ASTC Specification

Media Processing Division

Document Number: PR474-PRDC-013084 1.0
Date of Issue: 2012-07-19

Author(s): Sean Ellis, Jørn Nystad

Authorized by: Steve Steele

Abstract

This document specifies the ASTC texture compression standard.

Keywords

ASTC, Texture compression, Specification

Distribution List

Name	***Function***
Mali Developer Portal	Public release

CONTENTS

1 About This Document **3**

1.1 Change Control **3**
1.1.1 Current Status and Anticipated Changes 3
1.1.2 Change History 3

1.2 References **3**

1.3 Terms and Abbreviations **3**

1.4 Standards Terminology **4**
1.4.1 Meeting Requirements 4

1.5 Scope **4**

2 Introduction 5
2.1 What is ASTC? 5
2.2 Design Goals 5
2.3 Basic Concepts 5
2.4 Block Encoding 6
2.5 LDR and HDR Modes 7
2.6 Summary 8

3 Detailed Specification 9
3.1 Decode Procedure 9
3.2 Block Determination and Bit Rates 9
3.3 Block Layout 10
3.4 Block Mode 11
3.5 Color Endpoint Mode 13
3.6 Integer Sequence Encoding 14
3.7 Endpoint Unquantization 15
3.8 LDR Endpoint Decoding 16
3.9 HDR Endpoint Decoding 18
3.9.1 HDR Endpoint Mode 2 18
3.9.2 HDR Endpoint Mode 3 18
3.9.3 HDR Endpoint Mode 7 19
3.9.4 HDR Endpoint Mode 11 21
3.9.5 HDR Endpoint Mode 14 23
3.9.6 HDR Endpoint Mode 15 23
3.10 Weight Decoding 24
3.11 Weight Unquantization 24
3.12 Weight Infill 25
3.13 Weight Application 26
3.14 Dual-Plane Decoding 27
3.15 Partition Pattern Generation 27
3.16 Data Size Determination 28
3.17 Void-Extent Blocks 29
3.18 Illegal Encodings 30

4 LDR Profile Support 31

1 ABOUT THIS DOCUMENT

1.1 Change Control

1.1.1 Current Status and Anticipated Changes

It is intended that this should be a complete and correct specification of all ASTC features, both for the LDR profile and the full profile.

1.1.2 Change History

Issue	*Date*	*Author(s)*	*Change*
1.0	2012-07-19	Sean Ellis, Jørn Nystad	First Public Release.

Table 1 – Change History

1.2 References

This document does not refer to any additional documents.

Ref	*Document Number or URL*	*Author(s)*	*Title*
	None		

Table 2 – References

1.3 Terms and Abbreviations

This document refers to the following terms and abbreviations:

Term	*Description*
ASTC	Adaptive Scalable Texture Compression
Endpoint	A color that is encoded in the block information. Other colors may be calculated from two or more endpoint values.
Footprint	For a given block of encoded data, its footprint indicates which texels in the image are included in that block.
HDR	High Dynamic Range. Indicates a texture with color component values that may be beyond the usual 0..1 displayable range.
LDR	Low Dynamic Range. Indicates a texture with color component values within the usual 0..1 displayable range.
Partition	Where colors within a block do not fall neatly onto a single interpolated line in the color space, they can be separated into multiple independent sets, known as partitions.
Plane	Where color components in a texture are not strongly correlated, it may be useful to encode them separately. Each separate encoding is referred to as a plane.
PSNR	Peak Signal-to-Noise Ratio. A measure of similarity between two images, usually expressed in decibels. Often used to compare a decompressed image against the original, where higher PSNR values indicate higher fidelity.
Raster-order	A way of arranging a 2D array of items in memory. The item index in a raster-ordered block is formed by adding the X coordinate to the Y coordinate multiplied by the width of the block.
Simplex	A simplex is the simplest possible n-dimensional shape (with n+1 vertices). In 2D space, it is a triangle; in 3D space, it is a tetrahedron.

Table 3 – Terms and Abbreviations

1.4 Standards Terminology

This document uses the following terms, as defined in http://www.ietf.org/rfc/rfc2119.txt.

Must Shall Should May

1.4.1 Meeting Requirements

Requirements marked as "must" or "shall" are critical pieces of functionality that need to exist in the final product. If these pieces of functionality are missing then it is likely that we are not going to meet customer requirements, and may be forced to release at a lower quality than intended.

Where this document indicates that a feature "should" exist, then all reasonable efforts should be made to ensure that the feature exists in the product. If a feature at this level is not to be implemented in the release version then the effect of its removal should be well understood and documented.

Where this document states that a feature "may" exist as then it is not a critical part of the release, but should be considered if easily implemented and tested. These features may also become more important in future releases, so when designing the current version of the product some thought should be given to the potential ease of future implementation.

1.5 Scope

This document describes the ASTC data layout and decode procedure. It does not specify details of internal operation, internal hardware design, or timing. Nor does it specify encoding strategies.

2 INTRODUCTION

2.1 What is ASTC?

ASTC stands for Adaptive Scalable Texture Compression, and it is ARM's proposal for an open texture compression standard for next generation GPUs.

2.2 Design Goals

The design goals for the format are as follows:

Random access. This is a must for any texture compression format.

Bit exact decode. This is a must for conformance testing and reproducibility.

Suitable for mobile use. The format should be suitable for both desktop and mobile GPU environments. It should be low bandwidth and low in area.

Flexible choice of bit rate. Current formats only offer a few bit rates, leaving content developers with only coarse control over the size/quality tradeoff.

Scalable and long-lived. The format should support existing R, RG, RGB and RGBA image types, and also have high "headroom", allowing continuing use for several years and the ability to innovate in encoders. Part of this is the choice to include HDR and 3D.

Feature orthogonality. The choices for the various features of the format are all orthogonal to each other. This has three effects: first, it allows a large, flexible configuration space; second, it makes that space easier to understand; and third, it makes verification easier.

Best in class at given bit rate. We 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.

Open standard. Fragmentation of texture compression standards is something that is universally unpopular with content developers. A successful format needs to be freely implementable, open, and accessible. Accordingly, we are working with the Khronos group to standardize ASTC.

2.3 Basic Concepts

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

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

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.

2.4 Block Encoding

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

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

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

Figure 1 shows an example of a 4x4 block encoded in this way, using a 2-channel image (Red+Green) for legibility:

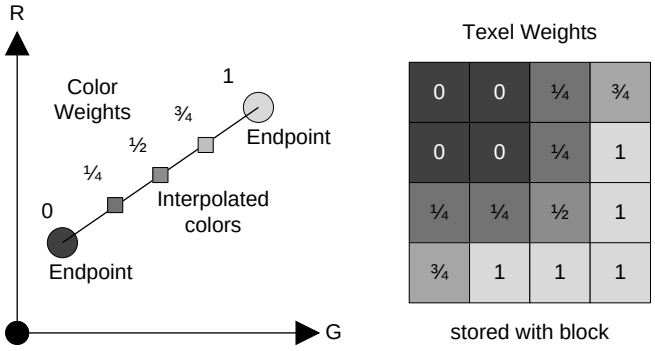

Figure 1 – Endpoint and Weight Representation

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 12x12 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.

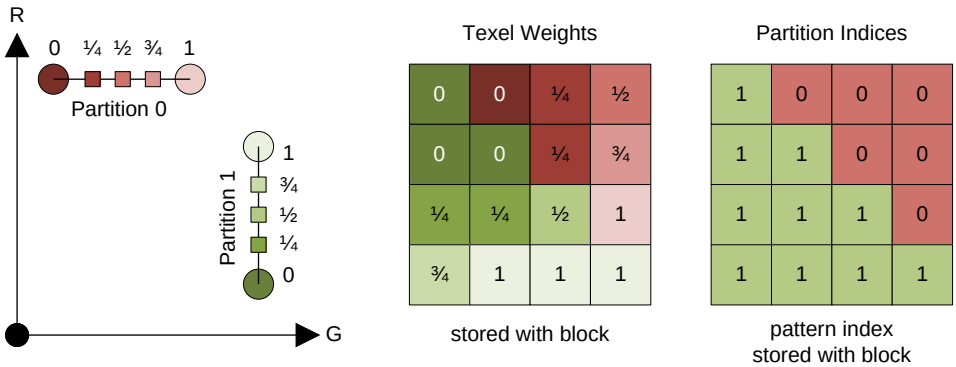

Figure 2 – Partitioning

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.

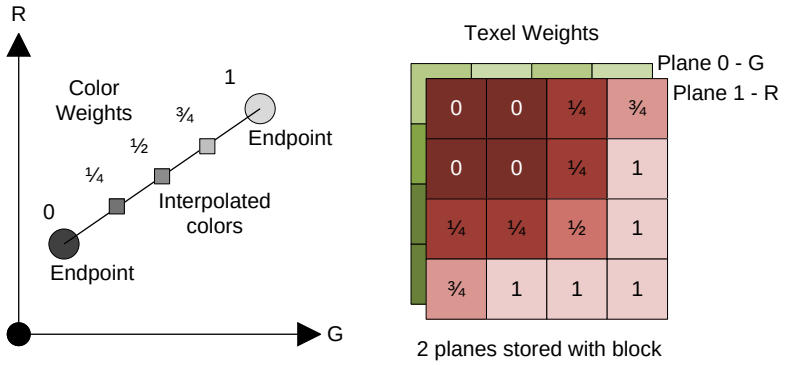

Figure 3 – Dual-Plane Mode

Since each of the above options is independent, it is possible to specify any combination of channels, endpoint color encoding, weight encoding, weight 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 is used to short-circuit fetching of what would be identical blocks, and further reduce memory bandwidth.

2.5 LDR and HDR Modes

The decoding process for LDR content can be simplified if it is known in advance that LDR output is required. Dynamic range selection is therefore included as part of the global configuration. The two modes differ in various ways.

Operation	***LDR Mode***	***HDR Mode***
Returned value	Vector of FP16 values, or Vector of UNORM8 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 4 – 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.

2.6 Summary

The global configuration data for the format is as follows:

- Block dimension (2D or 3D)
- Block footprint size
- Dynamic range (HDR or LDR)
- sRGB output enabled or not

The data specified per block is as follows:

- Texel weight grid size
- Texel weight range
- Texel weight values
- Number of partitions
- Partition pattern index
- Color endpoint modes
- Color endpoint data
- Number of planes
- Plane-to-channel assignment

3 DETAILED SPECIFICATION

3.1 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
```

3.2 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 following bit rates:

Block Footprint	***Bit Rate***	***Increment***
4x4	8.00	125%
5x4	6.40	125%
5x5	5.12	120%
6x5	4.27	120%
6x6	3.56	114%
8x5	3.20	120%
8x6	2.67	105%

Block Footprint	***Bit Rate***	***Increment***
10x5	2.56	120%
10x6	2.13	107%
8x8	2.00	125%
10x8	1.60	125%
10x10	1.28	120%
12x10	1.07	120%
12x12	0.89	

Table 5 – 2D Footprint and Bit Rates

The block footprint is given as width x height.

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

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

For cubic and near-cubic blocks, this gives the following bit rates:

Block Footprint	***Bit Rate***
3x3x3	4.74
4x3x3	3.56
4x4x3	2.67
4x4x4	2.00
5x4x4	1.60

Block Footprint	***Bit Rate***
5x5x4	1.28
5x5x5	1.02
6x5x5	0.85
6x6x5	0.71
6x6x6	0.59

Table 6 – 3D Footprint and Bit Rates

For images which are not an integer multiple of the block size, additional texels are added to the edges with maximum X and Y. 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* x *H* x *D* pixels in size, with block size *w* x *h* x *d*, the size of the image in blocks is:

B_w = ceiling(*W*/*w*)

B_h = ceiling(*H*/*h*)

B_d = ceiling(*D*/*d*)

3.3 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:

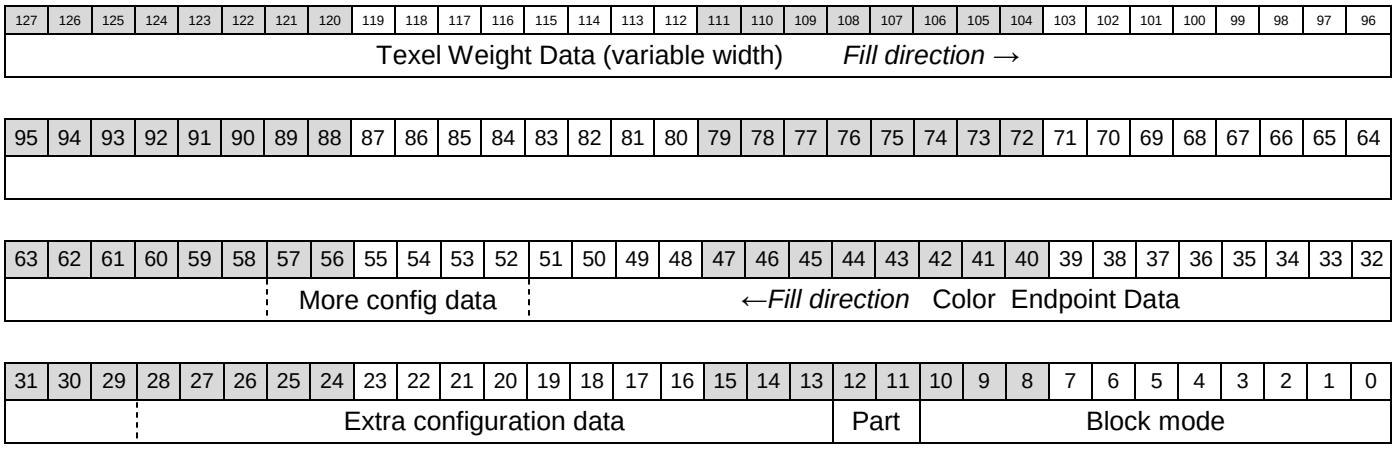

Table 7 –Block Layout Overview

Dotted partition lines indicate that the split position is 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 follows (only the bottom 32 bits are shown):

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
← Color Endpoint Data															CEM				00		Block mode										

Table 8 –Single-partition Block Layout

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.

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
←			CEM						Partition Index										Part		Block mode										

Table 9 –Multi-partition Block Layout

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 3.17.

3.4 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 follows:

	Low Precision Range (H=0)				***High Precision Range (H=1)***			
R	***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 10 –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 3.6 - *Integer Sequence Encoding*.

For 2D blocks, the block mode field is laid out as follows:

10	9	8	7	6	5	4	3	2	1	0	Width N	Height M	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[1]
x	x	x	x	x	x	x	0	0	0	0	-	-	Reserved

Table 11 –2D Block Mode Layout

Note that, due to the encoding of the R field, as described in the previous page, bits R_2 and R_1 cannot both be zero, which disambiguates the first five rows from the rest of the table.

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 follows:

10	9	8	7	6	5	4	3	2	1	0	Width N	Height M	Depth Q	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[1]
x	x	x	x	x	x	x	0	0	0	0	-	-	-	Reserved

Table 12 –3D Block Mode Layout

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

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.

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).

[1] Except for valid void-extent encodings.

3.5 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 follows:

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

CEM	***Description***	***Class***
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 13 –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:

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 14 –Multi-Partition Color Endpoint Modes

If the CEM selector value in bits [24:23] is not 00, then data layout is as follows:

Partitions													28	27	26	25	24	23
2	Weight		M_1										M_0		C_1	C_0	CEM	
3	Weight		M_2		M_1		M_0						M_0	C_2	C_1	C_0	CEM	
4	Weight		M_3		M_2		M_1		M_0				C_3	C_2	C_1	C_0	CEM	

Table 15 –Multi-Partition Color Endpoint Modes

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 3.7.

3.6 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 following forms:

Value range	***MSB encoding***	***LSB encoding***	***Value***	***Block***	***Packed block size***
0 .. 2^n-1	-	*n* bit value *m* (*n* <= 8)	*m*	1	*n*
0 .. (3 * 2^n)-1	Base-3 “trit” value *t*	*n* bit value *m* (*n* <= 6)	*t* * 2^n + *m*	5	8 + 5**n*
0 .. (5 * 2^n)-1	Base-5 “quint” value *q*	*n* bit value *m* (*n* <= 5)	*q* * 2^n + *m*	3	7 + 3**n*

Table 16 –Encoding for Different Ranges

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

The encoding scheme packs the trits or quints, and then interleaves the *n* additional bits in positions that satisfy the requirements of an arbitrary length stream. This makes it possible to correctly specify lists of values whose length is not an integer multiple of 3 or 5 values. It also makes it possible to easily select a value at random within the stream. If there are insufficient bits in the stream to fill the final block, then unused (higher order) bits are assumed to be 0 when decoding.

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

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

If the range is encoded using a trit, then each block contains 5 values (v_0 to v_4), each of which contains a trit (t_0 to t_4) and a corresponding LSB value (m_0 to m_4). The first bit of the packed block is bit floor(*i*/5)*(8+5**n*). The bits in the block are packed as follows (in this example, *n* is 4):

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
T7	m_4				T6	T5	m_3				T4	m_2				T3	T2	m_1				T1	T0	m_0			

Table 17 –Trit-based Packing

The five trits t_0 to t_4 are obtained by bit manipulations of the 8 bits T[7:0] as follows:

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

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

If the range is encoded using a quint, then each block contains 3 values (v_0 to v_2), each of which contains a quint (q_0 to q_2) and a corresponding LSB value (m_0 to m_2). The first bit of the packed block is bit floor(*i*/3)*(7+3**n*).

The bits in the block are packed as follows (in this example, *n* is 4):

18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Q6	Q5	m_2				Q4	Q3	m_1				Q2	Q1	Q0	m_0			

Table 18 –Quint-based Packing

The three quints q_0 to q_2 are obtained by bit manipulations of the 7 bits Q[6:0] as follows:

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

3.7 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, B, C and D and are decoded using the range as follows:

Range	***Trits***	***Quints***	***Bits***	***Bit value***	***A (9 bits)***	***B (9 bits)***	***C (9 bits)***	***D (3 bits)***
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 19 –Color Unquantization Parameters

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.

3.8 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 the table. They are documented in following section.

The LDR methods can be summarized as follows.

CEM	***Range***	***Description***	***n***	***Endpoints***
0	LDR	Luminance, direct	2	`e0=(v0,v0,v0,0xFF); e1=(v1,v1,v1,0xFF);`

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

2	HDR	Luminance, large range	2	*See Section 3.9.1.*
3	HDR	Luminance, small range	2	*See Section 3.9.2.*
4	LDR	Luminance+Alpha, Direct	4	`e0=(v0,v0,v0,v2);` `e1=(v1,v1,v1,v3);`

5	LDR	Luminance+Alpha,
		base+offset
			4	`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);`

6	LDR	RGB, base+scale	4	`e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, 0xFF);` `e1=(v0,v1,v2,0xFF);`
7	HDR	RGB, base+scale	4	*See Section 3.9.3.*

8	LDR	RGB,
		Direct
			6	`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); }`

9	LDR	RGB,
		base+offset
			6	`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);`

10	LDR	RGB, base+scale plus two A	6	`e0=(v0*v3>>8,v1*v3>>8,v2*v3>>8, v4);` `e1=(v0,v1,v2, v5);`
11	HDR	RGB	6	*See Section 3.9.4.*

CEM	Range	Description	n	Endpoints

12	LDR	RGBA,
		direct
			8	`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); }`

13	LDR	RGBA,
		base+offset
			8	`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);`

14	HDR	RGB + LDR Alpha	8	*See Section 3.9.5.*
15	HDR	RGB + HDR Alpha	8	*See Section 3.9.6.*

Table 20 –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.

The bit_transfer_signed procedure transfers a bit from one signed byte value (a) to another (b). The result is an 8-bit signed integer value and a 6-bit integer value sign extended to 8 bits. Note that, as is often the case, this is easier to express in hardware than in C:

```
bit_transfer_signed(uint16_t& a, uint16_t& b)
{
    b >>= 1;
    b |= a & 0x80;
    a >>= 1;
    a &= 0x3F;
    if( (a&0x20)!=0 ) a-=0x40;
}
```

For the purposes of this pseudocode, the signed bytes are passed in as unsigned 16-bit integers because the semantics of a right shift on a signed value in C are undefined.

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;}
}
```

3.9 HDR Endpoint Decoding

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

3.9.1 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);
```

3.9.2 HDR Endpoint Mode 3

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

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

Table 21 – 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);
```

3.9.3 HDR Endpoint Mode 7

Mode 7 packs the bits for a base RGB value, a scale factor, and some mode bits into the four values (v0, v1, v2, v3):

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

Table 22 – 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 follows:

	Number of bits			
Mode	***R***	***G***	***B***	***Scale***
0	11	5	5	7
1	11	6	6	5
2	10	5	5	8
3	9	6	6	7
4	8	7	7	6
5	7	7	7	7

Destination of extra bits						
X0	***X1***	***X2***	***X3***	***X4***	***X5***	***X6***
R[9]	R[8]	R[7]	R[10]	R[6]	S[6]	S[5]
R[8]	G[5]	R[7]	B[5]	R[6]	R[10]	R[9]
R[9]	R[8]	R[7]	R[6]	S[7]	S[6]	S[5]
R[8]	G[5]	R[7]	B[5]	R[6]	S[6]	S[5]
G[6]	G[5]	B[6]	B[5]	R[6]	R[7]	S[5]
G[6]	G[5]	B[6]	B[5]	R[6]	S[6]	S[5]

Table 23 – 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 modeval = ((v0 & 0xC0) >> 6) | ((v1 & 0x80) >> 5) | ((v2 & 0x80) >> 4);

int majcomp;
int mode;

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;
```

3.9.4 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):

Value	7	6	5	4	3	2	1	0
v0	a[7:0]							
v1	m[0]	a[8]	c[5:0]					
v2	m[1]	X0	b0[5:0]					
v3	m[2]	X1	b1[5:0]					
v4	mj[0]	X2	X4	d0[4:0]				
v5	mj[1]	X3	X5	d1[4:0]				

Table 24 – HDR Mode 11 Value Layout

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

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 25 – HDR Mode 11 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:

	Number of bits			
Mode	***a***	***b***	***c***	***d***
0	9	7	6	7
1	9	8	6	6
2	10	6	7	7
3	10	7	7	6
4	11	8	6	5
5	11	6	7	6
6	12	7	7	5
7	12	6	7	6

Destination of extra bits					
X0	***X1***	***X2***	***X3***	***X4***	***X5***
b0[6]	b1[6]	d0[6]	d1[6]	d0[5]	d1[5]
b0[6]	b1[6]	b0[7]	b1[7]	d0[5]	d1[5]
a[9]	c[6]	d0[6]	d1[6]	d0[5]	d1[5]
b0[6]	b1[6]	a[9]	c[6]	d0[5]	d1[5]
b0[6]	b1[6]	b0[7]	b1[7]	a[9]	a[10]
a[9]	a[10]	c[7]	c[6]	d0[5]	d1[5]
b0[6]	b1[6]	a[11]	c[6]	a[9]	a[10]
a[9]	a[10]	a[11]	c[6]	d0[5]	d1[5]

Table 26 – 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 ); }
```

3.9.5 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.

3.9.6 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:

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

Table 27 – 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.

3.10 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.

3.11 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 follows:

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

Table 28 –Weight Unquantization Values

For other values, we calculate the initial inputs to a bit manipulation procedure. These are denoted A, B, C and D and are decoded using the range as follows:

Range	***Trits***	***Quints***	***Bits***	***Bit value***	***A (7 bits)***	***B (7 bits)***	***C (7 bits)***	***D (3 bits)***
0..5	1		1	a	aaaaaaa	0000000	50	Trit
0..9		1	1	a	aaaaaaa	0000000	28	Quint
0..11	1		2	ba	aaaaaaa	b000b0b	23	Trit
0..19		1	2	ba	aaaaaaa	b0000b0	13	Quint
0..23	1		3	cba	aaaaaaa	cb000cb	11	Trit

Table 29 –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.

3.12 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)

```
Ds = floor( (1024 + floor(Bs/2)) / (Bs-1) );
Dt = floor( (1024 + floor(Bt/2)) / (Bt-1) );
Dr = floor( (1024 + floor(Br/2)) / (Br-1) );
```

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 p_{00} to p_{11} are the weights at the corner of the square in which the texel position resides. These are then weighted using the fractional position to produce the effective weight *i* as follows:

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

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

$f_s>f_t$	$f_t>f_r$	$f_s>f_r$	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 30 –Simplex Interpolation Parameters

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

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

3.13 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, 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 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 are passed to the external sRGB conversion block. Otherwise, if C = 65535, then the final result is 1.0 (0x3C00) otherwise C is divided by 2^{16} 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 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.

3.14 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 follows:

Value	*Weight 0*	*Weight 1*
0	GBA	R
1	RBA	G

Value	*Weight 0*	*Weight 1*
2	RGA	B
3	RGB	A

Table 31 –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.

3.15 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.

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);
```

…continued

```
    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

3.16 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 = 24 + 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_weights_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.

3.17 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 prevent redundant block fetches.

The layout of a 2D Void-Extent block is as follows:

127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
Block color A component																Block color B component															

95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
Block color G component																Block color R component															

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
Void-extent maximum T coordinate													Void-extent minimum T coordinate													Void-extent					

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
maximum S coordinate							Void-extent minimum S coordinate													1	1	D	1	1	1	1	1	1	1	0	0

Table 32 –2D Void-Extent Block Layout Overview

The layout of a 3D Void-Extent block is as follows:

127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
Block color A component																Block color B component															

95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
Block color G component																Block color R component															

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
Maximum R coordinate									Minimum R coordinate									Maximum T coordinate								Minimum					

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
T coordinate				Maximum S coordinate									Minimum S coordinate									D	1	1	1	1	1	1	1	0	0

Table 33 –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.

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.

3.18 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 Block color value upon encountering an illegal block. The standard Error Block color value is opaque magenta (R, G, B, A) = (0xFF, 0x00, 0xFF, 0xFF) in the LDR operation mode, and a vector of NaNs (R, G, B, A)=(NaN, NaN, NaN, NaN) in the HDR operation mode. It is recommended that the NaN be encoded as the bit-pattern 0xFFFF.

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.
- The block mode specified would require more than 64 weights total.
- The block mode specified would require more than 96 bits for the weights' Integer Sequence Encoding.
- The block mode specified would require fewer than 24 bits for the weights' Integer Sequence Encoding.
- 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 *ceil(13C/5)* 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 Block 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.

4 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 or Full Profile decoder.
- All features included only in the 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 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 5 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.