \times (bits_4 + 256 \times bits_5))))$$

red_0 and red_1 are 8-bit unsigned integers that are unpacked to red values RED_0 and RED_1.

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x,y) in the block using:

$$code(x,y) = bits[3 \times (4 \times y + x) + 2 \ldots 3 \times (4 \times y + x) + 0]$$

where bit 47 is the most significant and bit 0 is the least significant bit.

The red value R for a texel at location (x,y) in the block is given by Table 12.1.

RED_{min} and RED_{max} are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting RGBA value for the texel is $(R,0,0,1)$.

R value	Condition
RED_0	$red_0 > red_1, code(x,y) = 0$
RED_1	$red_0 > red_1, code(x,y) = 1$
$\frac{6RED_0+RED_1}{7}$	$red_0 > red_1, code(x,y) = 2$
$\frac{5RED_0+2RED_1}{7}$	$red_0 > red_1, code(x,y) = 3$
$\frac{4RED_0+3RED_1}{7}$	$red_0 > red_1, code(x,y) = 4$
$\frac{3RED_0+4RED_1}{7}$	$red_0 > red_1, code(x,y) = 5$
$\frac{2RED_0+5RED_1}{7}$	$red_0 > red_1, code(x,y) = 6$
$\frac{RED_0+6RED_1}{7}$	$red_0 > red_1, code(x,y) = 7$
RED_0	$red_0 \leq red_1, code(x,y) = 0$
RED_1	$red_0 \leq red_1, code(x,y) = 1$
$\frac{4RED_0+RED_1}{5}$	$red_0 \leq red_1, code(x,y) = 2$
$\frac{3RED_0+2RED_1}{5}$	$red_0 \leq red_1, code(x,y) = 3$
$\frac{2RED_0+3RED_1}{5}$	$red_0 \leq red_1, code(x,y) = 4$
$\frac{RED_0+4RED_1}{5}$	$red_0 \leq red_1, code(x,y) = 5$
RED_{min}	$red_0 \leq red_1, code(x,y) = 6$
RED_{max}	$red_0 \leq red_1, code(x,y) = 7$

Table 12.1: Block decoding for BC4

BC4 signed

Each 4×4 block of texels consists of 64 bits of signed red image data. The red values of a texel are extracted in the same way as BC4 unsigned except red_0, red_1, RED_0, RED_1, RED_{min}, and RED_{max} are signed values defined as follows:

$$RED_0 = \begin{cases} \frac{red_0}{127.0,} & red_0 > -128 \\ -1.0, & red_0 = -128 \end{cases}$$

$$RED_1 = \begin{cases} \frac{red_1}{127.0,} & red_1 > -128 \\ -1.0, & red_0 = -128 \end{cases}$$

$$RED_{min} = -1.0$$

$$RED_{max} = 1.0$$

red_0 and red_1 are 8-bit signed (twos complement) integers.

CAVEAT for signed red_0 and red_1 values: the expressions $red_0 > red_1$ and $red_0 \leq red_1$ above are considered undefined (read: may vary by implementation) when $red_0 = -127$ and $red_1 = -128$. This is because if red_0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed red-green formats should avoid encoding blocks where $red_0 = -127$ and $red_1 = -128$.

BC5 unsigned

Each 4×4 block of texels consists of 64 bits of compressed unsigned red image data followed by 64 bits of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like BC4 unsigned above.

The second 64 bits of compressed green are decoded exactly like BC4 unsigned above except the decoded value R for this second block is considered the resulting green value G.

Since the decoded texel has a red-green format, the resulting RGBA value for the texel is $(R, G, 0, 1)$.

BC5 signed

Each 4×4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like BC4 signed above.

The second 64 bits of compressed green are decoded exactly like BC4 signed above except the decoded value R for this second block is considered the resulting green value G.

Since this image has a red-green format, the resulting RGBA value is $(R, G, 0, 1)$.

Chapter 13

BPTC Compressed Texture Image Formats

This description is derived from the "BPTC Compressed Texture Image Formats" section of the OpenGL 4.5 specification.

Compressed texture images stored using the BPTC compressed image formats are represented as a collection of 4×4 texel blocks, where each block contains 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4×4 block is treated as a single pixel. If a BPTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When a BPTC image with a width of w, height of h, and block size of *blocksize* (16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times blocksize$$

When decoding a BPTC image, the block containing the texel at offset (x, y) begins at an offset (in bytes) relative to the base of the image of:

$$blocksize \times \left(\left\lceil \frac{w}{4} \right\rceil \times \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{x}{4} \right\rfloor \right)$$

The data corresponding to a specific texel (x, y) are extracted from a 4×4 texel block using a relative (x, y) value of:

$$(x \bmod 4, y \bmod 4)$$

There are two distinct BPTC image formats each of which has two variants. BC7 with or without an sRGB transform function used in the encoding of the RGB channels compresses 8-bit unsigned, normalized fixed-point data. BC6H in signed or unsigned form compresses high dynamic range floating-point values. The formats are similar, so the description of the BC6H format will reference significant sections of the BC7 description.

BC7

Each 4×4 block of texels consists of 128 bits of RGBA image data, of which the RGB components may be encoded linearly or with the sRGB transfer function.

Each block contains enough information to select and decode a pair of colors called endpoints, interpolate between those endpoints in a variety of ways, then remap the result into the final output.

Each block can contain data in one of eight modes. The mode is identified by the lowest bits of the lowest byte. It is encoded as zero or more zeros followed by a one. For example, using x to indicate a bit not included in the mode number, mode 0 is encoded as xxxxxxx1 in the low byte in binary, mode 5 is xx100000, and mode 7 is 10000000. Encoding the low byte as zero is reserved and should not be used when encoding a BPTC texture.

All further decoding is driven by the values derived from the mode listed in Table 13.3 and Table 13.4. The fields in the block are always in the same order for all modes. Starting at the lowest bit after the mode and going up, these fields are:

partition number, rotation, index selection, color, alpha, per-endpoint P-bit, shared P-bit, primary indices, and secondary indices. The number of bits to be read in each field is determined directly from the table.

Each block can be divided into between 1 and 3 groups of pixels with independent compression parameters called subsets. A texel in a block with one subset is always considered to be in subset zero. Otherwise, a number determined by the number of partition bits is used to look up in Table 13.5 or Table 13.6 for 2 and 3 subsets respectively. This partitioning is indexed by the X and Y within the block to generate the subset index.

Each block has two colors for each subset, stored first by endpoint, then by subset, then by color. For example, a format with two subsets and five color bits would have five bits of red for endpoint 0 of the first subset, then five bits of red for endpoint 1, then the two ends of the second subset, then green and blue stored similarly. If a block has non-zero alpha bits, the alpha data follows the color data with the same organization. If not, alpha is overridden to 1.0. These bits are treated as the high bits of a fixed-point value in a byte. If the format has a shared P-bit, there are two bits for endpoints 0 and 1 from low to high. If the format has a per-endpoint P-bits, then there are $2 \times subsets$ P-bits stored in the same order as color and alpha. Both kinds of P-bits are added as a bit below the color data stored in the byte. So, for a format with 5 red bits, the P-bit ends up in bit 2. For final scaling, the top bits of the value are replicated into any remaining bits in the byte. For the preceding example, bits 6 and 7 would be written to bits 0 and 1.

The endpoint colors are interpolated using index values stored in the block. The index bits are stored in x-major order. Each index has the number of bits indicated by the mode except for one special index per subset called the anchor index. Since the ordering of the endpoints is unimportant, we can save one bit on one index per subset by ordering the endpoints such that the highest bit is guaranteed to be zero. In partition zero, the anchor index is always index zero. In other partitions, the anchor index is specified by Table 13.7, Table 13.8, and Table 13.9. If secondary index bits are present, they are read in the same manner. The anchor index information is only used to determine the number of bits each index has when it's read from the block data.

The endpoint color and alpha values used for final interpolation are the decoded values corresponding to the applicable subset as selected above. The index value for interpolating color comes from the secondary index for the texel if the format has an index selection bit and its value is one and from the primary index otherwise. The alpha index comes from the secondary index if the block has a secondary index and the block either doesn't have an index selection bit or that bit is zero and the primary index otherwise.

Interpolation is always performed using a 6-bit interpolation factor. The effective interpolation factors for 2-, 3-, and 4-bit indices are given in Table 13.1.

2	0				21				43				64			
3	0		9		18		27		37		46		55		64	
4	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

Table 13.1: BPTC interpolation factors

The interpolation results in an RGBA color. If rotation bits are present, this color is remapped according to Table 13.2.

0	no change
1	swap(a,r)
2	swap(a,g)
3	swap(a,b)

Table 13.2: BPTC Rotation bits

These 8-bit values should be interpreted as RGBA 8-bit normalized channels, either linearly encoded or with the sRGB transfer function.

BC6H

Each 4×4 block of texels consists of 128 bits of RGB data. These formats are very similar and will be described together. In the description and pseudocode below, *signed* will be used as a condition which is true for the signed version of the

Mode	NS	PB	RB	ISB	CB	AB	EPB	SPB	IB	IB2
0	3	4	0	0	4	0	1	0	3	0
1	2	6	0	0	6	0	0	1	3	0
2	3	6	0	0	5	0	0	0	2	0
3	2	6	0	0	7	0	1	0	2	0
4	1	0	2	1	5	6	0	0	2	3
5	1	0	2	0	7	8	0	0	2	2
6	1	0	0	0	7	7	1	0	4	0
7	2	6	0	0	5	5	1	0	2	0

Table 13.3: Mode-dependent BPTC parameters.

Mode	As described previously
NS	Number of subsets in each partition
PB	Partition bits
RB	Rotation bits
ISB	Index selection bits
CB	Color bits
AB	Alpha bits
EPB	Endpoint P-bits
SPB	Shared P-bits
IB	Index bits per element
IB2	Secondary index bits per element}

Table 13.4: The full descriptions of the BPTC mode columns are as follows

format and false for the unsigned version of the format. Both formats only contain RGB data, so the returned alpha value is 1.0. If a block uses a reserved or invalid encoding, the return value is $(0,0,0,1)$.

Each block can contain data in one of 14 modes. The mode number is encoded in either the low two bits or the low five bits. If the low two bits are less than two, that is the mode number, otherwise the low five bits the mode number. Mode numbers not listed in Table 13.10 are reserved (19, 23, 27, and 31).

The data for the compressed blocks is stored in a different format for each mode. The formats are specified in Table 13.11. The format strings are intended to be read from left to right with the LSB on the left. Each element is of the form $v[a\colon b]$. If $a \geq b$, this indicates extracting $b-a+1$ bits from the block at that location and put them in the corresponding bits of the variable v. If $a < b$, then the bits are reversed. $v[a]$ is used as a shorthand for the one bit $v[a\colon a]$. As an example, $m[1\colon 0], g2[4]$ would move the low two bits from the block into the low two bits of m then the next bit of the block into bit 4 of $g2$. The variable names given in the table will be referred to in the language below.

Subsets and indices work in much the same way as described for the fixed-point formats above. If a float block has no partition bits, then it is a single-subset block. If it has partition bits, then it is a 2 subset block. The partition index references the first half of Table 13.5. Indices are read in the same way as the fixed-point formats including obeying the anchor values for index 0 and as needed by Table 13.7.

In a single-subset blocks, the two endpoints are contained in r_0, g_0, b_0 (hence e_0) and r_1, g_1, b_1 (hence e_1). In a two-subset block, the endpoints for the second subset are in r_2, g_2, b_2 and r_3, g_3, b_3. The value in e_0 is sign-extended if the format of the texture is signed. The values in e_1 (and e_2 and e_3 if the block is two-subset) are sign-extended if the format of the texture is signed or if the block mode has transformed endpoints. If the mode has transformed endpoints, the values from e_0 are used as a base to offset all other endpoints, wrapped at the number of endpoint bits. For example, $r_1 = (r_0 + r_1)\ \&\ ((1 \ll EPB) - 1)$.

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

Table 13.5: Partition table for BPTC 2 subset, with the 4×4 block of values for each partition index value

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

Table 13.6: Partition table for BPTC 3 subset, with the 4×4 block of values for each partition index value

15	15	15	15	15	15	15	15
15	15	15	15	15	15	15	15
15	2	8	2	2	8	8	15
2	8	2	2	8	8	2	2
15	15	6	8	2	8	15	15
2	8	2	2	2	15	15	6
6	2	6	8	15	15	2	2
15	15	15	15	15	2	2	15

Table 13.7: BPTC anchor index values for the second subset of two-subset partitioning. Values run right, then down.

3	3	15	15	8	3	15	15
8	8	6	6	6	5	3	3
3	3	8	15	3	3	6	10
5	8	8	6	8	5	15	15
8	15	3	5	6	10	8	15
15	3	15	5	15	15	15	15
3	15	5	5	5	8	5	10
5	10	8	13	15	12	3	3

Table 13.8: BPTC anchor index values for the second subset of three-subset partitioning. Values run right, then down.

15	8	8	3	15	15	3	8
15	15	15	15	15	15	15	8
15	8	15	3	15	8	15	8
3	15	6	10	15	15	10	8
15	3	15	10	10	8	9	10
6	15	8	15	3	6	6	8
15	3	15	15	15	15	15	15
15	15	15	15	3	15	15	8

Table 13.9: BPTC anchor index values for the third subset of three-subset partitioning. Values run right, then down.

Next, the endpoints are unquantized to maximize the usage of the bits and to ensure that the negative ranges are oriented properly to interpolate as a two's complement value. The following pseudocode assumes the computation is being done using sufficiently large intermediate values to avoid overflow. For the unsigned float format, we unquantize a value x to *unq* by:

```
if (EPB >= 15)
    unq = x;
else if (x == 0)
    unq = 0;
else if (x == ((1 << EPB)-1))
    unq = 0xFFFF;
else
    unq = ((x << 15) + 0x4000) >> (EPB-1);
```

The signed float unquantization is similar, but needs to worry about orienting the negative range:

```
s = 0;
if (EPB >= 16) {
    unq = x;
} else {
    if (x < 0) {
        s = 1;
        x = -x;
    }

    if (x == 0)
        unq = 0;
    else if (x >= ((1 << (EPB-1))-1))
        unq = 0x7FFF;
    else
        unq = ((x << 15) + 0x4000) >> (EPB-1);

    if (s)
        unq = -unq;
}
```

After the endpoints are unquantized, interpolation proceeds as in the fixed-point formats above including the interpolation weight table.

The interpolated values are passed through a final unquantization step. For the unsigned format, this step simply multiplies by $\frac{31}{64}$. The signed format negates negative components, multiplies by $\frac{31}{32}$, then ORs in the sign bit if the original value was negative.

The resultant value should be a legal 16-bit half float.

Mode Number	Transformed Endpoints	Partition Bits (PB)	Endpoint Bits (EPB)	Delta Bits
0	1	5	10	{5, 5, 5}
1	1	5	7	{6, 6, 6}
2	1	5	11	{5, 4, 4}
6	1	5	11	{4, 5, 4}
10	1	5	11	{4, 4, 5}
14	1	5	9	{5, 5, 5}
18	1	5	8	{6, 5, 5}
22	1	5	8	{5, 6, 5}
26	1	5	8	{5, 5, 6}
30	0	5	6	{6, 6, 6}
3	0	0	10	{10, 10, 10}
7	1	0	11	{9, 9, 9}
11	1	0	12	{8, 8, 8}
15	1	0	16	{4, 4, 4}

Table 13.10: Endpoint and partition parameters for BPTC block modes

Mode Number	Block Format
0	m[1:0], g2[4], b2[4], b3[4], r0[9(:0], g0[9:0], b0[9:0], r1[4:0], g3[4], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0],
1	m[1:0], g2[5], g3[4], g3[5], r0[6:0], b3[0], b3[1], b2[4], g0[6:0], b2[5], b3[2], g2[4], b0[6:0], b3[3], b3[5], b3[4], r1[5:0], g2[3:0], g1[5:0], g3[3:0], b1[5:0], b2[3:0], r2[5:0], r3[5:0]
2	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[4:0], r0[10], g2[3:0], g1[3:0], g0[10], b3[0], g3[3:0], b1[3:0], b0[10], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0],
6	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[3:0], r0[10], g3[4], g2[3:0], g1[4:0], g0[10], g3[3:0], b1[3:0], b0[10], b3[1], b2[3:0], r2[3:0], b3[0], b3[2], r3[3:0], g2[4], b3[3]
10	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[3:0], r0[10], b2[4], g2[3:0], g1[3:0], g0[10], b3[0], g3[3:0], b1[4:0], b0[10], b2[3:0], r2[3:0], b3[1], b3[2], r3[3:0], b3[4], b3[3]
14	m[4:0], r0[8:0], b2[4], g0[8:0], g2[4], b0[8:0], b3[4], r1[4:0], g3[4], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]
18	m[4:0], r0[7:0], g3[4], b2[4], g0[7:0], b3[2], g2[4], b0[7:0], b3[3], b3[4], r1[5:0], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[5:0], r3[5:0]
22	m[4:0], r0[7:0], b3[0], b2[4], g0[7:0], g2[5], g2[4], b0[7:0], g3[5], b3[4], r1[4:0], g3[4], g2[3:0], g1[5:0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]
26	m[4:0], r0[7:0], b3[1], b2[4], g0[7:0], b2[5], g2[4], b0[7:0], b3[5], b3[4], r1[4:0], g3[4], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[5:0], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]
30	m[4:0], r0[5:0], g3[4], b3[0], b3[1], b2[4], g0[5:0], g2[5], b2[5], b3[2], g2[4], b0[5:0], g3[5], b3[3], b3[5], b3[4], r1[5:0], g2[3:0], g1[5:0], g3[3:0], b1[5:0], b2[3:0], r2[5:0], r3[5:0]
3	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[9:0], g1[9:0], b1[9:0]
7	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[8:0], r0[10], g1[8:0], g0[10], b1[8:0], b0[10]
11	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[7:0], r0[10:11], g1[7:0], g0[10:11], b1[7:0], b0[10:11]
15	m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[3:0], r0[10:15], g1[3:0], g0[10:15], b1[3:0], b0[10:15]

Table 13.11: Block formats for BC6H block modes

Chapter 14

ETC1 Compressed Texture Image Formats

This description is derived from the OES_compressed_ETC1_RGB8_texture OpenGL extension.

The texture is described as a number of 4×4 pixel blocks. If the texture (or a particular mip-level) is smaller than 4 pixels in any dimension (such as a 2×2 or a 8×1 texture), the texture is found in the upper left part of the block(s), and the rest of the pixels are not used. For instance, a texture of size 4×2 will be placed in the upper half of a 4×4 block, and the lower half of the pixels in the block will not be accessed.

Pixel a_1 (see Table 14.4) of the first block in memory will represent the texture coordinate (u=0, v=0). Pixel a_2 in the second block in memory will be adjacent to pixel m1 in the first block, etc until the width of the texture. Then pixel a_3 in the following block (third block in memory for a 8×8 texture) will be adjacent to pixel d1 in the first block, etc until the height of the texture. The data storage for an 8×8 texture using the first, second, third and fourth block if stored in that order in memory would have the texels encoded in the same order as a simple linear format as if the bytes describing the pixels came in the following memory order: $a_1\ e_1\ i_1\ m_1\ a_2\ e_2\ i_2\ m_2\ b_1\ f_1\ j_1\ n_1\ b_2\ f_2\ j_2\ n_2\ c_1\ g_1\ k_1\ o_1\ c_2\ g_2\ k_2\ o_2\ d_1\ h_1\ l_1\ p_1\ d_2\ h_2\ l_2\ p_2\ a_3\ e_3\ i_3\ m_3\ a_4\ e_4\ i_4\ m_4\ b_3\ f_3\ j_3\ n_3\ b_4\ f_4\ j_4\ n_4\ c_3\ g_3\ k_3\ o_3\ c_4\ g_4\ k_4\ o_4\ d_3\ h_3\ l_3\ p_3\ d_4\ h_4\ l_4\ p_4$.

The number of bits that represent a 4×4 texel block is 64 bits.

The data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The 64 bits specifying the block are then represented by the following 64 bit integer:

$$int64bit = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$

Each 64-bit word contains information about a 4×4 pixel block as shown in Table 14.5. There are two modes in ETC1; the 'individual' mode and the 'differential' mode. Which mode is active for a particular 4×4 block is controlled by bit 33, which we call 'diffbit'. If *diffbit* = 0, the 'individual' mode is chosen, and if *diffbit* = 1, then the 'differential' mode is chosen. The bit layout for the two modes are different: The bit layout for the individual mode is shown in Table 14.1 part a and part c, and the bit layout for the differential mode is laid out in Table 14.1 part b and part c.

In both modes, the 4×4 block is divided into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we call 'flipbit'. If *flipbit*=0, the block is divided into two 2×4 subblocks side-by-side, as shown in Table 14.6. If *flipbit*=1, the block is divided into two 4×2 subblocks on top of each other, as shown in Table 14.7.

In both individual and differential mode, a 'base color' for each subblock is stored, but the way they are stored is different in the two modes:

In the 'individual' mode (*diffbit* = 0), the base color for subblock 1 is derived from the codewords R1 (bit 63-60), G1 (bit 55-52) and B1 (bit 47-44), see section a of Table 14.1. These four bit values are extended to RGB888 by replicating the four higher order bits in the four lower order bits. For instance, if R1 = 14 = 1110b, G1 = 3 = 0011b and B1 = 8 = 1000b, then the red component of the base color of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The base color for subblock 2 is decoded the same way, but using the 4-bit codewords R2 (bit 59-56), G2 (bit 51-48)and B2 (bit 43-40) instead. In summary, the base colors for the subblocks in the individual mode are:

$$base\ col\ subblock1 = extend\_4to8bits(R1, G1, B1)$$
$$base\ col\ subblock2 = extend\_4to8bits(R2, G2, B2)$$

In the 'differential' mode (*diffbit* = 1), the base color for subblock 1 is derived from the five-bit codewords $R1'$, $G1'$ and $B1'$. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if $R1'$ = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if $G1'$ = 4 = 00100b and $B1'$ = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five bit representation for the base color of subblock 2 is obtained by modifying the 5-bit codewords R1' G1' and B1' by the codewords dR2, dG2 and dB2. Each of dR2, dG2 and dB2 is a 3-bit two-complement number that can hold values between -4 and $+3$. For instance, if R1' = 28 as above, and dR2 = 100b = -4, then the five bit representation for the red color component is $28+(-4)=24$ = 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G1' = 4, dG2 = 2, B1' = 3 and dB2 = 0, the base color of subblock 2 will be RGB = (198, 49, 24). In summary, the base colors for the subblocks in the differential mode are:

$$base\ col\ subblock1 = extend\_5to8bits(R1', G1', B1')$$
$$base\ col\ subblock2 = extend\_5to8bits(R1'+dR2, G1'+dG2, B1'+dB2)$$

Note that these additions are not allowed to under- or overflow (go below zero or above 31). (The compression scheme can easily make sure they don't.) For over- or underflowing values, the behavior is undefined for all pixels in the 4×4 block. Note also that the extension to eight bits is performed *after* the addition.

After obtaining the base color, the operations are the same for the two modes 'individual' and 'differential'. First a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39-37), and for subblock 2, table codeword 2 is used (bits 36-34), see Table 14.1. The table codeword is used to select one of eight modifier tables, see Table 14.2. For instance, if the table code word is 010b = 2, then the modifier table [-29, -9, 9, 29] is selected. Note that the values in Table 14.2 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which modifier value to use from the modifier table using the two 'pixel index' bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Table 14.5) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see section c of Table 14.1. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits *diffbit* and *flipbit*. The pixel index bits are decoded using Table 14.3. If, for instance, the pixel index bits are 01b = 1, and the modifier table [-29, -9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see Table 14.3). This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: ($231+29$, $8+29$, $16+29$) resulting in (260, 37, 45). These values are then clamped to [0, 255], resulting in the color (255, 37, 45), and we are finished decoding the texel.

a) bit layout in bits 63 through 32 if diffbit = 0															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
base col1 R1 (4 bits)				base col2 R2 (4 bits)				base col1 G1 (4 bits)				base col2 G2 (4 bits)			
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
base col1 B1 (4 bits)				base col2 B2 (4 bits)				table cw 1				table cw 2		diff bit	flip bit
b) bit layout in bits 63 through 32 if diffbit = 1															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
base col1 R1' (5 bits)					dcol 2 dR2			base col1 G1' (4 bits)					dcol 2 dG2		
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
base col1 B1' (5 bits)					dcol 2 db2			table cw 1				table cw 2		diff bit	flip bit
c) bit layout in bits 31 through 0 (in both cases)															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
most significant pixel index bits															
p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
least significant pixel index bits															
p	o	n	m	l	k	j	i	h	g	f	e	d	c	b	a

Table 14.1: Texel Data format for ETC1 compressed textures

Table codeword	**Modifier table**			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table 14.2: Intensity modifier sets for ETC1 compressed textures

Pixel index value		**Resulting modifier value**
msb	lsb	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

Table 14.3: Mapping from pixel index values to modifier values for ETC1 compressed textures

First block in mem					Second block in mem				
a_1	e_1	i_1	m_1		a_2	e_2	i_2	m_2	$\rightarrow u$ direction
b_1	f_1	j_1	n_1		b_2	f_2	j_2	n_2	
c_1	g_1	k_1	o_1		c_2	g_2	k_2	o_2	
d_1	h_1	l_1	p_1		d_2	h_2	l_2	p_2	
a_3	e_3	i_3	m_3		a_4	e_4	i_4	m_4	
b_3	f_3	j_3	n_3		b_4	f_4	j_4	n_4	
c_3	g_3	k_3	o_3		c_4	g_4	k_4	o_4	
d_3	h_3	l_3	p_3		d_4	h_4	l_4	p_4	
Third block in mem					Fourth block in mem				$\downarrow v$ direction

Table 14.4: Pixel layout for an 8×8 texture using four ETC1 compressed blocks. Note how pixel a2 in the second block is adjacent to pixel m1 in the first block.

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Table 14.5: Pixel layout for an ETC1 compressed block

a	e		i	m
b	f		j	n
c	g		k	o
d	h		l	p

Table 14.6: Two 2×4-pixel ETC1 subblocks side-by-side

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Table 14.7: Two 4×2-pixel ETC1 subblocks on top of each other

Chapter 15

ETC2 Compressed Texture Image Formats

This description is derived from the "ETC Compressed Texture Image Formats" section of the OpenGL 4.5 specification.

The ETC formats form a family of related compressed texture image formats. They are designed to do different tasks, but also to be similar enough that hardware can be reused between them. Each one is described in detail below, but we will first give an overview of each format and describe how it is similar to others and the main differences.

RGB ETC2 is a format for compressing RGB data. It is a superset of the older ETC1 format. This means that an older ETC1 texture can be decoded using an ETC2-compliant decoder. The main difference is that the newer version contains three new modes; the 'T-mode' and the 'H-mode' which are good for sharp chrominance blocks and the 'Planar' mode which is good for smooth blocks.

RGB ETC2 with sRGB encoding is the same as linear RGB ETC2 with the difference that the values should be interpreted as being encoded with the sRGB transfer function instead of linear RGB-values.

RGBA ETC2 encodes RGBA 8-bit data. The RGB part is encoded exactly the same way as RGB ETC2. The alpha part is encoded separately.

RGBA ETC2 with sRGB encoding is the same as RGBA ETC2 but here the RGB-values (but not the alpha value) should be interpreted as being encoded with the sRGB transfer function.

Unsigned R11 EAC is a one-channel unsigned format. It is similar to the alpha part of RGBA ETC2 but not exactly the same; it delivers higher precision. It is possible to make hardware that can decode both formats with minimal overhead.

Unsigned RG11 EAC is a two-channel unsigned format. Each channel is decoded exactly as R11 EAC.

Signed R11 EAC is a one-channel signed format. This is good in situations when it is important to be able to preserve zero exactly, and still use both positive and negative values. It is designed to be similar enough to Signed R11 EAC so that hardware can decode both with minimal overhead, but it is not exactly the same. For example; the signed version does not add 0.5 to the base codeword, and the extension from 11 bits differ. For all details, see the corresponding sections.

Signed RG11 EAC is a two-channel signed format. Each channel is decoded exactly as signed R11 EAC.

RGB ETC2 with "punchthrough" alpha is very similar to RGB ETC2, but has the ability to represent "punchthrough"-alpha (completely opaque or transparent). Each block can select to be completely opaque using one bit. To fit this bit, there is no individual mode in RGB ETC2 with punchthrough alpha. In other respects, the opaque blocks are decoded as in RGB ETC2. For the transparent blocks, one index is reserved to represent transparency, and the decoding of the RGB channels are also affected. For details, see the corresponding sections.

RGB ETC2 with punchthrough alpha and sRGB encoding is the same as linear RGB ETC2 with punchthrough alpha but the RGB channel values should be interpreted as being encoded with the sRGB transfer function.

A texture compressed using any of the ETC texture image formats is described as a number of 4×4 pixel blocks.

Pixel a_1 (see Table 15.1) of the first block in memory will represent the texture coordinate $(u = 0, v = 0)$. Pixel a_2 in the second block in memory will be adjacent to pixel m_1 in the first block, etc. until the width of the texture. Then pixel a_3 in the following block (third block in memory for a 8×8 texture) will be adjacent to pixel d_1 in the first block, etc. until the height of the texture.

The data storage for an 8×8 texture using the first, second, third and fourth block if stored in that order in memory would have the texels encoded in the same order as a simple linear format as if the bytes describing the pixels came in the following memory order: $a_1\ e_1\ i_1\ m_1\ a_2\ e_2\ i_2\ m_2\ b_1\ f_1\ j_1\ n_1\ b_2\ f_2\ j_2\ n_2\ c_1\ g_1\ k_1\ o_1\ c_2\ g_2\ k_2\ o_2\ d_1\ h_1\ l_1\ p_1\ d_2\ h_2\ l_2\ p_2\ a_3\ e_3\ i_3\ m_3\ a_4\ e_4\ i_4\ m_4\ b_3\ f_3\ j_3\ n_3\ b_4\ f_4\ j_4\ n_4\ c_3\ g_3\ k_3\ o_3\ c_4\ g_4\ k_4\ o_4\ d_3\ h_3\ l_3\ p_3\ d_4\ h_4\ l_4\ p_4$.

First block in mem					Second block in mem				
a_1	e_1	i_1	m_1		a_2	e_2	i_2	m_2	$\rightarrow u$ direction
b_1	f_1	j_1	n_1		b_2	f_2	j_2	n_2	
c_1	g_1	k_1	o_1		c_2	g_2	k_2	o_2	
d_1	h_1	l_1	p_1		d_2	h_2	l_2	p_2	
a_3	e_3	i_3	m_3		a_4	e_4	i_4	m_4	
b_3	f_3	j_3	n_3		b_4	f_4	j_4	n_4	
c_3	g_3	k_3	o_3		c_4	g_4	k_4	o_4	
d_3	h_3	l_3	p_3		d_4	h_4	l_4	p_4	
Third block in mem					Fourth block in mem				$\downarrow v$ direction

Table 15.1: Pixel layout for an 8×8 texture using four ETC2 compressed blocks. Note how pixel a_3 in the third block is adjacent to pixel d_1 in the first block.

If the width or height of the texture (or a particular mip-level) is not a multiple of four, then padding is added to ensure that the texture contains a whole number of 4×4 blocks in each dimension. The padding does not affect the texel coordinates. For example, the texel shown as a_1 in Table 15.1 always has coordinates $i = 0, j = 0$. The values of padding texels are irrelevant, e.g., in a 3×3 texture, the texels marked as $m_1, n_1, o_1, d_1, h_1, l_1$ and p_1 form padding and have no effect on the final texture image.

The number of bits that represent a 4×4 texel block is 64 bits if the format is RGB ETC2, RGB ETC2 with sRGB encoding, RGBA ETC2 with punchthrough alpha, or RGB ETC2 with punchthrough alpha and sRGB encoding.

In those cases the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The 64 bits specifying the block are then represented by the following 64 bit integer:

$$int64bit = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$

The number of bits that represent a 4×4 texel block is 128 bits if the format is RGBA ETC2 with a linear or sRGB transfer function. In those cases the data for a block is stored as a number of bytes: $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}\}$, where byte q_0 is located at the lowest memory address and q_{15} at the highest. This is split into two 64-bit integers, one used for color channel decompression and one for alpha channel decompression:

$$int64bitAlpha = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$
$$int64bitColor = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_8 + q_9) + q_{10}) + q_{11}) + q_{12}) + q_{13}) + q_{14}) + q_{15}$$

Format RGB ETC2

For RGB ETC2, each 64-bit word contains information about a three-channel 4×4 pixel block as shown in Table 15.2.

The blocks are compressed using one of five different 'modes'. Section a of Table 15.3 shows the bits used for determining the mode used in a given block. First, if the bit marked 'D' is set to 0, the 'individual' mode is used. Otherwise, the three 5-bit values R, G and B, and the three 3-bit values dR, dG and dB are examined. R, G and B are treated as integers between 0 and 31 and dR, dG and dB as two's-complement integers between -4 and $+3$. First, R and dR are added, and if the sum is not within the interval $[0,31]$, the 'T' mode is selected. Otherwise, if the sum of G and dG is outside the interval $[0,31]$, the 'H' mode is selected. Otherwise, if the sum of B and dB is outside of the interval $[0,31]$, the 'planar' mode is selected. Finally, if the 'D' bit is set to 1 and all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

a	e	i	m	$\rightarrow u$ direction
b	f	j	n	
c	g	k	o	
d	h	l	p	
$\downarrow v$ direction				

Table 15.2: Pixel layout for an ETC2 compressed block

a) location of bits for mode selection

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					dR			G					dG			B					dB			.....						D	.

b) bit layout for bits 63 through 32 for 'individual' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R1					R2			G1					G2			B1					B2			table1			table2			0	F_B

c) bit layout for bits 63 through 32 for 'differential' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					dR			G					dG			B					dB			table1			table2			1	F_B

d) bit layout for bits 63 through 32 for 'T' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
...			R1a		.	R1b		G1					B1			R2					G2			B2				da		1	db

e) bit layout for bits 63 through 32 for 'H' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R1				G1 a			...			G1 b	B1 a	.	B1 b			R2				G2			B2				da		1	db

f) bit layout for bits 31 through 0 for 'individual', 'diff', 'T' and 'H' modes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p0	o0	n0	m0	l0	k0	j0	i0	h0	g0	f0	e0	d0	c0	b0	a0	p1	o1	n1	m1	l1	k1	j1	i1	h1	g1	f1	e1	d1	c1	b1	a1

g) bit layout for bits 63 through 0 for 'planar' mode:

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

Table 15.3: Texel Data format for ETC2 compressed texture formats

a	e		i	m
b	f		j	n
c	g		k	o
d	h		l	p

Table 15.4: Two 2-by-4-pixel ETC2 subblocks side-by-side.

The layout of the bits used to decode the ‘individual’ and ‘differential’ modes are shown in section b and section c of Table 15.3, respectively. Both of these modes share several characteristics. In both modes, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the ‘flip bit’. If the ‘flip bit’ is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Table 15.4. If the ‘flip bit’ is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Table 15.5. In both modes, a ‘base color’ for each subblock is stored, but the way they are stored is different in the two modes:

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Table 15.5: Two 4-by-2-pixel ETC2 subblocks on top of each other.

In the ‘individual’ mode, following the layout shown in section b of Table 15.3, the base color for subblock 1 is derived from the codewords R1 (bit 63—60), G1 (bit 55—52) and B1 (bit 47—44). These four bit values are extended to RGB888 by replicating the four higher order bits in the four lower order bits. For instance, if R1 = 14 = 1110 binary (1110b for short), G1 = 3 = 0011b and B1 = 8 = 1000b, then the red component of the base color of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The base color for subblock 2 is decoded the same way, but using the 4-bit codewords R2 (bit 59—56), G2 (bit 51—48)and B2 (bit 43—40) instead. In summary, the base colors for the subblocks in the individual mode are:

$$base\ col\ subblock1 = extend\_4to8bits(R1, G1, B1)$$
$$base\ col\ subblock2 = extend\_4to8bits(R2, G2, B2)$$

In the ‘differential’ mode, following the layout shown in section c of Table 15.3, the base color for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if R = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five-bit representation for the base color of subblock 2 is obtained by modifying the five-bit codewords R G and B by the codewords dR, dG and dB. Each of dR, dG and dB is a 3-bit two’s-complement number that can hold values between -4 and $+3$. For instance, if R = 28 as above, and dR = 100b = -4, then the five bit representation for the red color component is $28 + (-4) = 24 =$ 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, dG = 2, B = 3 and dB = 0, the base color of subblock 2 will be RGB = 198, 49, 24. In summary, the base colors for the subblocks in the differential mode are:

$$base\ col\ subblock1 = extend\_5to8bits(R, G, B)$$
$$base\ col\ subblock2 = extend\_5to8bits(R + dR, G + dG, B + dB)$$

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the ‘differential’ mode.

After obtaining the base color, the operations are the same for the two modes ‘individual’ and ‘differential’. First a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39—37), and for subblock 2, table codeword 2 is used (bits 36—34), see section b or section c of Table 15.3. The table codeword is used to select one of eight modifier tables, see Table 15.6. For instance, if the table code word is 010 binary = 2, then the modifier table -29, -9, 9, 29 is selected for the corresponding sub-block. Note that the values in Table 15.6 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which modifier value to use from the modifier table using the two ‘pixel index’ bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Table 15.2) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see section f of Table 15.3. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits ‘diffbit’ and ‘flipbit’. The pixel index bits are decoded using Table 15.7. If, for instance, the pixel index bits are 01 binary = 1, and the modifier table -29, -9, 9, 29 is used, then the modifier value selected for that pixel is 29 (see Table 15.7). This modifier value is now used to additively modify the base

Table codeword	Modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table 15.6: ETC2 intensity modifier sets for ‘individual’ and ‘differential’ modes

Pixel index value		Resulting modifier value
msb	lsb	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

Table 15.7: Mapping from pixel index values to modifier values for RGB ETC2 compressed textures.

color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: $(231+29,\ 8+29,\ 16+29)$ resulting in (260, 37, 45). These values are then clamped to [0, 255], resulting in the color (255, 37, 45), and we are finished decoding the texel.

The ‘T’ and ‘H’ compression modes also share some characteristics: both use two base colors stored using 4 bits per channel decoded as in the individual mode. Unlike the ‘individual’ mode however, these bits are not stored sequentially, but in the layout shown in section d and section e of Table 15.3. To clarify, in the ‘T’ mode, the two colors are constructed as follows:

$$base\ col\ 1 = extend\_4to8bits((R1a \ll 2) \mid R1b, G1, B1)$$
$$base\ col\ 2 = extend\_4to8bits(R2, G2, B2)$$

Here, $\ll$ denotes bit-wise left shift and $\mid$ denotes bit-wise OR. In the ‘H’ mode, the two colors are constructed as follows:

$$base\ col\ 1 = extend\_4to8bits(R1, (G1a \ll 1) \mid G1b, (B1a \ll 3) \mid B1b)$$
$$base\ col\ 2 = extend\_4to8bits(R2, G2, B2)$$

Both the ‘T’ and ‘H’ modes have four ‘paint colors’ which are the colors that will be used in the decompressed block, but they are assigned in a different manner. In the ‘T’ mode, ‘paint color 0’ is simply the first base color, and ‘paint color 2’ is the second base color. To obtain the other ‘paint colors’, a ‘distance’ is first determined, which will be used to modify the luminance of one of the base colors. This is done by combining the values ‘da’ and ‘db’ shown in section d of Table 15.3 by $(da \ll 1) \mid db$, and then using this value as an index into the small look-up table shown in Table 15.8. For example, if ‘da’ is 10 binary and ‘db’ is 1 binary, the index is 101 binary and the selected distance will be 32. ‘Paint color 1’ is then equal to the second base color with the ‘distance’ added to each channel, and ‘paint color 3’ is the second base color with the ‘distance’ subtracted.

Distance index	Distance
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

Table 15.8: Distance table for ETC2 ‘T’ and ‘H’ modes.

In summary, to determine the four ‘paint colors’ for a ‘T’ block:

$$\begin{aligned} \textit{paint color } 0 &= \textit{base col } 1 \\ \textit{paint color } 1 &= \textit{base col } 2 + (d,d,d) \\ \textit{paint color } 2 &= \textit{base col } 2 \\ \textit{paint color } 3 &= \textit{base col } 2 - (d,d,d) \end{aligned}$$

In both cases, the value of each channel is clamped to within [0,255].

A ‘distance’ value is computed for the ‘H’ mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table 15.8, ‘da’ and ‘db’ shown in section e of Table 15.3 are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (base col 1 value $\geq$ base col 2 value), the ‘value’ of a color for the comparison being equal to $(R \ll 16) + (G \ll 8) + B$. Once the ‘distance’ d has been determined for an ‘H’ block, the four ‘paint colors’ will be:

$$\begin{aligned} \textit{paint color } 0 &= \textit{base col } 1 + (d,d,d) \\ \textit{paint color } 1 &= \textit{base col } 1 - (d,d,d) \\ \textit{paint color } 2 &= \textit{base col } 2 + (d,d,d) \\ \textit{paint color } 3 &= \textit{base col } 2 - (d,d,d) \end{aligned}$$

Again, all color components are clamped to within [0,255]. Finally, in both the ‘T’ and ‘H’ modes, every pixel is assigned one of the four ‘paint colors’ in the same way the four modifier values are distributed in ‘individual’ or ‘differential’ blocks. For example, to choose a paint color for pixel d, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned paint color 2.

The final mode possible in an RGB ETC2-compressed block is the ‘planar’ mode. Here, three base colors are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three base colors are stored in RGB 676 format, and stored in the manner shown in section g of Table 15.3. The three colors are there labelled ‘O’, ‘H’ and ‘V’, so that the three components of color ‘V’ are RV, GV and BV, for example. Some color channels are split into non-consecutive bit-ranges, for example BO is reconstructed using BO1 as the most significant bit, BO2 as the two following bits, and BO3 as the three least significant bits.

Once the bits for the base colors have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the base colors in other modes. For example, the 6-bit blue and red channels are extended by replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three base colors in RGB888 format, the color of each pixel can then be determined as:

$$R(x,y) = \frac{x \times (RH - RO)}{4.0} + \frac{y \times (RV - RO)}{4.0} + RO$$
$$G(x,y) = \frac{x \times (GH - GO)}{4.0} + \frac{y \times (GV - GO)}{4.0} + GO$$
$$B(x,y) = \frac{x \times (BH - BO)}{4.0} + \frac{y \times (BV - BO)}{4.0} + BO$$

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Table 15.2 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

$$R(x,y) = clamp255((x \times (RH - RO) + y \times (RV - RO) + 4 \times RO + 2) \gg 2)$$
$$G(x,y) = clamp255((x \times (GH - GO) + y \times (GV - GO) + 4 \times GO + 2) \gg 2)$$
$$B(x,y) = clamp255((x \times (BH - BO) + y \times (BV - BO) + 4 \times BO + 2) \gg 2)$$

where $clamp255$ clamps the value to a number in the range [0, 255] and where $\gg$ performs bit-wise right shift.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

Format RGB ETC2 with sRGB encoding

Decompression of floating point sRGB values in RGB ETC2 with sRGB encoding follows that of floating point RGB values of linear RGB ETC2. The result is sRGB-encoded values between 0.0 and 1.0. The further conversion from an sRGB encoded component, cs, to a linear component, cl, is done according to the formulae in Section 7.7.3. Assume cs is the sRGB component in the range [0,1].

Format RGBA ETC2

Each 4×4 block of RGBA8888 information is compressed to 128 bits. To decode a block, the two 64-bit integers int64bitAlpha and int64bitColor are calculated as described in Section 15.1. The RGB component is then decoded the same way as for RGB ETC2 (see Section 15.1), using int64bitColor as the int64bit codeword.

a) bit layout in bits 63 through 48															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
base_codeword								multiplier				table index			
b) bit layout in bits 47 through 0, with pixels as name in Table 15.2, bits labelled from 0 being the LSB to 47 being the MSB.															
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
a0	a1	a2	b0	b1	b2	c0	c1	c2	d0	d1	d2	e0	e1	e2	f0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
f1	f2	g0	g1	g2	h0	h1	h2	i0	i1	i2	j0	j1	j2	k0	k1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
k2	l0	l1	l2	m0	m1	m2	n0	n1	n2	o0	o1	o2	p0	p1	p2

Table 15.9: Texel Data format for alpha part of RGBA ETC2 compressed textures

The 64-bits in int64bitAlpha used to decompress the alpha channel are laid out as shown in Table 15.9. The information is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier, which are used together to compute 8 pixel values to be used in the block. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of these 8 possible values for each pixel in the block.

The decoded value of a pixel is a value between 0 and 255 and is calculated the following way:

$$clamp255((base\_codeword) + modifier \times multiplier)$$

EQUATION 15.1: ETC2-base

where $clamp255(\cdot)$ maps values outside the range [0, 255] to 0.0 or 255.0.

The *base_codeword* is stored in the first 8 bits (bits 63—56) as shown in Table 15.9 part (a). This is the first term in Equation ETC2-base.

Next, we want to obtain the modifier. Bits 51—48 in Table 15.9 part (a) form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 15.10.

table index	**modifier table**							
0	-3	-6	-9	-15	2	5	8	14
1	-3	-7	-10	-13	2	6	9	12
2	-2	-5	-8	-13	1	4	7	12
3	-2	-4	-6	-13	1	3	5	12
4	-3	-6	-8	-12	2	5	7	11
5	-3	-7	-9	-11	2	6	8	10
6	-4	-7	-8	-11	3	6	7	10
7	-3	-5	-8	-11	2	4	7	10
8	-2	-6	-8	-10	1	5	7	9
9	-2	-5	-8	-10	1	4	7	9
10	-2	-4	-8	-10	1	3	7	9
11	-2	-5	-7	-10	1	4	6	9
12	-3	-4	-7	-10	2	3	6	9
13	-1	-2	-3	-10	0	1	2	9
14	-4	-6	-8	-9	3	5	7	8
15	-3	-5	-7	-9	2	4	6	8

Table 15.10: Intensity modifier sets for RGBA ETC2 alpha component

For example, a table index of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. As shown in Table 15.9 part (b), bits 47—0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel indices are stored in bit 44—42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our modifier, which is the starting point of our second term in the addition.

In the next step we obtain the multiplier value; bits 55—52 form a four-bit 'multiplier' between 0 and 15. This value should be multiplied with the modifier. An encoder is not allowed to produce a multiplier of zero, but the decoder should still be able to handle also this case (and produce $0\times$ modifier $= 0$ in that case).

The modifier times the multiplier now provides the third and final term in the sum in Equation ETC2-base. The sum is calculated and the value is clamped to the interval [0, 255]. The resulting value is the 8-bit output value.

For example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier of 2. We will then start with the base codeword 103 (01100111 binary). Next, a 'table index' of 13 selects table -1, -2, -3, -10, 0, 1, 2, 9, and using a pixel index of 3 will result in a modifier of -10. The multiplier is 2, forming $-10 \times 2 = -20$. We now add this to the base value and get $103 - 20 = 83$. After clamping we still get 83 = 01010011 binary. This is our 8-bit output value.

This specification gives the output for each channel in 8-bit integer values between 0 and 255, and these values all need to be divided by 255 to obtain the final floating point representation.

Note that hardware can be effectively shared between the alpha decoding part of this format and that of R11 EAC texture. For details on how to reuse hardware, see Section 15.5.

Format RGBA ETC2 with sRGB encoding

Decompression of floating point sRGB values in RGBA ETC2 with sRGB encoding follows that of floating point RGB values of linear RGBA ETC2. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, *cs*, to a linear component, *cl*, is according to the formula in Section 7.7.3. Assume *cs* is the sRGB component in the range [0,1].

The alpha component of RGBA ETC2 with sRGB encoding is done in the same way as for linear RGBA ETC2.

Format Unsigned R11 EAC

The number of bits to represent a 4×4 texel block is 64 bits. if format is R11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64 bit integer:

$$int64bit = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Table 15.2. The 64-bit word is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table 15.9.

The decoded value is calculated as:

$$clamp1((base\_codeword + 0.5) \times \frac{1}{255.875} + modifier \times multiplier \times \frac{1}{255.875})$$

where $clamp1(\cdot)$ maps values outside the range [0.0, 1.0] to 0.0 or 1.0.

We will now go into detail how the decoding is done. The result will be an 11-bit fixed point number where 0 represents 0.0 and 2047 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between 0 and 2047 we must multiply Equation R11 EAC-start by 2047.0:

$$clamp2((base\_codeword + 0.5) \times \frac{2047.0}{255.875} + modifier \times multiplier \times \frac{2047.0}{255.875})$$

where $clamp2(\cdot)$ clamps to the range [0.0, 2047.0]. Since $\frac{2047.0}{255.875}$ is exactly 8.0, the above equation can be written as

$$clamp2(base\_codeword \times 8 + 4 + modifier \times multiplier \times 8)$$

EQUATION 15.2: Equation R11 EAC simple

The base_codeword is stored in the first 8 bits as shown in Table 15.9 part (a). Bits 63—56 in each block represent an eight-bit integer (base_codeword) which is multiplied by 8 by shifting three steps to the left. We can add 4 to this value without addition logic by just inserting 100 binary in the last three bits after the shift. For example, if base_codeword is 129 = 10000001 binary (or 10000001b for short), the shifted value is 10000001000b and the shifted value including the +4 term is 10000001100b $= 1036 = 129 \times 8 + 4$. Hence we have summed together the first two terms of the sum in Equation R11 EAC simple.

Next, we want to obtain the modifier. Bits 51-48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 15.10. For example, a table index of 13 (1101 binary) means that we should use table [−1, −2, −3, −10, 0, 1, 2, 9]. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. Bits 47—0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel indices are stored in bit 44—42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is −10. This is now our modifier, which is the starting point of our second term in the sum.

In the next step we obtain the multiplier value; bits 55—52 form a four-bit 'multiplier' between 0 and 15. We will later treat what happens if the multiplier value is zero, but if it is nonzero, it should be multiplied width the modifier. This product should then be shifted three steps to the left to implement the $\times 8$ multiplication. The result now provides the third and final term in the sum in Equation R11 EAC simple. The sum is calculated and the result is clamped to a value in the interval [0, 2047]. The resulting value is the 11-bit output value.

For example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier of 2 . We will then first multiply the base_codeword 103 (01100111b) by 8 by left-shifting it (0110111000b) and then add 4 resulting in 0110111100b $= 828 = 103 \times 8 + 4$. Next, a 'table index' of 13 selects table [−1, −2, −3, −10, 0, 1, 2, 9], and using a pixel index of 3 will result in a modifier of −10. The multiplier is nonzero, which means that we should multiply it with the modifier, forming $-10 \times 2 = -20 =$ 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b $= -160$. We now add this to the base value and get $828 - 160 = 668$. After clamping we still get 668 = 01010011100b. This is our 11-bit output value, which represents the value $\frac{668}{2047} = 0.32633121\ldots$

If the multiplier_value is zero (i.e., the multiplier bits 55—52 are all zero), we should set the multiplier to $\frac{1.0}{8.0}$. Equation-r11eac-eqn-simple can then be simplified to

$$clamp2(base\_codeword \times 8 + 4 + modifier)$$

As an example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier_value of 0. We treat the base_codeword the same way, getting $828 = 103 \times 8 + 4$. The modifier is still -10. But the multiplier should now be $\frac{1}{8}$, which means that third term becomes $\frac{-10 \times (1}{8) \times 8 = -10}$. The sum therefore becomes $828 - 10 = 818$. After clamping we still get 818 = 01100110010b, and this is our 11-bit output value, and it represents $\frac{818}{2047} = 0.39960918\ldots$

Some OpenGL ES implementations may find it convenient to use 16-bit values for further processing. In this case, the 11-bit value should be extended using bit replication. An 11-bit value x is extended to 16 bits through $(x \ll 5) + (x \gg 6)$. For example, the value 668 = 01010011100b should be extended to 0101001110001010b = 21386.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that the method does not have the same reconstruction levels as the alpha part in the RGBA ETC2 format. For instance, for a *base_value* of 255 and a *table_value* of 0, the alpha part of the RGBA ETC2 format will represent a value of $\frac{(255+0)}{255.0} = 1.0$ exactly. In R11 EAC the same *base_value* and *table_value* will instead represent $\frac{(255.5+0)}{255.875} = 0.99853444\ldots$ That said, it is still possible to decode the alpha part of the RGBA ETC2-format using R11 EAC hardware. This is done by truncating the 11-bit number to 8 bits. As an example, if base_value = 255 and table_value = 0, we get the 11-bit value $(255 \times 8 + 4 + 0) = 2044 = 1111111100$b, which after truncation becomes the 8-bit value 11111111b = 255 which is exactly the correct value according to RGBA ETC2. Clamping has to be done to 0, 255 after truncation for RGBA ETC2 decoding. Care must also be taken to handle the case when the multiplier value is zero. In the 11-bit version, this means multiplying by $\frac{1}{8}$, but in the 8-bit version, it really means multiplication by 0. Thus, the decoder will have to know if it is an RGBA ETC2 texture or an R11 EAC texture to decode correctly, but the hardware can be 100\% shared.

As stated above, a base_value of 255 and a table_value of 0 will represent a value of $\frac{(255.5+0)}{255.875} = 0.99853444\ldots$, and this does not reach 1.0 even though 255 is the highest possible base_codeword. However, it is still possible to reach a pixel value of 1.0 since a modifier other than 0 can be used. Indeed, half of the modifiers will often produce a value of 1.0. As an example, assume we choose the base_value 255, a multiplier of 1 and the modifier table [−3 −5 −7 −9 2 4 6 8]. Starting with-r11eac-eqn-simple,

$$clamp1((base\_codeword + 0.5) \times \frac{1}{255.875} + table\_value \times multiplier \times \frac{1}{255.875})$$

we get

$$clamp1((255+0.5)\times\frac{1}{255.875}+\begin{bmatrix}-3 & -5 & -7 & -9 & 2 & 4 & 6 & 8\end{bmatrix}\times\frac{1}{255.875})$$

which equals

$$clamp1(\begin{bmatrix}0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.01 & 1.02 & 1.03\end{bmatrix})$$

or after clamping

$$\begin{bmatrix}0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.00 & 1.00 & 1.00\end{bmatrix}$$

which shows that several values can be 1.0, even though the base value does not reach 1.0. The same reasoning goes for 0.0.

Format Unsigned RG11 EAC

The number of bits to represent a 4×4 texel block is 128 bits if the format is RG11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0,q_1,q_2,q_3,q_4,q_5,q_6,q_7,p_0,p_1,p_2,p_3,p_4,p_5,p_6,p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

$$\text{int64bit0} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$
$$\text{int64bit1} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times p_0+p_1)+p_2)+p_3)+p_4)+p_5)+p_6)+p_7$$

The 64-bit word int64bit0 contains information about the red component of a two-channel 4x4 pixel block as shown in Table 15.2, and the word int64bit1 contains information about the green component. Both 64-bit integers are decoded in the same way as R11 EAC described in Section-r11eac-r11eac.

Format Signed R11 EAC

The number of bits to represent a 4×4 texel block is 64 bits if the format is signed R11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0,q_1,q_2,q_3,q_4,q_5,q_6,q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64 bit integer:

$$\text{int64bit} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Table 15.2. The 64-bit word is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table 15.9.

The decoded value is calculated as

$$clamp1(base\_codeword\times\frac{1}{127.875}+modifier\times multiplier\times\frac{1}{127.875})$$

EQUATION 15.3: R11-start

where $clamp1(\cdot)$ maps values outside the range $[-1.0, 1.0]$ to -1.0 or 1.0. We will now go into detail how the decoding is done. The result will be an 11-bit two's-complement fixed point number where -1023 represents -1.0 and 1023 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between −1023 and 1023 we must multiply Equation R11-start by 1023.0:

$$clamp2(base\_codeword \times \frac{1023.0}{127.875} + modifier \times multiplier \times \frac{1023.0}{127.875})$$

where clamp2(.) clamps to the range [−1023.0, 1023.0]. Since $\frac{1023.0}{127.875}$ is exactly 8, the above formula can be written as:

$$clamp2(base\_codeword \times 8 + modifier \times multiplier \times 8)$$

The *base_codeword* is stored in the first 8 bits as shown in Table 15.9 part (a). It is a two's-complement value in the range [−127, 127], and where the value −128 is not allowed; however, if it should occur anyway it must be treated as −127. The base_codeword is then multiplied by 8 by shifting it left three steps. For example the value 65 = 01000001 binary (or 01000001b for short) is shifted to 01000001000b $= 520 = 65 \times 8$.

Next, we want to obtain the modifier. Bits 51—48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table 15.10. For example, a table index of 13 (1101 binary) means that we should use table −1, −2, −3, −10, 0, 1, 2, 9. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. Bits 47—0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel *b*. Its pixel indices are stored in bit 44—42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is −10. This is now our modifier, which is the starting point of our second term in the sum.

In the next step we obtain the multiplier value; bits 55-52 form a four-bit 'multiplier' between 0 and 15. We will later treat what happens if the multiplier value is zero, but if it is nonzero, it should be multiplied with the modifier. This product should then be shifted three steps to the left to implement the ×8 multiplication. The result now provides the third and final term in the sum in Equation-signedr11eac-eqn-simple. The sum is calculated and the result is clamped to a value in the interval [−1023, 1023]. The resulting value is the 11-bit output value.

For example, assume a a base_codeword of 60, a 'table index' of 13, a pixel index of 3 and a multiplier of 2. We start by multiplying the base_codeword (00111100b) by 8 using bit shift, resulting in (00111100000b) $= 480 = 60 \times 8$. Next, a 'table index' of 13 selects table [−1, −2, −3, −10, 0, 1, 2, 9], and using a pixel index of 3 will result in a modifier of −10. The multiplier is nonzero, which means that we should multiply it with the modifier, forming $-10 \times 2 = -20 =$ 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b = −160. We now add this to the base value and get $480 - 160 = 320$. After clamping we still get 320 = 00101000000b. This is our 11-bit output value, which represents the value $\frac{320}{1023} = 0.31280547\ldots$.

If the multiplier_value is zero (i.e., the multiplier bits 55-52 are all zero), we should set the multiplier to $\frac{1.0}{8.0}$. Equation-signedr11eac-eqn-simple can then be simplified to:

$$clamp2(base\_codeword \times 8 + modifier)$$

As an example, assume a base_codeword of 65, a 'table index' of 13, a pixel index of 3 and a multiplier_value of 0. We treat the base_codeword the same way, getting $480 = 60 \times 8$. The modifier is still −10. But the multiplier should now be $\frac{1}{8}$, which means that third term becomes $-10 \times (\frac{1}{8}) \times 8 = -10$. The sum therefore becomes $480 - 10 = 470$. Clamping does not affect the value since it is already in the range [−1023, 1023], and the 11-bit output value is therefore 470 = 00111010110b. This represents $\frac{470}{1023} = 0.45943304\ldots$

Some OpenGL ES implementations may find it convenient to use two's-complement 16-bit values for further processing. In this case, a positive 11-bit value should be extended using bit replication on all the bits except the sign bit. An 11-bit value x is extended to 16 bits through $(x \ll 5) + (x \gg 5)$. Since the sign bit is zero for a positive value, no addition logic is needed for the bit replication in this case. For example, the value 470 = 00111010110b in the above example should be expanded to 0011101011001110b = 15054. A negative 11-bit value must first be made positive before bit replication, and then made negative again:

```
if (result11bit >= 0) {
  result16bit = (result11bit << 5) + (result11bit >> 5);
} else {
  result11bit = -result11bit;
  result16bit = (result11bit << 5) + (result11bit >> 5);
  result16bit = -result16bit;
}
```

Simply bit replicating a negative number without first making it positive will not give a correct result.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication according to the above should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that it is not possible to specify a base value of 1.0 or -1.0. The largest possible base_codeword is $+127$, which represents $\frac{127}{127.875} = 0.993\ldots$ However, it is still possible to reach a pixel value of 1.0 or -1.0, since the base value is modified by the table before the pixel value is calculated. Indeed, half of the modifiers will often produce a value of 1.0. As an example, assume the base_codeword is $+127$, the modifier table is [-3 -5 -7 -9 2 4 6 8] and the multiplier is one. Starting with Equation-signedr11eac-eqn-start,

$$base\_codeword \times \frac{1}{127.875} + modifier \times multiplier \times \frac{1}{127.875}$$

we get

$$\frac{127}{127.875} + \begin{bmatrix} -3 & -5 & -7 & -9 & 2 & 4 & 6 & 8 \end{bmatrix} \times \frac{1}{127.875}$$

which equals

$$\begin{bmatrix} 0.970 & 0.954 & 0.938 & 0.923 & 1.01 & 1.02 & 1.04 & 1.06 \end{bmatrix}$$

or after clamping

$$\begin{bmatrix} 0.970 & 0.954 & 0.938 & 0.923 & 1.00 & 1.00 & 1.00 & 1.00 \end{bmatrix}$$

This shows that it is indeed possible to arrive at the value 1.0. The same reasoning goes for -1.0.

Note also that Equations-signedr11eac-eqn-simple/signedr11eac-eqn-simpler are very similar to Equations-r11eac-eqn-simple/r11eac-eqn-simpler in the unsigned version EAC_R11. Apart from the $+4$, the clamping and the extension to bit sizes other than 11, the same decoding hardware can be shared between the two codecs.

Format Signed RG11 EAC

The number of bits to represent a 4×4 texel block is 128 bits if the format is signed RG11 EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

$$\text{int64bit0} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$
$$\text{int64bit1} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times p_0 + p_1) + p_2) + p_3) + p_4) + p_5) + p_6) + p_7$$

The 64-bit word int64bit0 contains information about the red component of a two-channel 4×4 pixel block as shown in Table 15.2, and the word int64bit1 contains information about the green component. Both 64-bit integers are decoded in the same way as signed R11 EAC described in Section 15.8.

Format RGB ETC2 with punchthrough alpha

For RGB ETC2 with punchthrough alpha, each 64-bit word contains information about a four-channel 4×4 pixel block as shown in Table 15.2.

The blocks are compressed using one of four different ‘modes’. Table 15.11 part (a) shows the bits used for determining the mode used in a given block.

To determine the mode, the three 5-bit values R, G and B, and the three 3-bit values dR, dG and dB are examined. R, G and B are treated as integers between 0 and 31 and dR, dG and dB as two’s-complement integers between -4 and $+3$. First, R and dR are added, and if the sum is not within the interval [0,31], the ‘T’ mode is selected. Otherwise, if the

a) location of bits for mode selection

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					dR			G					dG			B					dB			.....							Op	.

b) bit layout for bits 63 through 32 for 'differential' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
R					dR			G					dG			B					dB			table1			table2				Op	F_B

c) bit layout for bits 63 through 32 for 'T' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
...			R1a		.	R1b		G1					B1				R2					G2				B2				da		Op	db

d) bit layout for bits 63 through 32 for 'H' mode:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
.	R1				G1 a			...			G1 b	B1 a	.	B1 b			R2				G2				B2				da	Op	db

e) bit layout for bits 31 through 0 for 'diff', 'T' and 'H' modes

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p0	o0	n0	m0	l0	k0	j0	i0	h0	g0	f0	e0	d0	c0	b0	a0	p1	o1	n1	m1	l1	k1	j1	i1	h1	g1	f1	e1	d1	c1	b1	a1

f) bit layout for bits 63 through 0 for 'planar' mode:

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

Table 15.11: Texel Data format for punchthrough alpha ETC2 compressed texture formats

sum of G and dG is outside the interval [0,31], the 'H' mode is selected. Otherwise, if the sum of B and dB is outside of the interval [0,31], the 'planar' mode is selected. Finally, if all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

The layout of the bits used to decode the 'differential' mode is shown in Table 15.11 part (b). In this mode, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the 'flip bit'. If the 'flip bit' is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Table 15.4. If the 'flip bit' is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Table 15.5. For each subblock, a 'base color' is stored.

In the 'differential' mode, following the layout shown in Table 15.11 part (b), the base color for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if R = 28 = 11100 binary (11100b for short), the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five bit representation for the base color of subblock 2 is obtained by modifying the 5-bit codewords R, G and B by the codewords dR, dG and dB. Each of dR, dG and dB is a 3-bit two's-complement number that can hold values between -4 and $+3$. For instance, if R = 28 as above, and dR $=$ 100b $= -4$, then the five bit representation for the red color component is $28 + (-4) = 24 =$ 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, dG = 2, B = 3 and dB = 0, the base color of subblock 2 will be RGB = (198, 49, 24). In summary, the base colors for the subblocks in the differential mode are:

$$base\,col\,subblock1 = extend\_5to8bits(R,G,B)$$
$$base\,col\,subblock2 = extend\_5to8bits(R+dR,G+dG,B+dB)$$

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the 'differential' mode.

After obtaining the base color, a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39—37), and for subblock 2, table codeword 2 is used (bits 36—34), see Table 15.11 part (b). The table codeword is used to select one of eight modifier tables. If the 'opaque'-bit (bit 33) is set, Table 15.12 is used. If it is unset, Table 15.13 is

table codeword	modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table 15.12: ETC2 intensity modifier sets for the ‘differential’ if ‘opaque’ is set.

table codeword	modifier table			
0	-8	0	0	8
1	-17	0	0	17
2	-29	0	0	29
3	-42	0	0	42
4	-60	0	0	60
5	-80	0	0	80
6	-106	0	0	106
7	-183	0	0	183

Table 15.13: ETC2 intensity modifier sets for the ‘differential’ if ‘opaque’ is unset.

used. For instance, if the ‘opaque’-bit is 1 and the table code word is 010 binary = 2, then the modifier table [−29, −9, 9, 29] is selected for the corresponding sub-block. Note that the values in Table 15.12 and Table 15.13 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which modifier value to use from the modifier table using the two ‘pixel index’ bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Table 15.2) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table 15.11 part (e). Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of the ‘flipbit’.

If the ‘opaque’-bit (bit 33) is set, the pixel index bits are decoded using Table 15.14. If the ‘opaque’-bit is unset, Table 15.15 will be used instead. If, for instance, the ‘opaque’-bit is 1, and the pixel index bits are 01 binary = 1, and the modifier table [−29, −9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see Table 15.14). This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: (231 + 29, 8 + 29, 16 + 29) resulting in (260, 37, 45). These values are then clamped to [0, 255], resulting in the color (255, 37, 45).

Pixel index value		Resulting modifier value
msb	lsb	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

Table 15.14: ETC2 mapping from pixel index values to modifier values when ‘opaque’-bit is set.

The alpha component is decoded using the ‘opaque’-bit, which is positioned in bit 33 (see Table 15.11 part (b)). If the ‘opaque’-bit is set, alpha is always 255. However, if the ‘opaque’-bit is zero, the alpha-value depends on the pixel indices; if MSB==1 and LSB==0, the alpha value will be zero, otherwise it will be 255. Finally, if the alpha value equals 0, the red-, green- and blue components will also be zero.

Pixel index value		Resulting modifier value
msb	lsb	
1	1	-b (large negative value)
1	0	0 (zero)
0	0	0 (zero)
0	1	b (large positive value)

Table 15.15: ETC2 mapping from pixel index values to modifier values when 'opaque'-bit is unset.

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

Hence paint color 2 will equal RGBA = (0,0,0,0) if opaque == 0.

In the example above, assume that the 'opaque'-bit was instead 0. Then, since the MSB = 0 and LSB 1, alpha will be 255, and the final decoded RGBA-tuple will be (255, 37, 45, 255).

The 'T' and 'H' compression modes share some characteristics: both use two base colors stored using 4 bits per channel. These bits are not stored sequentially, but in the layout shown in Table 15.11 part (c) and Table 15.11 part (d). To clarify, in the 'T' mode, the two colors are constructed as follows:

$$base\,col\,1 = extend\_4to8bits(\,(R1a \ll 2)\,|\,R1b,\,G1,\,B1)$$
$$base\,col\,2 = extend\_4to8bits(R2,G2,B2)$$

In the 'H' mode, the two colors are constructed as follows:

$$base\,col\,1 = extend\_4to8bits(R1,\,(G1a \ll 1)\,|\,G1b,\,(B1a \ll 3)\,|\,B1b)$$
$$base\,col\,2 = extend\_4to8bits(R2,G2,B2)$$

The function extend_4to8bits() just replicates the four bits twice. This is equivalent to multiplying by 17. As an example, extend_4to8bits(1101b) equals 11011101b = 221.

Both the 'T' and 'H' modes have four 'paint colors' which are the colors that will be used in the decompressed block, but they are assigned in a different manner. In the 'T' mode, 'paint color 0' is simply the first base color, and 'paint color 2' is the second base color. To obtain the other 'paint colors', a 'distance' is first determined, which will be used to modify the luminance of one of the base colors. This is done by combining the values 'da' and 'db' shown in Table 15.11 part (c) by $(da \ll 1)|db$, and then using this value as an index into the small look-up table shown in Table 15.8. For example, if 'da' is 10 binary and 'db' is 1 binary, the index is 101 binary and the selected distance will be 32. 'Paint color 1' is then equal to the second base color with the 'distance' added to each channel, and 'paint color 3' is the second base color with the 'distance' subtracted. In summary, to determine the four 'paint colors' for a 'T' block:

$$paint\,color\,0 = base\,col\,1$$
$$paint\,color\,1 = base\,col\,2 + (d,d,d)$$
$$paint\,color\,2 = base\,col\,2$$
$$paint\,color\,3 = base\,col\,2 - (d,d,d)$$

In both cases, the value of each channel is clamped to within [0,255].

Just as for the differential mode, the RGB channels are set to zero if alpha is zero, and the alpha component is calculated the same way:

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

A 'distance' value is computed for the 'H' mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table 15.8, 'da' and 'db' shown in Table 15.11 part (d) are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (base col 1 value $\geq$ base col 2 value), the 'value' of a color for the comparison being equal to $(R \ll 16) + (G \ll 8) + B$. Once the 'distance' d has been determined for an 'H' block, the four 'paint colors' will be:

$$\begin{aligned} paint\,color\,0 &= base\,col\,1 + (d,d,d) \\ paint\,color\,1 &= base\,col\,1 - (d,d,d) \\ paint\,color\,2 &= base\,col\,2 + (d,d,d) \\ paint\,color\,3 &= base\,col\,2 - (d,d,d) \end{aligned}$$

Yet again, RGB is zeroed if alpha is 0 and the alpha component is determined the same way:

```
if (opaque == 0 && MSB == 1 && LSB == 0) {
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
} else {
  alpha = 255;
}
```

Hence paint color 2 will have R=G=B=alpha=0 if opaque == 0.

Again, all color components are clamped to within [0,255]. Finally, in both the 'T' and 'H' modes, every pixel is assigned one of the four 'paint colors' in the same way the four modifier values are distributed in 'individual' or 'differential' blocks. For example, to choose a paint color for pixel d, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned paint color 2.

The final mode possible in an RGB ETC2 with punchthrough alpha — compressed block is the 'planar' mode. In this mode, the 'opaque'-bit must be 1 (a valid encoder should not produce an 'opaque'-bit equal to 0 in the planar mode), but should the 'opaque'-bit anyway be 0 the decoder should treat it as if it were 1. In the 'planar' mode, three base colors are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three base colors are stored in RGB 676 format, and stored in the manner shown in Table 15.11 part (f). The three colors are there labelled 'O', 'H' and 'V', so that the three components of color 'V' are RV, GV and BV, for example. Some color channels are split into non-consecutive bit-ranges, for example BO is reconstructed using BO1 as the most significant bit, BO2 as the two following bits, and BO3 as the three least significant bits.

Once the bits for the base colors have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the base colors in other modes. For example, the 6-bit blue and red channels are extended by replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three base colors in RGB888 format, the color of each pixel can then be determined as:

$$
\begin{aligned}
R(x,y) &= \frac{x \times (RH - RO)}{4.0} + \frac{y \times (RV - RO)}{4.0} + RO \\
G(x,y) &= \frac{x \times (GH - GO)}{4.0} + \frac{y \times (GV - GO)}{4.0} + GO \\
B(x,y) &= \frac{x \times (BH - BO)}{4.0} + \frac{y \times (BV - BO)}{4.0} + BO \\
A(x,y) &= 255
\end{aligned}
$$

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Table 15.2 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

$$
\begin{aligned}
R(x,y) &= clamp255((x \times (RH - RO) + y \times (RV - RO) + 4 \times RO + 2) \gg 2) \\
G(x,y) &= clamp255((x \times (GH - GO) + y \times (GV - GO) + 4 \times GO + 2) \gg 2) \\
B(x,y) &= clamp255((x \times (BH - BO) + y \times (BV - BO) + 4 \times BO + 2) \gg 2) \\
A(x,y) &= 255
\end{aligned}
$$

where $clamp255$ clamps the value to a number in the range [0, 255].

Note that the alpha component is always 255 in the planar mode.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

Format RGB ETC2 with punchthrough alpha and sRGB encoding

Decompression of floating point sRGB values in RGB ETC2 with sRGB encoding and punchthrough alpha follows that of floating point RGB values of RGB ETC2 with punchthrough alpha. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, cs, to a linear component, cl, is according to the formula in Section 7.7.3. Assume cs is the sRGB component in the range [0,1]. Note that the alpha component is not gamma corrected, and hence does not use the above formula.

Chapter 16

ASTC Compressed Texture Image Formats

This description is derived from the Khronos OES_texture_compression_astc OpenGL extension.

What is ASTC?

ASTC stands for Adaptive Scalable Texture Compression. The ASTC formats form a family of related compressed texture image formats. They are all derived from a common set of definitions.

ASTC textures may be either 2D or 3D.

ASTC textures may be encoded using either high or low dynamic range. Low dynamic range images may optionally be specified using the sRGB transfer function for the RGB channels.

Two sub-profiles ("LDR Profile" and "HDR Profile") may be implemented, which support only 2D images at low or high dynamic range respectively.

ASTC textures may be encoded as 1, 2, 3 or 4 components, but they are all decoded into RGBA. ASTC has a variable block size.

Design Goals

The design goals for the format are as follows:

- Random access. This is a must for any texture compression format.
- Bit exact decode. This is a must for conformance testing and reproducibility.
- Suitable for mobile use. The format should be suitable for both desktop and mobile GPU environments. It should be low bandwidth and low in area.
- Flexible choice of bit rate. Current formats only offer a few bit rates, leaving content developers with only coarse control over the size/quality tradeoff.
- Scalable and long-lived. The format should support existing R, RG, RGB and RGBA image types, and also have high "headroom", allowing continuing use for several years and the ability to innovate in encoders. Part of this is the choice to include HDR and 3D.
- Feature orthogonality. The choices for the various features of the format are all orthogonal to each other. This has three effects: first, it allows a large, flexible configuration space; second, it makes that space easier to understand; and third, it makes verification easier.
- Best in class at given bit rate. It should beat or match the current best in class for peak signal-to-noise ratio (PSNR) at all bit rates.

- Fast decode. Texel throughput for a cached texture should be one texel decode per clock cycle per decoder. Parallel decoding of several texels from the same block should be possible at incremental cost.
- Low bandwidth. The encoding scheme should ensure that memory access is kept to a minimum, cache reuse is high and memory bandwidth for the format is low.
- Low area. It must occupy comparable die size to competing formats.

Basic Concepts

ASTC is a block-based lossy compression format. The compressed image is divided into a number of blocks of uniform size, which makes it possible to quickly determine which block a given texel resides in.

Each block has a fixed memory footprint of 128 bits, but these bits can represent varying numbers of texels (the block "footprint").

Note
The term "block footprint" in ASTC refers to the same concept as "compressed texel block dimensions" elsewhere in the Data Format Specification.

Block footprint sizes are not confined to powers-of-two, and are also not confined to be square. They may be 2D, in which case the block dimensions range from 4 to 12 texels, or 3D, in which case the block dimensions range from 3 to 6 texels.

Decoding one texel requires only the data from a single block. This simplifies cache design, reduces bandwidth and improves encoder throughput.

Block Encoding

To understand how the blocks are stored and decoded, it is useful to start with a simple example, and then introduce additional features.

The simplest block encoding starts by defining two color "endpoints". The endpoints define two colors, and a number of additional colors are generated by interpolating between them. We can define these colors using 1, 2, 3, or 4 components (usually corresponding to R, RG, RGB and RGBA textures), and using low or high dynamic range.

We then store a color interpolant weight for each texel in the image, which specifies how to calculate the color to use. From this, a weighted average of the two endpoint colors is used to generate the intermediate color, which is the returned color for this texel.

There are several different ways of specifying the endpoint colors, and the weights, but once they have been defined, calculation of the texel colors proceeds identically for all of them. Each block is free to choose whichever encoding scheme best represents its color endpoints, within the constraint that all the data fits within the 128 bit block.

For blocks which have a large number of texels (e.g. a 12×12 block), there is not enough space to explicitly store a weight for every texel. In this case, a sparser grid with fewer weights is stored, and interpolation is used to determine the effective weight to be used for each texel position. This allows very low bit rates to be used with acceptable quality. This can also be used to more efficiently encode blocks with low detail, or with strong vertical or horizontal features.

For blocks which have a mixture of disparate colors, a single line in the color space is not a good fit to the colors of the pixels in the original image. It is therefore possible to partition the texels into multiple sets, the pixels within each set having similar colors. For each of these "partitions", we specify separate endpoint pairs, and choose which pair of endpoints to use for a particular texel by looking up the partition index from a partitioning pattern table. In ASTC, this partition table is actually implemented as a function.

The endpoint encoding for each partition is independent.

For blocks which have uncorrelated channels — for example an image with a transparency mask, or an image used as a normal map — it may be necessary to specify two weights for each texel. Interpolation between the components of the

endpoint colors can then proceed independently for each "plane" of the image. The assignment of channels to planes is selectable.

Since each of the above options is independent, it is possible to specify any combination of channels, endpoint color encoding, weight encoding, interpolation, multiple partitions and single or dual planes.

Since these values are specified per block, it is important that they are represented with the minimum possible number of bits. As a result, these values are packed together in ways which can be difficult to read, but which are nevertheless highly amenable to hardware decode.

All of the values used as weights and color endpoint values can be specified with a variable number of bits. The encoding scheme used allows a fine-grained tradeoff between weight bits and color endpoint bits using "integer sequence encoding". This can pack adjacent values together, allowing us to use fractional numbers of bits per value.

Finally, a block may be just a single color. This is a so-called "void extent block" and has a special coding which also allows it to identify nearby regions of single color. This may be used to short-circuit fetching of what would be identical blocks, and further reduce memory bandwidth.

LDR and HDR Modes

The decoding process for LDR content can be simplified if it is known in advance that sRGB output is required. This selection is therefore included as part of the global configuration.

The two modes differ in various ways, as shown in Table 16.1.

Operation	**LDR Mode**	**HDR Mode**
Returned Value	Vector of FP16, or vector of 8-bit unsigned normalized values	Vector of FP16 values
sRGB compatible	Yes	No
LDR endpoint decoding precision	16 bits, or 8 bits for sRGB	16 bits
HDR endpoint mode results	Error color	As decoded
Error results	Error color	Vector of NaNs (0xFFFF)

Table 16.1: ASTC differences between LDR and HDR modes

The error color is opaque fully-saturated magenta $(R,G,B,A) = (0xFF, 0x00, 0xFF, 0xFF)$. This has been chosen as it is much more noticeable than black or white, and occurs far less often in valid images.

For linear RGB decode, the error color may be either opaque fully-saturated magenta $(R,G,B,A) = (1.0, 0.0, 1.0, 1.0)$ or a vector of four NaNs $(R,G,B,A) = (NaN, NaN, NaN, NaN)$. In the latter case, the recommended NaN value returned is $0xFFFF$.

The error color is returned as an informative response to invalid conditions, including invalid block encodings or use of reserved endpoint modes.

Future, forward-compatible extensions to ASTC may define valid interpretations of these conditions, which will decode to some other color. Therefore, encoders and applications must not rely on invalid encodings as a way of generating the error color.

Configuration Summary

The global configuration data for the format are as follows:

- Block dimension (2D or 3D)
- Block footprint size
- sRGB output enabled or not

The data specified per block are as follows:

- Texel weight grid size
- Texel weight range
- Texel weight values
- Number of partitions
- Partition pattern index
- Color endpoint modes (includes LDR or HDR selection)
- Color endpoint data
- Number of planes
- Plane-to-channel assignment

Decode Procedure

To decode one texel:

```
(Optimization: If within known void-extent, immediately return single color)

Find block containing texel
Read block mode
If void-extent block, store void extent and immediately return single color

For each plane in image
  If block mode requires infill
    Find and decode stored weights adjacent to texel, unquantize and interpolate
  Else
    Find and decode weight for texel, and unquantize

Read number of partitions
If number of partitions > 1
  Read partition table pattern index
  Look up partition number from pattern

Read color endpoint mode and endpoint data for selected partition
Unquantize color endpoints
Interpolate color endpoints using weight (or weights in dual-plane mode)
Return interpolated color
```

Block Determination and Bit Rates

The block footprint is a global setting for any given texture, and is therefore not encoded in the individual blocks.

For 2D textures, the block footprint's width and height are selectable from a number of predefined sizes, namely 4, 5, 6, 8, 10 and 12 pixels.

For square and nearly-square blocks, this gives the bit rates in Table 16.2.

The "Increment" column indicates the ratio of bit rate against the next lower available rate. A consistent value in this column indicates an even spread of bit rates.

For 3D textures, the block footprint's width, height and depth are selectable from a number of predefined sizes, namely 3, 4, 5, and 6 pixels.

Footprint		Bit Rate	Increment
Width	Height		
4	4	8.00	125%
5	4	6.40	125%
5	5	5.12	120%
6	5	4.27	120%
6	6	3.56	114%
8	5	3.20	120%
8	6	2.67	105%
10	5	2.56	120%
10	6	2.13	107%
8	8	2.00	125%
10	8	1.60	125%
10	10	1.28	120%
12	10	1.07	120%
12	12	0.89	

Table 16.2: ASTC 2D footprint and bit rates

Block Footprint			Bit Rate	Increment
Width	Height	Depth		
3	3	3	4.74	133%
4	3	3	3.56	133%
4	4	3	2.67	133%
4	4	4	2.00	125%
5	4	4	1.60	125%
5	5	4	1.28	125%
5	5	5	1.02	120%
6	5	5	0.85	120%
6	6	5	0.71	120%
6	6	6	0.59	

Table 16.3: ASTC 3D footprint and bit rates

For cubic and near-cubic blocks, this gives the bit rates in Table 16.3.

The full profile supports only those block footprints listed in Table 16.2 and Table 16.3. Other block sizes are not supported.

For images which are not an integer multiple of the block size, additional texels are added to the edges with maximum X and Y (and Z for 3D textures). These texels may be any color, as they will not be accessed.

Although these are not all powers of two, it is possible to calculate block addresses and pixel addresses within the block, for legal image sizes, without undue complexity.

Given an image which is $W \times H \times D$ pixels in size, with block size $w \times h \times d$, the size of the image in blocks is:

$$B_w = \left\lceil \frac{W}{w} \right\rceil$$
$$B_h = \left\lceil \frac{H}{h} \right\rceil$$
$$B_d = \left\lceil \frac{D}{d} \right\rceil$$

For a 3D image built from 2D slices, each 2D slice is a single texel thick, so that for an image which is $W \times H \times D$ pixels in size, with block size $w \times h$, the size of the image in blocks is:

$$B_w = \left\lceil \frac{W}{w} \right\rceil$$
$$B_h = \left\lceil \frac{H}{h} \right\rceil$$
$$B_d = D$$

Block Layout

Each block in the image is stored as a single 128-bit block in memory. These blocks are laid out in raster order, starting with the block at (0,0,0), then ordered sequentially by X, Y and finally Z (if present). They are aligned to 128-bit boundaries in memory.

The bits in the block are labeled in little-endian order — the byte at the lowest address contains bits 0..7. Bit 0 is the least significant bit in the byte.

Each block has the same basic layout, shown in Table 16.4.

Since the size of the “texel weight data” field is variable, the positions shown for the “more config data” field and “color endpoint data” field are only representative and not fixed.

The “Block mode” field specifies how the Texel Weight Data is encoded.

The “Part” field specifies the number of partitions, minus one. If dual plane mode is enabled, the number of partitions must be 3 or fewer. If 4 partitions are specified, the error value is returned for all texels in the block.

The size and layout of the extra configuration data depends on the number of partitions, and the number of planes in the image, as shown in Table 16.5 (only the bottom 32 bits are shown).

CEM is the color endpoint mode field, which determines how the Color Endpoint Data is encoded.

If dual-plane mode is active, the color component selector bits appear directly below the weight bits, as shown in Table 16.6.

The Partition Index field specifies which partition layout to use. CEM is the first 6 bits of color endpoint mode information for the various partitions. For modes which require more than 6 bits of CEM data, the additional bits appear at a variable position directly beneath the texel weight data.

If dual-plane mode is active, the color component selector bits then appear directly below the additional CEM bits.

The final special case is that if bits [8:0] of the block are “111111100”, then the block is a void-extent block, which has a separate encoding described in Section 16.23.

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

Table 16.4: ASTC block layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Color endpoint data															CEM
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CEM			0	0	Block mode										

Table 16.5: ASTC single-partition block layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			CEM						Partition index						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Partition index			Part		Block mode										

Table 16.6: ASTC multi-partition block layout

Block Mode

The Block Mode field specifies the width, height and depth of the grid of weights, what range of values they use, and whether dual weight planes are present. Since some these are not represented using powers of two (there are 12 possible weight widths, for example), and not all combinations are allowed, this is not a simple bit packing. However, it can be unpacked quickly in hardware.

The weight ranges are encoded using a 3 bit value R, which is interpreted together with a precision bit H, as shown in Table 16.7.

R	Low Precision Range (H=0)				High Precision Range (H=1)			
	Weight Range	Trits	Quints	Bits	Weight Range	Trits	Quints	Bits
000	Invalid				Invalid			
001	Invalid				Invalid			
010	0..1			1	0..9		1	1
011	0..2	1			0..11	1		2
100	0..3			2	0..15			4
101	0..4		1		0..19		1	2
110	0..5	1		1	0..23	1		3
111	0..7			3	0..31			5

Table 16.7: ASTC weight range encodings

Each weight value is encoded using the specified number of Trits, Quints and Bits. The details of this encoding can be found in Section 16.12.

For 2D blocks, the Block Mode field is laid out as shown in Table 16.8.

10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Width**	**Height**	**Notes**
D	H	B		A		R_0	0	0	R_2	R_1	B+4	A+2	
D	H	B		A		R_0	0	1	R_2	R_1	B+8	A+2	
D	H	B		A		R_0	1	0	R_2	R_1	A+2	B+8	
D	H	0	B	A		R_0	1	1	R_2	R_1	A+2	B+6	
D	H	1	B	A		R_0	1	1	R_2	R_1	B+2	A+2	
D	H	0	0	A		R_0	R_2	R_1	0	0	12	A+2	
D	H	0	1	A		R_0	R_2	R_1	0	0	A+2	12	
D	H	1	1	0	0	R_0	R_2	R_1	0	0	6	10	
D	H	1	1	0	1	R_0	R_2	R_1	0	0	10	6	
B		1	0	A		R_0	R_2	R_1	0	0	A+6	B+6	D=0, H=0
x	x	1	1	1	1	1	1	1	0	0	-	-	Void-extent
x	x	1	1	1	x	x	x	x	0	0	-	-	Reserved*
x	x	x	x	x	x	x	0	0	0	0	-	-	Reserved

Table 16.8: ASTC 2D block mode layout

Note that, due to the encoding of the R field, as described in the previous page, bits R2 and R1 cannot both be zero, which disambiguates the first five rows from the rest of the table.

Bit positions with a value of x are ignored for purposes of determining if a block is a void-extent block or reserved, but may have defined encodings for specific void-extent blocks.

The penultimate row of the table is reserved only if bits [5:2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

For 3D blocks, the Block Mode field is laid out as shown in Table 16.9.

The D bit is set to indicate dual-plane mode. In this mode, the maximum allowed number of partitions is 3.

10	9	8	7	6	5	4	3	2	1	0	Width	Height	Depth	Notes
D	H	B		A		R_0	C		R_2	R_1	A+2	B+2	C+2	
B		0	0	A		R_0	R_2	R_1	0	0	6	B+2	A+2	D=0, H=0
B		0	1	A		R_0	R_2	R_1	0	0	A+2	6	B+2	D=0, H=0
B		1	0	A		R_0	R_2	R_1	0	0	A+2	B+2	6	D=0, H=0
D	H	1	1	0	0	R_0	R_2	R_1	0	0	6	2	2	
D	H	1	1	0	1	R_0	R_2	R_1	0	0	2	6	2	
D	H	1	1	1	0	R_0	R_2	R_1	0	0	2	2	6	
x	x	1	1	1	1	1	1	1	0	0	-	-	-	Void-extent
x	x	1	1	1	1	x	x	x	0	0	-	-	-	Reserved*
x	x	x	x	x	x	x	0	0	0	0	-	-	-	Reserved

Table 16.9: ASTC 3D block mode layout

The penultimate row of the table is reserved only if bits [4:2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

The size of the grid in each dimension must be less than or equal to the corresponding dimension of the block footprint. If the grid size is greater than the footprint dimension in any axis, then this is an illegal block encoding and all texels will decode to the error color.

Color Endpoint Mode

In single-partition mode, the Color Endpoint Mode (CEM) field stores one of 16 possible values. Each of these specifies how many raw data values are encoded, and how to convert these raw values into two RGBA color endpoints. They can be summarized as shown in Table 16.10.

CEM	Description	Class
0	LDR Luminance, direct	0
1	LDR Luminance, base+offset	0
2	HDR Luminance, large range	0
3	HDR Luminance, small range	0
4	LDR Luminance+Alpha, direct	1
5	LDR Luminance+Alpha, base+offset	1
6	LDR RGB, base+scale	1
7	HDR RGB, base+scale	1
8	LDR RGB, direct	2
9	LDR RGB, base+offset	2
10	LDR RGB, base+scale plus two A	2
11	HDR RGB, direct	2
12	LDR RGBA, direct	3
13	LDR RGBA, base+offset	3
14	HDR RGB, direct + LDR Alpha	3
15	HDR RGB, direct + HDR Alpha	3

Table 16.10: ASTC color endpoint modes

In multi-partition mode, the CEM field is of variable width, from 6 to 14 bits. The lowest 2 bits of the CEM field specify how the endpoint mode for each partition is calculated as shown in Table 16.11.

Value	Meaning
00	All color endpoint pairs are of the same type. A full 4-bit CEM is stored in block bits [28:25] and is used for all partitions.
01	All endpoint pairs are of class 0 or 1.
10	All endpoint pairs are of class 1 or 2.
11	All endpoint pairs are of class 2 or 3.

Table 16.11: ASTC Multi-Partition Color Endpoint Modes

If the CEM selector value in bits [24:23] is not 00, then data layout is as shown in Table 16.12 and Table 16.13.

Part			n	m	l	k	j	i	h	g	
2	...	Weight	M1								...
3	...	Weight	M2		M1		M0				...
4	...	Weight	M3		M2		M1		M0		...

Table 16.12: ASTC multi-partition color endpoint mode layout

Part	28	27	26	25	24	23
2	M0		C1	C0	CEM	
3	M9	C2	C1	C0	CEM	
4	C3	C2	C1	C0	CEM	

Table 16.13: ASTC multi-partition color endpoint mode layout (2)

In this view, each partition *i* has two fields. C_i is the class selector bit, choosing between the two possible CEM classes (0 indicates the lower of the two classes), and M_i is a two-bit field specifying the low bits of the color endpoint mode within that class. The additional bits appear at a variable bit position, immediately below the texel weight data.

The ranges used for the data values are not explicitly specified. Instead, they are derived from the number of available bits remaining after the configuration data and weight data have been specified.

Details of the decoding procedure for Color Endpoints can be found in Section 16.13.

Integer Sequence Encoding

Both the weight data and the endpoint color data are variable width, and are specified using a sequence of integer values. The range of each value in a sequence (e.g. a color weight) is constrained.

Since it is often the case that the most efficient range for these values is not a power of two, each value sequence is encoded using a technique known as “integer sequence encoding”. This allows efficient, hardware-friendly packing and unpacking of values with non-power-of-two ranges.

In a sequence, each value has an identical range. The range is specified in one of the forms shown in Table 16.14 and Table 16.15.

Since 3^5 is 243, it is possible to pack five trits into 8 bits(which has 256 possible values), so a trit can effectively be encoded as 1.6 bits. Similarly, since 5^3 is 125, it is possible to pack three quints into 7 bits (which has 128 possible values), so a quint can be encoded as 2.33 bits.

The encoding scheme packs the trits or quints, and then interleaves the n additional bits in positions that satisfy the requirements of an arbitrary length stream. This makes it possible to correctly specify lists of values whose length is not an integer multiple of 3 or 5 values. It also makes it possible to easily select a value at random within the stream.

Value range	MSB encoding	LSB encoding
$0 \ldots 2^n - 1$	-	n bit value m ($n \leq 8$)
$0 \ldots (3 \times 2^n) - 1$	Base-3 "trit" value t	n bit value m ($n \leq 6$)
$0 \ldots (5 \times 2^n) - 1$	Base-5 "quint" value q	n bit value m ($n \leq 5$)

Table 16.14: ASTC range forms

Value range	Value	Block	Packed block size
$0 \ldots 2^n - 1$	m	1	n
$0 \ldots (3 \times 2^n) - 1$	$t \times 2^n + m$	5	$8 + 5 \times n$
$0 \ldots (5 \times 2^n) - 1$	$q \times 2^n + m$	3	$7 + 3 \times n$

Table 16.15: ASTC encoding for different ranges

If there are insufficient bits in the stream to fill the final block, then unused (higher order) bits are assumed to be 0 when decoding.

To decode the bits for value number i in a sequence of bits b, both indexed from 0, perform the following:

If the range is encoded as n bits per value, then the value is bits $b[i \times n + n - 1 : i \times n]$ — a simple multiplexing operation.

If the range is encoded using a trit, then each block contains 5 values (v0 to v4), each of which contains a trit (t0 to t4) and a corresponding LSB value (m0 to m4). The first bit of the packed block is bit $\lfloor \frac{i}{5} \rfloor \times (8 + 5 \times n)$. The bits in the block are packed as shown in Table 16.16 (in this example, n is 4).

				27	26	25	24	23	22	21	20	19	18	17	16
				T7	m4				T6	T5	m3				T4
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m2				T3	T2	m1				T1	T0	m0			

Table 16.16: ASTC trit-based packing

The five trits t0 to t4 are obtained by bit manipulations of the 8 bits T[7:0] as follows:

```
if T[4:2] = 111
    C = { T[7:5], T[1:0] }; t4 = t3 = 2
else
    C = T[4:0]
    if T[6:5] = 11
        t4 = 2; t3 = T[7]
    else
        t4 = T[7]; t3 = T[6:5]

if C[1:0] = 11
    t2 = 2; t1 = C[4]; t0 = { C[3], C[2]&~C[3] }
else if C[3:2] = 11
    t2 = 2; t1 = 2; t0 = C[1:0]
else
    t2 = C[4]; t1 = C[3:2]; t0 = { C[1], C[0]&~C[1] }
```

If the range is encoded using a quint, then each block contains 3 values (v0 to v2), each of which contains a quint (q0 to q2) and a corresponding LSB value (m0 to m2). The first bit of the packed block is bit $\lfloor \frac{i}{3} \rfloor \times (7+3\times n)$.

The bits in the block are packed as described in Table 16.17 and Table 16.18 (in this example, n is 4).

18	17	16
Q6	Q5	m2

Table 16.17: ASTC quint-based packing

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m2			Q4	Q3	m1				Q2	Q1	Q0	m0			

Table 16.18: ASTC quint-based packing (2)

The three quints q0 to q2 are obtained by bit manipulations of the 7 bits Q[6:0] as follows:

```
if Q[2:1] = 11 and Q[6:5] = 00
    q2 = { Q[0], Q[4]&~Q[0], Q[3]&~Q[0] }; q1 = q0 = 4
else
    if Q[2:1] = 11
        q2 = 4; C = { Q[4:3], ~Q[6:5], Q[0] }
    else
        q2 = Q[6:5]; C = Q[4:0]

    if C[2:0] = 101
        q1 = 4; q0 = C[4:3]
    else
        q1 = C[4:3]; q0 = C[2:0]
```

Both these procedures ensure a valid decoding for all 128 possible values (even though a few are duplicates). They can also be implemented efficiently in software using small tables.

Encoding methods are not specified here, although table-based mechanisms work well.

Endpoint Unquantization

Each color endpoint is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding, as a stream of bits stored from just above the configuration data, and growing upwards.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..255.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers. This procedure ensures correct scaling, but scrambles the order of the decoded values relative to the encoded values. This must be compensated for using a table in the encoder.

The initial inputs to the procedure are denoted A (9 bits), B (9 bits), C (9 bits) and D (3 bits) and are decoded using the range as described in Table 16.19.

These are then processed as follows:

```
T = D * C + B;
T = T ^ A;
T = (A & 0x80) | (T >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

Range	T	Q	B	Bits	A	B	C	D
0..5	1		1	a	aaaaaaaaa	000000000	204	Trit value
0..9		1	1	a	aaaaaaaaa	000000000	113	Quint value
0..11	1		2	ba	aaaaaaaaa	b000b0bb0	93	Trit value
0..19		1	2	ba	aaaaaaaaa	b0000bb00	54	Quint value
0..23	1		3	cba	aaaaaaaaa	cb000cbcb	44	Trit value
0..39		1	3	cba	aaaaaaaaa	cb0000cbc	26	Quint value
0..47	1		4	dcba	aaaaaaaaa	dcb000dcb	22	Trit value
0..79		1	4	dcba	aaaaaaaaa	dcb0000dc	13	Quint value
0..95	1		5	edcba	aaaaaaaaa	edcb000ed	11	Trit value
0..159		1	5	edcba	aaaaaaaaa	edcb0000e	6	Quint value
0..191	1		6	fedcba	aaaaaaaaa	fedcb000f	5	Trit value

Table 16.19: ASTC color unquantization parameters

LDR Endpoint Decoding

The decoding method used depends on the Color Endpoint Mode (CEM) field, which specifies how many values are used to represent the endpoint.

The CEM field also specifies how to take the *n* unquantized color endpoint values v_0 to v_{n-1} and convert them into two RGBA color endpoints e_0 and e_1.

The HDR Modes are more complex and do not fit neatly into this section. They are documented in following section.

The methods can be summarized as shown in Table 16.20.

CEM	Range	Description	n
0	LDR	Luminance, direct	2
1	LDR	Luminance, base+offset	2
2	HDR	Luminance, large range	2
3	HDR	Luminance, small range	2
4	LDR	Luminance+Alpha, direct	4
5	LDR	Luminance+Alpha, base+offset	4
6	LDR	RGB, base+scale	4
7	HDR	RGB, base+scale	4
8	LDR	RGB, direct	6
9	LDR	RGB, base+offset	6
10	LDR	RGB, base+scale plus two A	6
11	HDR	RGB	6
12	LDR	RGBA, direct	8
13	LDR	RGBA, base+offset	8
14	HDR	RGB + LDR Alpha	8
15	HDR	RGB + HDR Alpha	8

Table 16.20: ASTC LDR color endpoint modes

Mode 14 is special in that the alpha values are interpolated linearly, but the color components are interpolated logarithmically. This is the only endpoint format with mixed-mode operation, and will return the error value if encountered in LDR mode.

Decode the different LDR endpoint modes as follows:

Mode 0 LDR Luminance, direct

```
e0=(v0,v0,v0,0xFF); e1=(v1,v1,v1,0xFF);
```

Mode 1 LDR Luminance, base+offset

```
L0 = (v0>>2)|(v1&0xC0); L1=L0+(v1&0x3F);
if (L1>0xFF) { L1=0xFF; }
e0=(L0,L0,L0,0xFF); e1=(L1,L1,L1,0xFF);
```

Mode 4 LDR Luminance+Alpha,direct

```
e0=(v0,v0,v0,v2);
e1=(v1,v1,v1,v3);
```

Mode 5 LDR Luminance+Alpha, base+offset

```
bit_transfer_signed(v1,v0); bit_transfer_signed(v3,v2);
e0=(v0,v0,v0,v2); e1=(v0+v1,v0+v1,v0+v1,v2+v3);
clamp_unorm8(e0); clamp_unorm8(e1);
```

Mode 6 LDR RGB, base+scale

```
e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, 0xFF);
e1=(v0,v1,v2,0xFF);
```

Mode 8 LDR RGB, Direct

```
s0= v0+v2+v4; s1= v1+v3+v5;
if (s1>=s0){e0=(v0,v2,v4,0xFF);
            e1=(v1,v3,v5,0xFF); }
else { e0=blue_contract(v1,v3,v5,0xFF);
       e1=blue_contract(v0,v2,v4,0xFF); }
```

Mode 9 LDR RGB, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
if(v1+v3+v5 >= 0)
{ e0=(v0,v2,v4,0xFF); e1=(v0+v1,v2+v3,v4+v5,0xFF); }
else
{ e0=blue_contract(v0+v1,v2+v3,v4+v5,0xFF);
  e1=blue_contract(v0,v2,v4,0xFF); }
clamp_unorm8(e0); clamp_unorm8(e1);
```

Mode 10 LDR RGB, base+scale plus two A

```
e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, v4);
e1=(v0,v1,v2, v5);
```

Mode 12 LDR RGBA, direct

```
s0= v0+v2+v4; s1= v1+v3+v5;
if (s1>=s0){e0=(v0,v2,v4,v6);
            e1=(v1,v3,v5,v7); }
else { e0=blue_contract(v1,v3,v5,v7);
       e1=blue_contract(v0,v2,v4,v6); }
```

Mode 13 LDR RGBA, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
bit_transfer_signed(v7,v6);
if(v1+v3+v5>=0) { e0=(v0,v2,v4,v6);
       e1=(v0+v1,v2+v3,v4+v5,v6+v7); }
else { e0=blue_contract(v0+v1,v2+v3,v4+v5,v6+v7);
       e1=blue_contract(v0,v2,v4,v6); }
clamp_unorm8(e0); clamp_unorm8(e1);
```

The bit_transfer_signed procedure transfers a bit from one value (a) to another (b). Initially, both *a* and *b* are in the range 0..255. After calling this procedure, *a*'s range becomes -32..31, and *b* remains in the range 0..255. Note that, as is often the case, this is easier to express in hardware than in C:

```
bit_transfer_signed(int& a, int& b)
{
    b >>= 1;
    b |= a & 0x80;
    a >>= 1;
    a &= 0x3F;
    if( (a&0x20)!=0 ) a-=0x40;
}
```

The blue_contract procedure is used to give additional precision to RGB colors near grey:

```
color blue_contract( int r, int g, int b, int a )
{
    color c;
    c.r = (r+b) >> 1;
    c.g = (g+b) >> 1;
    c.b = b;
    c.a = a;
    return c;
}
```

The clamp_unorm8 procedure is used to clamp a color into the UNORM8 range:

```
void clamp_unorm8(color c)
{
    if(c.r < 0) {c.r=0;} else if(c.r > 255) {c.r=255;}
    if(c.g < 0) {c.g=0;} else if(c.g > 255) {c.g=255;}
    if(c.b < 0) {c.b=0;} else if(c.b > 255) {c.b=255;}
    if(c.a < 0) {c.a=0;} else if(c.a > 255) {c.a=255;}
}
```

HDR Endpoint Decoding

For HDR endpoint modes, color values are represented in a 12-bit pseudo-logarithmic representation.

HDR Endpoint Mode 2

Mode 2 represents luminance-only data with a large range. It encodes using two values (v0, v1). The complete decoding procedure is as follows:

```
if(v1 >= v0)
{
    y0 = (v0 << 4);
    y1 = (v1 << 4);
}
else
{
    y0 = (v1 << 4) + 8;
    y1 = (v0 << 4) - 8;
}
// Construct RGBA result (0x780 is 1.0f)
e0 = (y0, y0, y0, 0x780);
e1 = (y1, y1, y1, 0x780);
```

HDR Endpoint Mode 3

Mode 3 represents luminance-only data with a small range. It packs the bits for a base luminance value, together with an offset, into two values (v0, v1), according to Table 16.21.

Value	7	6	5	4	3	2	1	0
v0	M	L[6:0]						
v1	X[3:0]				d[3:0]			

Table 16.21: ASTC HDR mode 3 value layout

The bit field marked as X allocates different bits to L or d depending on the value of the mode bit M.

The complete decoding procedure is as follows:

```
// Check mode bit and extract.
if((v0&0x80) !=0)
{
    y0 = ((v1 & 0xE0) << 4) | ((v0 & 0x7F) << 2);
    d  =  (v1 & 0x1F) << 2;
}
else
{
    y0 = ((v1 & 0xF0) << 4) | ((v0 & 0x7F) << 1);
    d  =  (v1 & 0x0F) << 1;
}

// Add delta and clamp
y1 = y0 + d;
if(y1 > 0xFFF) { y1 = 0xFFF; }

// Construct RGBA result (0x780 is 1.0f)
e0 = (y0, y0, y0, 0x780);
e1 = (y1, y1, y1, 0x780);
```

HDR Endpoint Mode 7

Mode 7 packs the bits for a base RGB value, a scale factor, and some mode bits into the four values (v0, v1, v2, v3), as shown in Table 16.22.

Value	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
v0	M[3:2]		R[5:0]					
v1	M1	X0	X1	G[4:0]				
v2	M0	X2	X3	B[4:0]				
v3	X4	X5	X6	S[4:0]				

Table 16.22: ASTC HDR mode 7 value layout

The mode bits M0 to M3 are a packed representation of an endpoint bit mode, together with the major component index. For modes 0 to 4, the component (red, green, or blue) with the largest magnitude is identified, and the values swizzled to ensure that it is decoded from the red channel.

The endpoint bit mode is used to determine the number of bits assigned to each component of the endpoint, and the destination of each of the extra bits X0 to X6, as shown in Table 16.23.

	Number of bits					Destination of extra bits						
Mode	R	G	B	S		X0	X1	X2	X3	X4	X5	X6
0	11	5	5	7		R[9]	R[8]	R[7]	R[10]	R[6]	S[6]	S[5]
1	11	6	6	5		R[8]	G[5]	R[7]	B[5]	R[6]	R[10]	R[9]
2	10	5	5	8		R[9]	R[8]	R[7]	R[6]	S[7]	S[6]	S[5]
3	9	6	6	7		R[8]	G[5]	R[7]	B[5]	R[6]	S[6]	S[5]
4	8	7	7	6		G[6]	G[5]	B[6]	B[5]	R[6]	R[7]	S[5]
5	7	7	7	7		G[6]	G[5]	B[6]	B[5]	R[6]	S[6]	S[5]

Table 16.23: ASTC HDR mode 7 endpoint bit mode

As noted before, this appears complex when expressed in C, but much easier to achieve in hardware: bit masking, extraction, shifting and assignment usually ends up as a single wire or multiplexer.

The complete decoding procedure is as follows:

```
// Extract mode bits and unpack to major component and mode.
int majcomp; int mode; int modeval = ((v0&0xC0)>>6) | ((v1&0x80)>>5) | ((v2&0x80)>>4);

if( (modeval & 0xC ) != 0xC ) {
    majcomp = modeval >> 2; mode = modeval & 3;
} else if( modeval != 0xF ) {
    majcomp = modeval & 3;  mode = 4;
} else {
    majcomp = 0; mode = 5;
}

// Extract low-order bits of r, g, b, and s.
int red   = v0 & 0x3f; int green = v1 & 0x1f;
int blue  = v2 & 0x1f; int scale = v3 & 0x1f;

// Extract high-order bits, which may be assigned depending on mode
int x0 = (v1 >> 6) & 1; int x1 = (v1 >> 5) & 1; int x2 = (v2 >> 6) & 1;
int x3 = (v2 >> 5) & 1; int x4 = (v3 >> 7) & 1; int x5 = (v3 >> 6) & 1;
int x6 = (v3 >> 5) & 1;

// Now move the high-order xs into the right place.
int ohm = 1 << mode;
if( ohm & 0x30 ) green |= x0 << 6;
if( ohm & 0x3A ) green |= x1 << 5;
if( ohm & 0x30 ) blue |= x2 << 6;
if( ohm & 0x3A ) blue |= x3 << 5;
if( ohm & 0x3D ) scale |= x6 << 5;
if( ohm & 0x2D ) scale |= x5 << 6;
if( ohm & 0x04 ) scale |= x4 << 7;
if( ohm & 0x3B ) red |= x4 << 6;
if( ohm & 0x04 ) red |= x3 << 6;
if( ohm & 0x10 ) red |= x5 << 7;
if( ohm & 0x0F ) red |= x2 << 7;
if( ohm & 0x05 ) red |= x1 << 8;
if( ohm & 0x0A ) red |= x0 << 8;
if( ohm & 0x05 ) red |= x0 << 9;
if( ohm & 0x02 ) red |= x6 << 9;
if( ohm & 0x01 ) red |= x3 << 10;
if( ohm & 0x02 ) red |= x5 << 10;

// Shift the bits to the top of the 12-bit result.
static const int shamts[6] = { 1,1,2,3,4,5 };
int shamt = shamts[mode];
red <<= shamt; green <<= shamt; blue <<= shamt; scale <<= shamt;

// Minor components are stored as differences
if( mode != 5 ) { green = red - green; blue = red - blue; }

// Swizzle major component into place
if( majcomp == 1 ) swap( red, green );
if( majcomp == 2 ) swap( red, blue );

// Clamp output values, set alpha to 1.0
e1.r = clamp( red, 0, 0xFFF );
e1.g = clamp( green, 0, 0xFFF );
e1.b = clamp( blue, 0, 0xFFF );
e1.alpha = 0x780;
e0.r = clamp( red - scale, 0, 0xFFF );
e0.g = clamp( green - scale, 0, 0xFFF );
e0.b = clamp( blue - scale, 0, 0xFFF );
e0.alpha = 0x780;
```

HDR Endpoint Mode 11

Mode 11 specifies two RGB values, which it calculates from a number of bitfields (a, b0, b1, c, d0 and d1) which are packed together with some mode bits into the six values (v0, v1, v2, v3, v4, v5) as shown in Table 16.24.

Value	7	6	5	4	3	2	1	0
v0	a[7:0]							
v1	m0	a8	c[5:0]					
v2	m1	X0	b0[5:0]					
v3	m2	X1	b1[5:0]					
v4	mj0	X2	X4	d0[4:0]				
v5	mj1	X3	X5	d1[4:0]				

Table 16.24: ASTC HDR mode 11 value layout

If the major component bits mj[1:0] are both 1, then the RGB values are specified directly by Table 16.25.

Value	7	6	5	4	3	2	1	0
v0	R0[11:4]							
v1	R1[11:4]							
v2	G0[11:4]							
v3	G1[11:4]							
v4	1	B0[11:5]						
v5	1	B1[11:5]						

Table 16.25: ASTC HDR mode 11 direct value layout

The mode bits m[2:0] specify the bit allocation for the different values, and the destinations of the extra bits X0 to X5 as shown in Table 16.26.

	Number of bits					Destination of extra bits					
Mode	a	b	c	d		X0	X1	X2	X3	X4	X5
0	9	7	6	7		b0[6]	b1[6]	d0[6]	d1[6]	d0[5]	d1[5]
1	9	8	6	6		b0[6]	b1[6]	b0[7]	b1[7]	d0[5]	d1[5]
2	10	6	7	7		a[9]	c[6]	d0[6]	d1[6]	d0[5]	d1[5]
3	10	7	7	6		b0[6]	b1[6]	a[9]	c[6]	d0[5]	d1[5]
4	11	8	6	5		b0[6]	b1[6]	b0[7]	b1[7]	a[9]	a[10]
5	11	6	7	6		a[9]	a[10]	c[7]	c[6]	d0[5]	d1[5]
6	12	7	7	5		b0[6]	b1[6]	a[11]	c[6]	a[9]	a[10]
7	12	6	7	6		a[9]	a[10]	a[11]	c[6]	d0[5]	d1[5]

Table 16.26: ASTC HDR mode 11 endpoint bit mode

The complete decoding procedure is as follows:

```
// Find major component
int majcomp = ((v4 & 0x80) >> 7) | ((v5 & 0x80) >> 6);

// Deal with simple case first
if( majcomp == 3 ) {
    e0 = (v0 << 4, v2 << 4, (v4 & 0x7f) << 5, 0x780);
    e1 = (v1 << 4, v3 << 4, (v5 & 0x7f) << 5, 0x780);
    return;
}

// Decode mode, parameters.
int mode = ((v1&0x80)>>7) | ((v2&0x80)>>6) | ((v3&0x80)>>5);
int va  = v0 | ((v1 & 0x40) << 2);
int vb0 = v2 & 0x3f; int vb1 = v3 & 0x3f;
int vc  = v1 & 0x3f;
int vd0 = v4 & 0x7f; int vd1 = v5 & 0x7f;

// Assign top bits of vd0, vd1.
static const int dbitstab[8] = {7,6,7,6,5,6,5,6};
vd0 = signextend( vd0, dbitstab[mode] );
vd1 = signextend( vd1, dbitstab[mode] );

// Extract and place extra bits
int x0 = (v2 >> 6) & 1;
int x1 = (v3 >> 6) & 1;
int x2 = (v4 >> 6) & 1;
int x3 = (v5 >> 6) & 1;
int x4 = (v4 >> 5) & 1;
int x5 = (v5 >> 5) & 1;

int ohm = 1 << mode;
if( ohm & 0xA4 ) va |= x0 << 9;
if( ohm & 0x08 ) va |= x2 << 9;
if( ohm & 0x50 ) va |= x4 << 9;
if( ohm & 0x50 ) va |= x5 << 10;
if( ohm & 0xA0 ) va |= x1 << 10;
if( ohm & 0xC0 ) va |= x2 << 11;
if( ohm & 0x04 ) vc |= x1 << 6;
if( ohm & 0xE8 ) vc |= x3 << 6;
if( ohm & 0x20 ) vc |= x2 << 7;
if( ohm & 0x5B ) vb0 |= x0 << 6;
if( ohm & 0x5B ) vb1 |= x1 << 6;
if( ohm & 0x12 ) vb0 |= x2 << 7;
if( ohm & 0x12 ) vb1 |= x3 << 7;

// Now shift up so that major component is at top of 12-bit value
int shamt = (modeval >> 1) ^ 3;
va <<= shamt; vb0 <<= shamt; vb1 <<= shamt;
vc <<= shamt; vd0 <<= shamt; vd1 <<= shamt;

e1.r = clamp( va, 0, 0xFFF );
e1.g = clamp( va - vb0, 0, 0xFFF );
e1.b = clamp( va - vb1, 0, 0xFFF );
e1.alpha = 0x780;
e0.r = clamp( va - vc, 0, 0xFFF );
e0.g = clamp( va - vb0 - vc - vd0, 0, 0xFFF );
e0.b = clamp( va - vb1 - vc - vd1, 0, 0xFFF );
e0.alpha = 0x780;

if( majcomp == 1 )      { swap( e0.r, e0.g ); swap( e1.r, e1.g ); }
else if( majcomp == 2 ) { swap( e0.r, e0.b ); swap( e1.r, e1.b ); }
```

HDR Endpoint Mode 14

Mode 14 specifies two RGBA values, using the eight values (v0, v1, v2, v3, v4, v5, v6, v7). First, the RGB values are decoded from (v0..v5) using the method from Mode 11, then the alpha values are filled in from v6 and v7:

```
// Decode RGB as for mode 11
(e0,e1) = decode_mode_11(v0,v1,v2,v3,v4,v5)

// Now fill in the alphas
e0.alpha = v6;
e1.alpha = v7;
```

Note that in this mode, the alpha values are interpreted (and interpolated) as 8-bit unsigned normalized values, as in the LDR modes. This is the only mode that exhibits this behaviour.

HDR Endpoint Mode 15

Mode 15 specifies two RGBA values, using the eight values (v0, v1, v2, v3, v4, v5, v6, v7). First, the RGB values are decoded from (v0..v5) using the method from Mode 11. The alpha values are stored in values v6 and v7 as a mode and two values which are interpreted according to the mode, as shown in Table 16.27.

Value	7	6	5	4	3	2	1	0
v6	M0	A[6:0]						
v7	M1	B[6:0]						

Table 16.27: ASTC HDR mode 15 alpha value layout

The alpha values are decoded from v6 and v7 as follows:

```
// Decode RGB as for mode 11
(e0,e1) = decode_mode_11(v0,v1,v2,v3,v4,v5)

// Extract mode bits
mode = ((v6 >> 7) & 1) | ((v7 >> 6) & 2);
v6 &= 0x7F;
v7 &= 0x7F;

if(mode==3)
{
    // Directly specify alphas
    e0.alpha = v6 << 5;
    e1.alpha = v7 << 5;
}
else
{
    // Transfer bits from v7 to v6 and sign extend v7.
    v6 |= (v7 << (mode+1))) & 0x780;
    v7 &= (0x3F >> mode);
    v7 ^= 0x20 >> mode;
    v7 -= 0x20 >> mode;
    v6 <<= (4-mode);
    v7 <<= (4-mode);

    // Add delta and clamp
    v7 += v6;
    v7 = clamp(v7, 0, 0xFFF);
    e0.alpha = v6;
    e1.alpha = v7;
}
```

Note that in this mode, the alpha values are interpreted (and interpolated) as 12-bit HDR values, and are interpolated as for any other HDR component.

Weight Decoding

The weight information is stored as a stream of bits, growing downwards from the most significant bit in the block. Bit n in the stream is thus bit 127-n in the block.

For each location in the weight grid, a value (in the specified range) is packed into the stream. These are ordered in a raster pattern starting from location (0,0,0), with the X dimension increasing fastest, and the Z dimension increasing slowest. If dual-plane mode is selected, both weights are emitted together for each location, plane 0 first, then plane 1.

Weight Unquantization

Each weight plane is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..64. The procedure for doing so is similar to the color endpoint unquantization.

First, we unquantize the actual stored weight values to the range 0..63.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers.

For representations with no additional bits, the results are as shown in Table 16.28.

Range	0	1	2	3	4
0..2	0	32	63	-	-
0..4	0	16	32	47	63

Table 16.28: ASTC weight unquantization values

For other values, we calculate the initial inputs to a bit manipulation procedure. These are denoted A (7 bits), B (7 bits), C (7 bits), and D (3 bits) and are decoded using the range as shown in Table 16.29.

Range	T	Q	B	Bits	A	B	C	D
0..5	1		1	a	aaaaaaa	0000000	50	Trit value
0..9		1	1	a	aaaaaaa	0000000	28	Quint value
0..11	1		2	ba	aaaaaaa	b000b0b	23	Trit value
0..19		1	2	ba	aaaaaaa	b0000b0	13	Quint value
0..23	1		3	cba	aaaaaaa	cb000cb	11	Trit value

Table 16.29: ASTC weight unquantization parameters

These are then processed as follows:

```
T = D * C + B;
T = T ^ A;
T = (A & 0x20) | (T >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

As a final step, for all types of value, the range is expanded from 0..63 up to 0..64 as follows:

```
if (T > 32) { T += 1; }
```

This allows the implementation to use 64 as a divisor during interpolation, which is much easier than using 63.

Weight Infill

After unquantization, the weights are subject to weight selection and infill. The infill method is used to calculate the weight for a texel position, based on the weights in the stored weight grid array (which may be a different size). The procedure below must be followed exactly, to ensure bit exact results.

The block size is specified as three dimensions along the s, t and r axes (B_s, B_t, B_r). Texel coordinates within the block (s,t,r) can have values from 0 to one less than the block dimension in that axis. For each block dimension, we compute scale factors (D_s, D_t, D_r):

$$D_s = \left\lfloor \frac{(1024 + \lfloor \frac{B_s}{2} \rfloor)}{(B_s - 1)} \right\rfloor$$

$$D_t = \left\lfloor \frac{(1024 + \lfloor \frac{B_t}{2} \rfloor)}{(B_t - 1)} \right\rfloor$$

$$D_r = \left\lfloor \frac{(1024 + \lfloor \frac{B_r}{2} \rfloor)}{(B_r - 1)} \right\rfloor$$

Since the block dimensions are constrained, these are easily looked up in a table. These scale factors are then used to scale the (s,t,r) coordinates to a homogeneous coordinate (c_s, c_t, c_r):

```
cs = Ds * s;
ct = Dt * t;
cr = Dr * r;
```

This homogeneous coordinate (c_s, c_t, c_r) is then scaled again to give a coordinate (g_s, g_t, g_r) in the weight-grid space. The weight-grid is of size (N, M, Q), as specified in the block mode field:

```
gs = (cs*(N-1)+32) >> 6;
gt = (ct*(M-1)+32) >> 6;
gr = (cr*(Q-1)+32) >> 6;
```

The resulting coordinates may be in the range 0..176. These are interpreted as 4:4 unsigned fixed point numbers in the range 0.0 .. 11.0.

If we label the integral parts of these (j_s, j_t, j_r) and the fractional parts (f_s, f_t, f_r), then:

```
js = gs >> 4; fs = gs & 0x0F;
jt = gt >> 4; ft = gt & 0x0F;
jr = gr >> 4; fr = gr & 0x0F;
```

These values are then used to interpolate between the stored weights. This process differs for 2D and 3D.

For 2D, bilinear interpolation is used:

```
v0 = js + jt*N;
p00 = decode_weight(v0);
p01 = decode_weight(v0 + 1);
p10 = decode_weight(v0 + N);
p11 = decode_weight(v0 + N + 1);
```

The function decode_weight(n) decodes the n^{th} weight in the stored weight stream. The values p00 to p11 are the weights at the corner of the square in which the texel position resides. These are then weighted using the fractional position to produce the effective weight i as follows:

```
w11 = (fs*ft+8) >> 4;
w10 = ft - w11;
w01 = fs - w11;
w00 = 16 - fs - ft + w11;
i = (p00*w00 + p01*w01 + p10*w10 + p11*w11 + 8) >> 4;
```

For 3D, simplex interpolation is used as it is cheaper than a naïve trilinear interpolation. First, we pick some parameters for the interpolation based on comparisons of the fractional parts of the texel position as shown in Table 16.30.

$f_s > f_t$	$f_t > f_r$	$f_s > f_r$	s_1	s_2	w_0	w_1	w_2	w_3
True	True	*True*	1	N	$16 - f_s$	$f_s - f_t$	$f_t - f_r$	f_r
False	*True*	True	N	1	$16 - f_t$	$f_t - f_s$	$f_s - f_r$	f_r
True	False	True	1	N × M	$16 - f_s$	$f_s - f_r$	$f_r - f_t$	f_t
True	*False*	False	N × M	1	$16 - f_r$	$f_r - f_s$	$f_s - f_t$	f_t
False	True	False	N	N × M	$16 - f_t$	$f_t - f_r$	$f_r - f_s$	f_s
False	False	*False*	N × M	N	$16 - f_r$	$f_r - f_t$	$f_t - f_s$	f_s

Table 16.30: ASTC simplex interpolation parameters

Italicised test results are implied by the others. The effective weight *i* is then calculated as:

```
v0 = js + jt*N + jr*N*M;
p0 = decode_index(v0);
p1 = decode_index(v0 + s1);
p2 = decode_index(v0 + s1 + s2);
p3 = decode_index(v0 + N*M + N + 1);
i = (p0*w0 + p1*w1 + p2*w2 + p3*w3 + 8) >> 4;
```

Weight Application

Once the effective weight *i* for the texel has been calculated, the color endpoints are interpolated and expanded.

For LDR endpoint modes, each color component C is calculated from the corresponding 8-bit endpoint components C_0 and C_1 as follows:

If sRGB conversion is not enabled, or for the alpha channel in any case, C_0 and C_1 are first expanded to 16 bits by bit replication:

```
C0 = (C0 << 8) | C0;    C1 = (C1 << 8) | C1;
```

If sRGB conversion is enabled, C_0 and C_1 for the R, G, and B channels are expanded to 16 bits differently, as follows:

```
C0 = (C0 << 8) | 0x80;  C1 = (C1 << 8) | 0x80;
```

C_0 and C_1 are then interpolated to produce a UNORM16 result C:

```
C = floor( (C0*(64-i) + C1*i + 32)/64 )
```

If sRGB conversion is enabled, the top 8 bits of the interpolation result for the R, G and B channels are passed to the external sRGB conversion block. Otherwise, and for the alpha channel in any case, if C = 65535, then the final result is 1.0 (0x3C00); otherwise C is divided by 65536 and the infinite-precision result of the division is converted to FP16 with round-to-zero semantics.

For HDR endpoint modes, color values are represented in a 12-bit pseudo-logarithmic representation, and interpolation occurs in a piecewise-approximate logarithmic manner as follows:

In LDR mode, the error result is returned.

In HDR mode, the color components from each endpoint, C_0 and C_1, are initially shifted left 4 bits to become 16-bit integer values and these are interpolated in the same way as LDR. The 16-bit value C is then decomposed into the top five bits, E, and the bottom 11 bits M, which are then processed and recombined with E to form the final value C_f:

```
C = floor( (C0*(64-i) + C1*i + 32)/64 )
E = (C & 0xF800) >> 11; M = C & 0x7FF;
if (M < 512) { Mt = 3*M; }
else if (M >= 1536) { Mt = 5*M - 2048; }
else { Mt = 4*M - 512; }
Cf = (E<<10) + (Mt>>3)
```

This interpolation is a considerably closer approximation to a logarithmic space than simple 16-bit interpolation.

This final value C_f is interpreted as an IEEE FP16 value. If the result is +Inf or NaN, it is converted to the bit pattern 0x7BFF, which is the largest representable finite value.

Dual-Plane Decoding

If dual-plane mode is disabled, all of the endpoint components are interpolated using the same weight value.

If dual-plane mode is enabled, two weights are stored with each texel. One component is then selected to use the second weight for interpolation, instead of the first weight. The first weight is then used for all other components.

The component to treat specially is indicated using the 2-bit Color Component Selector (CCS) field as shown in Table 16.31.

Value	Weight 0	Weight 1
0	GBA	R
1	RBA	G
2	RGA	B
3	RGB	A

Table 16.31: ASTC dual plane color component selector values

The CCS bits are stored at a variable position directly below the weight bits and any additional CEM bits.

Partition Pattern Generation

When multiple partitions are active, each texel position is assigned a partition index. This partition index is calculated using a seed (the partition pattern index), the texel's x,y,z position within the block, and the number of partitions. An additional argument, small_block, is set to 1 if the number of texels in the block is less than 31, otherwise it is set to 0.

This function is specified in terms of x, y and z in order to support 3D textures. For 2D textures and texture slices, z will always be 0.

The full partition selection algorithm is as follows:

```
int select_partition(int seed, int x, int y, int z,
                     int partitioncount, int small_block)
{
    if( small_block ){ x <<= 1; y <<= 1; z <<= 1; }
    seed += (partitioncount-1) * 1024;
    uint32_t rnum = hash52(seed);
    uint8_t seed1  =  rnum        & 0xF;
    uint8_t seed2  = (rnum >>  4) & 0xF;
    uint8_t seed3  = (rnum >>  8) & 0xF;
    uint8_t seed4  = (rnum >> 12) & 0xF;
    uint8_t seed5  = (rnum >> 16) & 0xF;
    uint8_t seed6  = (rnum >> 20) & 0xF;
    uint8_t seed7  = (rnum >> 24) & 0xF;
    uint8_t seed8  = (rnum >> 28) & 0xF;
    uint8_t seed9  = (rnum >> 18) & 0xF;
    uint8_t seed10 = (rnum >> 22) & 0xF;
    uint8_t seed11 = (rnum >> 26) & 0xF;
    uint8_t seed12 = ((rnum >> 30) | (rnum << 2)) & 0xF;

    seed1  *= seed1;    seed2  *= seed2;
    seed3  *= seed3;    seed4  *= seed4;
    seed5  *= seed5;    seed6  *= seed6;
    seed7  *= seed7;    seed8  *= seed8;
    seed9  *= seed9;    seed10 *= seed10;
    seed11 *= seed11;   seed12 *= seed12;

    int sh1, sh2, sh3;
    if( seed & 1 )
        { sh1 = (seed&2 ? 4:5); sh2 = (partitioncount==3 ? 6:5); }
    else
        { sh1 = (partitioncount==3 ? 6:5); sh2 = (seed&2 ? 4:5); }
    sh3 = (seed & 0x10) ? sh1 : sh2:

    seed1 >>= sh1; seed2  >>= sh2; seed3  >>= sh1; seed4  >>= sh2;
    seed5 >>= sh1; seed6  >>= sh2; seed7  >>= sh1; seed8  >>= sh2;
    seed9 >>= sh3; seed10 >>= sh3; seed11 >>= sh3; seed12 >>= sh3;

    int a = seed1*x + seed2*y + seed11*z + (rnum >> 14);
    int b = seed3*x + seed4*y + seed12*z + (rnum >> 10);
    int c = seed5*x + seed6*y + seed9 *z + (rnum >>  6);
    int d = seed7*x + seed8*y + seed10*z + (rnum >>  2);

    a &= 0x3F; b &= 0x3F; c &= 0x3F; d &= 0x3F;

    if( partitioncount < 4 ) d = 0;
    if( partitioncount < 3 ) c = 0;

    if( a >= b && a >= c && a >= d ) return 0;
    else if( b >= c && b >= d ) return 1;
    else if( c >= d ) return 2;
    else return 3;
}
```

As has been observed before, the bit selections are much easier to express in hardware than in C.

The seed is expanded using a hash function hash52, which is defined as follows:

```
uint32_t hash52( uint32_t p )
{
    p ^= p >> 15;  p -= p << 17;  p += p << 7; p += p <<  4;
    p ^= p >>  5;  p += p << 16;  p ^= p >> 7; p ^= p >> 3;
    p ^= p <<  6;  p ^= p >> 17;
    return p;
}
```

This assumes that all operations act on 32-bit values

Data Size Determination

The size of the data used to represent color endpoints is not explicitly specified. Instead, it is determined from the block mode and number of partitions as follows:

```
config_bits = 17;
if(num_partitions>1)
    if(single_CEM)
        config_bits = 29;
    else
        config_bits = 25 + 3*num_partitions;

num_weights = M * N * Q; // size of weight grid

if(dual_plane)
    config_bits += 2;
    num_weights *= 2;

weight_bits = ceil(num_weights*8*trits_in_weight_range/5) +
              ceil(num_weights*7*quints_in_weight_range/3) +
              num_weights*bits_in_weight_range;

remaining_bits = 128 - config_bits - weight_bits;

num_CEM_pairs = base_CEM_class+1 + count_bits(extra_CEM_bits);
```

The CEM value range is then looked up from a table indexed by remaining bits and num_CEM_pairs. This table is initialized such that the range is as large as possible, consistent with the constraint that the number of bits required to encode num_CEM_pairs pairs of values is not more than the number of remaining bits.

An equivalent iterative algorithm would be:

```
num_CEM_values = num_CEM_pairs*2;

for(range = each possible CEM range in descending order of size)
{
    CEM_bits = ceil(num_CEM_values*8*trits_in_CEM_range/5) +
               ceil(num_CEM_values*7*quints_in_CEM_range/3) +
               num_CEM_values*bits_in_CEM_range;

    if(CEM_bits <= remaining_bits)
        break;
}
return range;
```

In cases where this procedure results in unallocated bits, these bits are not read by the decoding process and can have any value.

Void-Extent Blocks

A void-extent block is a block encoded with a single color. It also specifies some additional information about the extent of the single-color area beyond this block, which can optionally be used by a decoder to reduce or prevent redundant block fetches.

The layout of a 2D Void-Extent block is as shown in Table 16.32.

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

Table 16.32: ASTC 2D void-extent block layout overview

The layout of a 3D Void-Extent block is as shown in Table 16.33.

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

Table 16.33: ASTC 3D void-extent block layout overview

Bit 9 is the Dynamic Range flag, which indicates the format in which colors are stored. A 0 value indicates LDR, in which case the color components are stored as UNORM16 values. A 1 indicates HDR, in which case the color components are stored as FP16 values.

The reason for the storage of UNORM16 values in the LDR case is due to the possibility that the value will need to be passed on to sRGB conversion. By storing the color value in the format which comes out of the interpolator, before the conversion to FP16, we avoid having to have separate versions for sRGB and linear modes.

If a void-extent block with HDR values is decoded in LDR mode, then the result will be the error color, opaque magenta, for all texels within the block.

In the HDR case, if the color component values are infinity or NaN, this will result in undefined behavior. As usual, this must not lead to GL interruption or termination.

Bits 10 and 11 are reserved and must be 1.

The minimum and maximum coordinate values are treated as unsigned integers and then normalized into the range 0..1 (by dividing by 2^{13}-1 or 2^{9}-1, for 2D and 3D respectively). The maximum values for each dimension must be greater than the corresponding minimum values, unless they are all all-1s.

If all the coordinates are all-1s, then the void extent is ignored, and the block is simply a constant-color block.

The existence of single-color blocks with void extents must not produce results different from those obtained if these single-color blocks are defined without void-extents. Any situation in which the results would differ is invalid. Results from invalid void extents are undefined.

If a void-extent appears in a MIPmap level other than the most detailed one, then the extent will apply to all of the more detailed levels too. This allows decoders to avoid sampling more detailed MIPmaps.

If the more detailed MIPmap level is not a constant color in this region, then the block may be marked as constant color, but without a void extent, as detailed above.

If a void-extent extends to the edge of a texture, then filtered texture colors may not be the same color as that specified in the block, due to texture border colors, wrapping, or cube face wrapping.

Care must be taken when updating or extracting partial image data that void-extents in the image do not become invalid.

Illegal Encodings

In ASTC, there is a variety of ways to encode an illegal block. Decoders are required to recognize all illegal blocks and emit the standard error color value upon encountering an illegal block.

Here is a comprehensive list of situations that represent illegal block encodings:

- The block mode specified is one of the modes explicitly listed as Reserved.
- A 2D void-extent block that has any of the reserved bits not set to 1.
- A block mode has been specified that would require more than 64 weights total.
- A block mode has been specified that would require more than 96 bits for integer sequence encoding of the weight grid.
- A block mode has been specified that would require fewer than 24 bits for integer sequence encoding of the weight grid.
- The size of the weight grid exceeds the size of the block footprint in any dimension.
- Color endpoint modes have been specified such that the color integer sequence encoding would require more than 18 integers.
- The number of bits available for color endpoint encoding after all the other fields have been counted is less than $\lceil \frac{13C}{5} \rceil$ where C is the number of color endpoint integers (this would restrict color integers to a range smaller than 0..5, which is not supported).
- Dual weight mode is enabled for a block with 4 partitions.
- Void-Extent blocks where the low coordinate for some texture axis is greater than or equal to the high coordinate.

Note also that, in LDR mode, a block which has both HDR and LDR endpoint modes assigned to different partitions is not an error block. Only those texels which belong to the HDR partition will result in the error color. Texels belonging to a LDR partition will be decoded as normal.

LDR PROFILE SUPPORT

In order to ease verification and accelerate adoption, an LDR-only subset of the full ASTC specification has been made available.

Implementations of this LDR Profile must satisfy the following requirements:

- All textures with valid encodings for LDR Profile must decode identically using either a LDR Profile, HDR Profile, or Full Profile decoder.
- All features included only in the HDR Profile or Full Profile must be treated as reserved in the LDR Profile, and return the error color on decoding.
- Any sequence of API calls valid for the LDR Profile must also be valid for the HDR Profile or Full Profile and return identical results when given a texture encoded for the LDR Profile.

The feature subset for the LDR profile is:

- 2D textures only.
- Only those block sizes listed in Table 16.2 are supported.
- LDR operation mode only.
- Only LDR endpoint formats must be supported, namely formats 0, 1, 4, 5, 6, 8, 9, 10, 12, 13.
- Decoding from a HDR endpoint results in the error color.
- Interpolation returns UNORM8 results when used in conjunction with sRGB.
- LDR void extent blocks must be supported, but void extents may not be checked.

HDR PROFILE SUPPORT

In order to ease verification and accelerate adoption, a second subset of the full ASTC specification has been made available, known as the HDR profile.

Implementations of the HDR Profile must satisfy the following requirements:

- The HDR profile is a superset of the LDR profile and therefore all valid LDR encodings must decode identically using a HDR profile decoder.
- All textures with valid encodings for HDR Profile must decode identically using either a HDR Profile or Full Profile decoder.
- All features included only in the Full Profile must be treated as reserved in the HDR Profile, and return the error color on decoding.
- Any sequence of API calls valid for the HDR Profile must also be valid for the Full Profile and return identical results when given a texture encoded for the HDR Profile.

The feature subset for the HDR profile is:

- 2D textures only.
- Only those block sizes listed in Table 16.2 are supported.
- All endpoint formats must be supported.
- 2D void extent blocks must be supported, but void extents may not be checked.

Chapter 17

Floating-point formats

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

16-bit floating-point numbers

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

$$V = \begin{cases} (-1)^S \times 0.0, & E = 0, M = 0 \\ (-1)^S \times 2^{-14} \times \frac{M}{2^{10}}, & E = 0, M \neq 0 \\ (-1)^S \times 2^{E-15} \times \left(1 + \frac{M}{2^{10}}\right), & 0 < E < 31 \\ (-1)^S \times Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

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

$$S = \left\lfloor \frac{N \bmod 65536}{32768} \right\rfloor$$

$$E = \left\lfloor \frac{N \bmod 32768}{1024} \right\rfloor$$

$$M = N \bmod 1024.$$

Unsigned 11-bit floating-point numbers

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

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{64}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{64}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

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

$$E = \left\lfloor \frac{N}{64} \right\rfloor$$

$$M = N \bmod 64.$$

Unsigned 10-bit floating-point numbers

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

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{32}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{32}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

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

$$E = \left\lfloor \frac{N}{32} \right\rfloor$$

$$M = N \bmod 32.$$

Non-standard floating point formats

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

An example of use of the 16-bit floating point format described in Section 17.1 but described in terms of a custom floating point format is provided in Table 18.13. Note that this is provided for example only, and this particular format would be better described using the standard 16-bit floating point format as documented in Table 18.14.

The mantissa

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

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

For example, the shared exponent formats shown in Table 18.8 does not have an implicit "1" bit, and therefore the **sample_upper** values for the 9-bit mantissas are 256 — this being the mantissa value for 1.0 when the exponent is set to 0.

For the 16-bit signed floating point format described in Section 17.1, **sample_upper** should be set to 1024, indicating the implicit "1" bit which is above the 10 bits representable in the mantissa. **sample_lower** should be 0 in this case, since the mantissa uses a sign-magnitude representation.

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

The exponent

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

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

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

Special values

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

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

Floating point values encoded with an exponent larger than the exponent's **sample_upper** value and with a mantissa of 0 are interpreted as representing +/- infinity, depending on the value of an explicit sign bit. Note that in some formats, no exponent above **sample_upper** is possible — for example, Table 18.8.

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

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

Conversion formulae

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

$$V = \begin{cases} (-1)^S \times 0.0, & E = 0, M = 0 \\ (-1)^S \times 2^{-(E_{sample\_lower}-1)} \times \frac{M}{M_{sample\_upper}}, & E = 0, M \neq 0 \\ (-1)^S \times 2^{E-E_{sample\_lower}} \times \left(1 + \frac{M}{M_{sample\_upper}}\right), & 0 < E \leq E_{sample\_upper} \\ (-1)^S \times Inf, & E > E_{sample\_upper}, M = 0 \\ NaN, & E > E_{sample\_upper}, M \neq 0. \end{cases}$$

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

$$V = \begin{cases} (-1)^S \times 2^{E-E_{sample\_lower}} \times \left(\frac{M}{M_{sample\_upper}}\right), & 0 < E \leq E_{sample\_upper} \\ (-1)^S \times Inf, & E > E_{sample\_upper}, M = 0 \\ NaN, & E > E_{sample\_upper}, M \neq 0. \end{cases}$$

A descriptor block for a format without an implicit "1" (and with the added complication of having the same exponent bits shared across multiple channels, which is why an implicit "1" bit does not make sense) is shown in Table 18.8. In the case of this particular example, the above equations simplify to:

$$red = red_{shared} \times 2^{(exp_{shared}-B-N)}$$

$$green = green_{shared} \times 2^{(exp_{shared}-B-N)}$$

$$blue = blue_{shared} \times 2^{(exp_{shared}-B-N)}$$

Where:

$$N = 9 \text{ (= number of mantissa bits per component)}$$

$$B = 15 \text{ (= exponent bias)}$$

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

Chapter 18

Example format descriptors

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		88 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	SRGB (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
4 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for second sample			
8 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for third sample			
16 (bit_offset)		8 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for fourth sample			
24 (bit_offset)		8 (bit_length)	31 (channel_type) (alpha+linear)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			

Table 18.1: Four co-sited 8-bit sRGB channels, assuming premultiplied alpha

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
44 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		40 (descriptor_block_size)	
YUVSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	ALPHA_STRAIGHT (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
4 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (Y)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			

Table 18.2: A single 8-bit monochrome channel

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
156 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		152 (descriptor_block_size)	
YUVSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	ALPHA_STRAIGHT (flags)
7 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
1 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
0 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for second sample			
1 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
2 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for third sample			
2 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
4 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for fourth sample			
3 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
6 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for fifth sample			
4 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
8 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for sixth sample			
5 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
10 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for seventh sample			
6 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
12 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			
Sample information for eighth sample			
7 (bit_offset)		1 (bit_length)	0 (channel_type) (Y)
14 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
1 (sample_upper)			

Table 18.3: 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 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		88 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	SRGB (transfer_function)	PREMULTIPLIED (flags)
1 (texel_block_dimension_0)	1 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
2 (bytes_plane_0)	2 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for second sample			
8 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
2 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for third sample			
16 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	2 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for fourth sample			
24 (bit_offset)		8 (bit_length)	2 (channel_type) (blue)
2 (sample_position_0)	2 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			

Table 18.4: 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 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		104 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	SRGB (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
0 (bytes_plane_0)	4 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for the palette index			
0 (bit_offset)		3 (bit_length)	0 (channel_type) (irrelevant)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
4 (sample_upper) — this specifies that there are 5 palette entries			
Sample information for first sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for second sample			
8 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for third sample			
16 (bit_offset)		8 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for fourth sample			
24 (bit_offset)		8 (bit_length)	31 (channel_type) (alpha+linear)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			

Table 18.5: 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 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		120 (descriptor_block_size)	
YUVSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	PREMULTIPLIED (flags)
1 (texel_block_dimension_0)	1 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
2 (bytes_plane_0)	2 (bytes_plane_1)	1 (bytes_plane_2)	1 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first Y sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (Y)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
235 (sample_upper)			
Sample information for second Y sample			
8 (bit_offset)		8 (bit_length)	0 (channel_type) (Y)
2 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
235 (sample_upper)			
Sample information for third Y sample			
16 (bit_offset)		8 (bit_length)	0 (channel_type) (Y)
0 (sample_position_0)	2 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
235 (sample_upper)			
Sample information for fourth Y sample			
24 (bit_offset)		8 (bit_length)	0 (channel_type) (Y)
2 (sample_position_0)	2 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
235 (sample_upper)			
Sample information for U sample			
32 (bit_offset)		8 (bit_length)	1 (channel_type) (U)
1 (sample_position_0)	1 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
240 (sample_upper)			
Sample information for V sample			
36 (bit_offset)		8 (bit_length)	2 (channel_type) (V)
1 (sample_position_0)	1 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
16 (sample_lower)			
240 (sample_upper)			

Table 18.6: YUV4:2:0: BT.709 reduced-range data, with U and V aligned to the midpoint of the Y samples.

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		88 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	SRGB (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
2 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample: bit 0 belongs to green, bits 0..2 of channel in 13..15			
13 (bit_offset)		3 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
63 (sample_upper)			
Sample information for second sample: bits 3..5 of green in 0..2			
0 (bit_offset)		3 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower) — ignored, taken from first sample			
0 (sample_upper) — ignored, taken from first sample			
Sample information for third sample			
3 (bit_offset)		5 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
31 (sample_upper)			
Sample information for fourth sample			
8 (bit_offset)		5 (bit_length)	1 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
31 (sample_upper)			

Table 18.7: 565 RGB format as represented on a big-endian architecture

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
124 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		120 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
4 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for R mantissa			
0 (bit_offset)		9 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
256 (sample_upper) - mantissa at 1.0 (no implicit “1”)			
Sample information for R exponent			
27 (bit_offset)		5 (bit_length)	32 (channel_type) (R+exponent)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
15 (sample_lower) - exponent bias			
31 (sample_upper) - no infinity/NaN support			
Sample information for G mantissa			
16 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
256 (sample_upper) - mantissa at 1.0 (no implicit “1”)			
Sample information for G exponent			
27 (bit_offset)		5 (bit_length)	33 (channel_type) (G+exponent)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
15 (sample_lower) - exponent bias			
31 (sample_upper) - no infinity/NaN support			
Sample information for B mantissa			
32 (bit_offset)		8 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
256 (sample_upper) - mantissa at 1.0 (no implicit “1”)			
Sample information for B exponent			
27 (bit_offset)		5 (bit_length)	34 (channel_type) (B+exponent)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
15 (sample_lower) - exponent bias			
31 (sample_upper) - no infinity/NaN support			

Table 18.8: R9G9B9E5 shared-exponent format

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
124 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		120 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
1 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for R tint (shared low bits)			
0 (bit_offset)		2 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
0 (sample_upper) - ignored, not unique			
Sample information for R unique (high) bits			
2 (bit_offset)		2 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
15 (sample_upper) - unique R upper value			
Sample information for G tint (shared low bits)			
0 (bit_offset)		2 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
0 (sample_upper) - ignored, not unique			
Sample information for G unique (high) bits			
4 (bit_offset)		2 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
15 (sample_upper) - unique G upper value			
Sample information for B tint (shared low bits)			
0 (bit_offset)		2 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
0 (sample_upper) - ignored, not unique			
Sample information for B unique (high) bits			
6 (bit_offset)		2 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
15 (sample_upper) - unique B upper value			

Table 18.9: Acorn 256-color format (2 bits each independent RGB, 2 bits shared “tint”)

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
220 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		216 (descriptor_block_size) — 12 samples	
YUVSDA (color_model)	BT709 (color_primaries)	ITU (transfer_function)	PREMULTIPLIED (flags)
5 (dimension_0)	0 (dimension_1)	0 (dimension_2)	0 (dimension_3)
16 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
0 (bit_offset)		10 (bit_length)	1 (channel_type) (U0+1)
1 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
10 (bit offset)		10 (bit_length)	0 (channel_type) (Y0)
0	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
20 (bit_offset)		10 (bit_length)	2 (channel_type) (V0+1)
1 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
32 (bit offset)		10 (bit_length)	0 (channel_type) (Y1)
2	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
42 (bit_offset)		10 (bit_length)	1 (channel_type) (U2+3)
5 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
52 (bit offset)		10 (bit_length)	0 (channel_type) (Y2)
4	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
64 (bit_offset)		10 (bit_length)	2 (channel_type) (V2+3)
5 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
74 (bit offset)		10 (bit_length)	0 (channel_type) (Y3)
6	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
84 (bit_offset)		10 (bit_length)	1 (channel_type) (U4+5)
9 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
96 (bit offset)		10 (bit_length)	0 (channel_type) (Y4)
8	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
106 (bit_offset)		10 (bit_length)	2 (channel_type) (V4+5)
9 (assume mid-sited)	0	0	0
0 (sample_lower)			
1023 (sample_upper)			
116 (bit offset)		10 (bit_length)	0 (channel_type) (Y5)
10	0	0	0
0 (sample_lower)			
1023 (sample_upper)			

Table 18.10: V210 format (full-range YUV)

Byte 0 (LSB)	**Byte 1**	**Byte 2**	**Byte 3 (MSB)**
92 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		88 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	LINEAR (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
1 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
0 (bit_offset)		8 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for second sample			
0 (bit_offset)		8 (bit_length)	1 (channel_type) (green)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for third sample			
0 (bit_offset)		8 (bit_length)	2 (channel_type) (blue)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			
Sample information for fourth sample			
0 (bit_offset)		8 (bit_length)	31 (channel_type) (alpha+linear)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
255 (sample_upper)			

Table 18.11: Intensity-alpha format showing aliased samples

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
76 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		72 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	LINEAR (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
6 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample			
32 (bit_offset)		16 (bit_length)	64 (channel_type) (red | signed)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower) - ignored, overridden by second sample			
0 (sample_upper) - ignored, overridden by second sample			
Sample information for second sample			
16 (bit_offset)		16 (bit_length)	64 (channel_type) (red | signed)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0xFFFFFFFF (sample_lower) - bottom 32 bits of sample_lower			
0xFFFFFFFF (sample_upper) - bottom 32 bits of sample_upper			
Sample information for third sample			
0 (bit_offset)		16 (bit_length)	64 (channel_type) (red | signed)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0xFFFFFFFF (sample_lower) - top 16 bits of sample_lower, sign-extended			
0x7FFF (sample_upper) - top 16 bits of sample_upper			

Table 18.12: Three co-sited 16-bit samples with the high word first, comprising a single 48-bit signed red channel

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
76 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		72 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	LINEAR (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
2 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information for first sample (mantissa)			
0 (bit_offset)		10 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0 (sample_lower)			
2048 (sample_upper) - implicit 1			
Sample information for second sample (sign bit)			
15 (bit_offset)		1 (bit_length)	64 (channel_type) (red | signed)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0xFFFFFFFF (sample_lower)			
0x0 (sample_upper)			
Sample information for third sample (exponent)			
10 (bit_offset)		5 (bit_length)	32 (channel_type) (red | exponent)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
15 (sample_lower) - bias			
30 (sample_upper) - support for infinities (because 31 can be encoded)			

Table 18.13: A single 16-bit floating-point red value, described explicitly (example only!)

Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
44 (total_size)			
0 (vendor_id)		0 (descriptor_type)	
0 (version number)		40 (descriptor_block_size)	
RGBSDA (color_model)	BT709 (color_primaries)	LINEAR (transfer_function)	PREMULTIPLIED (flags)
0 (texel_block_dimension_0)	0 (texel_block_dimension_1)	0 (texel_block_dimension_2)	0 (texel_block_dimension_3)
2 (bytes_plane_0)	0 (bytes_plane_1)	0 (bytes_plane_2)	0 (bytes_plane_3)
0 (bytes_plane_4)	0 (bytes_plane_5)	0 (bytes_plane_6)	0 (bytes_plane_7)
Sample information			
0 (bit_offset)		10 (bit_length)	0 (channel_type) (red)
0 (sample_position_0)	0 (sample_position_1)	0 (sample_position_2)	0 (sample_position_3)
0xbf80000 (sample_lower) = -1.0			
0x3f80000 (sample_upper) = 1.0			

Table 18.14: A single 16-bit floating-point red value, described normally