Intel® UHD Graphics Open Source

Programmer's Reference Manual

For the 2020 Intel Core™ Processors with Intel Hybrid Technology based on the "Lakefield" Platform

Volume 5: Memory Data Formats

May 2021, Revision 1.0

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

Customer is responsible for safety of the overall system, including compliance with applicable safety-related requirements or standards.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Table of Contents

Memory Data Formats

This chapter describes the attributes associated with the memory-resident data objects operated on by the graphics pipeline. This includes object types, pixel formats, memory layouts, and rules/restrictions placed on the dimensions, physical memory location, pitch, alignment, etc. with respect to the specific operations performed on the objects.

Unsigned Normalized (UNORM)

An unsigned normalized value with *n* bits is interpreted as a value between 0.0 and 1.0. The minimum value (all 0's) is interpreted as 0.0, the maximum value (all 1's) is interpreted as 1.0. Values in between are equally spaced. For example, a 2-bit UNORM value would have the four values 0, 1/3, 2/3, and 1.

If the incoming value is interpreted as an n-bit integer, the interpreted value can be calculated by dividing the integer by 2^n-1.

Gamma Conversion (SRGB)

Gamma conversion is only supported on UNORM formats. If this flag is included in the surface format name, it indicates that a reverse gamma conversion is to be done after the source surface is read, and a forward gamma conversion is to be done before the destination surface is written.

Signed Normalized (SNORM)

Programming Note	
Context:	Signed normalized value in memory data formats.
A signed normalized value with *n* bits is interpreted as a value between -1 and +1.0. If the incoming value is interpreted as a 2's-complement n-bit integer, the interpreted value can be calculated by dividing the integer by 2^{n-1}-1. The most negative value of -2^{n-1} will result in a value slightly smaller than -1.0. This value is clamped to -1.0; thus, there are two representations of -1.0 in SNORM format.	

Unsigned Integer (UINT/USCALED)

The UINT and USCALED formats interpret the source as an unsigned integer value with *n* bits with a range of 0 to 2^n-1.

The UINT formats copy the source value to the destination (zero-extending if required), keeping the value as an integer.

The USCALED formats convert the integer into the corresponding floating point value (e.g., 0x03 --> 3.0f). For 32-bit sources, the value is rounded to nearest even.

Signed Integer (SINT/SSCALED)

A signed integer value with *n* bits is interpreted as a 2's complement integer with a range of -2^{n-1} to $+2^{n-1}-1$.

The SINT formats copy the source value to the destination (sign-extending if required), keeping the value as an integer.

The SSCALED formats convert the integer into the corresponding floating point value (e.g., 0xFFFD --> -3.0f). For 32-bit sources, the value is rounded to nearest even.

Floating Point (FLOAT)

Refer to IEEE Standard 754 for Binary Floating-Point Arithmetic. The IA-32 Intel (R) Architecture Software Developer's Manual also describes floating point data types .

64-bit Floating Point

Bit	Description
63	**Sign (s)**
62:52	**Exponent (e)** Biased Exponent
51:0	**Fraction (f)** Does not include "hidden one"

The value of this data type is derived as:

- if e == b'11..11' and f != 0, then v is NaN regardless of s
- if e == b'11..11' and f == 0, then v = $(-1)^s$*infinity (signed infinity)
- if 0 < e < b'11..11', then v = $(-1)^s*2^{(e-1023)}*(1.f)$
- if e == 0 and f != 0, then v = $(-1)^s*2^{(e-1022)}*(0.f)$ (denormalized numbers)
- if e == 0 and f == 0, then v = $(-1)^s$*0 (signed zero)

32-bit Floating Point

Bit	Description
31	**Sign (s)**
30:23	**Exponent (e)** Biased Exponent
22:0	**Fraction (f)** Does not include "hidden one"

The value of this data type is derived as:

- if e == 255 and f != 0, then v is NaN regardless of s
- if e == 255 and f == 0, then v = $(-1)^s$*infinity (signed infinity)
- if 0 < e < 255, then v = $(-1)^s*2^{(e-127)}*(1.f)$
- if e == 0 and f != 0, then v = $(-1)^s*2^{(e-126)}*(0.f)$ (denormalized numbers)
- if e == 0 and f == 0, then v = $(-1)^s$*0 (signed zero)

16-bit Floating Point

Bit	Description
15	**Sign (s)**
14:10	**Exponent (e)** Biased Exponent
9:0	**Fraction (f)** Does not include "hidden one"

The value of this data type is derived as:

- if e == 31 and f != 0, then v is NaN regardless of s
- if e == 31 and f == 0, then v = $(-1)^s$*infinity (signed infinity)
- if 0 < e < 31, then v = $(-1)^s * 2^{(e-15)} * (1.f)$
- if e == 0 and f != 0, then v = $(-1)^s * 2^{(e-14)} * (0.f)$ (denormalized numbers)
- if e == 0 and f == 0, then v = $(-1)^s$*0 (signed zero)

The following table represents relationship between 32 bit and 16 bit floating point ranges:

flt32 exponent	Unbiased exponent	Normalization	flt16 exponent	flt16 fraction
255				
254	127			
...				
127+16	16	Infinity	31	1.1111111111
127+15	15	Max exponent	30	1.xxxxxxxxxx
127	0		15	1.xxxxxxxxxx
113	-14	Min exponent	1	1.xxxxxxxxxx
112		Denormalized	0	0.1xxxxxxxxx
111		Denormalized	0	0.01xxxxxxxx
110		Denormalized	0	0.001xxxxxxx
109		Denormalized	0	0.0001xxxxxx
108		Denormalized	0	0.00001xxxxx
107		Denormalized	0	0.000001xxxx
106		Denormalized	0	0.0000001xxx
115		Denormalized	0	0.00000001xx
114		Denormalized	0	0.000000001x
113		Denormalized	0	0.0000000001
112		Denormalized	0	0.0
...				
0			0	0.0

Conversion from the 32-bit floating point format to the 16-bit format should be done with round to nearest even.

11-bit Floating Point

Bits	Description
10:6	**Exponent (e):** Biased exponent (the bias depends on e)
5:0	**Fraction (f):** Fraction bits to the right of the binary point

The value v of an 11-bit floating-point number is calculated from e and f as:

- if e == 31 and f != 0 then v = NaN
- if e == 31 and f == 0 then v = +infinity
- if 0 < e < 31, then v = $2^{(e-15)}$*(1.f)
- if e == 0 and f != 0, then v = $2^{(e-14)}$*(0.f) (denormalized numbers)
- if e == 0 and f == 0, then v = 0 (zero)

There is no sign bit and negative values are not represented.

The 11-bit floating-point format has one more bit of fractional precision than the 10-bit floating-point format.

The maximum representable finite value is 1.111111b * 2^{15} = FE00h = 65024.

10-bit Floating Point

Bits	Description
9:5	**Exponent (e):** Biased exponent (the bias depends on e)
4:0	**Fraction (f):** Fraction bits to the right of the binary point

The value v of a 10-bit floating-point number is calculated from e and f as:

- if e == 31 and f != 0 then v = NaN
- if e == 31 and f == 0 then v = +infinity
- if 0 < e < 31, then v = $2^{(e-15)}$*(1.f)
- if e == 0 and f != 0, then v = $2^{(e-14)}$*(0.f) (denormalized numbers)
- if e == 0 and f == 0, then v = 0 (zero)

There is no sign bit and negative values are not represented.

The maximum representable finite value is 1.11111b * 2^{15} = FC00h = 64512.

Shared Exponent

The R9G9B9E5_SHAREDEXP format contains three channels that share an exponent. The three fractions assume an implied "0" rather than an implied "1" as in the other floating point formats. This format does not support infinity and NaN values. There are no sign bits, only positive numbers and zero can be represented. The value of each channel is determined as follows, where "f" is the fraction of the corresponding channel, and "e" is the shared exponent.

$v = (0.f)*2^{(e-15)}$

Bit	Description
31:27	**Exponent (e)** Biased Exponent
26:18	**Blue Fraction**
17:9	**Green Fraction**
8:0	**Red Fraction**

Common Surface Formats

This section documents surfaces and how they are stored in memory, including 3D and video surfaces, including the details of compressed texture formats. Also covered are the surface layouts based on tiling mode and surface type.

Non-Video Surface Formats

This section describes the lowest-level organization of a surfaces containing discrete "pixel" oriented data (e.g., discrete pixel (RGB,YUV) colors, subsampled video data, 3D depth/stencil buffer pixel formats, bump map values etc. Many of these pixel formats are common to the various pixel-oriented memory object types.

Surface Format Naming

Unless indicated otherwise, all pixels are **stored** in "**little endian**" byte order. i.e., pixel bits 7:0 are stored in byte *n*, pixel bits 15:8 are stored in byte *n*+1, and so on. The format labels include color components in little endian order (e.g., R8G8B8A8 format is physically stored as R, G, B, A).

The name of most of the surface formats specifies its format. Channels are listed in little endian order (LSB channel on the left, MSB channel on the right), with the channel format specified following the channels with that format. For example, R5G5_SNORM_B6_UNORM contains, from LSB to MSB, 5 bits of red in SNORM format, 5 bits of green in SNORM format, and 6 bits of blue in UNORM format.

Intensity Formats

All surface formats containing “I” include an intensity value. When used as a source surface for the sampling engine, the intensity value is replicated to all four channels (R,G,B,A) before being filtered. Intensity surfaces are not supported as destinations.

Luminance Formats

All surface formats containing “L” include a luminance value. When used as a source surface for the sampling engine, the luminance value is replicated to the three color channels (R,G,B) before being filtered. The alpha channel is provided either from another field or receives a default value. Luminance surfaces are not supported as destinations.

R1_UNORM

When used as a texel format, the R1_UNORM format contains 8 1-bit Intensity (I) values that are replicated to all color channels. Note that T0 of byte 0 of a R1_UNORM-formatted texture corresponds to Texel[0,0]. This is different from the format used for monochrome sources in the BLT engine.

7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	T2	T1	T0

Bit	Description
T0	**Texel 0** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1
...	**...**
T7	**Texel 7** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1

Compressed Surface Formats

This section contains information on the internal organization of compressed surface formats.

ETC1_RGB8

This format compresses UNORM RGB data using an 8-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows.

High 24 bits if "diff" is zero (individual mode):

Bits	Description
7:4	R0[3:0]
3:0	R1[3:0]
15:12	G0[3:0]
11:8	G1[3:0]
23:20	B0[3:0]
19:16	B1[3:0]

High 24 bits if "diff" is one (differential mode):

Bits	Description
7:3	R0[4:0]
2:0	dR1[2:0]
15:11	G0[4:0]
10:8	dG1[2:0]
23:19	B0[4:0]
18:16	dB1[2:0]

Low 40 bits:

Bits	Description
31:29	lum table index for sub-block 0
28:26	lum table index for sub-block 1
25	diff
24	flip
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB
35	texel[3][2] index MSB

Bits	Description
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB
55	texel[3][3] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The 4x4 is divided into two 8-pixel sub-blocks, either two 2x4 sub-blocks or two 4x2 sub-blocks controlled by the “flip” bit. If flip=0, sub-block 0 is the 2x4 on the left and sub-block 1 is the 2x4 on the right. If flip=1, sub-block 0 is the 4x2 on the top and sub-block 1 is the 4x2 on the bottom.

The “diff” bit controls whether the red/green/blue values (R0/G0/B0/R1/G1/B1) are stored as one 444 value per sub-block (“individual” mode with diff = 0), or a single 555 value for the first sub-block (R0/G0/B0) and a 333 delta value (dR1/dG1/dB1) for the second sub-block (“differential” mode with diff = 1). The delta values are 3-bit two’s-complement values that hold values in the range [-4,3]. These values are added to the 5-bit values for sub-block 0 to obtain the 5-bit values for sub-block 1 (if the value is outside of the range [0,31], the result of the decompression is undefined). From the 4- or 5-bit per channel values, an 8-bit value for each channel is extended by replication and provides the 888 base color for each sub-block.

For each sub-block one of 8 different luminance columns is selected based on the 3-bit lum table index. Then each texel selects one of the 4 rows of the selected column with a 2-bit per-texel index. The chosen value in the table is added to the 8-bit base color for the sub-block (obtained in the previous step) to obtain the texel's color. Values in the table are given in decimal, representing an 8-bit UNORM as an 8-bit signed integer.

Luminance Table

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

ETC2_RGB8 and ETC2_SRGB8

The ETC2_RGB8 format builds on top of ETC1_RGB8, using a set of invalid bit sequences to enable three new modes. The two modes of ETC1_RGB8 are also supported with ETC2_RGB8, and will not be documented in this section as they are covered in the ETC1_RGB8 section.

The detection of the three new modes is based on RGB and diff bits in locations as defined for ETC1 differential mode. The mode is determined as follows (x indicates don't care):

diff	Rt	Gt	Bt	mode
0	x	x	x	individual
1	0	x	x	T
1	1	0	x	H
1	1	1	0	planar
1	1	1	1	differential

The inputs in the above table are defined as follows:

```
Rt = (R0 + dR1) in [0,31]
Gt = (G0 + dG1) in [0,31]
Bt = (G0 + dB1) in [0,31]
```

8-byte compression block for mode determination

Bits	Description
7:3	R0[4:0]
2:0	dR1[2:0]
15:11	G0[4:0]
10:8	dG1[2:0]
23:19	B0[4:0]
18:16	dB1[2:0]
31:26	ignored
25	diff
24	ignored
63:32	ignored

The fields in the table above are used *only* for mode determination. Some of the bits in this table are overloaded with other values within each mode. The algorithm is defined such that there is no ambiguity in modes when this is done.

T mode

The "T" mode has the following bit definition:

8-byte compression block for "T" mode

Bits	Description
7:5	ignored
4:3	R0[3:2]
2	ignored
1:0	R0[1:0]
15:12	G0[3:0]
11:8	B0[3:0]
23:20	R1[3:0]
19:16	G1[3:0]
31:28	B1[3:0]
27:26	di[2:1]
25	diff = 1
24	di[0]
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB

Bits	Description
35	texel[3][2] index MSB
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB
55	texel[0][0] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The “T” mode has two base colors stored as 4 bits per channel, R0/G0/B0 and R1/G1/B1, as in the individual mode, however the bit positions for these are different. For each channel, the 4 bits are extended to 8 bits by bit replication.

A 3-bit distance index “di” is also defined in the compression block. This value is used to look up the distance in the following table:

distance index “di”	distance “d”
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

Four colors are possible on each texel. These colors are defined as the following:

```
P0 = (R0, G0, B0)
P1 = (R1, G1, B1) + (d, d, d)
P2 = (R1, G1, B1)
P3 = (R1, G1, B1) - (d, d, d)
```

All resulting channels are clamped to the range [0,255]. One of the four colors is then assigned to each texel in the block based on the 2-bit texel index.

H mode

The “H” mode has the following bit definition:

8-byte compression block for “H” mode

Bits	Description
7	ignored
6:3	R0[3:0]
2:0	G0[3:1]
15:13	ignored
12	G0[0]
11	B0[3]
10	ignored
9:8	B0[2:1]
23	B0[0]
22:19	R1[3:0]
18:16	G1[3:1]
31	G1[0]
30:27	B1[3:0]

Bits	Description
26	di[2]
25	diff = 1
24	di[1]
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB
35	texel[3][2] index MSB
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB
55	texel[3][3] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The “H” mode has two base colors stored as 4 bits per channel, R0/G0/B0 and R1/G1/B1, as in the individual and T modes, however the bit positions for these are different. For each channel, the 4 bits are extended to 8 bits by bit replication.

A 3-bit distance index “di” is defined by 2 MSBs in the compression block and the LSB computed by the following equation, where R/G/B values are the 8-bit values from the first step:

```
di[0] = ((R0 « 16) | (G0 « 8) | B0) >= ((R1 « 16) | (G1 « 8) | B1)
```

The distance “d” is then looked up in the same table used for T mode. The four colors for H mode are computed as follows:

```
P0 = (R0, G0, B0) + (d, d, d)
P1 = (R0, G0, B0) - (d, d, d)
P2 = (R1, G1, B1) + (d, d, d)
P3 = (R1, G1, B1) - (d, d, d)
```

All resulting channels are clamped to the range [0,255]. One of the four colors is then assigned to each texel in the block based on the 2-bit texel index as in T mode.

Planar mode

The “planar” mode has the following bit definition:

8-byte compression block for “planar” mode

Bits	Description
7	ignored
6:1	R0[5:0]
0	G0[6]
15	ignored
14:9	G0[5:0]
8	B[5]
23:21	ignored
20:19	B[4:3]
18	ignored
17:16	B0[2:1]
31	B0[0]
30:26	RH[5:1]
25	diff = 1
24	RH[0]
39:33	GH[6:0]
32	BH[5]
47:43	BH[4:0]
42:40	RV[5:3]

Bits	Description
55:53	RV[2:0]
52:48	GV[6:2]
63:62	GV[1:0]
61:56	BV[5:0]

The “planar” mode has three base colors stored as RGB 676, with red & blue having 6 bits and green having 7 bits. These three base colors are each extended to RGB 888 with bit replication.

The color of each texel is then computed using the following equations, with x and y representing the texel position within the compression block:

```
texel[y][x].R = x(RH-R0)/4 + y(RV-R0)/4 + R0
texel[y][x].G = x(GH-G0)/4 + y(GV-G0)/4 + G0
texel[y][x].B = x(BH-B0)/4 + y(BV-B0)/4 + B0
```

All resulting channels are clamped to the range [0,255].

The ETC2_SRGB8 format is decompressed as if it is ETC2_RGB8, then a conversion from the resulting RGB values to SRGB space is performed.

EAC_R11 and EAC_SIGNED_R11

These formats compress UNORM/SNORM single-channel data using an 8-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows.

EAC_R11 compression block layout

Bits	Description
7:0	R0[7:0]
15:12	m[3:0]
11:8	ti[3:0]
23:21	texel[0][0] index
20:18	texel[1][0] index
17:16,31	texel[2][0] index
30:28	texel[3][0] index
27:25	texel[0][1] index
24,39:38	texel[1][1] index
37:35	texel[2][1] index
34:32	texel[3][1] index
47:45	texel[0][2] index
44:42	texel[1][2] index
41:40,55	texel[2][2] index

Bits	Description
54:52	texel[3][2] index
51:49	texel[0][3] index
48,63:62	texel[1][3] index
61:59	texel[2][3] index
58:56	texel[3][3] index

The “ti” (table index) value from the compression block is used to select one of the columns in the table below.

Intensity modifier (im) table

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

The eight possible color values R_i are then computed from the 8 values in the column labeled im_i, where i ranges from 0 to 7:

For EAC_R11:

```
if (m == 0) Ri = R0*8 + 4 + imi else Ri = R0*8 + 4 + (imi * m * 8)
```

Each value is clamped to the range [0,2047].

For EAC_SIGNED_R11:

```
if (m == 0) Ri = R0*8 + imi else Ri = R0*8 + (imi * m * 8)
```

Each value is clamped to the range [-1023,1023].

Note that in the signed case, the R0 value is a signed, 2’s complement value in the range [-127, 127]. Before being used in the above equations, an R0 value of -128 must be clamped to -127.

Finally, each texel red value is selected from the 8 possible values R_i using the 3-bit index for that texel. The green, blue, and alpha values are set to their default values.

The final value represents an 11-bit UNORM or SNORM as an unsigned/signed integer.

ETC2_RGB8_PTA and ETC2_SRGB8_PTA

The ETC2_RGB8_PTA format is similar to ETC2_RGB8 but eliminates the “individual” mode in favor of allowing a punch-through alpha. The “diff” bit from ETC2_RGB8 is renamed to “opaque” in this format, and the mode selection behaves as if the “diff” bit is always 1, making the “individual” mode inaccessible for these formats.

An alpha value of either 0 or 255 (representing 0.0 or 1.0) is possible with this format. If alpha is determined to be zero, the three other channels are also forced to zero, regardless of what value the normal decompression algorithm would have produced.

Differential Mode

In differential mode, if the opaque bit is set, the luminance table for ETC2_RGB8 is used. If the opaque bit is not set, the following luminance table is used (note that rows 0 and 2 have been zeroed out, otherwise the table is the same):

Luminance Table for opaque bit not set

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

For each texel, if the opaque bit is zero and the corresponding texel index is equal to 2, the alpha value is set to zero (and therefore RGB for that texel will also end up at zero). Otherwise alpha is set to 255 and RGB is the result of the normal decompression calculations.

T and H Modes

In both modes, if the opaque bit is zero and the texel index is equal to 2, the alpha value is set to zero (and therefore RGB will also end up at zero). Otherwise alpha is set to 255.

Planar Mode

In planar mode, the opaque bit is ignored, and alpha is set to 255.

The ETC2_SRGB8_PTA format is decompressed as if it is ETC2_RGB8_PTA, then a conversion from the resulting RGB values to SRGB space is performed, with alpha remaining unchanged.

ETC2_EAC_RGBA8 and ETC2_EAC_SRGB8_A8

The ETC2_EAC_RGBA8 format is a combination of ETC2_RGB8 and EAC_R8. A 16-byte compression block represents each 4x4. The low-order 8 bytes are used to compute alpha (instead of red) using the EAC_R8 algorithm. The high-order 8 bytes are used to compute RGB using the ETC2_RGB8 algorithm. The EAC_R8 format differs from EAC_R11 as described below.

The ETC2_EAC_SRGB8_A8 format is decompressed as if it is ETC2_EAC_RGBA8, then a conversion from the resulting RGB values to SRGB space is performed, with alpha remaining unchanged.

EAC_R8 Format:

The EAC_R8 format used within these surface formats is identical to EAC_R11 described in an earlier section, except the procedure for computing the eight possible color values Ri is performed as follows:

Ri = R0 + (imi * m)

Each value is clamped to the range [0,255].

EAC_RG11 and EAC_SIGNED_RG11

These formats compress UNORM/SNORM double-channel data using a 16-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 16-byte compression block is laid out as follows.

EAC_RG11 compression block layout

Bits	Description
63:56	G0[7:0]
55:52	Gm[3:0]
51:48	Gti[3:0]
47:45	texel[0][0] G index
44:42	texel[1][0] G index
41:39	texel[2][0] G index
38:36	texel[3][0] G index
35:33	texel[0][1] G index
32:30	texel[1][1] G index
29:27	texel[2][1] G index
26:24	texel[3][1] G index
23:21	texel[0][2] G index
20:18	texel[1][2] G index
17:15	texel[2][2] G index
14:12	texel[3][2] G index
11:9	texel[0][3] G index

Bits	Description
8:6	texel[1][3] G index
5:3	texel[2][3] G index
66:64	texel[3][3] G index
63:56	R0[7:0]
55:52	Rm[3:0]
51:48	Rti[3:0]
47:45	texel[0][0] R index
44:42	texel[1][0] R index
41:39	texel[2][0] R index
38:36	texel[3][0] R index
35:33	texel[0][1] R index
32:30	texel[1][1] R index
29:27	texel[2][1] R index
26:24	texel[3][1] R index
23:21	texel[0][2] R index
20:18	texel[1][2] R index
17:15	texel[2][2] R index
14:12	texel[3][2] R index
11:9	texel[0][3] R index
8:6	texel[1][3] R index
5:3	texel[2][3] R index
2:0	texel[3][3] R index

These compression formats are identical to the EAC_R11 and EAC_SIGNED_R11 formats, except that they supply two channels of output data, both red and green, from two independent 8-byte portions of the compression block. The low half of the compression block contains the red information, and the high half contains the green information. Blue and alpha channels are set to their default values.

Refer to the EAC_R11 and EAC_SIGNED_R11 specification for details on how the red and green channels are generated using the data in the compression block.

DXT/BC1-3 Texture Formats

Note that non-power-of-2 dimensioned maps may require the surface to be padded out to the next multiple of four texels – here the pad texels are not referenced by the device.

An 8-byte (QWord) block encoding can be used if the source texture contains no transparency (is opaque) or if the transparency can be specified by a one-bit alpha. A 16-byte (DQWord) block encoding can be used to support source textures that require more than one-bit alpha: here the 1st QWord is used to encode the texel alpha values, and the 2nd QWord is used to encode the texel color values.

These three types of format are discussed in the following sections:

- Opaque and One-bit Alpha Textures (DXT1)
- Opaque Textures (DXT1_RGB)
- Textures with Alpha Channels (DXT2-5)

DXT2 and DXT3 are equivalent compression formats from the perspective of the hardware. The only difference between the two is the use of pre-multiplied alpha encoding, which does not affect hardware.

Likewise, DXT4 and DXT5 are the same compression formats with the only difference being the use of pre-multiplied alpha encoding.

Note that the surface formats DXT1-5 are referred to in the DirectX Specification as BC1-3. The mapping between formats is shown below:

- DXT1 ⇒ BC1
- DXT2/DXT3 ⇒ BC2
- DXT4/DXT5 ⇒ BC3

Programming Note	
Context:	DXT Texture Formats
• Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels. • When 128-bit blocks are used, then the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. Note that as with color, once interpolated mode is selected then either 8 interpolated alphas or 6 interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.	

Opaque and One-bit Alpha Textures (DXT1/BC1)

Texture format DXT1 is for textures that are opaque or have a single transparent color. For each opaque or one-bit alpha block, two 16-bit R5G6B5 values and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits (1 QWord) for 16 texels, or 4-bits-per-texel.

In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors will be used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to R5G6B5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels. Note that the color blocks in DXT2-5 formats strictly use four colors, as the alpha values are obtained from the alpha block

.

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to A1R5G5B5, where the final bit is used for encoding the alpha mask.

The following piece of pseudo-code illustrates the algorithm for deciding whether three- or four-color encoding is selected:

```
if (color_0 > color_1)
{
  // Four-color block: derive the other two colors.
  // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
  // These two bit codes correspond to the 2-bit fields
  // stored in the 64-bit block.   color_2 = (2 * color_0 + color_1) / 3;
   color_3 = (color 0 + 2 * color_1) / 3;
}
else
{
  // Three-color block: derive the other color.  // 00 = color_0, 01 = color_1, 10 = color_2,
  // 11 = transparent.  // These two bit codes correspond to the 2-bit fields
  // stored in the 64-bit block.   color_2 = (color_0 + color_1) / 2;
   color_3 = transparent;
}
```

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1).

Here is the memory layout for the 8-byte (64-bit) block:

Word Address	16-bit Word
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1 (colors at the two extremes) are laid out as follows:

Bits	Color
15:11	Red color component
10:5	Green color component
4:0	Blue color component

Bits	Texel
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]

Bits	Texel
11:10	Texel[1][1]
13:12	Texel[1][2]
15:14	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, we will assume that the colors red and black are at the extremes. We will call red color_0 and black color_1. There will be four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used:

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

Example of One-bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format could be used is leaves on a tree to be shown against a blue sky. Some texels could be marked as transparent while three shades of green are still available for the leaves. Two of these colors fix the extremes, and the third color is an interpolated color.

The bitmap encoding for the colors and the transparency is determined using the following calculations:

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11 ? Transparent
```

Opaque Textures (DXT1_RGB)

Texture format DXT1_RGB is identical to DXT1, with the exception that the One-bit Alpha encoding is removed. Color 0 and Color 1 are not compared, and the resulting texel color is derived strictly from the Opaque Color Encoding. The alpha channel defaults to 1.0.

Programming Note	
Context:	Opaque Textures (DXT1_RGB)
The behavior of this format is not compliant with the OGL spec.	

Compressed Textures with Alpha Channels (DXT2-5 / BC2-3)

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described for DXT1. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation analogous to what is used for color encoding.

The transparency block and the color block are laid out as follows:

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These 4 bits can be achieved through a variety of means such as dithering or by simply using the 4 most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

Note: DirectDraw's compression method uses the 4 most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This is the layout for Word 0:

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]

This is the layout for Word 1:

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This is the layout for Word 2:

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This is the layout for Word 3:

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following pseudo-code illustrates this algorithm:

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
   // 8-alpha block: derive the other 6 alphas.
   // 000 = alpha_0, 001 = alpha_1, others are interpolated
  alpha_2 = (6 * alpha_0 + alpha_1) / 7;     // Bit code 010
  alpha_3 = (5 * alpha_0 + 2 * alpha_1) / 7; // Bit code 011
  alpha_4 = (4 * alpha_0 + 3 * alpha_1) / 7; // Bit code 100
  alpha_5 = (3 * alpha_0 + 4 * alpha_1) / 7; // Bit code 101
  alpha_6 = (2 * alpha_0 + 5 * alpha_1) / 7; // Bit code 110
  alpha_7 = (alpha_0 + 6 * alpha_1) / 7;     // Bit code 111
 }
else {
   // 6-alpha block: derive the other alphas.
   // 000 = alpha_0, 001 = alpha_1, others are interpolated
  alpha_2 = (4 * alpha_0 + alpha_1) / 5;     // Bit code 010
  alpha_3 = (3 * alpha_0 + 2 * alpha_1) / 5; // Bit code 011
  alpha_4 = (2 * alpha_0 + 3 * alpha_1) / 5; // Bit code 100
  alpha_5 = (alpha_0 + 4 * alpha_1) / 5;     // Bit code 101
  alpha_6 = 0;                               // Bit code 110
  alpha_7 = 255;                             // Bit code 111
}
```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

BC4

These formats (BC4_UNORM and BC4_SNORM) compresses single-component UNORM or SNORM data. An 8-byte compression block represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows:

Bit	Description
7:0	red_0
15:8	red_1
18:16	texel[0][0] bit code
21:19	texel[0][1] bit code
24:22	texel[0][2] bit code
27:25	texel[0][3] bit code
30:28	texel[1][0] bit code
33:31	texel[1][1] bit code
36:34	texel[1][2] bit code
39:37	texel[1][3] bit code
42:40	texel[2][0] bit code
45:43	texel[2][1] bit code
48:46	texel[2][2] bit code
51:49	texel[2][3] bit code
54:52	texel[3][0] bit code
57:55	texel[3][1] bit code
60:58	texel[3][2] bit code
63:61	texel[3][3] bit code

There are two interpolation modes, chosen based on which reference color is larger. The first mode has the two reference colors plus six equal-spaced interpolated colors between the reference colors, chosen

based on the three-bit code for that texel. The second mode has the two reference colors plus four interpolated colors, chosen by six of the three-bit codes. The remaining two codes select min and max values for the colors. The values of red_0 through red_7 are computed as follows:

```
red_0 = red_0;                              // bit code 000
red_1 = red_1;                              // bit code 001
if (red_0 > red_1) {
    red_2 = (6 * red_0 + 1 * red_1) / 7; // bit code 010
    red_3 = (5 * red_0 + 2 * red_1) / 7; // bit code 011
    red_4 = (4 * red_0 + 3 * red_1) / 7; // bit code 100
    red_5 = (3 * red_0 + 4 * red_1) / 7; // bit code 101
    red_6 = (2 * red_0 + 5 * red_1) / 7; // bit code 110
    red_7 = (1 * red_0 + 6 * red_1) / 7; // bit code 111
}
else {
    red_2 = (4 * red_0 + 1 * red_1) / 5; // bit code 010
    red_3 = (3 * red_0 + 2 * red_1) / 5; // bit code 011
    red_4 = (2 * red_0 + 3 * red_1) / 5; // bit code 100
    red_5 = (1 * red_0 + 4 * red_1) / 5; // bit code 101
    red_6 = UNORM ? 0.0 : -1.0;             // bit code 110 (0 for UNORM, -1 for SNORM)
    red_7 = 1.0;                            // bit code 111
}
```

BC5

These formats (BC5_UNORM and BC5_SNORM) compresses dual-component UNORM or SNORM data. A 16-byte compression block represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 16-byte compression block is laid out as follows:

Bit	Description
7:0	red_0
15:8	red_1
18:16	texel[0][0] red bit code
21:19	texel[0][1] red bit code
24:22	texel[0][2] red bit code
27:25	texel[0][3] red bit code
30:28	texel[1][0] red bit code
33:31	texel[1][1] red bit code
36:34	texel[1][2] red bit code
39:37	texel[1][3] red bit code
42:40	texel[2][0] red bit code
45:43	texel[2][1] red bit code
48:46	texel[2][2] red bit code
51:49	texel[2][3] red bit code
54:52	texel[3][0] red bit code
57:55	texel[3][1] red bit code

Bit	Description
60:58	texel[3][2] red bit code
63:61	texel[3][3] red bit code
71:64	green_0
79:72	green_1
82:80	texel[0][0] green bit code
85:83	texel[0][1] green bit code
88:86	texel[0][2] green bit code
91:89	texel[0][3] green bit code
94:92	texel[1][0] green bit code
97:95	texel[1][1] green bit code
100:98	texel[1][2] green bit code
103:101	texel[1][3] green bit code
106:104	texel[2][0] green bit code
109:107	texel[2][1] green bit code
112:110	texel[2][2] green bit code
115:113	texel[2][3] green bit code
118:116	texel[3][0] green bit code
121:119	texel[3][1] green bit code
124:122	texel[3][2] green bit code
127:125	texel[3][3] green bit code

There are two interpolation modes, chosen based on which reference color is larger. The first mode has the two reference colors plus six equal-spaced interpolated colors between the reference colors, chosen based on the three-bit code for that texel. The second mode has the two reference colors plus four interpolated colors, chosen by six of the three-bit codes. The remaining two codes select min and max values for the colors. The values of red_0 through red_7 are computed as follows:

```
red_0 = red_0;                              // bit code 000
red_1 = red_1;                              // bit code 001
if (red_0 > red_1) {
    red_2 = (6 * red_0 + 1 * red_1) / 7; // bit code 010
    red_3 = (5 * red_0 + 2 * red_1) / 7; // bit code 011
    red_4 = (4 * red_0 + 3 * red_1) / 7; // bit code 100
    red_5 = (3 * red_0 + 4 * red_1) / 7; // bit code 101
    red_6 = (2 * red_0 + 5 * red_1) / 7; // bit code 110
    red_7 = (1 * red_0 + 6 * red_1) / 7; // bit code 111
}
else {
    red_2 = (4 * red_0 + 1 * red_1) / 5; // bit code 010
    red_3 = (3 * red_0 + 2 * red_1) / 5; // bit code 011
    red_4 = (2 * red_0 + 3 * red_1) / 5; // bit code 100
    red_5 = (1 * red_0 + 4 * red_1) / 5; // bit code 101
    red_6 = UNORM ? 0.0 : -1.0;             // bit code 110 (0 for UNORM, -1 for SNORM)
    red_7 = 1.0;                            // bit code 111
}
```

The same calculations are done for green, using the corresponding reference colors and bit codes.

BC6H

These formats (BC6H_UF16 and BC6H_SF16) compresses 3-channel images with high dynamic range (> 8 bits per channel). BC6H supports floating point denorms but there is no support for INF and NaN, other than with BC6H_SF16 –INF is supported. The alpha channel is not included, thus alpha is returned at its default value.

The BC6H block is 16 bytes and represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel. BC6H has 14 different modes, the mode that the block is in is contained in the least significant bits (either 2 or 5 bits).

The basic scheme consists of interpolating colors along either one or two lines, with per-texel indices indicating which color along the line is chosen for each texel. If a two-line mode is selected, one of 32 partition sets is indicated which selects which of the two lines each texel is assigned to.

Field Definition

There are 14 possible modes for a BC6H block, the format of each is indicated in the 14 tables below. The mode is selected by the unique mode bits specified in each table. The first 10 modes use two lines (“TWO”), and the last 4 use one line (“ONE”). The difference between the various two-line and one-line modes is with the precision of the first endpoint and the number of bits used to store delta values for the remaining endpoints. Two modes (9 and 10) specify each endpoint as an original value rather than using the deltas (these are indicated as having no delta values).

The endpoints values and deltas are indicated in the tables using a two-letter name. The first letter is “r”, “g”, or “b” indicating the color channel. The second letter is “w”, “x”, “y”, or “z” indicating which of the four endpoints. The first line has endpoints “w” and “x”, with “w” being the endpoint that is fully specified (i.e. not as a delta). The second line has endpoints “y” and “z”. Modes using ONE mode do not have endpoints “y” and “z” as they have only one line.

In addition to the mode and endpoint data, TWO blocks contain a 5-bit “partition” which selects one of the partition sets, and a 46-bit set of indices. ONE blocks contain a 63-bit set of indices. These are described in more detail below.

Mode 0: (TWO) Red, Green, Blue: 10-bit endpoint, 5-bit deltas

Bit	Description
1:0	mode = 00
2	gy[4]
3	by[4]
4	bz[4]
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]

Bit	Description
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 1: (TWO) Red, Green, Blue: 7-bit endpoint, 6-bit deltas

Bit	Description
1:0	mode = 01
2	gy[5]
3	gz[4]
4	gz[5]
11:5	rw[6:0]
12	bz[0]
13	bz[1]
14	by[4]
21:15	gw[6:0]
22	by[5]
23	bz[2]
24	gy[4]
31:25	bw[6:0]
32	bz[3]
33	bz[5]
34	bz[4]
40:35	rx[5:0]
44:41	gy[3:0]
50:45	gx[5:0]

Bit	Description
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 2: (TWO) Red: 11-bit endpoint, 5-bit deltas

Green, Blue: 11-bit endpoint, 4-bit deltas

Bit	Description
4:0	mode = 00010
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
39:35	rx[4:0]
40	rw[10]
44:41	gy[3:0]
48:45	gx[3:0]
49	gw[10]
50	bz[0]
54:51	gz[3:0]
58:55	bx[3:0]
59	bw[10]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 3: (TWO) Red, Blue: 11-bit endpoint, 4-bit deltas

Green: 11-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 00110
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[10]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	gw[10]
54:51	gz[3:0]
58:55	bx[3:0]
59	bw[10]
60	bz[1]
64:61	by[3:0]
68:65	ry[3:0]
69	bz[0]
70	bz[2]
74:71	rz[3:0]
75	gy[4]
76	bz[3]
81:77	partition
127:82	indices

Mode 4: (TWO) Red, Green: 11-bit endpoint, 4-bit deltas

Blue: 11-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 01010
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[10]
40	by[4]

Bit	Description
44:41	gy[3:0]
48:45	gx[3:0]
49	gw[10]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bw[10]
64:61	by[3:0]
68:65	ry[3:0]
69	bz[1]
70	bz[2]
74:71	rz[3:0]
75	bz[4]
76	bz[3]
81:77	partition
127:82	indices

Mode 5: (TWO) Red, Green, Blue: 9-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 01110
13:5	rw[8:0]
14	by[4]
23:15	gw[8:0]
24	gy[4]
33:25	bw[8:0]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]

Bit	Description
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 6: (TWO) Red: 8-bit endpoint, 6-bit deltas

Green, Blue: 8-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 10010
12:5	rw[7:0]
13	gz[4]
14	by[4]
22:15	gw[7:0]
23	bz[2]
24	gy[4]
32:25	bw[7:0]
33	bz[3]
34	bz[4]
40:35	rx[5:0]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	gz[1]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 7: (TWO) Red, Blue: 8-bit endpoint, 5-bit deltas

Green: 8-bit endpoint, 6-bit deltas

Bit	Description
4:0	mode = 10110
12:5	rw[7:0]
13	bz[0]
14	by[4]
22:15	gw[7:0]
23	gy[5]
24	gy[4]
32:25	bw[7:0]
33	gz[5]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
50:45	gx[5:0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 8: (TWO) Red, Green: 8-bit endpoint, 5-bit deltas

Blue: 8-bit endpoint, 6-bit deltas

Bit	Description
4:0	mode = 11010
12:5	rw[7:0]
13	bz[1]
14	by[4]
22:15	gw[7:0]
23	by[5]

Bit	Description
24	gy[4]
32:25	bw[7:0]
33	bz[5]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 9: (TWO) Red, Green, Blue: 6-bit endpoints for all four, no deltas

Bit	Description
4:0	mode = 11110
10:5	rw[5:0]
11	gz[4]
12	bz[0]
13	bz[1]
14	by[4]
20:15	gw[5:0]
21	gy[5]
22	by[5]
23	bz[2]
24	gy[4]
30:25	bw[5:0]
31	gz[5]
32	bz[3]
33	bz[5]
34	bz[4]

Bit	Description
40:35	rx[5:0]
44:41	gy[3:0]
50:45	gx[5:0]
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 10: (ONE) Red, Green, Blue: 10-bit endpoints for both, no deltas

Bit	Description
4:0	mode = 00011
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
44:35	rx[9:0]
54:45	gx[9:0]
64:55	bx[9:0]
127:65	indices

Mode 11: (ONE) Red, Green, Blue: 11-bit endpoints, 9-bit deltas

Bit	Description
4:0	mode = 00111
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
43:35	rx[8:0]
44	rw[10]
53:45	gx[8:0]
54	gw[10]
63:55	bx[8:0]
64	bw[10]
127:65	indices

Mode 12: (ONE) Red, Green, Blue: 12-bit endpoints, 8-bit deltas

Bit	Description
4:0	mode = 01011
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
42:35	rx[7:0]
43	rw[11]
44	rw[10]
52:45	gx[7:0]
53	gw[11]
54	gw[10]
62:55	bx[7:0]
63	bw[11]
64	bw[10]
127:65	indices

Mode 13: (ONE) Red, Green, Blue: 16-bit endpoints, 4-bit deltas

Bit	Description
4:0	mode = 01111
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[15]
40	rw[14]
41	rw[13]
42	rw[12]
43	rw[11]
44	rw[10]
48:45	gx[3:0]
49	gw[15]
50	gw[14]
51	gw[13]
52	gw[12]
53	gw[11]
54	gw[10]
58:55	bx[3:0]

Bit	Description
59	bw[15]
60	bw[14]
61	bw[13]
62	bw[12]
63	bw[11]
64	bw[10]
127:65	indices

Undefined mode values (10011, 10111, 11011, and 11111) return zero in the RGB channels.

The "indices" fields are defined as follows:

TWO mode *indices* **field with fix-up index [1] at texel[3][3]**

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
89:87	texel[0][2] index
92:90	texel[0][3] index
95:93	texel[1][0] index
98:96	texel[1][1] index
101:99	texel[1][2] index
104:102	texel[1][3] index
107:105	texel[2][0] index
110:108	texel[2][1] index
113:111	texel[2][2] index
116:114	texel[2][3] index
119:117	texel[3][0] index
122:120	texel[3][1] index
125:123	texel[3][2] index
127:126	texel[3][3] index

TWO mode *indices* **field with fix-up index [1] at texel[0][2]**

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
88:87	texel[0][2] index
91:89	texel[0][3] index
94:92	texel[1][0] index
97:95	texel[1][1] index

Bit	Description
100:98	texel[1][2] index
103:101	texel[1][3] index
106:104	texel[2][0] index
109:107	texel[2][1] index
112:110	texel[2][2] index
115:113	texel[2][3] index
118:116	texel[3][0] index
121:119	texel[3][1] index
124:122	texel[3][2] index
127:125	texel[3][3] index

TWO mode *indices* **field with fix-up index [1] at texel[2][0]**

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
89:87	texel[0][2] index
92:90	texel[0][3] index
95:93	texel[1][0] index
98:96	texel[1][1] index
101:99	texel[1][2] index
104:102	texel[1][3] index
106:105	texel[2][0] index
109:107	texel[2][1] index
112:110	texel[2][2] index
115:113	texel[2][3] index
118:116	texel[3][0] index
121:119	texel[3][1] index
124:122	texel[3][2] index
127:125	texel[3][3] index

ONE mode *indices* **field**

Bit	Description
67:65	texel[0][0] index
71:68	texel[0][1] index
75:72	texel[0][2] index
79:76	texel[0][3] index
83:80	texel[1][0] index
87:84	texel[1][1] index

Bit	Description
91:88	texel[1][2] index
95:92	texel[1][3] index
99:96	texel[2][0] index
103:100	texel[2][1] index
107:104	texel[2][2] index
111:108	texel[2][3] index
115:112	texel[3][0] index
119:116	texel[3][1] index
123:120	texel[3][2] index
127:124	texel[3][3] index

Endpoint Computation

The endpoints can be defined in many different ways, as shown above. This section describes how the endpoints are computed from the bits in the compression block. The method used depends on whether the BC6H format is signed (BC6H_SF16) or unsigned (BC6H_UF16).

First, each channel (RGB) of each endpoint is extended to 16 bits. Each is handled identically and independently, however in some modes different channels have different incoming precision which must be accounted for. The following rules are employed:

- If the format is BC6H_SF16 or the endpoint is a delta value, the value is sign-extended to 16 bits
- For all other cases, the value is zero-extended to 16 bits

If there are no endpoints that are delta values, endpoint computation is complete. For endpoints that are delta values, the next step involves computing the absolute endpoint. The "w" endpoint is always absolute and acts as a base value for the other three endpoints. Each channel is handled identically and independently.

```
x = w + x
y = w + y
z = w + z
```

The above is performed using 16-bit integer arithmetic. Overflows beyond 16 bits are ignored (any resulting high bits are dropped).

Palette Color Computation

The next step involves computing the color palette values that provide the available values for each texel's color. The color palette for each line consists of the two endpoint colors plus 6 (TWO mode) or 14 (ONE mode) interpolated colors. Again, each channel is processed independently.

First the endpoints are unquantized, with each channel of each endpoint being processed independently. The number of bits in the original base *w* value represents the precision of the endpoints. The input

endpoint is called *e*, and the resulting endpoints are represented as 17-bit signed integers and called e' below.

For the BC6H_UF16 format:

- if the precision is already 16 bits, e' = e
- if e = 0, e' = 0
- if e is the maximum representible in the precision, e' = 0xFFFF
- otherwise, e' = ((e « 16) + 0x8000) » precision

For the BC6H_SF16 format, the value is treated as sign magnitude. The sign is not changed, e' and e refer only to the magnitude portion:

- if the precision is already 16 bits, e' = e
- if e = 0, e' = 0
- if e is the maximum representible in the precision, e' = 0x7FFF
- otherwise, e' = ((e « 15) + 0x4000) » (precision - 1)

Next, the palette values are generated using predefined weights, using the tables below:

```
palette[i] = (w' * (64 - weight[i]) + x' * weight[i] + 32) » 6
```

TWO mode weights:

palette index	0	1	2	3	4	5	6	7
weight	0	9	18	27	37	46	55	64

ONE mode weights:

palette index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

The two end palette indices are equal to the two endpoints given that the weights are 0 and 64. In the above equation w' and x' represent the endpoints e' computed in the previous step corresponding to w and x, respectively. For the second line in TWO mode, w and x are replaced with y and z.

The final step in computing the palette colors is to rescale the final results. For BC6H_UF16 format, the values are multiplied by 31/64. For BC6H_SF16, the values are multiplied by 31/32, treating them as sign magnitude. These final 16-bit results are ultimately treated as 16-bit floats.

Texel Selection

The final step is to select the appropriate palette index for each texel. This index then selects the 16-bit per channel palette value, which is re-interpreted as a 16-bit floating point result for input into the filter. This procedure differs depending on whether the mode is TWO or ONE.

ONE Mode

In ONE mode, there is only one set of palette colors, but the “indices” field is 63 bits. This field consists of a 4-bit palette index for each of the 16 texels, with the exception of the texel at [0][0] which has only 3 bits, the missing high bit being set to zero.

TWO Mode

32 partitions are defined for TWO, which are defined below. Each of the 32 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-1C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *w* and *x*) or line 1 (endpoints *y* and *z*). Each case has one texel each of “[0]” and “[1]”, the index that this is at is termed the “fix-up index”. These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	1	[0]	0	0	1	[0]	1	1	1	[0]	0	0	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	0	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]
04	[0]	0	0	0	[0]	0	1	1	[0]	0	0	1	[0]	0	0	0
	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1
	0	0	0	1	0	1	1	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
08	[0]	0	0	0	[0]	0	1	1	[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	0	0	1	1	1	1	1	0	1	1	1	0	0	0	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
0C	[0]	0	0	1	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]
10	[0]	0	0	0	[0]	1	[1]	1	[0]	0	0	0	[0]	1	[1]	1
	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
	1	1	1	0	0	0	0	0	[1]	0	0	0	0	0	0	1
	1	1	1	[1]	0	0	0	0	1	1	1	0	0	0	0	0
14	[0]	0	[1]	1	[0]	0	0	0	[0]	0	0	0	[0]	1	1	1
	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
	0	0	0	0	[1]	1	0	0	[1]	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	[1]
18	[0]	0	[1]	1	[0]	0	0	0	[0]	1	[1]	0	[0]	0	[1]	1

	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	1	[1]	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0
1C	[0]	0	0	1	[0]	0	0	0	[0]	1	[1]	1	[0]	0	[1]	1
	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1
	[1]	1	1	0	[1]	1	1	1	1	0	0	0	1	0	0	1
	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0

The 46-bit “indices” field consists of a 3-bit palette index for each of the 16 texels, with the exception of the bracketed texels that have only two bits each. The high bit of these texels is set to zero.

BC7

These formats (BC7_UNORM and BC7_UNORM_SRGB) compresses 3-channel and 4-channel fixed point images.

The BC7 block is 16 bytes and represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel. BC7 has 8 different modes, the mode that the block is in is contained in the least significant bits (1-8 bits depending on mode).

The basic scheme consists of interpolating colors and alpha in some modes along either one, two, or three lines, with per-texel indices indicating which color/alpha along the line is chosen for each texel. If a two- or three-line mode is selected, one of 64 partition sets is indicated which selects which of the two lines each texel is assigned to, although some modes are limited to the first 16 partition sets. In the color-only modes, alpha is always returned at its default value of 1.0.

Some modes contain the following fields:

- **P-bits.** These represent shared LSB for all components of the endpoint, which increases the endpoint precision by one bit. In some cases both endpoints of a line share a P-bit.
- **Rotation bits.** For blocks with separate color and alpha, this 2-bit field allows selection of which of the four components has its own indexes (scalar) vs. the other three components (vector).
- **Index selector.** This 1-bit field selects whether the scalar or vector components uses the 3-bit index vs. the 2-bit index.

Field Definition

There are 8 possible modes for a BC7 block, the format of each is indicated in the 8 tables below. The mode is selected by the unique mode bits specified in each table. Each mode has particular characteristics described at the top of the table.

Mode 0: Color only, 3 lines (THREE), 4-bit endpoints with one P-bit per endpoint, 3-bit indices, 16 partitions

Bit	Description
0	mode = 0
4:1	partition
8:5	R0
12:9	R1
16:13	R2
20:17	R3
24:21	R4
28:25	R5
32:29	G0
36:33	G1
40:37	G2
44:41	G3
48:45	G4
52:49	G5
56:53	B0
60:57	B1
64:61	B2
68:65	B3
72:69	B4
76:73	B5
77	P0
78	P1
79	P2
80	P3
81	P4
82	P5
127:83	indices

Mode 1: Color only, 2 lines (TWO), 6-bit endpoints with one shared P-bit per line, 3-bit indices, 64 partitions

Bit	Description
1:0	mode = 10
7:2	partition
13:8	R0
19:14	R1

Bit	Description
25:20	R2
31:26	R3
37:32	G0
43:38	G1
49:44	G2
55:50	G3
61:56	B0
67:62	B1
73:68	B2
79:74	B3
80	P0
81	P1
127:82	indices

Mode 2: Color only, 3 lines (THREE), 5-bit endpoints, 2-bit indices, 64 partitions

Bit	Description
2:0	mode = 100
8:3	partition
13:9	R0
18:14	R1
23:19	R2
28:24	R3
33:29	R4
38:34	R5
43:39	G0
48:44	G1
53:49	G2
58:54	G3
63:59	G4
68:64	G5
73:69	B0
78:74	B1
83:79	B2
88:84	B3
93:89	B4
98:94	B5
127:99	indices

Mode 3: Color only, 2 lines (TWO), 7-bit endpoints with one P-bit per endpoint, 2-bit indices, 64 partitions

Bit	Description
3:0	mode = 1000
9:4	partition
16:10	R0
23:17	R1
30:24	R2
37:31	R3
44:38	G0
51:45	G1
58:52	G2
65:59	G3
72:66	B0
79:73	B1
86:80	B2
93:87	B3
94	P0
95	P1
96	P2
97	P3
127:98	indices

Mode 4: Color and alpha, 1 line (ONE), 5-bit color endpoints, 6-bit alpha endpoints, 16 2-bit indices, 16 3-bit indices, 2-bit component rotation, 1-bit index selector

Bit	Description
4:0	mode = 10000
6:5	rotation
7	index selector
12:8	R0
17:13	R1
22:18	G0
27:23	G1
32:28	B0
37:33	B1
43:38	A0
49:44	A1
80:50	2-bit indices

Bit	Description
127:81	3-bit indices

Mode 5: Color and alpha, 1 line (ONE), 7-bit color endpoints, 8-bit alpha endpoints, 2-bit color indices, 2-bit alpha indices, 2-bit component rotation

Bit	Description
5:0	mode = 100000
7:6	rotation
14:8	R0
21:15	R1
28:22	G0
35:29	G1
42:36	B0
49:43	B1
57:50	A0
65:58	A1
96:66	color indices
127:97	alpha indices

Mode 6: Combined color and alpha, 1 line (ONE), 7-bit endpoints with one P-bit per endpoint, 4-bit indices

Bit	Description
6:0	mode = 1000000
13:7	R0
20:14	R1
27:21	G0
34:28	G1
41:35	B0
48:42	B1
55:49	A0
62:56	A1
63	P0
64	P1
127:65	indices

Mode 7: Combined color and alpha, 2 lines (TWO), 5-bit endpoints with one P-bit per endpoint, 2-bit indices, 64 partitions

Bit	Description
7:0	mode = 10000000
13:8	partition
18:14	R0
23:19	R1
28:24	R2
33:29	R3
38:34	G0
43:39	G1
48:44	G2
53:49	G3
58:54	B0
63:59	B1
68:64	B2
73:69	B3
78:74	A0
83:79	A1
88:84	A2
93:89	A3
94	P0
95	P1
96	P2
97	P3
127:98	indices

Undefined mode values (bits 7:0 = 00000000) return zero in the RGB channels.

The indices fields are variable in length and due to the different locations of the fix-up indices depending on partition set there are a very large number of possible configurations. Each mode above indicates how many bits each index has, and the fix-up indices (one in ONE mode, two in TWO mode, and three in THREE mode) each have one less bit than indicated. However, the indices are always packed into the index fields according to the table below, with the specific bit assignments of each texel following the rules just given.

Bit	Description
LSBs	texel[0][0] index
	texel[0][1] index
	texel[0][2] index
	texel[0][3] index
	texel[1][0] index
	texel[1][1] index
	texel[1][2] index
	texel[1][3] index
	texel[2][0] index
	texel[2][1] index
	texel[2][2] index
	texel[2][3] index
	texel[3][0] index
	texel[3][1] index
	texel[3][2] index
MSBs	texel[3][3] index

Endpoint Computation

The endpoints can be defined with different precision depending on mode, as shown above. This section describes how the endpoints are computed from the bits in the compression block. Each component of each endpoint follows the same steps.

If a P-bit is defined for the endpoint, it is first added as an additional LSB at the bottom of the endpoint value. The endpoint is then bit-replicated to create an 8-bit fixed point endpoint value with a range from 0x00 to 0xFF.

Palette Color Computation

The next step involves computing the color palette values that provide the available values for each texel's color. The color palette for each line consists of the two endpoint colors plus 2, 6, or 14 interpolated colors, depending on the number of bits in the indices. Again each channel is processed independently.

The equation to compute each palette color with index i, given two endpoints is as follows, using the tables below to determine the weight for each palette index:

```
palette[i] = (E0 * (64 - weight[i]) + E1 * weight[i] + 32) » 6
```

2-bit index weights:

palette index	0	1	2	3
weight	0	21	43	64

3-bit index weights:

palette index	0	1	2	3	4	5	6	7
weight	0	9	18	27	37	46	55	64

4-bit index weights:

palette index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

The two end palette indices are equal to the two endpoints given that the weights are 0 and 64. In the above equation E0 and E1 represent the even-numbered and odd-numbered endpoints computed in the previous step for the component and line currently being computed.

Texel Selection

The final step is to select the appropriate palette index for each texel. This index then selects the 8-bit per channel palette value, which is interpreted as an 8-bit UNORM value for input into the filter (In BC7_UNORM_SRGB to UNORM values first go through inverse gamma conversion). This procedure differs depending on whether the mode is ONE, TWO, or THREE.

ONE Mode

In ONE mode, there is only one set of palette colors, thus there is only a single “partition set” defined, with all texels selecting line 0 and texel [0][0] being the “fix-up index” with one less bit in the index.

TWO Mode

64 partitions are defined for TWO, which are defined below. Each of the 64 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-3C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *0* and *1*) or line 1 (endpoints *2* and *3*). Each case has one texel each of "[0]" and "[1]", the index that this is at is termed the "fix-up index". These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	1	[0]	0	0	1	[0]	1	1	1	[0]	0	0	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	0	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]
04	[0]	0	0	0	[0]	0	1	1	[0]	0	0	1	[0]	0	0	0
	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1
	0	0	0	1	0	1	1	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
08	[0]	0	0	0	[0]	0	1	1	[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	0	0	1	1	1	1	1	0	1	1	1	0	0	0	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
0C	[0]	0	0	1	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]
10	[0]	0	0	0	[0]	1	[1]	1	[0]	0	0	0	[0]	1	[1]	1
	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
	1	1	1	0	0	0	0	0	[1]	0	0	0	0	0	0	1
	1	1	1	[1]	0	0	0	0	1	1	1	0	0	0	0	0
14	[0]	0	[1]	1	[0]	0	0	0	[0]	0	0	0	[0]	1	1	1
	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
	0	0	0	0	[1]	1	0	0	[1]	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	[1]
18	[0]	0	[1]	1	[0]	0	0	0	[0]	1	[1]	0	[0]	0	[1]	1
	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	1	[1]	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0
1C	[0]	0	0	1	[0]	0	0	0	[0]	1	[1]	1	[0]	0	[1]	1
	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1

	[1]	1	1	0	[1]	1	1	1	1	0	0	0	1	0	0	1
	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0
20	[0]	1	0	1	[0]	0	0	0	[0]	1	0	1	[0]	0	1	1
	0	1	0	1	1	1	1	1	1	0	[1]	0	0	0	1	1
	0	1	0	1	0	0	0	0	0	1	0	1	[1]	1	0	0
	0	1	0	[1]	1	1	1	[1]	1	0	1	0	1	1	0	0
24	[0]	0	[1]	1	[0]	1	0	1	[0]	1	1	0	[0]	1	0	1
	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0
	0	0	1	1	[1]	0	1	0	0	1	1	0	1	0	1	0
	1	1	0	0	1	0	1	0	1	0	0	[1]	0	1	0	[1]
28	[0]	1	[1]	1	[0]	0	0	1	[0]	0	[1]	1	[0]	0	[1]	1
	0	0	1	1	0	0	1	1	0	0	1	0	1	0	1	1
	1	1	0	0	[1]	1	0	0	0	1	0	0	1	1	0	1
	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0
2C	[0]	1	[1]	0	[0]	0	1	1	[0]	1	1	0	[0]	0	0	0
	1	0	0	1	1	1	0	0	0	1	1	0	0	1	[1]	0
	1	0	0	1	1	1	0	0	1	0	0	1	0	1	1	0
	0	1	1	0	0	0	1	[1]	1	0	0	[1]	0	0	0	0
30	[0]	1	0	0	[0]	0	[1]	0	[0]	0	0	0	[0]	0	0	0
	1	1	[1]	0	0	1	1	1	0	0	[1]	0	0	1	0	0
	0	1	0	0	0	0	1	0	0	1	1	1	[1]	1	1	0
	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
34	[0]	1	1	0	[0]	0	1	1	[0]	1	[1]	0	[0]	0	[1]	1
	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1
	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0	0
	0	0	1	[1]	1	0	0	[1]	1	1	0	0	0	1	1	0
38	[0]	1	1	0	[0]	1	1	0	[0]	1	1	1	[0]	0	0	1
	1	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0
	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0
	1	0	0	[1]	1	0	0	[1]	0	0	0	[1]	0	1	1	[1]
3C	[0]	0	0	0	[0]	0	[1]	1	[0]	0	[1]	0	[0]	1	0	0
	1	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1
	0	0	1	[1]	0	0	0	0	1	1	1	0	0	1	1	[1]

THREE Mode

64 partitions are defined for THREE, which are defined below. Each of the 64 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-3C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *0* and *1*), line 1 (endpoints *2* and *3*), or line 2 (endpoints *4* and *5*). Each case has one texel each of "[0]", "[1]", and "[2]", the index that this is at is termed the "fix-up index". These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	[1]	[0]	0	0	[1]	[0]	0	0	0	[0]	2	2	[2]
	0	0	1	1	0	0	1	1	2	0	0	1	0	0	2	2
	0	2	2	1	[2]	2	1	1	[2]	2	1	1	0	0	1	1
	2	2	2	[2]	2	2	2	1	2	2	1	[1]	0	1	1	[1]
04	[0]	0	0	0	[0]	0	1	[1]	[0]	0	2	[2]	[0]	0	1	1
	0	0	0	0	0	0	1	1	0	0	2	2	0	0	1	1
	[1]	1	2	2	0	0	2	2	1	1	1	1	[2]	2	1	1
	1	1	2	[2]	0	0	2	[2]	1	1	1	[1]	2	2	1	[1]
08	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0	[0]	0	1	2
	0	0	0	0	1	1	1	1	1	1	[1]	1	0	0	[1]	2
	[1]	1	1	1	[1]	1	1	1	2	2	2	2	0	0	1	2
	2	2	2	[2]	2	2	2	[2]	2	2	2	[2]	0	0	1	[2]
0C	[0]	1	1	2	[0]	1	2	2	[0]	0	1	[1]	[0]	0	1	[1]
	0	1	[1]	2	0	[1]	2	2	0	1	1	2	2	0	0	1
	0	1	1	2	0	1	2	2	1	1	2	2	[2]	2	0	0
	0	1	1	[2]	0	1	2	[2]	1	2	2	[2]	2	2	2	0
10	[0]	0	0	[1]	[0]	1	1	[1]	[0]	0	0	0	[0]	0	2	[2]
	0	0	1	1	0	0	1	1	1	1	2	2	0	0	2	2
	0	1	1	2	[2]	0	0	1	[1]	1	2	2	0	0	2	2
	1	1	2	[2]	2	2	0	0	1	1	2	[2]	1	1	1	[1]
14	[0]	1	1	[1]	[0]	0	0	[1]	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	1	0	0	[1]	1	1	1	0	0
	0	2	2	2	[2]	2	2	1	0	1	2	2	[2]	2	[1]	0
	0	2	2	[2]	2	2	2	1	0	1	2	[2]	2	2	1	0
18	[0]	1	2	[2]	[0]	0	1	2	[0]	1	1	0	[0]	0	0	0
	0	[1]	2	2	0	0	1	2	1	2	[2]	1	0	1	[1]	0
	0	0	1	1	[1]	1	2	2	[1]	2	2	1	1	2	[2]	1
	0	0	0	0	2	2	2	[2]	0	1	1	0	1	2	2	1
1C	[0]	0	2	2	[0]	1	1	0	[0]	0	1	1	[0]	0	0	0

	1	1	0	2	0	[1]	1	0	0	1	2	2	2	0	0	0
	[1]	1	0	2	2	0	0	2	0	1	[2]	2	[2]	2	1	1
	0	0	2	[2]	2	2	2	[2]	0	0	1	[1]	2	2	2	[1]
20	[0]	0	0	0	[0]	2	2	[2]	[0]	0	1	[1]	[0]	1	2	0
	0	0	0	2	0	0	2	2	0	0	1	2	0	[1]	2	0
	[1]	1	2	2	0	0	1	2	0	0	2	2	0	1	[2]	0
	1	2	2	[2]	0	0	1	[1]	0	2	2	[2]	0	1	2	0
24	[0]	0	0	0	[0]	1	2	0	[0]	1	2	0	[0]	0	1	1
	1	1	[1]	1	1	2	0	1	2	0	1	2	2	2	0	0
	2	2	[2]	2	[2]	0	[1]	2	[1]	[2]	0	1	1	1	[2]	2
	0	0	0	0	0	1	2	0	0	1	2	0	0	0	1	[1]
28	[0]	0	1	1	[0]	1	0	[1]	[0]	0	0	0	[0]	0	2	2
	1	1	[2]	2	0	1	0	1	0	0	0	0	1	[1]	2	2
	2	2	0	0	2	2	2	2	[2]	1	2	1	0	0	2	2
	0	0	1	[1]	2	2	2	[2]	2	1	2	[1]	1	1	2	[2]
2C	[0]	0	2	[2]	[0]	2	2	0	[0]	1	0	1	[0]	0	0	0
	0	0	1	1	1	2	[2]	1	2	2	[2]	2	2	1	2	1
	0	0	2	2	0	2	2	0	2	2	2	2	[2]	1	2	1
	0	0	1	[1]	1	2	2	[1]	0	1	0	[1]	2	1	2	[1]
30	[0]	1	0	[1]	[0]	2	2	[2]	[0]	0	0	2	[0]	0	0	0
	0	1	0	1	0	1	1	1	1	[1]	1	2	2	[1]	1	2
	0	1	0	1	0	2	2	2	0	0	0	2	2	1	1	2
	2	2	2	[2]	0	1	1	[1]	1	1	1	[2]	2	1	1	[2]
34	[0]	2	2	2	[0]	0	0	2	[0]	1	1	0	[0]	0	0	0
	0	[1]	1	1	1	1	1	2	0	[1]	1	0	0	0	0	0
	0	1	1	1	[1]	1	1	2	0	1	1	0	2	1	[1]	2
	0	2	2	[2]	0	0	0	[2]	2	2	2	[2]	2	1	1	[2]
38	[0]	1	1	0	[0]	0	2	2	[0]	0	2	2	[0]	0	0	0
	0	[1]	1	0	0	0	1	1	1	1	2	2	0	0	0	0
	2	2	2	2	0	0	[1]	1	[1]	1	2	2	0	0	0	0
	2	2	2	[2]	0	0	2	[2]	0	0	2	[2]	2	[1]	1	[2]
3C	[0]	0	0	[2]	[0]	2	2	2	[0]	1	0	[1]	[0]	1	1	[1]
	0	0	0	1	1	2	2	2	2	2	2	2	2	0	1	1
	0	0	0	2	0	2	2	2	2	2	2	2	[2]	2	0	1
	0	0	0	[1]	[1]	2	2	[2]	2	2	2	[2]	2	2	2	0

Adaptive Scalable Texture Compression (ASTC)

This section describes the data structure of the Adaptive Scalable Texture Compression (ASTC) format, as well as the decoding flow of ASTC. Also described are the header format and mipmap layout in the compressed texture file of *.astc. This is based on the reference encoder and decoder from the Khronos committee, with an extension to support multiple miplevel texture.

ASTC is a new compressed texture format with following characteristics:

1. ASTC compression format is currently only used for static texture, due to the large amount of computation and high latency required to find the optimal configuration in compression. It cannot be used to compress dynamic textures such as a shadow map.
2. ASTC is a lossy compression technique that cannot be used to compress dynamic textures which do not tolerate quality degradation.
3. ASTC has a huge range of compression ratio and block size, but these choices are fixed for each texture for all blocks at all mipmap levels.
4. ASTC has options to support compression from 1 to 4 channels for texture data.
5. ASTC can support both high and low dynamic textures.
6. ASTC can support both 2D and 3D textures.

Supported Formats
2D LDR profile.

ASTC Fundamentals

This section describes some background details and new surface formats for ASTC.

Background

ASTC is a more advanced texture compression technique than most BC and ETC formats, and can reduce footprint and bandwidth of static texture further in Graphics application by providing a texture compression solution at higher compression ratios. To best find the balance point of visual quality and compression, it provides a wide range of bit rate selection from 8bpp to 0.89 bpp in 2D, and 4.6bpp to 0.6 bpp in 3D at various block size of footprints. It also has flexibility to specify 1-4 components, selection of dual plane mode among the specified color components.

It extends the existing linear model on color distribution of each block in multiple partitions (up to 4), with flexible compact supporting on index/weight for color interpolation.

The mixture of HDR and LDR data is within each block level allows a great flexibility to represent high dynamics variation at fine granularity. The support of 3D texture explores the data coherency in all 3 dimensions, without the need to mimic 3D map with 2D slices. On top of everything, void-extent regions are introduced for both 2D and 3D maps as further optimization on large constant region.

Due to the computational complexity and processing delay of the encoding process, ASTC compression encoding is always offline, and can only be used for static texture. It does not support auto mipmap generation and cannot be considered as a format for render target.

The ASTC provides a wide spectrum of bit per pixel for both 2D and 3D texture for both LDR and HDR images, hence a wide range of compression to any 2D and 3D texture.

2D HDR and 3D LDR/HDR Profiles are not supported in 3D Sampler decompression engine.

LDR Compression Ratios:

2D Block Footprint	Bit Rate (bpp)	Compression ratio (LDR 32bpp)
4x4	8.00	4.0
5x4	6.40	5.0
5x5	5.12	6.3
6x5	4.27	7.5
6x6	3.56	9.0
8x5	3.20	10.0
8x6	2.67	12.0
10x5	2.56	12.5
10x6	2.13	15.0
8x8	2.00	16.0
10x8	1.60	20.0
10x10	1.28	25.0
12x10	1.07	29.9
12x12	0.89	36.0

HDR Compression Ratios:

Compared against fixed compression ratios of 4x or 8x on BC* formats, ASTC provides compression ratios from 4x to 36x for 2D LDR, 8x to 72x in 2D HDR maps, 7x to 54x on 3D LDR (32bpp) maps, and 14x to 108x in 3D HDR (64bpp). This can reduce bandwidth and footprint of a static 2D HDR or 3D textures to a small fractional of the existing BC formats, and greatly improve the performance on the graphic applications using these textures intensively.

Another benefit of ASTC is that, with the large range of selection of footprints and bpp, it can provide a good trade-off between quality degradation of the compressed texture and performance, due to the

bandwidth and footprints reduction. This could not be achieved by any previously existing texture compression technologies.

Although ASTC has a huge benefit of bandwidth reduction, the expected performance gain in real 3D application from this technique depends on how much texture bandwidth bottleneck is relative to the throughput of computing in EU, Sampler, and other fixed function components.

New Surface Formats for ASTC Texture

The ASTC data format natively supports 14 2D block size, 10 3D block size, and each decoded format should support either UN8 (with sRGB con version) or Float16 at each color component. Following is the full list of all different surface formats as the full combination of different block shapes and UN8 or Float16 options.

Programming Note	
Context:	Supported ASTC Formats
Only 2D LDR ASTC Formats are supported. The table below lists all possible ASTC formats, but not all are supported.	

Value	[26] LDR/Full [25] 2D/3D [24] U8srgb /FLT16	Width 2D [23:21] 3D [23:22]	Height 2D [20:18] 3D [21:20]	Depth 2D: n/a 3D: [19:18]	Binary form	Name	(BPE)
000h	000	0	0		000 000 000	ASTC_LDR_2D_4x4_U8sRGB	8.00
008h	000	1	0		000 001 000	ASTC_LDR_2D_5x4_U8sRGB	6.40
009h	000	1	1		000 001 001	ASTC_LDR_2D_5x5_U8sRGB	5.12
011h	000	2	1		000 010 001	ASTC_LDR_2D_6x5_U8sRGB	4.27
012h	000	2	2		000 010 010	ASTC_LDR_2D_6x6_U8sRGB	3.56
021h	000	4	1		000 100 001	ASTC_LDR_2D_8x5_U8sRGB	3.20
022h	000	4	2		000 100 010	ASTC_LDR_2D_8x6_U8sRGB	2.67

Value	[26] LDR/Full [25] 2D/3D [24] U8srgb /FLT16	Width 2D [23:21] 3D [23:22]	Height 2D [20:18] 3D [21:20]	Depth 2D: n/a 3D: [19:18]	Binary form	Name	(BPE)
031h	000	6	1		000 110 001	ASTC_LDR_2D_10x5_U8sRGB	2.56
032h	000	6	2		000 110 010	ASTC_LDR_2D_10x6_U8sRGB	2.13
024h	000	4	4		000 100 100	ASTC_LDR_2D_8x8_U8sRGB	2.00
034h	000	6	4		000 110 100	ASTC_LDR_2D_10x8_U8sRGB	1.60
036h	000	6	6		000 110 110	ASTC_LDR_2D_10x10_U8sRGB	1.28
03eh	000	7	6		000 111 110	ASTC_LDR_2D_12x10_U8sRGB	1.07
03fh	000	7	7		000 111 111	ASTC_LDR_2D_12x12_U8sRGB	0.89
040h	001	0	0		001 000 000	ASTC_LDR_2D_4x4_FLT16	8.00
048h	001	1	0		001 001 000	ASTC_LDR_2D_5x4_FLT16	6.40
049h	001	1	1		001 001 001	ASTC_LDR_2D_5x5_FLT16	5.12
051h	001	2	1		001 010 001	ASTC_LDR_2D_6x5_FLT16	4.27
052h	001	2	2		001 010 010	ASTC_LDR_2D_6x6_FLT16	3.56
061h	001	4	1		001 100 001	ASTC_LDR_2D_8x5_FLT16	3.20
062h	001	4	2		001 100 010	ASTC_LDR_2D_8x6_FLT16	2.67
071h	001	6	1		001 110	ASTC_LDR_2D_10x5_FLT16	2.56

Value	[26] LDR/Full [25] 2D/3D [24] U8srgb /FLT16	Width 2D [23:21] 3D [23:22]	Height 2D [20:18] 3D [21:20]	Depth 2D: n/a 3D: [19:18]	Binary form	Name	(BPE)
					001		
072h	001	6	2		001 110 010	ASTC_LDR_2D_10x6_FLT16	2.13
064h	001	4	4		001 100 100	ASTC_LDR_2D_8x8_FLT16	2.00
074h	001	6	4		001 110 100	ASTC_LDR_2D_10x8_FLT16	1.60
076h	001	6	6		001 110 110	ASTC_LDR_2D_10x10_FLT16	1.28
07eh	001	7	6		001 111 110	ASTC_LDR_2D_12x10_FLT16	1.07
07fh	001	7	7		001 111 111	ASTC_LDR_2D_12x12_FLT16	0.89
080h	010	0	0	0	010 000 000	ASTC_LDR_3D_3x3x3_U8sRGB	4.74
090h	010	1	0	0	010 010 000	ASTC_LDR_3D_4x3x3_U8sRGB	3.56
094h	010	1	1	0	010 010 100	ASTC_LDR_3D_4x4x3_U8sRGB	2.67
095h	010	1	1	1	010 010 101	ASTC_LDR_3D_4x4x4_U8sRGB	2.00
0a5h	010	2	1	1	010 100 101	ASTC_LDR_3D_5x4x4_U8sRGB	1.60
0a9h	010	2	2	1	010 101 001	ASTC_LDR_3D_5x5x4_U8sRGB	1.28
0aah	010	2	2	2	010 101 010	ASTC_LDR_3D_5x5x5_U8sRGB	1.02
0bah	010	3	2	2	010 111 010	ASTC_LDR_3D_6x5x5_U8sRGB	0.85

Value	[26] LDR/Full [25] 2D/3D [24] U8srgb /FLT16	Width 2D [23:21] 3D [23:22]	Height 2D [20:18] 3D [21:20]	Depth 2D: n/a 3D: [19:18]	Binary form	Name	(BPE)
0beh	010	3	3	2	010 111 110	ASTC_LDR_3D_6x6x5_U8sRGB	0.71
0bfh	010	3	3	3	010 111 111	ASTC_LDR_3D_6x6x6_U8sRGB	0.59
140h	101	0	0	n/a	101 000 000	ASTC_FULL_2D_4x4_FLT16	8.00
148h	101	1	0	n/a	101 001 000	ASTC_FULL_2D_5x4_FLT16	6.40
149h	101	1	1	n/a	101 001 001	ASTC_FULL_2D_5x5_FLT16	5.12
151h	101	2	1	n/a	101 010 001	ASTC_FULL_2D_6x5_FLT16	4.27
152h	101	2	2	n/a	101 010 010	ASTC_FULL_2D_6x6_FLT16	3.56
161h	101	4	1	n/a	101 100 001	ASTC_FULL_2D_8x5_FLT16	3.20
162h	101	4	2	n/a	101 100 010	ASTC_FULL_2D_8x6_FLT16	2.67
171h	101	6	1	n/a	101 110 001	ASTC_FULL_2D_10x5_FLT16	2.56
172h	101	6	2	n/a	101 110 010	ASTC_FULL_2D_10x6_FLT16	2.13
164h	101	4	4	n/a	101 100 100	ASTC_FULL_2D_8x8_FLT16	2.00
174h	101	6	4	n/a	101 110 100	ASTC_FULL_2D_10x8_FLT16	1.60
176h	101	6	6	n/a	101 110 110	ASTC_FULL_2D_10x10_FLT16	1.28

Value	[26] LDR/Full [25] 2D/3D [24] U8srgb /FLT16	Width 2D [23:21] 3D [23:22]	Height 2D [20:18] 3D [21:20]	Depth 2D: n/a 3D: [19:18]	Binary form	Name	(BPE)
17eh	101	7	6	n/a	101 111 110	ASTC_FULL_2D_12x10_FLT16	1.07
17fh	101	7	7	n/a	101 111 111	ASTC_FULL_2D_12x12_FLT16	0.89
1c0h	111	0	0	0	111 000 000	ASTC_FULL_3D_3x3x3_FLT16	4.74
1d0h	111	1	0	0	111 010 000	ASTC_FULL_3D_4x3x3_FLT16	3.56
1d4h	111	1	1	0	111 010 100	ASTC_FULL_3D_4x4x3_FLT16	2.67
1d5h	111	1	1	1	111 010 101	ASTC_FULL_3D_4x4x4_FLT16	2.00
1e5h	111	2	1	1	111 100 101	ASTC_FULL_3D_5x4x4_FLT16	1.60
1e9h	111	2	2	1	111 101 001	ASTC_FULL_3D_5x5x4_FLT16	1.28
1eah	111	2	2	2	111 101 010	ASTC_FULL_3D_5x5x5_FLT16	1.02
1fah	111	3	2	2	111 111 010	ASTC_FULL_3D_6x5x5_FLT16	0.85
1feh	111	3	3	2	111 111 110	ASTC_FULL_3D_6x6x5_FLT16	0.71
1ffh	111	3	3	3	111 111 111	ASTC_FULL_3D_6x6x6_FLT16	0.59

ASTC File Format and Memory Layout

This section describes how ASTC-compressed data is stored in files and accessed by software.

ASTC Header Data Structure and Amendment

The 1st block of an ASTC compression texture is a header file. Its byte layout in the original header structure in *.astc file is:

```
struct astc_header
{
    uint8_t magic[4];
    uint8_t blockdim_x;
    uint8_t blockdim_y;
    uint8_t blockdim_z;
    uint8_t xsize[3]; // x-size = xsize[0] + xsize[1] + xsize[2]
    uint8_t ysize[3]; // x-size, y-size and z-size are given in texels;
    uint8_t zsize[3]; // block count is inferred
};
```

Since there are limited ranges for block dimensions in x, y and z directions as described in following, we could store additional information in the unused upper bits of these byte fields

Block Dimension	2D	3D
blockdim_x	4, 5, 6, 8, 10, 12	3, 4, 5, 6
blockdim_y	4, 5, 6, 8, 10, 12	3, 4, 5, 6
blockdim_z	1	3, 4, 5, 6

Since blockdim_z is in the range of [1,6], only lower 3 bits of blockdim_z is used. We proposed the **Intel astc extension format** with *numLODs* stored in the upper 5 bits of the byte field used for blockdim_z. This new byte field can be defined as:

numLODs_blockdim_z = (numLODs-1) « 3 | (blockdim_z & 0x7) ;

New header:

```
struct astc_header
{
    uint8_t magic[4];
    uint8_t blockdim_x;
    uint8_t blockdim_y;
uint8_t numLODs_blockdim_z;
uint8_t xsize[3]; // width = xsize[0] + (xsize[1]«8) + (xsize[2]«16)
uint8_t ysize[3]; // height= ysize[0] + (ysize[1]«8) + (ysize[2]«16)
uint8_t zsize[3]; // depth = zsize[0] + (zsize[1]«8) + (zsize[2]«16)
// x_size, y_size and z_size are given in texels;

// block count is inferred
```

};

The driver or the software responsible for managing the memory resource will get numLODs and blockdim_z in:

numLODs = ((numLODs_blockdim_z » 3) & 0x1F) + 1;

blockdim_z = numLODs_blockdim_z & 0x7;

Data Layout in ASTC Compression File

A number of parameters are useful to determine where given pixels are located on the 2D & 3D surface. First, the width and height for each LOD level "L" is computed as:

$$W_L = \big((width \gg L) > 0\big)?\, width \gg L : 1\big)$$

$$H_L = \big((height \gg L) > 0\big)?\, height \gg L : 1\big)$$

$$D_L = \big((depth \gg L) > 0\big)?\, depth \gg L : 1\big)$$

The numbers of blocks in width, height and depth slab in each LOD are:

Nw(L) = Ceil(WL /Bw);

Nh (L) = Ceil(HL /Bh);

Ns (L) = Ceil(DL /Bd),

Where Bw, Bh and Bd is the block width, height and depth respectively.

Since ASTC has a native tile format specified by the encoding block size, the total number of blocks in each LOD level of the mipmap is described by nBL = Nw(L) * Nh (L) * Ns (L), The total number of blocks in the entire texture map is a summation of nBL's from all mipmap levels and all slabs, which are all pre-compressed via ASTC encoder. All the blocks in each LOD are in raster sequenced in width, height and then depth slab order.

Total ASTC Data Block Layout in All Mipmap Levels

The entire layout of the compression texture file looks like:

Address	Data Description
Addr0 (Base Address)	Header structure
Addr0+16	1st Data Block in LOD_0
Addr0+32	2nd Data Block in LOD_0
…	…
Addr1 = Addr0+16*nB0	Last Data Block in LOD_0

Address	Data Description
Addr1 +16	1st Data Block in LOD_1
Addr1+32	2nd Data Block in LOD_1
…	…
Addr2 = Addr1+16*nB1	Last Data Block in LOD_1
Addr2+16	1st Data Block in LOD_2
Addr2+32	2nd Data Block in LOD_2
…	…
Addr3 = Addr2+16*nB2	Last Data Block in LOD_2
…	…

Data Layout in Memory for All Mipmap Levels

The following equations for give the base address (U_offset, V_offset) in Cartesian coordinates for the starting point of each mip map at LOD L and depth slab q:

LOD=0:

U_offset (0, q) = 0;

*V_offset (0, q) = q * h0;*

LOD=1:

*U_offset (1, q) = (q%2)*w1;*

*V_offset (1, q) = D0*h0 + (q»1)*h1;*

LOD=2:

*U_offset (2, q) = (q%4)*w2;*

*V_offset (2, q) = D0*h0 + ceil(D1/2) * h1 + (q»2)*h2;*

LOD=3:

*U_offset (3, q) = (q%8)*w3;*

*V_offset (3, q) = D0*h0 + ceil(D1/2) * h1 + ceil(D2/2) * h2 + (q»3)*h3;*

……

Since ASTC has a native tile format specified by the encoding block size, the total number of blocks in each LOD level of the mipmap is described by nBL = Nw(L) * Nh (L) * Ns (L). The memory layout for

TileY format are considered with 512bit (16Bx4) in 1 cacheline granularity, the total number of blocks is: 4*((Ceil(HL /Bh)+3)/4 * Ceil(WL /Bw) * Ceil(DL /Bd):

Here is the full list describing the total number of rows and columns of data in each mipmap for texture in ASTC format:

Table for block dimension in 2D

Block Size	ASTC Block Height (in line)	ASTC Block Width (in Byte)
4	((Ceil(HL /4) +3)/4) *4	Ceil(WL /4) * 16
5	((Ceil(HL /5) +3)/4) *4	Ceil(WL /5) * 16
6	((Ceil(HL /6) +3)/4) *4	Ceil(WL /6) * 16
8	((Ceil(HL /8) +3)/4) *4	Ceil(WL /8) * 16
10	((Ceil(HL /10) +3)/4) *4	Ceil(WL /10) * 16
12	((Ceil(HL /12) +3)/4) *4	Ceil(WL /12) * 16

Table for block dimension in 3D

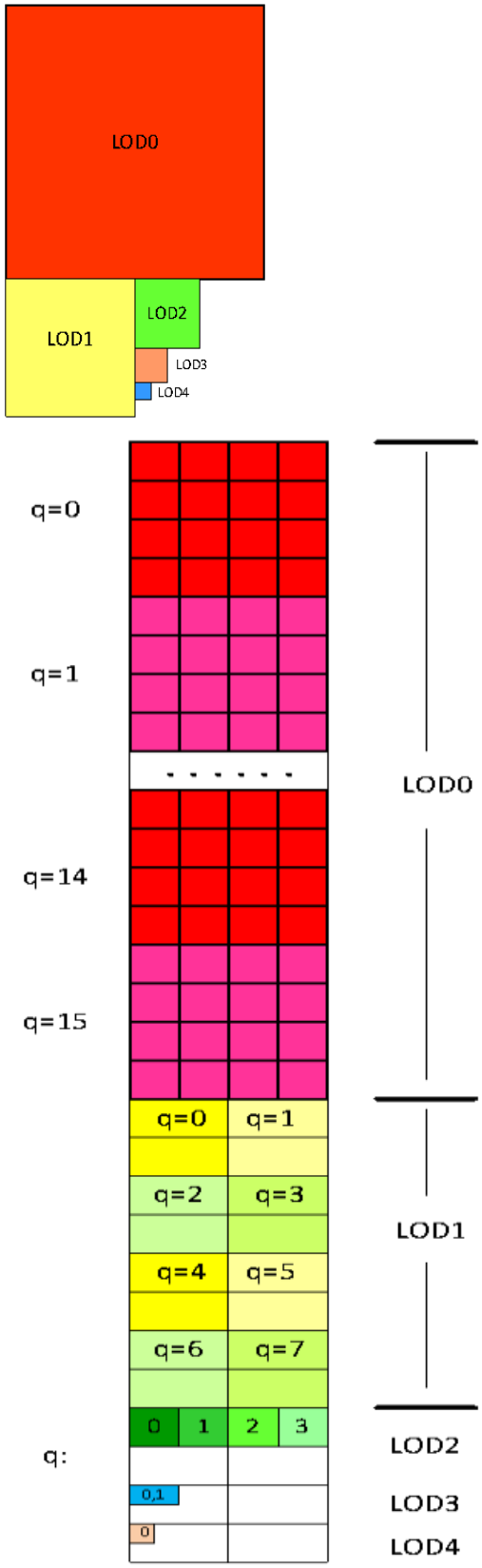
Block Size	ASTC Block Height (in line)	ASTC Block Width (in Byte)	ASTC Block Depth/slab (in slice)
3	((Ceil(HL /3) +3)/4) *4	Ceil(WL /3) * 16	Ceil(DL /3)
4	((Ceil(HL /4) +3)/4) *4	Ceil(WL /4) * 16	Ceil(DL /4)
5	((Ceil(HL /5) +3)/4) *4	Ceil(WL /5) * 16	Ceil(DL /5)
6	((Ceil(HL /6) +3)/4) *4	Ceil(WL /6) * 16	Ceil(DL /6)

For example, an image of 64x64 with 5x5 block coding in LOD0 will have:

Block Height: (13+3)/4*4=16 (lines)

Block Width: 13 *16 = 208 (Bytes)

The following diagram illustrate the memory layout for 2D and 3D map respectively.


LOD0
LOD1
LOD2
LOD3
LOD4
q=0
q=1
q=14
q=15
q=0
q=1
q=2
q=3
q=4
q=5
q=6
q=7
0
1
2
3
0,1
0
q:
LOD0
LOD1
LOD2
LOD3
LOD4

ASTC Data Structure

This section describes how the ASTC-compressed data is formatted in memory for use by hardware.

Layout and Description of Block Data

The block data structure is described in the following table in the categories of the block being partition enabled (2-4 partitions) or disabled (only 1 partition), as well as 1 plane or dual-plane mode. Where CEM refers to Color Endpoint Mode, and CCS stands for Color Channel Selection:

Layout of Partitioning Disabled (1 partition) and Enabled (multi-partition) blocks

Partition Disable	127:19			18:17	16:13	12:11	10:0
(1 Plane)	Index Data		Color Endpoint Data		CEM	00	Index Mode
(2 Planes)	Index Data		Color Endpoint Data	CCS	CEM	00	Index Mode
Multi-partitions	127:29			28:23	22:13	12:11	10:0
(1 Plane)	Index Data	Rest of CEM	Color Endpoint Data	CEM (Initial 6 bits)	Partition Index	Part	Index Mode
(2 Planes)	Index Data	CCS	Rest of CEM	Color Endpoint Data	CEM (Initial 6 bits)	Partition Index	Part	Index Mode

The 11 bit “Index mode” field specifies how the Texel Index Data is encoded. The bit encoding of this field is listed in next two tables, one for the 2D and one for the 3D.

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. In case 4 partitions in such situation 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 partition, and the number of planes in the image.

Partitioning

For any non-void extend region, each block is subdivided into 1, 2, 3 or 4 partitions, with a separate color endpoint pair for each partition. The number of partitions is specified by the partition count-1 in bits [12:11] of block data. If 2 or more partitions are selected, partitioning is enabled, the 10 bit partition index is then used to select one from 1024 partitioning patterns, where the total set of patterns supported in ASTC depends on the partition count and block size. The partitioning patterns are produced generatively, which supports a very large set of partitioning patterns for different block sizes with a modest number of hardware gates implementation.

Index Mode

The “Index mode” field specifies how the Texel Index Data is encoded. The bit encoding of this field is listed in next two tables, one for the 2D and one for the 3D.

The Index Mode field specifies the width (N), height (M) and depth (Q) of the grid of indices, what range of values they use, and whether dual index planes are present. The index ranges are encoded using a 3 bit value R, which is interpreted together with a precision bit H, as follows:

Mode R(r2 r1 r0)	Low-precision (H=0)					High-precision (H=1)			
	Index Range	Trits	Quints	Bits		Index Range	Trits	Quints	Bits
0 0 0	Invalid					Invalid			
0 0 1	Invalid					Invalid			
0 1 0	[0,1]			1		[0,9]		1	1
0 1 1	[0,2]	1				[0,11]	1		2
1 0 0	[0,3]			2		[0,15]			4
1 0 1	[0,4]		1			[0,19]		1	2
1 1 0	[0,5]	1		1		[0,23]	1		3
1 1 1	[0,7]			3		[0,31]			5

Each index value is encoded using the specified number of Trits, Quints and Bits. The details of this encoding can be found in Section - *Integer Sequence Encoding*. Due to the encoding of the R field, bits r2 and r1 cannot both be zero,

The number of indices provided for a block is not tied to the block size in any way, instead, the indices form an N*M*Q ordered grid. N, M and Q are specified on a per-block basis rather then being a global texture property. For 2D blocks, N and M can be set to any value from 2 to 12 while Q is fixed at 1; for 3D blocks, N, M and Q can be set to any value from 2 to 5. The range used for each index can be set separately for each block. The Index Bit Mode field species the values of N, M, Q and the range; it also specifies whether Dual Index Planes are present or not as well.

The D bit in following tables 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.

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

The bit encoding of the index mode field for 2D Blocks

Bits									Width	Height	Notes
10	9	8	7	6	5	4	3:2	1:0	N	M	
D	H	B		A		r0	0 0	r2 r1	B+4	A+2	
D	H	B		A		r0	0 1	r2 r1	B+8	A+2	
D	H	B		A		r0	1 0	r2 r1	A+2	B+8	
D	H	0	B	A		r0	1 1	r2 r1	A+2	B+6	
D	H	1	B	A		r0	1 1	r2 r1	B+2	A+2	
D	H	0	0	A		r0	r2 r1	0 0	12	A+2	
D	H	0	1	A		r0	r2 r1	0 0	A+2	12	
D	H	1	1	0	0	r0	r2 r1	0 0	6	10	
D	H	1	1	0	1	r0	r2 r1	0 0	10	6	
B		1	0	A		r0	r2 r1	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	-	-	Reserve
x	x	x	x	x	x	x	0 0	0 0	-	-	Reserve

Note that, due to the encoding of the R field (r0, r1, r2), bits r2 and r1 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 index mode field is laid out as follows:

3D Index Mode Layout

Bits											Notes			
10	9	8	7	6	5	4	3	2	1	0	N	M	Q	(D, H)
D	H	B		A		r0	C		r2	r1	A+2	B+2	C+2	
B		0	0	A		r0	r2	r1	0	0	6	B+2	A+2	(0, 0)
B		0	1	A		r0	r2	r1	0	0	A+2	6	B+2	(0, 0)
B		1	0	A		r0	r2	r1	0	0	A+2	B+2	6	(0, 0)
D	H	1	1	0	0	r0	r2	r1	0	0	6	2	2	
D	H	1	1	0	1	r0	r2	r1	0	0	2	6	2	
D	H	1	1	1	0	r0	r2	r1	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	-	-	-	Reserve
x	x	x	x	x	x	x	0	0	0	0	-	-	-	Reserve

The D bit is set to indicate dual-plane mode:

1: dual index planes are used

0: single index plane is used

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).

H: Index Range Bit:

1: the High-Precision group is selected.

0: The Low-Precision group is selected.

Here is the detail description:

- The encoding of xx111111100 is for the void-extent block.
- The pattern xxxxxxx0000 (the bottom 4 bits being 0000b) is reserved for future extension, and should result a NaN-vector when such a pattern is decoded.

- Any encodings not listed in the table are considered invalid and result in undened behavior if encountered by decoders.

Given the limitation of the fix length of 128 bits per block, there are restrictions that will not allow every possible encoding:

- The total number of indexes (N*M*Q for single index plane, 2*N*M*Q for dual index planes) must not exceed 64.
- The length of the Index Integer Sequence must not exceed 96 bits.
- The length of the Index Integer Sequence must be at least 24 bits.
- The above restriction, combined with the other field widths of the format, implicitly restricts the Color Integer Sequence to a maximum of 75 bits.
- Blocks that violate these restrictions are not legally produced by the encoder, result a vector of NaNs if encountered by decoders.

Here is how the indices in each block are encoded and stored:

- They are encoded using the Integer Sequence Encoding method described in Appendix.
- The resulting bit-sequence is then bit-reversed, and stored from the top of the block downwards. The ordering of the indices in the Integer Sequence is a simple scan line-like ordering.

The indices are used in two steps to interpolate between two endpoint colors for each texel.

- First, they are scaled from whatever interval they were to the range [0,64];
- The resulting value is then used as a weight to interpolate between the two endpoints.

Index Planes

Depending on the Index Bits mode selected, an ASTC compressed block may offer 1 or 2 index planes. In the case of 2 index planes, two indices rather than just one are supplied for each texel that receives indices. Of these two indices, the first one is used for a weighted sum of three of the color components; the second is used for a weighted sum of the fourth color component. If only 1 index plane is present, it applies to all four color components.

If two index planes are used, then a 2-bit bit field is needed to indicate which of the color components the second index plane applies to. These two bits are stored just below the index bits, except in the case where leftover color endpoint type bits are present; in that case, these two bits are stored just below the leftover color endpoint type bits. This two-bit bit-field has the following layout:

Channel	Red	Green	Blue	Alpha
Value	0	1	2	3

If index infill is present while two index planes are being used, then index infill is performed on each index plane separately. If two index planes are used, the indexes are stored interleaved: the first index belongs to the first index plane, the second index belongs to the second index plane, the third index belongs to the first index plane, and so on.

Index Infill Procedure

In ASTC, each block has an N*M*Q ordered grid of indices. N, M and Q may or may not match the dimensions of the actual block (e.g. it is possible to encode a 5x3 grid for an 8x8 block); if they don't match, then the grid is scaled so that its corner indexes align with the corner texels of the block, a bilinear index infill procedure is defined to interpolate an index for each texel. This procedure picks 1 to 4 indexes, and assigns each of them a weight; these weights are always a multiple of 1/16. The exact details of this interpolation procedure are specified below.

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:

List of Color Endpoint Modes

CEM	Description	Class	# of integers to represent each pair of color end points
0	LDR Luminance or Alpha, direct	0	2
1	LDR Luminance, base+offset	0	2
2	HDR Luminance, large range	0	2
3	HDR Luminance, small range	0	2
4	LDR Luminance+Alpha, direct	1	4
5	LDR Luminance+Alpha, base+offset	1	4
6	LDR RGB, base+scale	1	4
7	HDR RGB, base+scale	1	4
8	LDR RGB, direct	2	6
9	LDR RGB, base+offset	2	6
10	LDR RGB, base+scale plus two A	2	6
11	HDR RGB, direct	2	6
12	LDR RGBA, direct	3	8: D=0; 6: D=1
13	LDR RGBA, base+offset	3	8: D=0; 6: D=1

CEM	Description	Class	# of integers to represent each pair of color end points
14	HDR RGB, direct + LDR Alpha	3	8: D=0; 6: D=1
15	HDR RGB, direct + HDR Alpha	3	8: D=0; 6: D=1

Description
LDR modes are supported in ASTC LDR profile, which is enabled.
HDR modes are only supported in ASTC HDR mode, which is enabled.

In 2-4 partition modes, the encoding of Color Endpoint Modes are listed in following tables, where the endpoint mode representation may take from 6 to 14 bits, of which the first 6 bits are stored just after the partition indices, and the remaining bits are stored just below the index bits at variable position in the remaining space.

Partition / Class Types		High bits				[1:0]
Same Class	**6b [5:0]**	[5:2] Color Endpoint Mode				0 0
Different Classes	**2-Partions 8b [7:0]**	[7:6] Mode in P1	[5:4] Mode in P0	[3:3] Class Select for P1	[2:2] Class Select for P0	0 1 (Class 0 & 1)
	3-Partions 11b [10:0]			[10:9] Mode in P2	[8:7] Mode in P1	1 0 (Class 1 & 2)
		[6:5] Mode in P0	[4:4] Class Select for P2	[3:3] Class Select for P1	[2:2] Class Select for P0	
	4-Partions 14b [13:0]	[13:12] Mode in P2	[11:10] Mode in P1	[9:8] Mode in P2	[7:6] Mode in P1	1 1 (Class 2 & 3)
		[5:5] Class Select for P3	[4:4] Class Select for P2	[3:3] Class Select for P1	[2:2] Class Select for P0	

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

List of Color Endpoint Class Types encoding under multi-partitions

Partitions							...	28	27	26	25	24	23
2	...	Index	M1				...	M0		C1	C0	CEM	
3	...	Index	M2	M1	M0		...	M0	C2	C1	C0	CEM	
4	...	Index	M3	M2	M1	M0	...	C3	C2	C1	C0	CEM	

In this view, each partition *i* has two fields. C*i* is the class selector bit, choosing between the two possible CEM classes (0 indicates the lower of the two classes), and M*i* is a two-bit field specifying the low bits of the color endpoint mode within that class. The additional bits appear at a variable bit position, immediately below the texel index 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 index data have been specified. Details of the decoding procedure for Color Endpoints can be found later.

Color Endpoint Data Size Determination

The size of the data used to represent color endpoints is not explicitly specified. Instead, it is determined from the index 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_indices = M * N * Q; // size of index grid


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


index_bits = floor(num_indices*8*trits_in_index_range/5) +
floor(num_indices*7*quints_in_index_range/3) +
num_indices*bits_in_index_range;


remaining_bits = 128 - config_bits - index_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 = floor(num_CEM_values*8*trits_in_CEM_range/5) +
      floor(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

As noted in the index mode, a specifically type of encoding is the void-extended type (2D), an efficient way to encode a constant color for large blocks of regions in texture. The data structure of a void extent is listed in following 2 tables as 2D and 3D blocks respectively.

Layout of 2D Void-Extend Block, being supported in LDR.

127:112	111:96	95:80	79:64	63:51	50:38	37:25	24:12	11:10	9	8:0
A	B	G	R	T_high	T_low	S_high	S_low	Res:11	H	111111100

Layout of a 3D Void-Extent Block, only supported in Full Profile

127:112	111:96	95:80	79:64	63:55	54:46	45:37	36:28	27:19	18:10	9	8:0
A	B	G	R	R_high	R_low	T_high	T_low	S_high	S_low	H	111111100

Bit 9 H 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. 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 low and height coordinate values are treated as unsigned integers and then normalized into the range 0..1 (by dividing by 2^{13}-1 for 2D or 2^{9}-1, for 3D respectively). The high values for each dimension must be greater than the corresponding low 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.

Decoding Process

This section contains the high-level decoding algorithm for ASTC decompression.

Overview Decoding Flow

The goal for this feature is to reconstruct a cacheline (512b) of a target texture data at 4x4 region in UNORM8 A8R8G8B8 or 4x2 in FLT16 A16R16G16B16 with certain performance target, given the input texture coordinate (s,t,r). The scope of the u-architecture includes

- The additional surface format of the post decoding block, and the footprint (equivalent bpp). These are both global to each texture surface, and can be passed to the Sampler in the surface state via sampler messages.
- With post-scaled texture coordinate (u, v, p), the additional address calculation in FT to find the particular block location relative to the native block size specified in the surface state, as well as the relative texel position within that block. Assuming the block size for the block is Bu, Bv, Bp, the dimensions of a 2D surface as measured in block size tsize is:

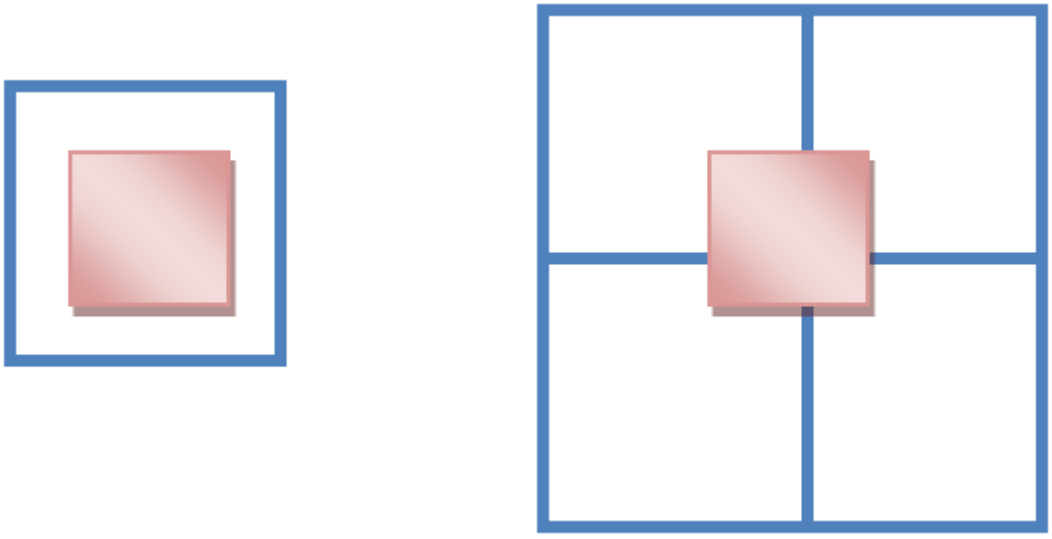
```
bw = MAX ( 2, (w+ tsize -1)/ tsize )
bh  = MAX (2, (h+ tsize -1)/ tsize)
```

Here the division is an integer division. The relationship between non-negative image coordinates [row,col] =[u, v] and block coordinates is

```
bu  = u / tsize  ;  buu = u % tsize;
bv  = v / tsize  ;  bvv =  v % tsize;
bp  = p / tsize  ;  bpp = p % tsize;
```

- With the selected sets of block size from 4x4 to 12x12 in 2D and 3x3x3 to 6x6x6 in 3D maps,1~4 blocks of source texture needs to be fetched, depending on whether the destination tile size (4x2 in FLT16 or 4x4 in UNORM8888) is inclusive or come across a few source blocks, as shown in Fig.

Destination tile is inclusive within one tile or across up to 4 tiles in source texture region



- Decode 1 to 4 128-bit ASTC compressed blocks fetched from DRAM in Sampler from ASTC compression format to either UNORM8 (LDR) or FLT16(HDR), reconstruct the texels needed in the texture filtering stage. The total decoding processing include:

Front End Decoding Processing:

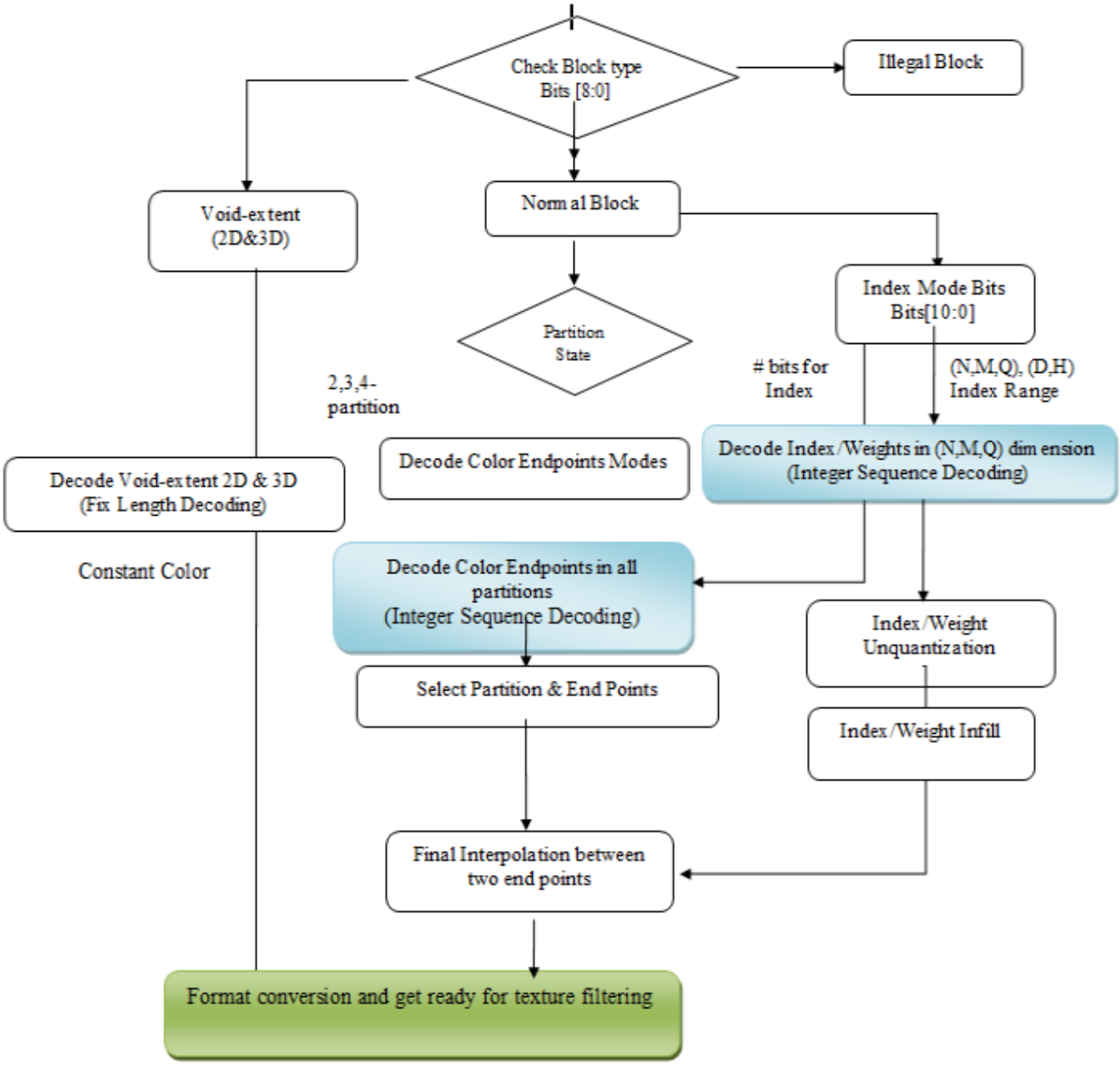
1. Detect if an ASTC block is a void-extent type, illegal type, or a normal non-void-extent type.
2. Decode the partition state – number of partitions in the current block.
3. Decode the index mode for the block include the partition seed and (N,M,Q) dimension of the compact sampling domain.
4. Decode the color endpoints modes in each partition.
5. Calculate the bit position and total # of bits used for Index.
6. Calculate the bit position and total # of bits and # of integers used in the Color endpoints in all partitions within the block.
7. With Integer Sequence Decoding, get all the indices in the compact domain defined by NxMxQ grid.
8. With Integer Sequence Decoding, get all the color end points from 16 modes in FLT16 for all partitions.

Back End Decoding Processing:

1. Reconstruct the indices at the selected sampling locations with infill scaling.
2. Find the partition from the partition seed at each sampling location.
3. Reconstruct the texture color value with the index and the pair of color end points at each sampling location.
4. If Block type is void extent, get the constant color from the high 64 bits and assign to the sampling location.

5. Convert the data to UNORM8 if LDR data is needed for the subsequent FL filtering process. Under void-extent block type,

Following is the flow diagram of the decoding process:



Integer Sequence Encoding

Both the index 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 index) 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

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 (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 floor(*i*/5)*(8+5**n*). The bits in the block are packed as follows (in this example, *n* is 4):

Trit-based Packing

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			m4		T6		T5		m3		T4		m2		T3	T2		m1				T1	T0	m0				

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] }
```

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

These are then processed as follows:

T= D * C + B;

T = T ^ A;

T = (A & 0x80) | (T » 2);

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

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 v0 to vn-1 and convert them into two RGBA color endpoints e0 and e1. 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.

Color Endpoint Modes

CEM	Range	Description	# of end points	Endpoints Reconstruction
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 next Section
3	HDR	Luminance, small range	2	See next Section
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 next Section

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); }

CEM	Range	Description	# of end points	Endpoints Reconstruction

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 next Section

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 next Section
15	HDR	RGB + HDR Alpha	8	See next Section

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;}
}
```

HDR Endpoint Decoding

The 6 HDR CEM modes on color endpoints reconstruction and surface formats are only used in full-profile ASTC texture in float 16 bit.

- HDR Endpoint Mode 2: HDR Luminance, large range
- HDR Endpoint Mode 3: HDR Luminance, small range
- HDR Endpoint Mode 7: HDR RGB, base + scale
- HDR Endpoint Mode 11: HDR RGB, direct
- HDR Endpoint Mode 14: HDR RGB, direct + LDR Alpha
- HDR Endpoint Mode 15: HDR RGB, direct + HDR Alpha

HDR Endpoint Mode 2 (HDR Luminance, Large Range)

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 (HDR Luminance, Small Range)

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	Bit[7]	Bit[6]	Bit[5]	Bit[4]	Bit[3]	Bit[2]	Bit[1]	Bit[0]
V0	M	L[6:0]						
V1	X[3:0]				D[3:0]			

The bit field marked as X allocates different bits to L or d depending on the value of the mode bit M. The complete decoding procedure is as follows:

```
// Check mode bit and extract.
If ((v0&0x80) !=0)
{
        y0 = ((v1 & 0xE0) « 4) | ((v0 & 0x7F) « 2);
        d = (v1 & 0x1F) « 2;
}
else {
        y0 = ((v1 & 0xF0) « 4) | ((v0 & 0x7F) « 1);
        d = (v1 & 0x0F) « 1;
}
// Add delta and clamp
y1 = y0 + d;
if(y1 > 0xFFF) { y1 = 0xFFF; }
// Construct RGBA result (0x780 is 1.0f)
e0 = (y0, y0, y0, 0x780);
e1 = (y1, y1, y1, 0x780);
```

HDR Endpoint Mode 7 (HDR RGB, Base+Scale)

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).

HDR Mode 7 Value Layout

Value	Bit[7]	Bit[6]	Bit[5]	Bit[4]	Bit[3]	Bit[2]	Bit[1]	Bit[0]
V0	M[3]	M[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]				

The mode bits M[0:3] are a packed representation of an endpoint bit mode, together with the major component index. For modes 0 to 4, the component (red, green, or blue) with the largest magnitude is identified, and the values are 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:

Endpoint Bit Mode

Mode	Number of bits				Description of Extra Bits						
	R	G	B	Scale	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]

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;

HDR Endpoint Mode 11 (HDR RGB, Direct)

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):

HDR Mode 11 Value Layout

Value	Bit[7]	Bit[6]	Bit[5]	Bit[3]	Bit[2]	Bit[1]	Bit[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]			

If the major component bits mj[1:0] = b11, then the RGB values are specified directly as

HDR Mode 11 Value Layout

Value	Bit[7]	Bit[6]	Bit[5]	Bit[3]	Bit[2]	Bit[1]	Bit[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]					

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

Endpoint Bit Mode

Mode	Number of Bits					Description of Extra Bits					
	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]

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 (HDR RGB, Direct + LDR Alpha)

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;
```

HDR Endpoint Mode 15 (HDR RGB, Direct + HDR Alpha)

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:

HDR Mode 15 Alpha Value Layout

Value	Bit[7]	Bit[6]	Bit[5]	Bit[4]	Bit[3]	Bit[2]	Bit[1]	Bit[0]
V6	M0	A[6:0]						
V7	M1	B[6:0]						

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;
}
```

Restrictions on Number of Partitions Per Block

Following table gives total number of partitions for each CEM mode given the restriction of total up to 16 integer values being decoded from the Integer Sequence Coding sequence.

Groups	Max Number of Partition	CEM Modes
(v0,v1)	4	0,1,2,3
(v0,v1,v2,v3)	4	4,5,6,7
(v0,v1,v2,v3,v4,v5)	3	8,9,10,11
(v0,v1,v2,v3,v4,v5,v6,v7)	2	12,13,14,15

Index Decoding

The index 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 index 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 indices are emitted together for each location, plane 0 first, then plane 1.

Index Unquantization

Each index 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 index 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:

Index Unquantization Values

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

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

Index Unquantization Parameters

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

These are then processed as follows:

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

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.

Infill Process

After unquantization, the indexes are subject to index selection and infill. The infill method is used to calculate the index for a texel position, based on the indices in the stored index 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 (Bs, Bt, Br). 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 (Ds, Dt, Dr)

```
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 (cs, ct, cr):

```
cs = Ds * s;
```

ct = Dt * t;

cr = Dr * r;

This homogeneous coordinate (cs, ct, cr) is then scaled again to give a coordinate (gs, gt, gr) in the index-grid space . The index-grid is of size (N, M, Q), as specified in the index 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 (js, jt, jr) and the fractional parts (fs, ft, fr), 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 indices. This process differs for 2D and 3D.

For 2D, bilinear interpolation is used:

v0 = js + jt*N;

p00 = decode_index(v0);

p01 = decode_index(v0 + 1);

p10 = decode_index(v0 + N);

p11 = decode_index(v0 + N + 1);

The function decode_index(n) decodes the nth index in the stored index stream. The values p00 to p11 are the indices at the corner of the square in which the texel position resides. These are then weighted using the fractional position to produce the effective index 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:

fs>ft	ft>fr	fs>fr	s1	s2	w0	w1	w2	w3
True	True	True	1	N	16-fs	fs-ft	ft-fr	fr
False	True	True	N	1	16-ft	ft-fs	fs-fr	fr
True	False	True	1	N*M	16-fs	fs-fr	fr-ft	ft
True	False	False	N*M	1	16-fr	fr-fs	fs-ft	ft
False	True	False	N	N*M	16-ft	ft-fr	fr-fs	fs
False	False	False	N*M	N	16-fr	fr-ft	ft-fs	fs

The effective index *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;
```

Index Application

Once the effective index *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 C0 and C1 as follows:

If sRGB conversion is not enabled, C0 and C1 are first expanded to 16 bits by bit replication:

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

If sRGB conversion is enabled, C0 and C1 are expanded to 16 bits differently, as follows:

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

C0 and C1 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 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 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, C0 and C1, 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 Cf:

```
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 final value Cf 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 index value. If dual-plane mode is enabled, two indices are stored with each texel. One component is then selected to use the second index for interpolation, instead of the first index. The first index 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:

Dual Plane Color Component Selector Values

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

The CCS bits are stored at a variable position directly below the index 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. 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 index 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_indices = M * N * Q; // size of index grid
if(dual_plane)
config_bits += 2;
num_indices *= 2;
```

```
index_bits = ceil(num_indices*8*trits_in_index_range/5) +
        ceil(num_indices*7*quints_in_index_range/3) +
                num_indices*bits_in_index_range;
remaining_bits = 128 – config_bits – index_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.

3D Void-Extent Blocks

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

127:112	111:96	95:80	79:64	63:55	54:46	45:37	36:28	27:19	18:10	9:9	8:0
A	B	G	R	P_high	P_low	T_high	T_low	S_high	S_low	D	111111100

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

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 index bit mode specified is one of the modes explicitly listed as Reserved.
- An index bit mode has been specified that would require more than 64 indexes total.
- An index bit mode has been specified that would require more than 96 bits for the Index Integer Sequence Encoding.
- An index bit mode has been specified that would require fewer than 24 bits for the Index Integer Sequence Encoding.
- The size of the index 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 Index 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.
- Under 3D mode, the depth (Q) is not 1

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.

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.

Video Pixel/Texel Formats

This section describes the "video" pixel/texel formats with respect to memory layout. See the Overlay chapter for a description of how the Y, U, V components are sampled.

Packed Memory Organization

Color components are all 8 bits in size for YUV formats. For YUV 4:2:2 formats each DWord will contain two pixels and only the byte order affects the memory organization.

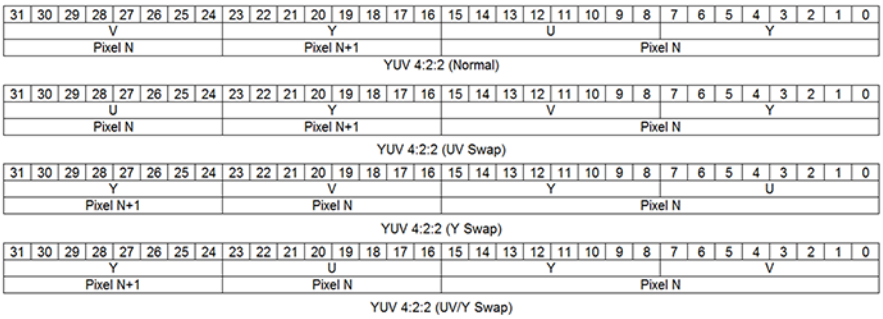
The following four YUV 4:2:2 surface formats are supported, listed with alternate names:

- YCRCB_NORMAL (YUYV/YUY2)
- YCRCB_SWAPUVY (VYUY) (R8G8_B8G8_UNORM)
- YCRCB_SWAPUV(YVYU) (G8R8_G8B8_UNORM)
- YCRCB_SWAPY (UYVY)

The channels are mapped as follows:

Cr (V)	Red
Y	Green
Cb (U)	Blue

Memory layout of packed YUV 4:2:2 formats



Planar Memory Organization

Planar formats use what could be thought of as separate buffers for the three-color components. Because there is a separate stride for the Y and U/V data buffers, several memory footprints can be supported.

The 3D sampler supports direct sampling and filtering of planar video surfaces such as YV12 and NV12.

Programming Note	
Context:	Tiling of Planar Surface
Tiling of planar surfaces (tileX, tileY, tileYf, or tileYs) is only supported for planar surfaces where the chroma plane is full-pitch (e.g. NV21). In this case, the field **Y Offset for U or UV Plane** in the RENDER_SURFACE_STATE must be programmed to force the UV plane to be at the start of a tile.	

Programming Note	
Context:	YV12/YV21 Surface Pitch Restriction
The Surface Pitch defined in RENDER_SURFACE_STATE must be a multiple of 64Bytes for YV12 and YV21 surfaces.	

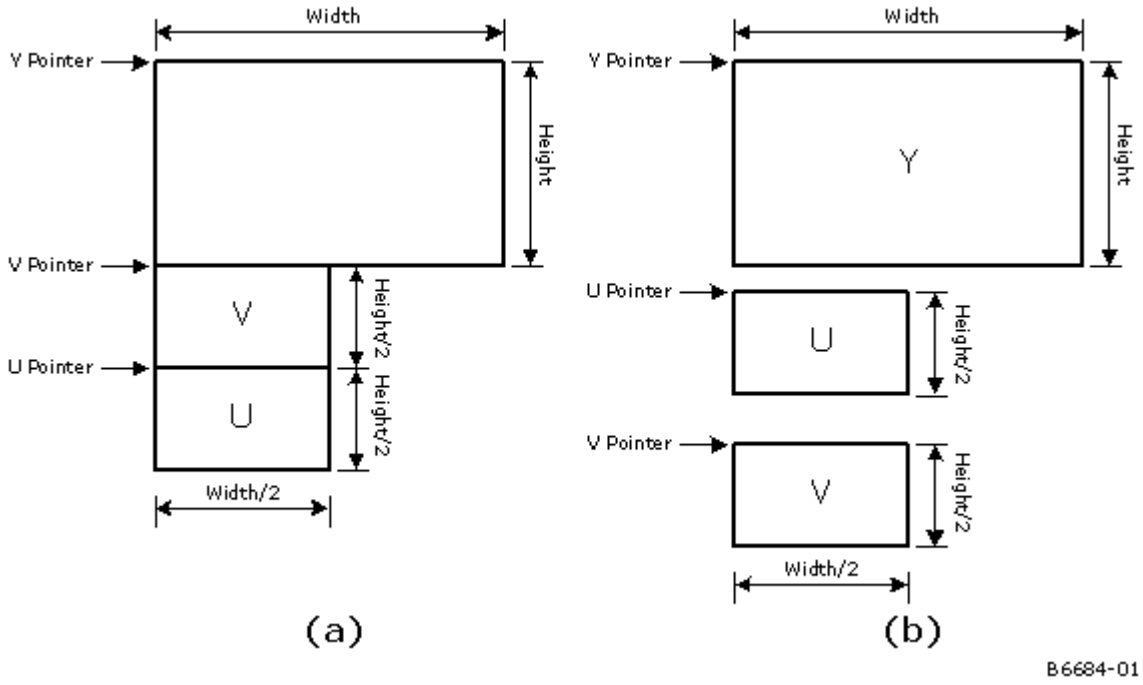
Programming Note	
Context:	NV21 Support
Sampling of NV21 surface format is supported by swapping the U and V channels when sampling the surface. This can be done by programming the **Shader Channel Select** in the RENDER_SURFACE_STATE for the Red and Blue Channels.	

The 3D sampler supports 12, and 16-bit planar video surface formats known collectively as P016. They are only supported for Sample_Unorm.

The following figure shows two types of memory organization for the YUV 4:2:0 planar video data:

1. The memory organization of the common YV12 data, where all three planes are contiguous and the strides of U and V components are half of that of the Y component.
2. An alternative memory structure that the addresses of the three planes are independent but satisfy certain alignment restrictions.

YUV 4:2:0 Format Memory Organization

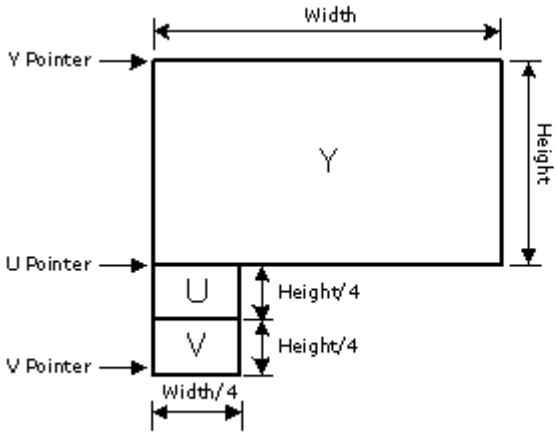


The following figure shows memory organization of the planar YUV 4:1:0 format where the planes are contiguous.

Note: The chroma planes (U and V), when separate (case b above) are treated as half-pitch with respect to the Y plane.

Workaround
When using Planar formats for YUV with half-pitch chroma planes (e.g. YV12), and fenced tiling is not supported
LINEAR filtering of Planar YUV surfaces such as YV12 using the 3D sampler is done after the U and V have been replicated to form a YUV444 texels. This means that the U and V components will effectively be point-sampled rather than filtered. Acheive true filtering of the U and V components, the 3 planes of the YUV surface must be bound as separate surfaces, and the filtering must be done on each individually.

YUV 4:1:0 Format Memory Organization



B6685-01

The table below shows how position within a Planar YUV surface chroma plane is calculated for various cases ot U and V pitch and position. It also shows restrictions on the alignment of the chroma planes in memory for non-interleaved (YV12) and interleaved chroma (e.g. NV12) is used.

Case	Interleave Chroma	Pitch	Vertical U/V Offset
YUV with Half Pitch Chroma	No	Half	When U is below Y Y_Uoffset = Y_Height * 2 Y_Voffset = Y_Height * 2 + V_Height When V is below Y Y_Uoffset = Y_Height * 2 + V_Height Y_Voffset = Y_Height * 2
YUV with Full Pitch Chroma	Yes	Full	When U is below Y Y_Uoffset = Y_Height Y_Voffset = Y_Height + V_Height When V is below Y Y_Uoffset = Y_Height + V_Height Y_Voffset = Y_Height
YUV for Media Sampling	Yes	Always Full	Same as 3D full pitch

Programming Note	
Context: Planar YUV surfaces cannot be 1D surface types.	
Because there is a requirement that the height of the Y plane of a planar surface must be a greater than 1, it cannot be programmed to be a **Surface Type** of SURFTYPE_1D.	

Programming Note	
Context:	MIP Filtering
Surface state cannot have (**MIP Mode Filter** != NONE) for Planar YUV surfaces (e.g. PLANAR_420_8).	

Programming Note	
Context:	Standard Tiling
Planar YUV does not support MIP Tails as part of Standard Tiling. The MIP Tail Start field in RENDER_SURFACE_STATE must be programmed to 15.	

Programming Note	
Context:	Quilted and Planar
Planar YUV is not supported for Quilted surfaces.	

Programming Note	
Context:	
Planar YUV is not supported with Corner Texel Mode	

Sample operations to a Planar YUV surface with the SURFACE_STATE disabled may not return 0's as the result.

Additional Video Formats

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
DXGI_FORMAT_AYUV	DXGI_FORMAT_R8G8B8A8_UNORM (V->R8, U->G8, Y->B8, A->A8)			Packed	1	R8G8B8A8_UNORM	NA	NA	Sampler, PB
DXGI_FORMAT_AYUV	DXGI_FORMAT_R8G8B8A8_UINT (V->R8, U->G8, Y->B8, A->A8)			Packed	1	R8G8B8A8_UINT	NA	NA	Sampler, HDC, PB
DXGI_FORMAT_YUY2	DXGI_FORMAT_R8G8B8A8_UNORM (Y0->R8, U0->G8, Y1->B8, V0->A8)			Packed	1	R8G8B8A8_UNORM	NA	NA	Sampler,
NA	NA	CL_YCbYCr (Y0->R16, U0->G16, Y1->B16, V0->A16	CL_UNORM_INT16	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_YCbYCr (Y0->R16, U0->G16, Y1->B16, V0->A16	CL_UNORM_INT12	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_YCbYCr (Y0->R16, U0->G16, Y1->B16, V0->A16	CL_UNORM_INT10	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
DXGI_FORMAT_YUY2	DXGI_FORMAT_R8G8B8A8_UINT (Y0->R8, U0->G8, Y1->B8, V0->A8)	CL_YCbYCr (Y0->R8, U0->G8, Y1->B8, V0->A8	CL_UNORM_INT8	Packed	1	R8G8B8A8_UINT	NA	NA	Sampler, HDC
DXGI_FORMAT_YUY2	DXGI_FORMAT_R8G8_B8G8_UNORM			Packed	1	R8G8B8A8_UNORM In this case the width of the view will appear to be twice the R8G8B8A8 view, with hardware	NA	NA	Sampler

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
						reconstruction of RGBA done automatically on read (and before filtering).			
NA	NA	CL_CbYCrY (U0->R16, Y0->G16, V0->B16, Y1->A16	CL_UNORM_INT16	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbYCrY (U0->R16, Y0->G16, V0->B16, Y1->A16	CL_UNORM_INT12	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbYCrY (U0->R16, Y0->G16, V0->B16, Y1->A16	CL_UNORM_INT10	Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbYCrY (U0->R8, Y0->G8, V0->B8, Y1->A8	CL_UNORM_INT8	Packed	1	R8G8B8A8_UINT	NA	NA	Sampler, HDC
DXGI_FORMAT_NV12	Y = DXGI_FORMAT_R8_UNORM U/V = DXGI_FORMAT_R8G8_UNORM (U->R8, V->G8)			Planar	2	R8_UNORM	R8G8_UNORM chomra pixel dimensions 1/2 in both x and y from the Luma view	NA	Sampler, PB
NA	NA	CL_Y_Cr_Cb (Y -> R16) (U->R16) (V->R16)	CL_UNORM_INT16	Planar	3	R16_UNIT	R16_UNIT	R16_UNIT	Sampler

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
NA	NA	CL_Y_Cr_Cb (Y -> R16) (U->R16) (V->R16)	CL_UNORM_INT12	Planar	3	R16_UNIT	R16_UNIT	R16_UNIT	Sampler
NA	NA	CL_Y_Cr_Cb (Y -> R16) (U->R16) (V->R16)	CL_UNORM_INT10	Planar	3	R16_UNIT	R16_UNIT	R16_UNIT	Sampler
NA	NA	CL_Y_Cr_Cb (Y -> R8) (U->R8) (V->R8)	CL_UNORM_INT8	Planar	3	R8_UNIT	R8_UNIT	R8_UNIT	Sampler
NA	NA	CL_Y_CrCb (Y -> R16) (U->R16, V->G16)	CL_UNORM_INT16	Planar	2	R16_UNIT	R16G16_UINT chomra pixel dimensions halved in both x and y from the Luma view	NA	Sampler, HDC
NA	NA	CL_Y_CrCb (Y -> R16) (U->R16, V->G16)	CL_UNORM_INT12	Planar	2	R16_UNIT	R16G16_UINT chomra pixel dimensions halved in both x and y from the Luma view	NA	Sampler, HDC
NA	NA	CL_Y_CrCb (Y -> R16) (U->R16, V->G16)	CL_UNORM_INT10	Planar	2	R16_UNIT	R16G16_UINT chomra pixel dimensions halved in both x and y from the Luma view	NA	Sampler, HDC

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
DXGI_FORMAT_NV12	Y = DXGI_FORMAT_R8_UINT U/V = DXGI_FORMAT_R8G8_UINT (U->R8, V->G8)	CL_Y_CrCb (Y -> R8) (U->R8, V->G8)	CL_UNORM_INT8	Planar	2	R8_UNIT	R8G8_UINT chomra pixel dimensions halved in both x and y from the Luma view	NA	Sampler, HDC, PB
DXGI_FORMAT_NV11	Y = DXGI_FORMAT_R8_UNORM U/V = DXGI_FORMAT_R8G8_UNORM (U->R8, V->G8)			Planar	2	R8_UNORM	R8G8_UNORM chomra pixel dimensions 1/4 in both x and y from the Luma view	NA	Sampler, PB
DXGI_FORMAT_NV11	Y = DXGI_FORMAT_R8_UINT U/V = DXGI_FORMAT_R8G8_UINT (U->R8, V->G8)			Planar	2	R8_UNIT	R8G8_UINT chomra pixel dimensions 1/4 in both x and y from the Luma view	NA	Sampler, HDC, PB
DXGI_FORMAT_P016	Y = DXGI_FORMAT_R16_UNORM U/V = DXGI_FORMAT_R16G16_UNORM (U->R16, V->G16)			Planar	2	R16_UNORM	R16G16_UNORM chomra pixel dimensions 1/2 in both x and y from the Luma view	NA	Sampler, HDC, PB
DXGI_FORMAT_P016	Y = DXGI_FORMAT_R16_UINT U/V = DXGI_FORMAT_R16G16_UINT (U->R16, V->G16)			Planar	2	R16_UNIT	R16G16_UINT chomra pixel dimensions 1/2 in both x and y from the Luma view	NA	Sampler, PB
DXGI_FORMAT_P010	Y =			Planar	2	R16_UNORM	R16G16_U	NA	Sampler, HDC, PB

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
	DXGI_FORMAT_R16_UNORM U/V = DXGI_FORMAT_R16G16_UNORM (U->R16, V->G16)						NORM chroma pixel dimensions 1/2 in both x and y from the Luma view		
DXGI_FORMAT_P010	Y = DXGI_FORMAT_R16_UINT U/V = DXGI_FORMAT_R16G16_UINT (U->R16, V->G16)			Planar	2	R16_UNIT	R16G16_UINT chomra pixel dimensions 1/2 in both x and y from the Luma view	NA	Sampler, HDC, PB
DXGI_FORMAT_Y216	DXGI_FORMAT_R16G16B16A16_UNORM (Y0->R16, U->G16, Y1->B16, V->A16).			Packed	1	R16G16B16A16_UNORM	NA	NA	Sampler,
DXGI_FORMAT_Y216	DXGI_FORMAT_R16G16B16A16_UINT (Y0->R16, U->G16, Y1->B16, V->A16).			Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
DXGI_FORMAT_Y210	DXGI_FORMAT_R16G16B16A16_UNORM (Y0->R16, U->G16, Y1->B16, V->A16).			Packed	1	R16G16B16A16_UNORM	NA	NA	Sampler,
DXGI_FORMAT_Y210	DXGI_FORMAT_R16G16B16A16_UINT (Y0->R16, U->G16, Y1->B16, V->A16).			Packed	1	R16G16B16A16_UINT	NA	NA	Sampler, HDC
DXGI_FORMAT_Y416	DXGI_FORMAT_R16G16B16A16_UNORM (U->R16, Y->G16, V->B16, A->A16)			Packed	1	R16G16B16A16_UNORM	NA	NA	Sampler, HDC
DXGI_FORMAT_Y416	DXGI_FORMAT_R16G16B16A16_UINT (U->R16, Y->G16, V-			Packed	1	R16G16B16A16_UINT	NA	NA	Sampler,

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
	>B16, A->A16)								
DXGI_FORMAT_Y410	DXGI_FORMAT_R16G16B16A16_UNORM (U->R16, Y->G16, V->B16, A->A16)			Packed	1	R16G16B16A16_UNORM	NA	NA	Sampler, HDC
DXGI_FORMAT_Y410	DXGI_FORMAT_R16G16B16A16_UINT (U->R16, Y->G16, V->B16, A->A16)			Packed	1	R16G16B16A16_UINT	NA	NA	Sampler,
NA	NA	CL_CbCr (U->R16, V->G16)	CL_UNORM_INT16	Packed	1	R16G16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbCr (U->R16, V->G16)	CL_UNORM_INT12	Packed	1	R16G16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbCr (U->R16, V->G16)	CL_UNORM_INT10	Packed	1	R16G16_UINT	NA	NA	Sampler, HDC
NA	NA	CL_CbCr (U->R8, V->G8)	CL_UNORM_INT8	Packed	1	R8G8_UINT	NA	NA	Sampler, HDC
NA	NA	CL_Cb_Cr (U->R16) (V->R16)	CL_UNORM_INT16	Planar	2	R16_UNIT	R16_UNIT	NA	Sampler, HDC
NA	NA	CL_Cb_Cr (U->R16) (V->R16)	CL_UNORM_INT12	Planar	2	R16_UNIT	R16_UNIT	NA	Sampler, HDC
NA	NA	CL_Cb_Cr (U->R16) (V->R16)	CL_UNORM_INT10	Planar	2	R16_UNIT	R16_UNIT	NA	Sampler, HDC
NA	NA	CL_Cb_Cr (U->R8)	CL_UNORM_INT8	Planar	2	R8_UNIT	R8_UNIT	NA	Sampler, HDC

Dx YUV format name	DxRGB format name	GPGPU Image Type	GPGPU Image Data Type	Packed / Planar	# Surface States	Surface Format #1	Surface Format #2	Surface Format #3	Support By
		(V->R8)							
NA	NA	CL_Y (Y->R16)	CL_UNORM_INT16	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Y (Y->R16)	CL_UNORM_INT12	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Y (Y->R16)	CL_UNORM_INT10	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Y (Y->R8)	CL_UNORM_INT8	Packed	1	R8_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (V->R16)	CL_UNORM_INT16	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (V->R16)	CL_UNORM_INT12	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (V->R16)	CL_UNORM_INT10	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (V->R8)	CL_UNORM_INT8	Packed	1	R8_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (U->R16)	CL_UNORM_INT16	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (U->R16)	CL_UNORM_INT12	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (U->R16)	CL_UNORM_INT10	Packed	1	R16_UNIT	NA	NA	Sampler, HDC
NA	NA	CL_Cb (U->R8)	CL_UNORM_INT8	Packed	1	R8_UNIT	NA	NA	Sampler, HDC

Raw Format

A format called "RAW" is available that is only supported with the untyped surface read/write, block, scattered, and atomic operation data port messages. It means that the surface has no inherent format. Surfaces of type RAW are addressed with byte-based offsets. The RAW surface format can be applied only to surface types of BUFFER and STRBUF.

Surface Memory Organizations

See *Memory Interface Functions* chapter for a discussion of tiled vs. linear surface formats.

Display, Overlay, Cursor Surfaces

These surfaces are memory image buffers (planes) used to refresh a display device in non-VGA mode. See the Display chapter for specifics on how these surfaces are defined/used.

2D Render Surfaces

These surfaces are used as general source and/or destination operands in 2D BLT operations.

Note that there is no coherency between 2D render surfaces and the texture cache. Software must explicitly invalidate the texture cache before using a texture that has been modified via the BLT engine.

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

2D Monochrome Source

These 1 BPP (bit per pixel) surfaces are used as source operands to certain 2D BLT operations, where the BLT engine expands the 1 BPP source to the required color depth.

The texture cache stores any monochrome sources. There is no mechanism to maintain coherency between 2D render surfaces and texture-cached monochrome sources. Software must explicitly invalidate the texture cache before using a memory-based monochrome source that has been modified via the BLT engine. (Here the assumption is that SW enforces memory-based monochrome source surfaces as read-only surfaces.)

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, coherency rules, etc.

2D Color Pattern

Color pattern surfaces are used as special pattern operands in 2D BLT operations.

The device uses the texture cache to store color patterns. There is no mechanism to maintain coherency between 2D render surfaces and (texture)-cached color patterns. Software is required to explicitly invalidate the texture cache before using a memory-based color pattern that has been modified via the BLT engine. (Here the assumption is that SW enforces memory-based color pattern surfaces as read-only surfaces.)

See the *2D Instruction* and *2D Rendering* chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

3D Color Buffer (Destination) Surfaces

3D Color Buffer surfaces hold per-pixel color values for use in the 3D Pipeline. The 3D Pipeline always requires a Color Buffer to be defined.

See the Non-Video Pixel/Texel Formats section in this chapter for details on the Color Buffer pixel formats. See the 3D Instruction and 3D Rendering chapters for Color Buffer usage details.

The Color Buffer is defined as the BUFFERID_COLOR_BACK memory buffer via the 3DSTATE_BUFFER_INFO instruction. That buffer can be mapped to LM or SM (snooped or unsnooped), and can be linear or tiled. When both the Depth and Color Buffers are tiled, the respective Tile Walk directions must match.

When a linear Color Buffer and a linear Depth Buffer are used together:

- The buffers may have different pitches, though both pitches must be a multiple of 32 bytes.
- The buffers must be co-aligned with a 32-byte region.

3D Depth Buffer Surfaces

Depth Buffer surfaces hold per-pixel depth values and per-pixel stencil values for use in the 3D Pipeline. The 3D Pipeline does not require a Depth Buffer in general, though a Depth Buffer is required to perform non-trivial Depth Test and Stencil Test operations.

The Depth Buffer is specified via the 3DSTATE_DEPTH_BUFFER command. See the description of that instruction in *Windower* for restrictions.

See *Depth Buffer Formats* below for a summary of the possible depth buffer formats. See the Depth Buffer Formats section in this chapter for details on the pixel formats. See the *Windower* and *DataPort* chapters for details on the usage of the Depth Buffer.

Depth Buffer Formats

DepthBufferFormat / DepthComponent	BPP (Bits Per Pixel)	Description
D32_FLOAT_S8X24_UINT	64	32-bit floating point Z depth value in first DWord, 8-bit stencil in lower byte of second DWord
D32_FLOAT	32	32-bit floating point Z depth value
D24_UNORM_S8_UINT	32	24-bit fixed point Z depth value in lower 3 bytes, 8-bit stencil value in upper byte
D16_UNORM	16	16-bit fixed point Z depth value

3D Separate Stencil Buffer Surfaces

Separate Stencil Buffer surfaces hold per-pixel stencil values for use in the 3D Pipeline. Note that the 3D Pipeline does not require a Stencil Buffer to be allocated, though a Stencil Buffer is required to perform non-trivial Stencil Test operations.

Depth Buffer Formats summarizes Stencil Buffer formats. Refer to the Stencil Buffer Formats section in this chapter for details on the pixel formats. Refer to the *Windower* chapters for Stencil Buffer usage details.

The Stencil buffer is specified via the 3DSTATE_STENCIL_BUFFER command. See that instruction description in *Windower* for restrictions.

Depth Buffer Formats

DepthBufferFormat / DepthComponent	BPP (bits per pixel)	Description
R8_ UNIT	8	8-bit stencil value in a byte

Surface Layout and Tiling

This section explains how various surface types (1D, 2D, 3D, and Cube) are laid out in memory. Most of the information in this section is independent of tiling. The concept of tiling can be laid on top of information. Wherever there is a specific difference it will be called out.

For Tiling (TileY, TileYs etc.), see the Address Tiling Function Introduction section which provides detailed information on how tiles are organized and laid out.

Maximum Surface Size in Bytes

In addition to restrictions on maximum height, width, and depth, surfaces are also restricted to a maximum size of 2^44 bytes. All pixels within the surface must be contained within 2^44 bytes of the base address.

NULL Page Support for Media Sampler

NULL support for VA Media sampler will process the returned data for NULL pages as if it was all zeros. Except for feature matching function, in which the NULL page indication will cause the distance function calculation to be ignored and a maximum distance value of 0xFF to be returned.

Tiling

To improve efficiency in memory accesses, most surfaces can be laid out using a tiling scheme.

Supported Legacy Tiling Modes:

- **TileY**
- **TileX**
- **TileW**

Supported Tiled Resource Modes

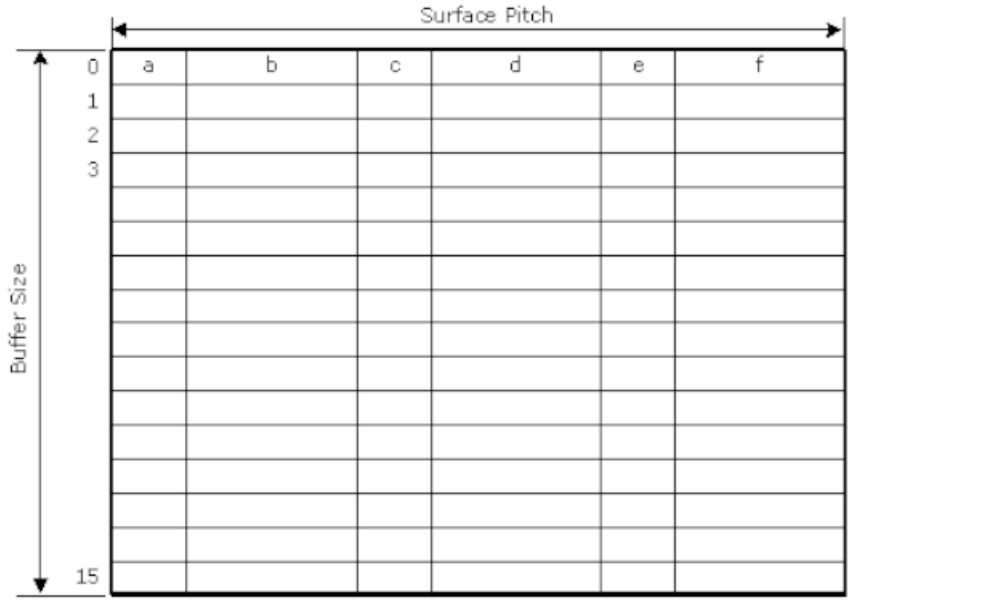
- **TileYF:** 4KB tiling mode based on TileY
- **TileYS:** 64KB tiling mode based on TileY

These modes are described in the **Address Tiling Function Introduction** volume.

Typed Buffers

A typed buffer is an array of structures. Each structure contains up to 2048 bytes of elements. Each element is a single surface format using one of the supported surface formats depending on how the surface is being accessed. The surface pitch state for the surface specifies the size of each structure in bytes.

The buffer is stored in memory contiguously with each element in the structure packed together, and the first element in the next structure immediately following the last element of the previous structure. Buffers are supported only in linear memory.



Typed buffers are accessed using a surface state for each structure element (a,b,c, etc. in the diagram above). The surface state for element “b” (for example) contains the surface format of element “b” (which may differ from other elements), the base address points to element “b” in the first structure (slice 0 of the array). The pitch for all the elements in the buffer is the same value, and the surface type of each element is SURFTYPE_BUFFER.

The offset into the typed buffer is given by the following equation:

```
Offset = (V * Pitch) + U
```

MIP Layout

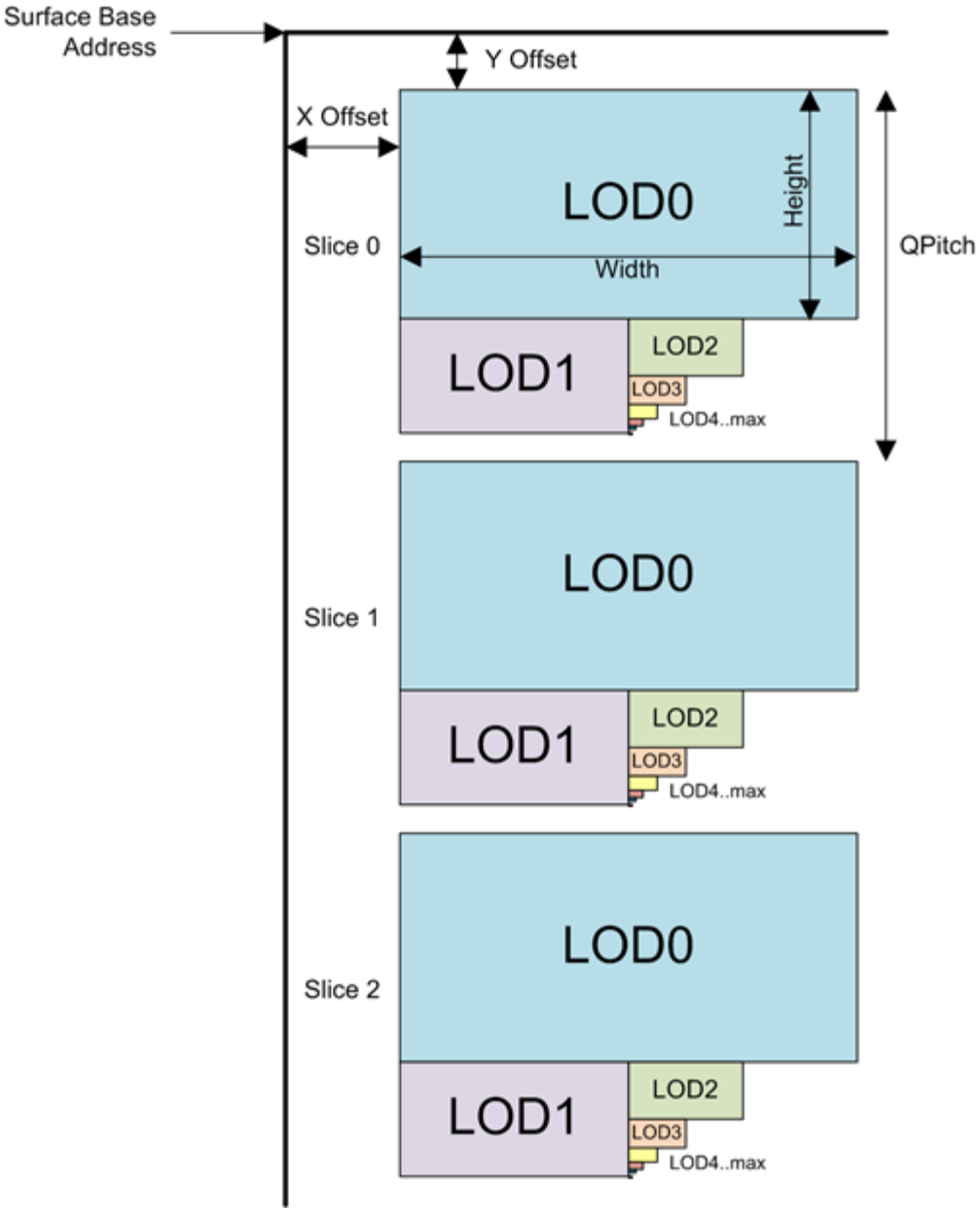
A surface can support multiple levels of details (LODs) or MIPs. The MIPCOUNT field in the RENDER_SURFACE_STATE defines how many MIPs a surface contains.

MIP0 or LOD0 is the largest, highest-detail MIP. The height, width and depth of this LOD is what is defined in the RENDER_SURFACE_STATE for that surface. Each subsequent

MIP is exactly one-half the height and width of the previous, making it 1/4th the size in memory.

The MIPs of a surface a layed out in memory using a 2-dimensional method as shown below. Volumetric and arrayed surfaces use multiple "slices" of this MIP layout, with each slice separted by QPITCH number of rows.

The diagram below shows many of the parameters of a 2D,2D Arrayed and 3D surface.



This 2-dimensional layout implies that there is padding required on the rows below LOD0 in order to ensure each row is the same number of texels.

If Tiling is enabled, then each MIP is layed out using one or more tiles. If TileYf or TileYs tiling is enabled (TR_MODE != NONE), then some of the MIPs may actually be stored in a MIPTail which fits in a single 64K or 4K tile. The layout above, then only applied to MIPs which are not packed in the MIP Tail. Note that, depending on surface height the Vertical Alignment that surface can actually have the last few mips layed out below LOD1. Using MIP Tail (if supported) eliminates this possibility.

Raw (Untyped) Buffers

Raw buffers also use the surface type of SURFTYPE_BUFFER, but the surface format is RAW. These buffers are one-dimensional. They are accessed with a single U parameter which is a byte offset into the buffer. Raw buffers are also supported only in linear memory.

The offset into the raw buffer is given directly by the U parameter.

```
Offset = U
```

Structured Buffers

A structured buffer is a surface type that is accessed by a 2-dimensional coordinate. It can be thought of as an array of structures, where each structure is a predefined number of DWords in size. The first coordinate (U) defines the array index, and the second coordinate (V) is a byte offset into the structure which must be a multiple of 4 (DWord-aligned). A structured buffer must be defined with **Surface Format** RAW.

The structured buffer has only one dimension programmed in SURFACE_STATE which indicates the array size. The byte offset dimension (V) is assumed to be bounded only by the **Surface Pitch**.

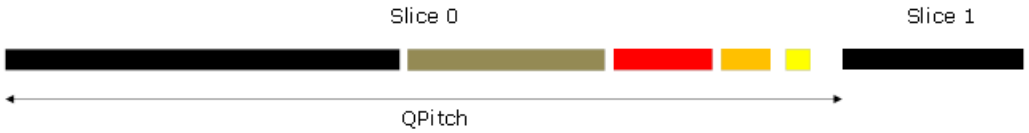
The two dimensional offset into the surface is defined directly by the U and V parameters. Structured buffers are linear.

1D Surfaces

One-dimensional surfaces use a tiling mode of linear. Technically, they are not tiled resources, but the Tiled Resource Mode field in RENDER_SURFACE_STATE is still used to indicate the alignment requirements for this linear surface (See 1D Alignment requirements for how 4K and 64KB Tiled Resource Modes impact alignment). 1D surfaces are stored linearly in memory.

Programming Note	
Context:	Legacy 1D Tiling
There is a legacy mode for representing a 1D surface as a 2D surface with a height of 1 texel. However this mode is not recommended due to API compatibility. The **Sampler Legacy 1D Map Layout Disable** MMIO bit (bit 0, E194h) *must* be set to 1h to allow true 1D surfaces.	

Linear 1D surfaces are stored in a one-dimensional view of memory as follows:



Surface Pitch is ignored for 1D surfaces. **Surface QPitch** specifies the distance in pixels between array slices. QPitch should allow at least enough space for any mips that may be present.

A number of parameters are useful to determine where given pixels will be located on the 1D surface. First, the width for each LOD “L” is computed:

$$W_L = ((width >> L) > 0 ? width >> L : 1$$

When **Corner Texel Mode** is enabled via the RENDER_SURFACE_STATE, the width of a 1D surface is calculated as shown below:

W_L = MAX(1,(W_{L-1}-1)»1)+1

There is a restriction that the smallest map dimension is 2 texels for **Corner Texel Mode** (W_0 >1)

Next, the aligned width parameter for each LOD "L" is computed. The "i" parameter is the horizontal alignment parameter set by a state field or defined as a constant, depending on the surface. The equation uses the I value that applies to the LOD being computed.

$$w_L = i * ceil\left(\frac{W_L}{i}\right)$$

Next, the offset to each LOD is determined. The offset has one dimension for 1D surfaces. The single element in the LOD_L vector is named LODUL.

$$LOD_0 = (0)$$
$$LOD_1 = (w_0)$$
$$LOD_2 = (w_0 + w_1)$$
$$LOD_3 = (w_0 + w_1 + w_2)$$
$$LOD_4 = (w_0 + w_1 + w_2 + w_3)$$
...

Based on the above parameters and the U and R (pixel address and array index, respectively), and the bytes per pixel of the surface format (Bpp), the offset "u" in bytes from the base address of the surface is given by:

```
u = [(R * QPitch) + LODUL + U] * Bpp
```

Programming Note	
Context:	Packed YUV Surfaces
Packed YUV surface formats such as YCRCB_NORMAL, YCRCB_SWAPUVY etc. will be treated as 16bpp surface, not 32bpp, which may impact how they are laid out in memory.	

Tiling and Mip Tail for 1D Surfaces

There is no MIP Tail allowed for 1D surfaces because they are not allowed to be tiled. They must be declared as linear.

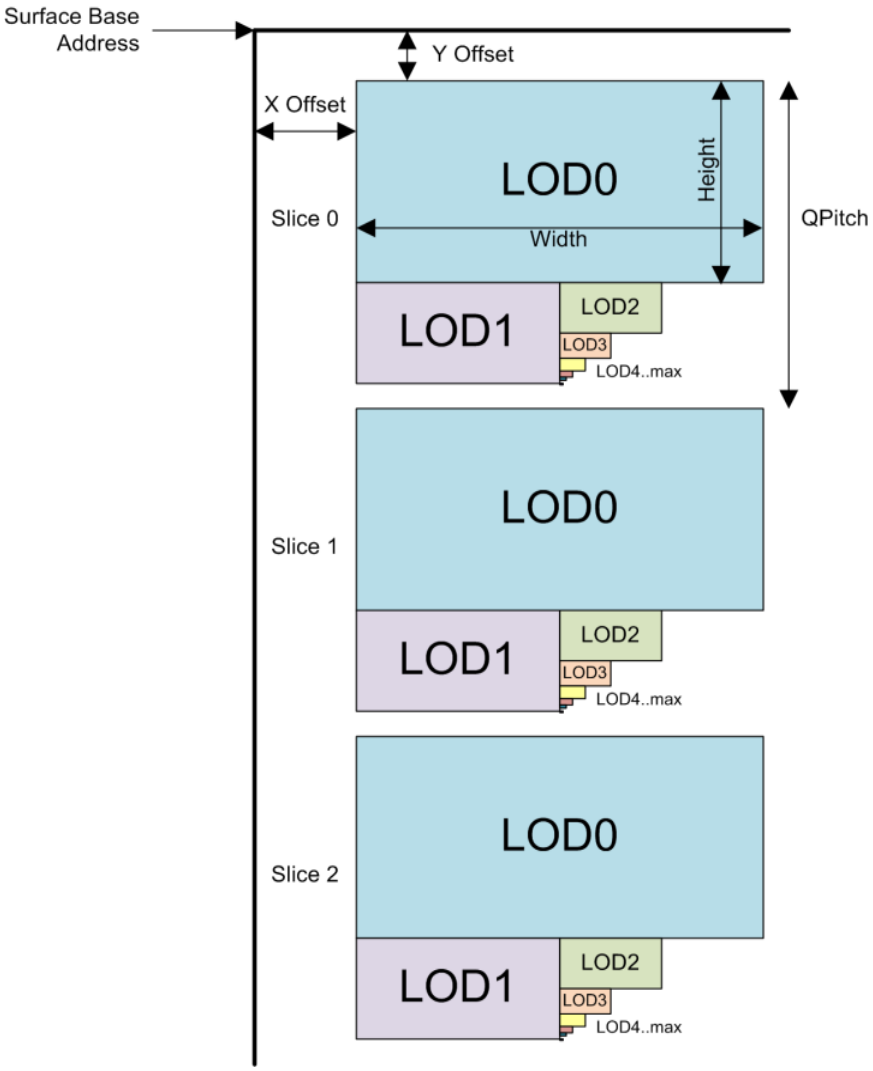
1D Alignment Requirements

1D surfaces are not tiled but laid out linearly in memory.

Tiled Resource Mode	Bits per Element	Horizontal Alignment
TRMODE_NONE	Any	64

2D Surfaces

2D surfaces represent two-dimensional bitmaps, which can also be mip-mapped and/or consist of array slices, effectively representing multiple 2D sub-surfaces within a single surface. The diagram below shows many of the parameters of a 2D surface or Arrayed 2D Surface and what they mean.



All surface parameters are defined in terms of texels (agnostic to the surface format).

Surface Pitch defines the distance in bytes between rows of the surface and is a function of the **Width** of LOD. **QPitch** specifies the distance in rows between array slices and is a function of the **Height**. **QPitch** should allow at least enough space for any Mips that may be present.

There can also be non-zero offsets (**X_Offset** and **Y_Offset)** defined from the base address which can be used to provide padding or provide an offset to a lower-detail LOD.

There are limitations to the physical size of an LOD in the sampler texture cache. An LOD must be aligned to a cache-line except for some special cases related to Planar YUV surfaces. In general, the cache-alignment restriction implies there is a minimum height for an LOD of 4 texels. So, LODs which are smaller than 4 high are padded.

For tiled surfaces with **TR_MODE** != TR_NONE this restriction is not significant because the MIP tail will be used for smaller MIPs and the slots are a minimum of 64B. For non-tiled surface or surfaces where **TR_MODE == TR_NONE**, Mips smaller than 4 high start at the top of the region, and they are padded. This padding leads to a case where the smallest LOD starts "below" LOD1 vertically.

Calculating Texel Location

This section describes how the texel location is calculated once the Surface State and LOD are known. A number of parameters are useful to determine where given pixels are located on the 2D surface. The width ($\mathbf{W}_L$) and height ($\mathbf{H}_L$) for each LOD "L" is computed by the formula:

W_L = ((width»L) > 0 ? width»L:1)

H_L = ((height»L) > 0? height»L:1)

When **Corner Texel Mode** is enabled via the RENDER_SURFACE_STATE, the width and height of a 2D surface are calculated as shown below:

WL = MAX(1,(W0-1)»L)+1

HL = MAX(1,(H0-1)»L)+1

There is a restriction that the smallest map dimension is 2 texels for **Corner Texel Mode (**W0 >1, H0 > 1). This also applies to 2D arrays and 2D arrays viewed as cubes.

The LOD width and height for each subsequent LOD is one-half the previous LOD, with the minimum dimension being 1 texel. If the surface is multisampled and it is a depth or stencil surface or **Multisampled Surface Storage Format** in SURFACE_STATE is MSFMT_DEPTH_STENCIL, $\mathbf{W}_L$ and $\mathbf{H}_L$ must be adjusted as follows:

Number of Multisamples	W_L =	H_L =
2	ceiling(W_L / 2) * 4	H_L [no adjustment]
4	ceiling(W_L / 2) * 4	ceiling(H_L / 2) * 4
8	ceiling(W_L / 2) * 8	ceiling(H_L / 2) * 4
16	ceiling(W_L / 2) * 8	ceiling(H_L / 2) * 8

Next, aligned width, height, and depth parameters for each LOD "L" must be computed. The "i" and "j" parameters are horizontal and vertical alignment parameters set by state fields or defined as constants, depending on the surface. Depth has no alignment parameter (effectively it is 1).

The equation uses the i and j values that apply to the LOD being computed. The "p" and "q" parameters define the width and height in texels of the compression block for compressed surface formats. Both p and q are defined to equal 1 for uncompressed surface formats.

$$w_L = i * p * ceil\left(\frac{W_L}{i * p}\right)$$

$$h_L = j * q * ceil\left(\frac{H_L}{j * q}\right)$$

Once the height (h_i) and width (w_i) of each LOD is computed, the offset to each LOD can be determined. The offset is a vector with two dimensions. The elements in the LOD_L vector are named in order $LODU_L$, $LODV_L$.

LOD offset computation for when no Mip Tail is used or when L < Mip Tail Start LOD:

$LOD_0 = (0,0)$

$LOD_1 = (0,h_0)$

$LOD_2 = (w_1,h_0)$

$LOD_3 = (w_1,h_0 + h_2)$

$LOD_4 = (w_1,h_0 + h_2 + h_3)$

…

$LOD_N = (w_1, h_0 + h_2 + h_3 \ldots + h_{N-1})$

Where N = MIP_COUNT for the surface. As noted previous in this section, the value of $h_2 + h_3 \ldots + h_{N-1}$ may be greater than h_1 due to alignment requirements.

Based on the above parameters and the U, V, and R (two dimensional pixel address U/V and array index R), and the *bytes* per pixel of the surface format (Bpp), the offsets u in bytes and v in rows are given by:

$u = (U + LODU_L) * Bpp$

$v = (R * QPitch) + LODV_L + V$

For a description of how the Mip Tail is laid out and offsets into the Mip Tail are calculated see the sub-section on 2D Surface Layout for Mip Tails.

Programming Note	
Context:	Packed YUV Surfaces
Packed YUV surface formats such as YCRCB_NORMAL, YCRCB_SWAPUVY etc. will be treated as 16bpp surface, not 32bpp, which may impact how they are layed out in memory.	

The two-dimensional offset into the surface (for non-MipTail cases) is defined by the u and v values computed above. The lower virtual address bits are determined by the following table, based on the bits of u and v. An *element* is defined as a pixel for uncompressed surface formats and a compression block for compressed surface formats. Empty bit positions indicate that the bit is not part of the tile swizzle and is filled in with equations given next (note that linear mode has all bits empty—there is no swizzling in linear mode).

The table below shows the mapping of u and v address bits within a tile for the supported tiling modes for a 2D surface. The u bits are a Byte address within a row and the v bits are a Row address within the tile.

Programming Note																			
Context:				Tiling Definition															
Tile Mode	Bits per Element	TileID constants		Virtual Address Bits															
		Cv	Cu	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TileYS	64 & 128	6	10	u9	v5	u8	v4	u7	v3	u6	v2	u5	u4	v1	v0	u3	u2	u1	u0
	16 & 32	7	9	u8	v6	u7	v5	u6	v4	u5	v3	u4	v2	v1	v0	u3	u2	u1	u0
	8	8	8	u7	v7	u6	v6	u5	v5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0
TileYF	64 & 128	4	8					u7	v3	u6	v2	u5	u4	v1	v0	u3	u2	u1	u0
	16 & 32	5	7					u6	v4	u5	v3	u4	v2	v1	v0	u3	u2	u1	u0
	8	6	6					u5	v5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0
TileY	all	5	7					u6	u5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0
TileX	all	3	9					v2	v1	v0	u8	u7	u6	u5	u4	u3	u2	u1	u0
TileW	all	6	6					u5	u4	u3	v5	v4	v3	v2	u2	v1	u1	v0	u0
Linear	all	0	0																

The TileID fills the upper bits of the virtual address (starting with the lowest blank bit in the above table):

TileID = (v » Cv) * (Pitch » Cu) + (u » Cu)

Where Pitch is the **Surface_Pitch** field from RENDER_SURFACE_STATE.

Note: Multisampled CMS and UMS surfaces use a modified address bit swizzling table rather than the one above. Refer to the *Multisampled*2D Surfaces section for details.

Tiling and Mip Tails for 2D Surfaces

When surface is Tiled (Tile_Mode=YMAJOR) and Tile Resources are enabled (TR_MODE != TR_NONE), a 2D surface can contain a Mip Tail for smaller Mip sizes.

When LOD (L) is less than the **Mip Tail Start LOD** (S) declared in the Surface State the offset to the start of LOD is calculated as shown above.

If the LOD is greater than or equal to **Mip Tail Start LOD** field in the surface state then the MIP Tail layout below is used.

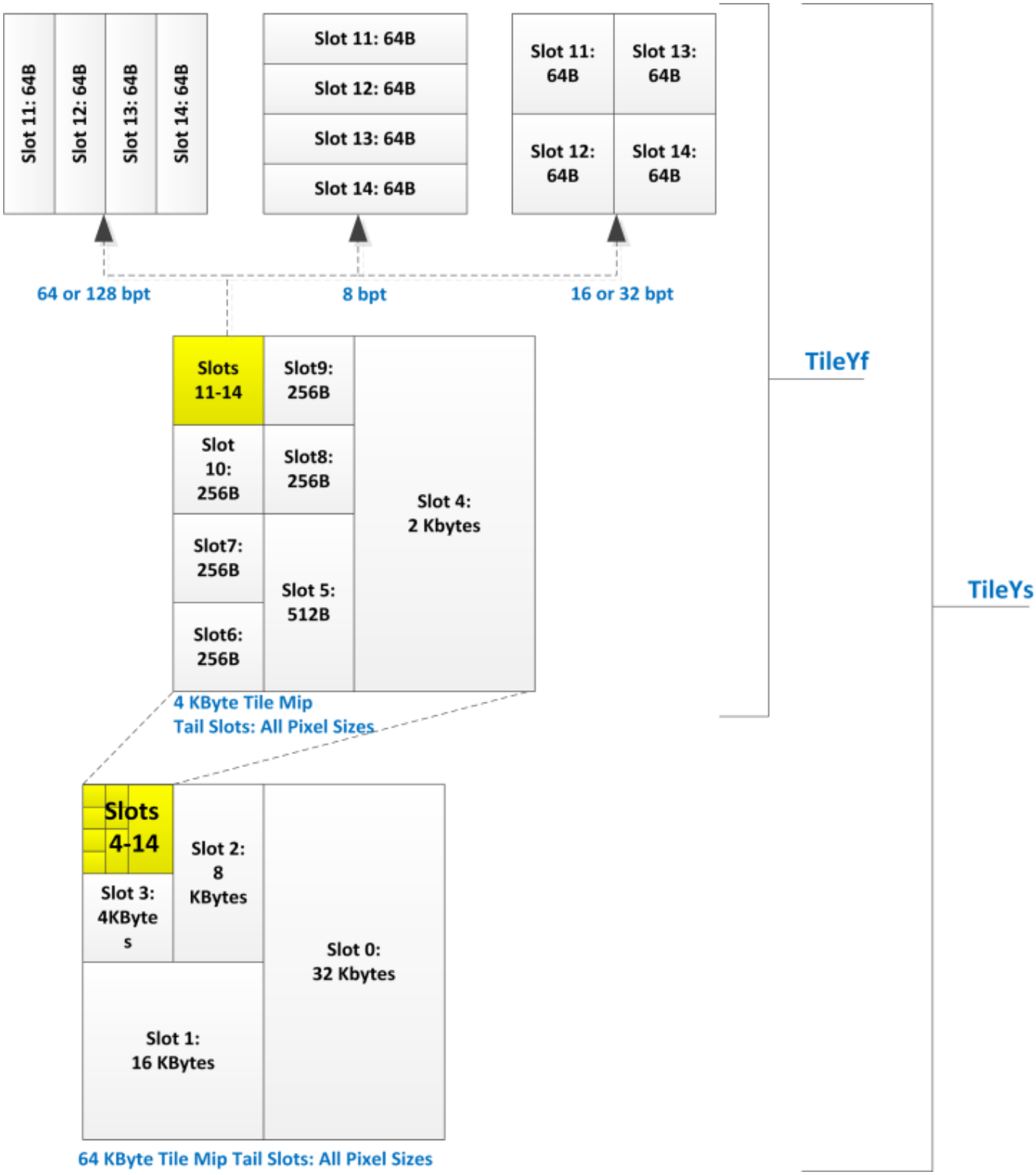
For tiled resources, the mip tail offset is given by the following, where s is the Mip Tail Start LOD:

$LOD_S = (w_1, h_0 + h_2 + h_3 + \ldots + h_{S-1})$

The LOD's in the Mip Tail are arranged differently than the other LOD's.

The diagram below shows the 64KB TileYS Mip Tail layout of LODs within it, with "slots" indicating the LOD contained within (slot 0 corresponds to LOD_s above). LOD's are aligned to the upper left corner of the space available. The block marked "Slots 4-14" is a 4KB tile arrangement as shown. Within this 4KB tile slots 11 thru 14 are arranged differently depending on the number bits per texel (bpt).

A TileYf (4KByte) Mip Tail will start with the 4KByte tile shown, but the slots will be renumbered to start at Slot0 rather than Slot4. The layout of slots 11 through 14 remain the same. Note that Slots 12-14 are NOT 256-Byte aligned which is not compliant with the standard MIP Tail layout. These slots are not supported for Standard Tiling.



The offsets *into the Mip Tail tile* are given by the following table for each LOD in the Mip tail. Each entry in the table is a horizontal (M_U) and vertical (M_V) position (in texels) from the upper left corner of the Mip Tail. If LOD >= S (starting LOD for MIP Tail), then these Mip Tail offsets must be added to the $LODU_L$ and $LODV_L$ calculated above.

Note that many of the higher LODs are not possible given surface size constraints, but they are listed here for reference. The offsets given here need to be added to the LODs offset computed earlier to obtain the offset into the surface LOD_L.

Slot 11 is 256-byte aligned. Slots 12 through 14 are 64-byte aligned.

TileYS LOD					TileYF LOD					128 bpe	64 bpe	32 bpe	16 bpe	8 bpe
1x	2x	4x	8x	16x	1x	2x	4x	8x	16x					
s										(32,0)	(64,0)	(64,0)	(128,0)	(128,0)
s+1	s									(0,32)	(0,32)	(0,64)	(0,64)	(0,128)
s+2	s+1	s								(16,0)	(32,0)	(32,0)	(64,0)	(64,0)
s+3	s+2	s+1	s							(0,16)	(0,16)	(0,32)	(0,32)	(0,64)
s+4	s+3	s+2	s+1	s	s					(8,0)	(16,0)	(16,0)	(32,0)	(32,0)
s+5	s+4	s+3	s+2	s+1	s+1	s				(4, 8)	(8, 8)	(8, 16)	(16, 16)	(16, 32)
s+6	s+5	s+4	s+3	s+2	s+2	s+1				(0, 12)	(0, 12)	(0, 24)	(0, 24)	(0, 48)
s+7	s+6	s+5	s+4	s+3	s+3	s+2				(0, 8)	(0, 8)	(0, 16)	(0, 16)	(0, 32)
s+8	s+7	s+6	s+5	s+4	s+4	s+3	s			(4, 4)	(8, 4)	(8, 8)	(16, 8)	(16, 16)
s+9	s+8	s+7	s+6	s+5	s+5	s+4	s+1			(4, 0)	(8, 0)	(8, 0)	(16, 0)	(16, 0)
s+10	s+9	s+8	s+7	s+6	s+6	s+5	s+2	s		(0, 4)	(0, 4)	(0, 8)	(0, 8)	(0, 16)
s+11	s+10	s+9	s+8	s+7	s+7	s+6	s+3	s+1	s	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)
s+12	s+11	s+10	s+9	s+8	s+8	s+7	s+4	s+2	s+1	(1, 0)	(2, 0)	(0, 4)	(0, 4)	(0, 4)
s+13	s+12	s+11	s+10	s+9	s+9	s+8	s+5	s+3	s+2	(2, 0)	(4, 0)	(4, 0)	(8, 0)	(0, 8)
s+14	s+13	s+12	s+11	s+10	s+10	s+9	s+6	s+4	s+3	(3, 0)	(6, 0)	(4, 4)	(8, 4)	(0, 12)

If the LOD is located in the MIP Tail then the equation for calculating the byte positions for u and v become:

$u = (U + LODU_S + M_U) * Bpp$
$v = (R * QPitch) + LODV_S + M_V + V$

where M_U and M_V are the offset parameters from the table above for the given slot in the MIP Tail.

Programming Note	
Context:	Lossless Compression and MIP Tail
Lossless compression must not be used on surfaces which have MIP Tail which contains MIPs for Slots greater than 11.	

Stencil Buffer Layout

This section details the layout of the stencil buffer.

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

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

4x Screen space(pixel, sample)							
(0,0)	(1,0)	(0,1)	(1,1)	(4,0)	(5,0)	(4,1)	(5,1)
(2,0)	(3,0)	(2,1)	(3,1)	(6,0)	(7,0)	(6,1)	(7,1)
(0,2)	(1,2)	(0,3)	(1,3)	(4,2)	(5,2)	(4,3)	(5,3)
(2,2)	(3,2)	(2,3)	(3,3)	(6,2)	(7,2)	(6,3)	(7,3)
(8,0)	(9,0)	(8,1)	(9,1)	(12,0)	(13,0)	(12,1)	(13,1)
(10,0)	(11,0)	(10,1)	(11,1)	(14,0)	(15,0)	(14,1)	(15,1)
(8,2)	(9,2)	(8,3)	(9,3)	(12,2)	(13,2)	(12,3)	(13,3)
(10,2)	(11,2)	(10,3)	(11,3)	(14,2)	(15,2)	(14,3)	(15,3)

4x Physical space(pixel, sample)															
(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)
(4,0)	(5,0)	(6,0)	(7,0)	(4,1)	(5,1)	(6,1)	(7,1)	(4,2)	(5,2)	(6,2)	(7,2)	(4,3)	(5,3)	(6,3)	(7,3)
(8,0)	(9,0)	(10,0)	(11,0)	(8,1)	(9,1)	(10,1)	(11,1)	(8,2)	(9,2)	(10,2)	(11,2)	(8,3)	(9,3)	(10,3)	(11,3)
(12,0)	(13,0)	(14,0)	(15,0)	(12,1)	(13,1)	(14,1)	(15,1)	(12,2)	(13,2)	(14,2)	(15,2)	(12,3)	(13,3)	(14,3)	(15,3)

8x Screen space(pixel, sample)							
(0,0)	(1,0)	(0,1)	(1,1)	(0,4)	(1,4)	(0,5)	(1,5)
(2,0)	(3,0)	(2,1)	(3,1)	(2,4)	(3,4)	(2,5)	(3,5)
(0,2)	(1,2)	(0,3)	(1,3)	(0,6)	(1,6)	(0,7)	(1,7)
(2,2)	(3,2)	(2,3)	(3,3)	(2,6)	(3,6)	(2,7)	(3,7)
(4,0)	(5,0)	(4,1)	(5,1)	(4,4)	(5,4)	(4,5)	(5,5)
(6,0)	(7,0)	(6,1)	(7,1)	(6,4)	(7,4)	(6,5)	(7,5)
(4,2)	(5,2)	(4,3)	(5,3)	(4,6)	(5,6)	(4,7)	(5,7)
(6,2)	(7,2)	(6,3)	(7,3)	(6,6)	(7,6)	(6,7)	(7,7)

8x Physical space(pixel, sample)															
(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)
(0,4)	(1,4)	(2,4)	(3,4)	(0,5)	(1,5)	(2,5)	(3,5)	(0,6)	(1,6)	(2,6)	(3,6)	(0,7)	(1,7)	(2,7)	(3,7)
(4,0)	(5,0)	(6,0)	(7,0)	(4,1)	(5,1)	(6,1)	(7,1)	(4,4)	(5,4)	(6,4)	(7,4)	(4,5)	(5,5)	(6,5)	(7,5)
(4,2)	(5,2)	(6,2)	(7,2)	(4,3)	(5,3)	(6,3)	(7,3)	(4,6)	(5,6)	(6,6)	(7,6)	(4,7)	(5,7)	(6,7)	(7,7)

2D/CUBE Alignment Requirement

The vertical and horizontal alignment fields in the RENDER_SURFACE_STATE are ignored for standard tiling formats (TRMODE = NONE). In the case of standard tiling formats the alignment requirements are fixed and are provided for by the tables below for 2D and CUBE surface.

Tile Mode	Bits per Element	Horizontal Alignment	Vertical Alignment
TileYS	128	64	64
	64	128	64
	32	128	128
	16	256	128
	8	256	256
TileYF	128	16	16
	64	32	16
	32	32	32
	16	64	32
	8	64	64

For MSFMT_MSS type multi-sampled TileYS and TileYF surfaces, the alignments given above must be divided by the appropriate value from the table below.

Number of Multisamples	Horizontal Alignment is divided by	Vertical Alignment is divided by
2	2	1
4	2	2
8	4	2
16	4	4

Multisampled 2D Surfaces

There are three types of multisampled surface layouts designated as follows:

• **IMS** Interleaved Multisampled Surface

• **CMS** Compressed Mulitsampled Surface

• **UMS** Uncompressed Multisampled Surface

These surface layouts are described in the following sections.

Interleaved Multisampled Surfaces

IMS surfaces are supported in all generations for depth and stencil surfaces. These surfaces contain the samples in an interleaved fashion, with the underlying surface in memory having a height and width that is larger than the non-multisampled surface as follows:

- 4x MSAA: 2x width and 2x height of non-multisampled surface.
- 8x MSAA: 4x width and 2x height of non-multisampled surface.
- 16x MSAA: 4X width and 4X height of the non-multisampled surface.

When sampling from an IMS surface (e.g. ld2dms), the coordinates are automatically scaled to handle the increased physical size of the map.

The tables below shows the layout of 16-bit Depth (Z) values for different IMS formats. It shows layout of each 64-Byte chunk.

1X:

	bit 0							**bit 127**
Bytes 15:0	P(0,0)	P(1,0)	P(2,0)	P(3,0)	P(4,0)	P(5,0)	P(6,0)	P(7,0)
Bytes 16:31	P(0,1)	P(1,1)	P(2,1)	P(3,1)	P(4,1)	P(5,1)	P(6,1)	P(7,1)
Bytes 32:47	P(0,2)	P(1,2)	P(2,2)	P(3,2)	P(4,2)	P(5,2)	P(6,2)	P(7,2)
Bytes 48:63	P(0,3)	P(1,3)	P(2,3)	P(3,3)	P(4,3)	P(5,3)	P(6,3)	P(7,3)

2X:

	bit 0							**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1	P(2,0) s0	P(3,0) s0	P(2,0) s1	P(3,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1	P(2,1) s0	P(3,1) s0	P(2,1) s1	P(3,1) s1
Bytes 32:47	P(0,2) s0	P(1,2) s0	P(0,2) s1	P(1,2) s1	P(2,2) s0	P(3,2) s0	P(2,2) s1	P(3,2) s1
Bytes 48:63	P(0,3) s0	P(1,3) s0	P(0,3) s1	P(1,3) s1	P(2,3) s0	P(3,3) s0	P(2,3) s1	P(3,3) s1

4X:

	bit 0							**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1	P(2,0) s0	P(3,0) s0	P(2,0) s1	P(3,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1	P(2,1) s0	P(3,1) s0	P(2,1) s1	P(3,1) s1
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3	P(2,0) s2	P(3,0) s2	P(2,0) s3	P(3,0) s3
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3	P(2,1) s2	P(3,1) s2	P(2,1) s3	P(3,1) s3

8X:

	bit 0							**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1	P(0,0) s4	P(1,0) s4	P(0,0) s5	P(1,0) s5
Bytes 16:31	P(0,1) s0	P(1,1) s1	P(0,1) s1	P(1,1) s1	P(0,1) s4	P(1,1) s4	P(0,1) s5	P(1,1) s5
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3	P(0,0) s6	P(1,0) s6	P(0,0) s7	P(1,0) s7
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3	P(0,1) s6	P(1,1) s6	P(0,1) s7	P(1,1) s7

16X:

	bit 0							**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1	P(0,0) s4	P(1,0) s4	P(0,0) s5	P(1,0) s5
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1	P(0,1) s4	P(1,1) s4	P(0,1) s5	P(1,1) s5
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3	P(0,0) s6	P(1,0) s6	P(0,0) s7	P(1,0) s7
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3	P(0,1) s6	P(1,1) s6	P(0,1) s7	P(1,1) s7

	bit 0							**bit 127**
Bytes 64:79	P(0,0) s8	P(1,0) s8	P(0,0) s9	P(1,0) s9	P(0,0) s12	P(1,0) s12	P(0,0) s13	P(1,0) s13
Bytes 80:95	P(0,1) s8	P(1,1) s8	P(0,1) s9	P(1,1) s9	P(0,1) s12	P(1,1) s12	P(0,1) s13	P(1,1) s13
Bytes 96:111	P(0,0) s10	P(1,0) s10	P(0,0) s11	P(1,0) s11	P(0,0) s14	P(1,0) s14	P(0,0) s15	P(1,0) s15
Bytes 112:127	P(0,1) s10	P(1,1) s10	P(0,1) s11	P(1,1) s11	P(0,1) s14	P(1,1) s14	P(0,1) s15	P(1,1) s15

The table below shows the layout of 32-bit Depth (Z) values for different IMS formats. It shows layout of each 64-Byte chunk.

1X:

	bit 0			**bit 127**	
Bytes 15:0	P(0,0)	P(1,0)	P(2,0)	P(3,0)	
Bytes 16:31	P(0,1)	P(1,1)	P(2,1)	P(3,1)	
Bytes 32:47	P(0,2)	P(1,2)	P(2,2)	P(3,2)	
Bytes 48:63	P(0,3)	P(1,3)	P(2,3)	P(3,3)	

2X:

	bit 0			**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1
Bytes 32:47	P(0,2) s0	P(1,2) s0	P(0,2) s1	P(1,2) s1
Bytes 48:63	P(0,3) s0	P(1,3) s0	P(0,3) s1	P(1,3) s1

4X:

	bit 0			**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3

8X:

	bit 0			**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3

16X:

	bit 0			**bit 127**
Bytes 15:0	P(0,0) s0	P(1,0) s0	P(0,0) s1	P(1,0) s1
Bytes 16:31	P(0,1) s0	P(1,1) s0	P(0,1) s1	P(1,1) s1
Bytes 32:47	P(0,0) s2	P(1,0) s2	P(0,0) s3	P(1,0) s3
Bytes 48:63	P(0,1) s2	P(1,1) s2	P(0,1) s3	P(1,1) s3

	bit 0			**bit 127**
Bytes 64:79	P(0,0) s4	P(1,0) s4	P(0,0) s5	P(1,0) s5
Bytes 80:95	P(0,1) s4	P(1,1) s4	P(0,1) s5	P(1,1) s5
Bytes 96:111	P(0,0) s6	P(1,0) s6	P(0,0) s7	P(1,0) s7
Bytes 112:127	P(0,1) s6	P(1,1) s6	P(0,1) s7	P(1,1) s7

	bit 0			bit 127
Bytes 128:143	P(0,0) s8	P(1,0) s8	P(0,0) s9	P(1,0) s9
Bytes 144:159	P(0,1) s8	P(1,1) s8	P(0,1) s9	P(1,1) s9
Bytes 160:175	P(0,0) s10	P(1,0) s10	P(0,0) s11	P(1,0) s11
Bytes 176:191	P(0,1) s10	P(1,1) s10	P(0,1) s11	P(1,1) s11

	bit 0			bit 127
Bytes 192:207	P(0,0) s12	P(1,0) s12	P(0,0) s13	P(1,0) s13
Bytes 208:223	P(0,1) s12	P(1,1) s12	P(0,1) s13	P(1,1) s13
Bytes 224:239	P(0,0) s14	P(1,0) s14	P(0,0) s15	P(1,0) s15
Bytes 240:255	P(0,1) s14	P(1,1) s14	P(0,1) s15	P(1,1) s15

The table below shows the layout of Depth Stencil values for different IMS formats. It shows layout of each 64-Byte chunk.

1X:

Bytes 0-7	P(0,0)	P(1,0)	P(0,1)	P(1,1)	P(2,0)	P(3,0)	P(2,1)	P(3,1)
Bytes 8-15	P(0,2)	P(1,2)	P(0,3)	P(1,3)	P(2,2)	P(3,2)	P(2,3)	P(3,3)
Bytes 16-23	P(4,0)	P(5,0)	P(4,1)	P(5,1)	P(6,0)	P(7,0)	P(6,1)	P(7,1)
Bytes 24-31	P(4,2)	P(5,2)	P(4,3)	P(5,3)	P(6,2)	P(7,2)	P(6,3)	P(7,3)
Bytes 32-39	P(0,4)	P(1,4)	P(0,5)	P(1,5)	P(2,4)	P(3,4)	P(2,5)	P(3,5)
Bytes 40-47	P(0,6)	P(1,6)	P(0,7)	P(1,7)	P(2,6)	P(3,6)	P(2,7)	P(3,7)
Bytes 48-55	P(4,4)	P(5,4)	P(4,5)	P(5,5)	P(6,4)	P(7,4)	P(6,5)	P(7,5)
Bytes 56-63	P(4,6)	P(5,6)	P(4,7)	P(5,7)	P(6,6)	P(7,6)	P(6,7)	P(7,7)

2X:

Bytes 0-7	P(0,0) s0	P(1,0) s0	P(0,1) s0	P(1,1) s0	P(0,0) s1	P(1,0) s1	P(0,1) s1	P(1,1) s1
Bytes 8-15	P(0,2) s0	P(1,2) s0	P(0,3) s0	P(1,3) s0	P(0,2) s1	P(1,2) s1	P(0,3) s1	P(1,3) s1
Bytes 16-23	P(2,0) s0	P(3,0) s0	P(2,1) s0	P(3,1) s0	P(2,0) s1	P(3,0) s1	P(2,1) s1	P(3,1) s1
Bytes 24-31	P(2,2) s0	P(3,2) s0	P(2,3) s0	P(3,3) s0	P(2,2) s1	P(3,2) s1	P(2,3) s1	P(3,3) s1
Bytes 32-39	P(0,4) s0	P(1,4) s0	P(0,5) s0	P(1,5) s0	P(0,4) s1	P(1,4) s1	P(0,5) s1	P(1,5) s1
Bytes 40-47	P(0,6) s0	P(1,6) s0	P(0,7) s0	P(1,7) s0	P(0,6) s1	P(1,6) s1	P(0,7) s1	P(1,7) s1
Bytes 48-55	P(2,4) s0	P(3,4) s0	P(2,5) s0	P(3,5) s0	P(2,4) s1	P(3,4) s1	P(2,5) s1	P(3,5) s1
Bytes 56-63	P(2,6) s0	P(3,6) s0	P(2,7) s0	P(3,7) s0	P(2,6) s1	P(3,6) s1	P(2,7) s1	P(3,7) s1

4X:

Bytes 0-7	P(0,0) s0	P(1,0) s0	P(0,1) s0	P(1,1) s0	P(0,0) s1	P(1,0) s1	P(0,1) s1	P(1,1) s1
Bytes 8-15	P(0,0) s2	P(1,0) s2	P(0,1) s2	P(1,1) s2	P(0,0) s3	P(1,0) s3	P(0,1) s3	P(1,1) s3
Bytes 16-23	P(2,0) s0	P(3,0) s0	P(2,1) s0	P(3,1) s0	P(2,0) s1	P(3,0) s1	P(2,1) s1	P(3,1) s1
Bytes 24-31	P(2,0) s2	P(3,0) s2	P(2,1) s2	P(3,1) s2	P(2,0) s3	P(3,0) s3	P(2,1) s3	P(3,1) s3
Bytes 32-39	P(0,2) s0	P(1,2) s0	P(0,3) s0	P(1,3) s0	P(0,2) s1	P(1,2) s1	P(0,3) s1	P(1,3) s1
Bytes 40-47	P(0,2) s2	P(1,2) s2	P(0,3) s2	P(1,3) s2	P(0,2) s3	P(1,2) s3	P(0,3) s3	P(1,3) s3
Bytes 48-55	P(2,2) s0	P(3,2) s0	P(2,3) s0	P(3,3) s0	P(2,2) s1	P(3,2) s1	P(2,3) s1	P(3,3) s1
Bytes 56-63	P(2,2) s2	P(3,2) s2	P(2,3) s2	P(3,3) s2	P(2,2) s3	P(3,2) s3	P(2,3) s3	P(3,3) s3

8X:

Bytes 0-7	P(0,0) s0	P(1,0) s0	P(0,1) s0	P(1,1) s0	P(0,0) s1	P(1,0) s1	P(0,1) s1	P(1,1) s1
Bytes 8-15	P(0,0) s2	P(1,0) s2	P(0,1) s2	P(1,1) s2	P(0,0) s3	P(1,0) s3	P(0,1) s3	P(1,1) s3
Bytes 16-23	P(0,0) s4	P(1,0) s4	P(0,1) s4	P(1,1) s4	P(0,0) s5	P(1,0) s5	P(0,1) s5	P(1,1) s5
Bytes 24-31	P(0,0) s6	P(1,0) s6	P(0,1) s6	P(1,1) s6	P(0,0) s7	P(1,0) s7	P(0,1) s7	P(1,1) s7
Bytes 32-39	P(0,2) s0	P(1,2) s0	P(0,3) s0	P(1,3) s0	P(0,2) s1	P(1,2) s1	P(0,3) s1	P(1,3) s1
Bytes 40-47	P(0,2) s2	P(1,2) s2	P(0,3) s2	P(1,3) s2	P(0,2) s3	P(1,2) s3	P(0,3) s3	P(1,3) s3
Bytes 48-55	P(0,2) s4	P(1,2) s4	P(0,3) s4	P(1,3) s4	P(0,2) s5	P(1,2) s5	P(0,3) s5	P(1,3) s5
Bytes 56-63	P(0,2) s6	P(1,2) s6	P(0,3) s6	P(1,3) s6	P(0,2) s7	P(1,2) s7	P(0,3) s7	P(1,3) s7

16X:

Bytes 0-7	P(0,0) s0	P(1,0) s0	P(0,1) s0	P(1,1) s0	P(0,0) s1	P(1,0) s1	P(0,1) s1	P(1,1) s1
Bytes 8-15	P(0,0) s2	P(1,0) s2	P(0,1) s2	P(1,1) s2	P(0,0) s3	P(1,0) s3	P(0,1) s3	P(1,1) s3
Bytes 16-23	P(0,0) s4	P(1,0) s4	P(0,1) s4	P(1,1) s4	P(0,0) s5	P(1,0) s5	P(0,1) s5	P(1,1) s5
Bytes 24-31	P(0,0) s6	P(1,0) s6	P(0,1) s6	P(1,1) s6	P(0,0) s7	P(1,0) s7	P(0,1) s7	P(1,1) s7
Bytes 32-39	P(0,0) s8	P(1,0) s8	P(0,1) s8	P(1,1) s8	P(0,0) s9	P(1,0) s9	P(0,1) s9	P(1,1) s9
Bytes 40-47	P(0,0) s10	P(1,0) s10	P(0,1) s10	P(1,1) s10	P(0,0) s11	P(1,0) s11	P(0,1) s11	P(1,1) s11
Bytes 48-55	P(0,0) s12	P(1,0) s12	P(0,1) s12	P(1,1) s12	P(0,0) s13	P(1,0) s13	P(0,1) s13	P(1,1) s13
Bytes 56-63	P(0,0) s14	P(1,0) s14	P(0,1) s14	P(1,1) s14	P(0,0) s15	P(1,0) s15	P(0,1) s15	P(1,1) s15

Compressed Multisampled Surfaces

Multisampled render targets can be compressed. If **Auxiliary Surface Mode** in SURFACE_STATE is set to AUX_CCS, hardware handles the compression using a software-invisible algorithm. However, performance optimizations in the multisample resolve kernel using the sampling engine are possible if the internal format of these surfaces is understood by software. This section documents the formats of the Multisample Control Surface (MCS) and Multisample Surface (MSS).

MCS Surface

The MCS surface consists of one element per pixel, with the element size being an 8-bit unsigned integer value for 4x multisampled surfaces, a 32-bit unsigned integer value for 8x multisampled surfaces and a 64-bit unsigned integer value for 16x multisampled surface. Each field within the element indicates which sample slice (SS) the sample resides on.

2x MCS

The 2x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

7:2	1	0
reserved	sample 1 SS	sample 0 SS

Each 1-bit field indicates which sample slice (SS) the sample's color value is stored. An MCS value of 0x00 indicates that both samples are stored in sample slice 0 (thus have the same color). This is the fully compressed case. An MCS value of 0x03 indicates that all samples in the pixel are in the clear state, and none of the sample slices are valid. The pixel's color must be replaced with the surface's clear value.

4x MCS

The 4x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

7:6	5:4	3:2	1:0
sample 3 SS	sample 2 SS	sample 1 SS	sample 0 SS

Each 2-bit field indicates which sample slice (SS) the sample's color value is stored. An MCS value of 0x00 indicates that all four samples are stored in sample slice 0 (thus all have the same color). This is the fully compressed case. An MCS value of 0xff indicates that all samples in the pixel are in the clear state, and none of the sample slices are valid. The pixel's color must be replaced with the surface's clear value. See the section below on Clear Pixel Conditions for additional encoding information.

8x MCS

Extending the mechanism used for the 4x MCS to 8x requires 3 bits per sample times 8 samples, or 24 bits per pixel. The 24-bit MCS value per pixel is placed in a 32-bit footprint, with the upper 8 bits unused as shown below. See the section below on Clear Pixel Conditions for additional encoding information.

31:24	23:21	20:18	17:15	14:12	11:9	8:6	5:3	2:0
reserved (MBZ)	sample 7 SS	sample 6 SS	sample 5 SS	sample 4 SS	sample 3 SS	sample 2 SS	sample 1 SS	sample 0 SS

16x MCS

The 16x MCS is 64 bits per pixel. The 64 bits are encoded as follows:

63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32
sample 15 SS	sample 14 SS	sample 13 SS	sample 12 SS	sample 11 SS	sample 10 SS	sample 9 SS	sample 8 SS

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
sample 7 SS	sample 6 SS	sample 5 SS	sample 4 SS	sample 3 SS	sample 2 SS	sample 1 SS	sample 0 SS

Other than this, the 16x algorithm is the same as the 8x algorithm. The MCS value indicating clear state is 0xffffffff_ffffffff. See the section below on Clear Pixel Conditions for additional encoding information.

Clear Pixel Conditions

The MCS format allows for the encoding of clear value for one or more planes of the multi-sampled surface. A value of all 1's for defined MCS bits indicates that all planes of the multi-sampled surface are clear. For example, a value of 0x3 for 2X MSAA MCS byte means that both planes of the pixel are clear. Likewise, a value of 0xff for X4, 0xffffffff for X8 and 0xffffffff_ffffffff for X16 MSAA means that all planes of the pixel are clear.

In the case where not all planes are clear, but at least 2 planes are clear the encoding of the MCS given above is changed. If the MCS value for plane 0 is non-zero, then all planes which are at all 1's are clear and all other planes are referencing the plane indicated by their respective MCS value minus 1. For example, a 4X MSAA MCS value of 01 10 11 11 means that MCS 0 is referencing plane 0, and MCS 1 is referencing plane 1, and MCS 2 and 3 are clear.

MSS Surface

The physical MSS surface is stored identically to a 2D array surface, with the height and width matching the pixel dimensions of the logical multisampled surface. The number of array slices in the physical surface is 2, 4, 8, or 16 times that of the logical surface (depending on the number of multisamples). Sample slices belonging to the same logical surface array slice are stored in adjacent physical slices. The sampling engine ld2dss message gives direct access to a specific sample slice.

Tiling for CMS and UMS Surfaces

Multisampled CMS and UMS use a modified table from non-mulitsampled 2D surfaces.

TileY,

TileX, TileW, and Linear: Treat as 2D array, with the array index “R” modified as follows. “n” is the number of multisamples, “ss” is the sample slice index with range 0..n-1.

```
R(new) = ( R(old) « log2(n) ) | ss
```

TileYS: In addition to u and v, the sample slice index “ss” is included in the address swizzling according to the following table. Because of this, the mip tail holds one less LOD for each successive number of multisamples. Refer to the mip tail table in the previous section for behavior of the mip tail for each number of multisamples.

Number of Multisamples	Bits per Element	TileID constants		Virtual Address Bits									
		Cv	Cu	15	14	13	12	11	10	9	8	7	6
2x	64 & 128	6	9	ss0	v5	u8	v4	u7	v3	u6	v2	u5	u4
	16 & 32	7	8	ss0	v6	u7	v5	u6	v4	u5	v3	u4	v2
	8	8	7	ss0	v7	u6	v6	u5	v5	u4	v4	v3	v2
4x	64 & 128	5	9	ss1	ss0	u8	v4	u7	v3	u6	v2	u5	u4
	16 & 32	6	8	ss1	ss0	u7	v5	u6	v4	u5	v3	u4	v2
	8	7	7	ss1	ss0	u6	v6	u5	v5	u4	v4	v3	v2
8x	64 & 128	5	8	ss2	ss1	ss0	v4	u7	v3	u6	v2	u5	u4
	16 & 32	6	7	ss2	ss1	ss0	v5	u6	v4	u5	v3	u4	v2
	8	7	6	ss2	ss1	ss0	v6	u5	v5	u4	v4	v3	v2
16x	64 & 128	4	8	ss3	ss2	ss1	ss0	u7	v3	u6	v2	u5	u4
	16 & 32	5	7	ss3	ss2	ss1	ss0	u6	v4	u5	v3	u4	v2
	8	6	6	ss3	ss2	ss1	ss0	u5	v5	u4	v4	v3	v2

Note that Cv and Cu are also different that the values for non-multisampled 2D surfaces.

TileYF: In addition to u and v, the sample slice index “ss” is included in the address swizzling according to the following table. Because of this, the mip tail holds one less LOD for each successive number of multisamples. Refer to the mip tail table in the previous section for behavior of the mip tail for each number of multisamples.

Number of Multisamples	Bits per Element	TileID constants		Virtual Address Bits					
		Cv	Cu	11	10	9	8	7	6
2x	128 & 64	4	7	ss0	v3	u6	v2	u5	u4
	32 & 16	5	6	ss0	v4	u5	v3	u4	v2
	8	6	5	ss0	v5	u4	v4	v3	v2
4x	128 & 64	3	7	ss1	ss0	u6	v2	u5	u4
	32 & 16	4	6	ss1	ss0	u5	v3	u4	v2
	8	5	5	ss1	ss0	u4	v4	v3	v2
8x	128 & 64	3	6	ss2	ss1	ss0	v2	u5	u4
	32 & 16	4	5	ss2	ss1	ss0	v3	u4	v2
	8	5	4	ss2	ss1	ss0	v4	v3	v2
16x	128 & 64	2	6	ss3	ss2	ss1	ss0	u5	u4
	32 & 16	3	5	ss3	ss2	ss1	ss0	u4	v2
	8	4	4	ss3	ss2	ss1	ss0	v3	v2

Uncompressed Multisampled Surfaces

UMS surfaces similar to CMS, except that the **Auxiliary Surface Mode** is set to AUX_NONE, meaning that there is no MCS surface. UMS contains only an MSS surface, where each sample is stored on its sample slice (SS) of the same index.

Programming Note	
Context:	3D Sampler
See 3D Sampler Messages section for a description of how the ld2dms and ld2dms_w messages work for UMS surfaces	

Quilted Textures

A quilted texture is a 2D texture made up of quilt slices, each of which is a portion of the surface up to 16k x 16k texels in size. The quilt slices themselves are organized in a matrix up to 32 x 32. “Quilt Width” and “Quilt Height” fields indicate the dimensions of the surface in quilt slices. “Height” and “Width” fields indicate the size of each quilt slice in texels. The total size of the quilted texture can be up to 512k x 512k texels.

In addition, arrays of quilted textures are supported. The total number of array slices is limited to 2048 / (QuiltWidth * QuiltHeight).

A surface is defined as a “Quilted Texture” if either the “Quilt Width” or “Quilt Height” field in SURFACE_STATE is nonzero. A quilted texture is stored in the storage format as a 2D array, with each quilt square occupying one array slice. The following equation indicates how the array slice is computed from the Qx, Qy, and R parameters, where Qx and Qy are the quilt slice coordinates and R is the array index.

```
ArraySliceIndex = (R * QuiltWidth * QuiltHeight) + Qy * QuiltWidth + Qx
```

Quilted textures do NOT support wrapping. U and V coordinates must be in the range of [0.0,1.0).

Cube Surfaces

The 3D Pipeline supports *cubic environment maps*, conceptually arranged as a cube surrounding the origin of a 3D coordinate system aligned to the cube faces. These maps can be used to supply texel (color/alpha) data of the environment in any direction from the enclosed origin, where the direction is supplied as a 3D “vector” texture coordinate. These cube maps can also be mipmapped.

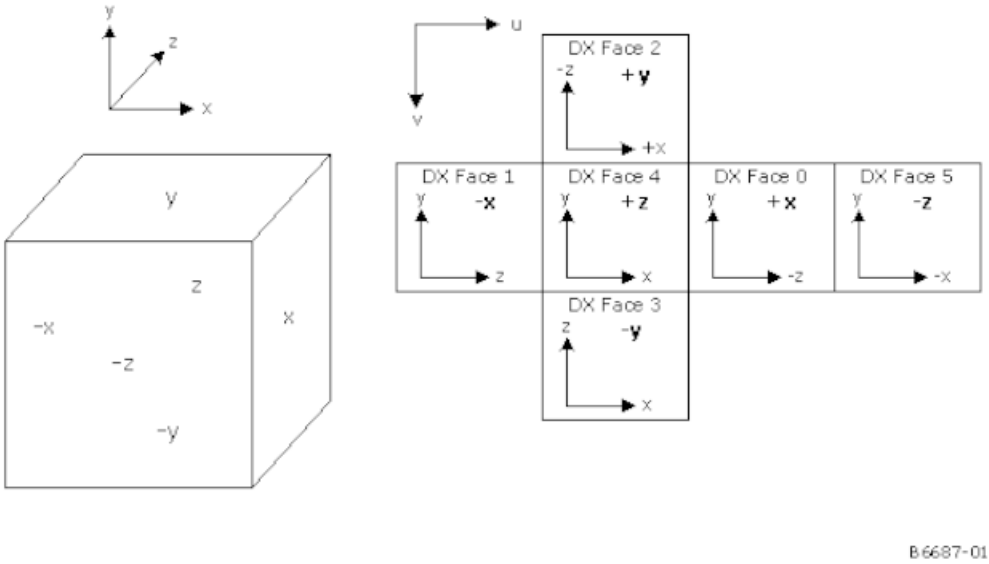
Each texture map level is represented as a group of six, square *cube face* texture surfaces. The faces are identified by their relationship to the 3D texture coordinate system. The subsections below describe the cube maps as described at the API as well as the memory layout dictated by the hardware.

The diagram below describes the cube map faces as they are defined at the DirectX API. It shows the axes on the faces as they would be seen from the inside (at the origin).

The 3D sampler converts the incoming U,V,R coordinates on the sampler

This will be looking directly at face 4, the +z –face. Y is up by default.

DirectX Cube Map Definition



The coordinates on each face are relative to the center of the cube, and they range from -1.0 to 1.0 rather than the normal 0 to 1.0 normalized coordinate system in a 2D array surface.

Each face has a corresponding face identifier "f" as indicated in the following table:

face	face identifier "f"
+x	0
-x	1
+y	2
-y	3
+z	4
-z	5

A cube surface is stored in memory the same as a 2D array, with the face identifier "f" and array index "ai" being transformed into the "R" coordinate used in storing 2D arrays using the following equation:

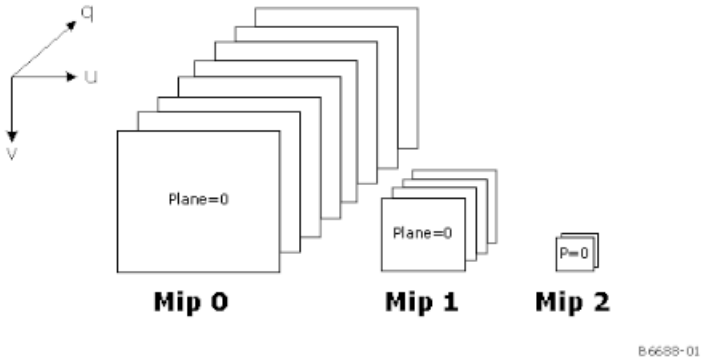
```
R = (ai * 6) + f
```

Refer to the "2D Surfaces" section for details on how 2D arrays are stored.

3D Surfaces

Multiple texture map surfaces (and their respective mipmap chains) can be arranged into a structure known as a Texture3D (volume) texture. A volume texture map consists of many *planes* of 2D texture maps. See *Sampler* for a description of how volume textures are used.

Volume Texture Map



Surface Pitch defines the distance in bytes between rows of the surface. **Surface QPitch** specifies the distance in rows between R-slices. QPitch should allow at least enough space for any mips that may be present.

A number of parameters are useful to determine where given pixels are located on the 3D surface. First, the width, height, and depth for each LOD "L" is computed:

$$W_L = ((width >> L) > 0\,?\,width >> L : 1)$$
$$H_L = ((height >> L) > 0\,?\,height >> L : 1)$$
$$D_L = ((depth >> L) > 0\,?\,depth >> L : 1)$$

When **Corner Texel Mode** is enabled via the RENDER_SURFACE_STATE, the width and height of a 3D surface are calculated as shown below:

$W_L = MAX(1,(W_0-1)»L)+1$

$H_L = MAX(1,(H_0-1)»L)+1$

$D_L = MAX(1,(D_0-1)»L)+1$

There is a restriction that the smallest map dimension is 2 texels for **Corner Texel Mode** ($W_0 > 1$, $H_0 > 1$, $D_0 > 1$)

Next, aligned width, height, and depth parameters for each LOD "L" are computed. The "i", "j", and "k" parameters are the horizontal, vertical, and depth alignment parameters set by state fields or defined as constants. The alignment parameters may change at one point in the mip chain based on **Mip Tail Start LOD**. The equation uses the i/j values that apply to the LOD being computed. The "p", "q", and "s" parameters define the width, height, and depth in texels of the compression block for compressed surface formats. These are all defined to equal 1 for uncompressed surface formats.

$$w_L = i * ceil\left(\frac{W_L}{i}\right)$$
$$h_L = j * ceil\left(\frac{H_L}{j}\right)$$
$$d_L = D_L$$
$$d_L = k * s * ceil\left(\frac{D_L}{k * s}\right)$$

Next, the offset to each LOD is determined. The offset is a vector with three dimensions. The elements in the LOD_L vector are named in order $LODU_L$, $LODV_L$, $LODR_L$.

LOD offset computation for **Tiled Resource Mode** == TR_NONE or when L < **Mip Tail Start LOD**:

$$LOD_0 = (0,0,0)$$
$$LOD_1 = (0,h_0,0)$$
$$LOD_2 = (w_1,h_0,0)$$
$$LOD_3 = (w_1,h_0+h_2,0)$$
$$LOD_4 = (w_1,h_0+h_2+h_3,0)$$
...

For the Primary Surface

Based on the above parameters and the U, V, and R (three-dimensional pixel address), and the bytes per pixel of the surface format (Bpp), the offsets u in bytes, v in rows, and r in slices are given by:

```
u = [U + LODUL] * Bpp
v = LODVL + V
r = LODRL + R
```

Programming Note	
Context:	Packed YUV Surfaces
Packed YUV surface formats such as YCRCB_NORMAL, YCRCB_SWAPUVY etc. will be treated as 16bpp surface, not 32bpp, which may impact how they are layed out in memory.	

The three dimensional offset into the surface is defined by the u, v, and r values computed above. The lower virtual address bits are determined by the following table, based on the bits of u, v, and r. An *element* is defined as a pixel for uncompressed surface formats and a compression block for compressed surface formats.

Empty bit positions indicate that the bit is not part of the tile swizzle and is filled in with the equations given next (note that linear mode has all bits empty—there is no swizzling in linear mode).

Tile Mode	Bits per Element	TileID constants			Virtual Address Bits															
		Cr	Cv	Cu	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TileYS	128 & 64	4	4	8	u7	v3	r3	u6	v2	r2	u5	u4								
	32	4	5	7	u6	v4	r3	u5	v3	r2	u4	v2	r1	r0	v1	v0	u3	u2	u1	u0
	16 & 8	5	5	6	u5	v4	r4	u4	v3	r3	v2	r2	r1	r0	v1	v0	u3	u2	u1	u0
TileYF	128 & 64	3	3	6					v2	r2	u5	u4	r1	r0	v1	v0	u3	u2	u1	u0
	32	3	4	5					v3	r2	u4	v2	r1	r0	v1	v0	u3	u2	u1	u0
	16 & 8	4	4	4					v3	r3	v2	r2	r1	r0	v1	v0	u3	u2	u1	u0
TileY	all	0	5	7					u6	u5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0
Linear	all	0	0	0																

The table below is enabled by use of the Tile Address Mapping Mode bi in **RENDER_SURFACE_STATE**.

This mapping must never be used.

Tile Mode	Bits per Element	TileID constants			Virtual Address Bits															
		Cr	Cv	Cu	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TileYS	128 & 64	4	4	8	u7	v3	r3	u6	v2	r2	u5	v1	r1	u4	r0	v0	u3	u2	u1	u0
	32	4	5	7	u6	v4	r3	u5	v3	r2	u4	v2	r1	u3	v1	v0	r0	u2	u1	u0
	16 & 8	5	5	6	u5	v4	r4	u4	v3	r3	u3	v2	r2	u2	v1	v0	r1	r0	u1	u0
TileYF	128 & 64	3	3	6					v2	r2	u5	v1	r1	u4	r0	v0	u3	u2	u1	u0
	32	3	4	5					v3	r2	u4	v2	r1	u3	v1	v0	r0	u2	u1	u0
	16 & 8	4	4	4					v3	r3	u3	v2	r2	u2	v1	v0	r1	r0	u1	u0
TileY	all	0	5	7					u6	u5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0
Linear	all	0	0	0																

The TileID fills the upper bits of the virtual address (starting with the lowest blank bit in the above table):

```
TileID = [(r » Cr) * (QPitch » Cv) + (v » Cv)] * (Pitch » Cu) + (u » Cu)
```

For the CCS Auxiliary Surface

The CCS is stored differently for the 3D surface type. CCS supports only TileY tile mode, which does not have a three-dimensional offset. Instead, the 3D CCS follows a scheme similar to 2D surfaces. Based on the above parameters and the U, V, and R (three-dimensional pixel address, shifted to adjust for control block size in bytes), the offsets u in bytes and v in rows are given by:

```
u = [U + LODUL]
v = (R * QPitch) + LODVL + V
```

The two-dimensional offset into the surface is defined by the u and v values computed above. The lower virtual address bits are determined by the following table, based on the bits of u and v.

Empty bit positions indicate that the bit is not part of the tile swizzle and is filled in with equations given next.

Tile Mode	Bits per Element	TileID constants			Virtual Address Bits															
		Cr	Cv	Cu	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TileY	5	7							u6	u5	u4	v4	v3	v2	v1	v0	u3	u2	u1	u0

The TileID fills the upper bits of the virtual address (starting with the lowest blank bit in the above table):

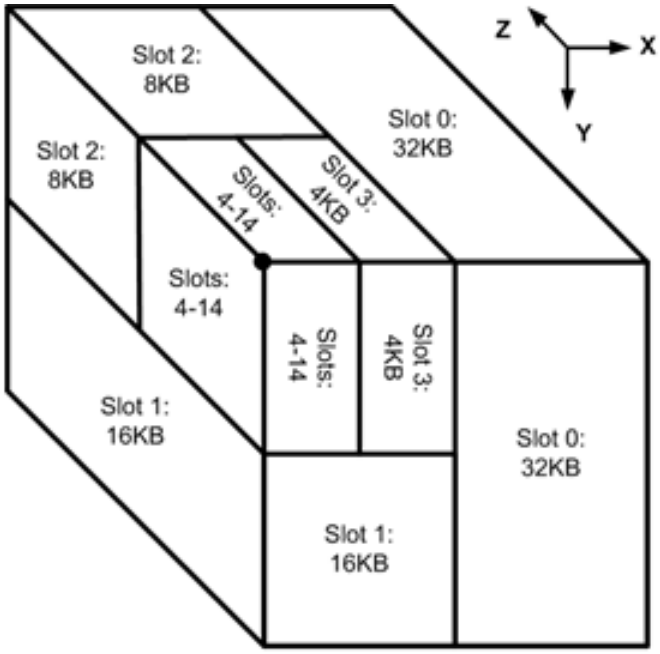
```
TileID = (v » Cv) * (Pitch » Cu) + (u » Cu)
```

Tiling and Mip Tails for 3D Surfaces

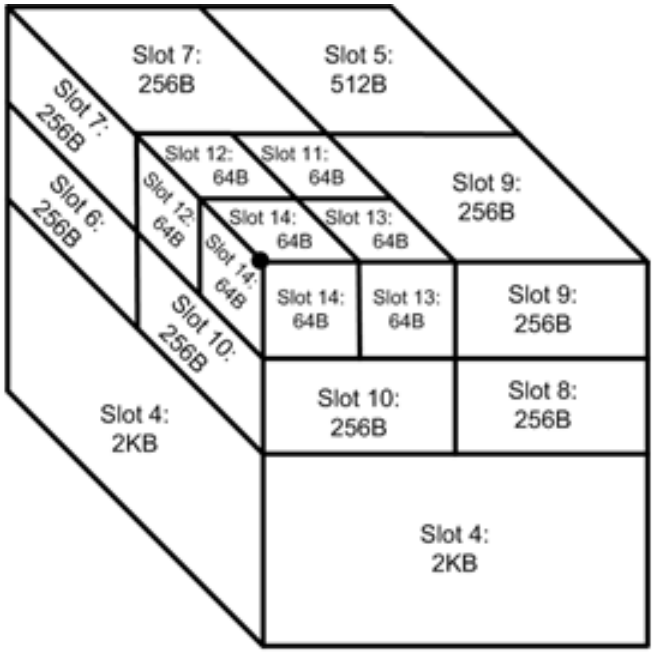
For tiled surfaces where Tiled Resource Mode != TR_NONE, the surface may contain a mip tail. The Mip tail offset is given by the following, where S is the Mip Tail Start LOD:

$$LOD_S = (w_1, h_0+h_2+h_3+ \ldots + h_{S-1}, 0)$$

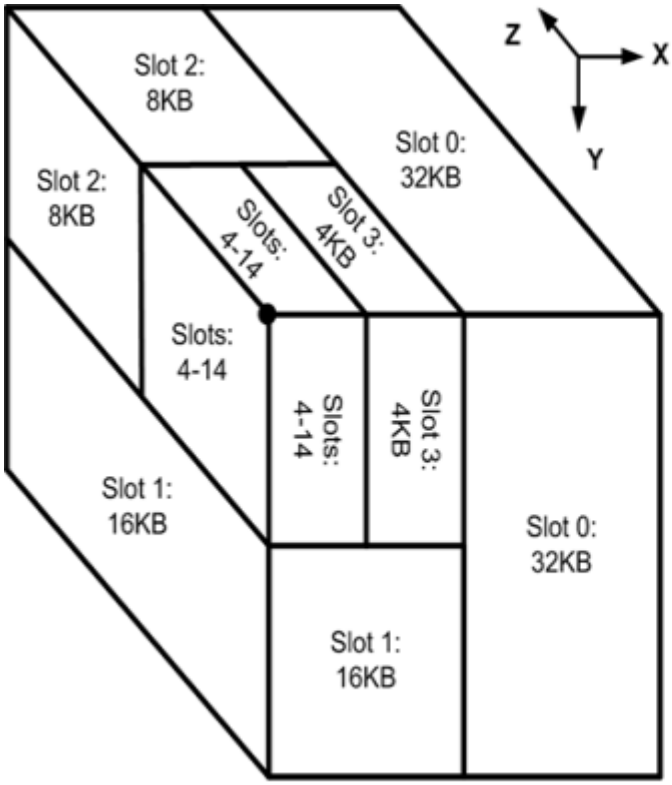
The mip tail exhibits a different arrangement than the rest of the surface. The diagram below shows the 64KB TileYS mip tail and the arrangement of LODs within it, with "slots" indicating the LOD contained within (slot 0 corresponds to LOD s). LODs are aligned to the front upper left corner of the space available. The block marked "Slots 4-15" contains one of the 4KB tile arrangements within, depending on the surface format bits per element. For TileYF, only the 4KB tile exists, with 4 subtracted from each slot number.



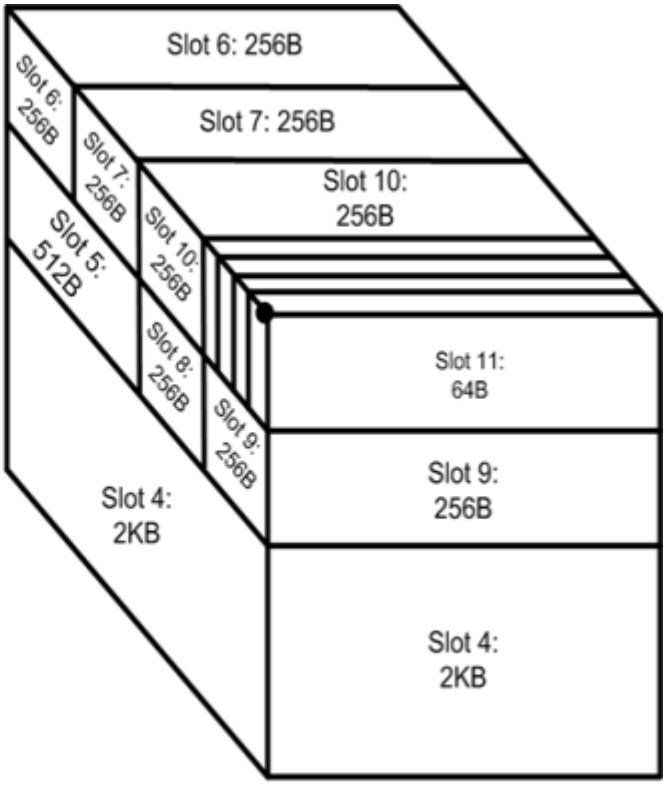
64KB Tile Mip Tail Slots: All Pixel Depths



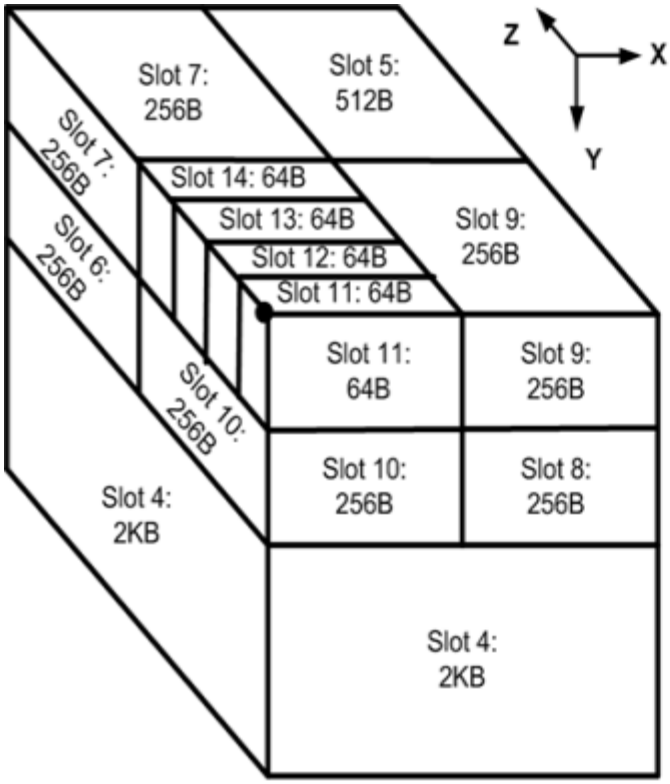
4KB Tile Mip Tail Slots: All Pixel Depths



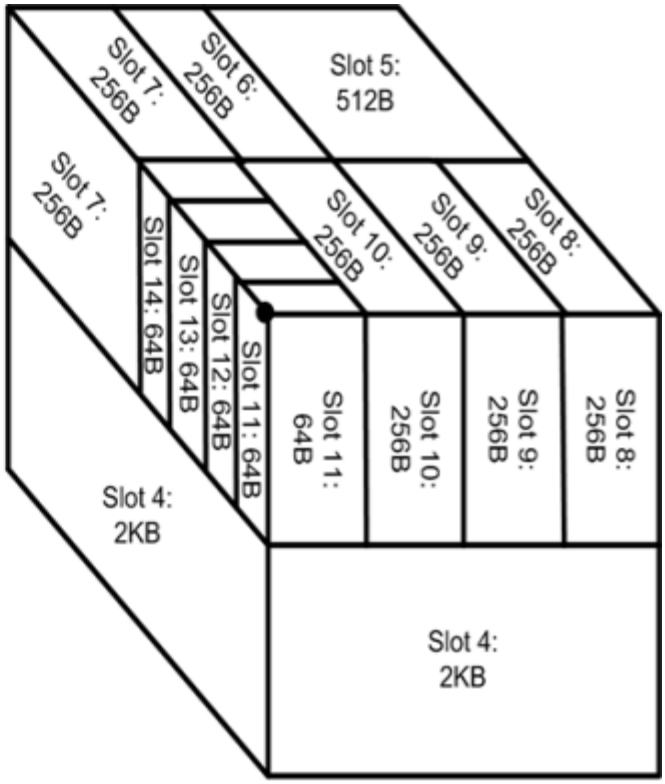
64KB Tile Mip Tail Slots: All Pixel Depths



4KB Tile Mip Tail Slots: 8-bit & 16-bit Pixels



4KB Tile Mip Tail Slots: 32-bit Pixels



4KB Tile Mip Tail Slots: 64-bit & 128-bit Pixels

The offsets into the mip tail tile are given by the following table for each LOD in the mip tail. Note that many of the higher LODs are not possible given surface size constraints, but they are listed here for reference. The offsets given here need to be added to the LODs offset computed earlier to obtain the offset into the surface LOD_L.

TileYS LOD	TileYF LOD	128 bpe	64 bpe	32 bpe	16 bpe	8 bpe
s		(8, 0, 0)	(16, 0, 0)	(16, 0, 0)	(16, 0, 0)	(32, 0, 0)
s+1		(0, 8, 0)	(0, 8, 0)	(0, 16, 0)	(0, 16, 0)	(0, 16, 0)
s+2		(0, 0, 8)	(0, 0, 8)	(0, 0, 8)	(0, 0, 16)	(0, 0, 16)
s+3		(4, 0, 0)	(8, 0, 0)	(8, 0, 0)	(8, 0, 0)	(16, 0, 0)
s+4	s	(0, 4, 0)	(0, 4, 0)	(0, 8, 0)	(0, 8, 0)	(0, 8, 0)
s+5	s+1	(2, 0, 4)	(4, 0, 4)	(4, 0, 4)	(4, 0, 8)	(8, 0, 8)
s+6	s+2	(0, 2, 4)	(0, 2, 4)	(0, 4, 4)	(0, 4, 8)	(0, 4, 8)
s+7	s+3	(0, 0, 4)	(0, 0, 4)	(0, 0, 4)	(0, 0, 8)	(0, 0, 8)
s+8	s+4	(2, 2, 0)	(4, 2, 0)	(4, 4, 0)	(4, 4, 0)	(8, 4, 0)
s+9	s+5	(2, 0, 0)	(4, 0, 0)	(4, 0, 0)	(4, 0, 0)	(8, 0, 0)
s+10	s+6	(0, 2, 0)	(0, 2, 0)	(0, 4, 0)	(0, 4, 0)	(0, 4, 0)
s+11	s+7	(1, 0, 2)	(2, 0, 2)	(2, 0, 2)	(2, 0, 4)	(4, 0, 4)
s+12	s+8	(0, 0, 2)	(0, 0, 2)	(0, 0, 2)	(0, 0, 4)	(0, 0, 4)
s+13	s+9	(1, 0, 0)	(2, 0, 0)	(2, 0, 0)	(2, 0, 0)	(4, 0, 0)
s+14	s+10	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)

Slot Number	Slot Size In Bytes	Byte Offset	128-bit 3D Offset	64-bit 3D Offset	32-bit 3D Offset	16-bit 3D Offset	8-bit 3D Offset
0	32KB	32KB	(8, 0, 0)	(16, 0, 0)	(16, 0, 0)	(16, 0, 0)	(32, 0, 0)
1	16KB	16KB	(0, 8, 0)	(0, 8, 0)	(0, 16, 0)	(0, 16, 0)	(0, 16, 0)
2	8KB	8KB	(0, 0, 8)	(0, 0, 8)	(0, 0, 8)	(0, 0, 16)	(0, 0, 16)
3	4KB	4KB	(4, 0, 0)	(8, 0, 0)	(8, 0, 0)	(8, 0, 0)	(16, 0, 0)
4	2KB	2KB	(0, 4, 0)	(0, 4, 0)	(0, 8, 0)	(0, 8, 0)	(0, 8, 0)
5	512B	1536B	(2, 0, 4)	(4, 0, 4)	(4, 0, 4)	(0, 4, 8)	(0, 4, 8)
6	256B	1280B	(1, 0, 4)	(2, 0, 4)	(0, 4, 4)	(0, 0, 12)	(0, 0, 12)
7	256B	1024B	(0, 0, 4)	(0, 0, 4)	(0, 0, 4)	(0, 0, 8)	(0, 0, 8)
8	256B	768B	(3, 0, 0)	(6, 0, 0)	(4, 4, 0)	(0, 4, 4)	(0, 4, 4)
9	256B	512B	(2, 0, 0)	(4, 0, 0)	(4, 0, 0)	(0, 4, 0)	(0, 4, 0)
10	256B	256B	(1, 0, 0)	(2, 0, 0)	(0, 4, 0)	(0, 0, 4)	(0, 0, 4)
11	64B	0B	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
[12]	64B	64B	(0, 0, 1)	(0, 0, 1)	(0, 0, 1)	(0, 0, 1)	(0, 0, 1)
[13]	64B	128B	(0, 0, 2)	(0, 0, 2)	(0, 0, 2)	(0, 0, 2)	(0, 0, 2)
[14]	64B	192B	(0, 0, 3)	(0, 0, 3)	(0, 0, 3)	(0, 0, 3)	(0, 0, 3)

The following table gives the maximum MIP size which can be included in the

MIP Tail based on bpe. MIP Tail is only supported for Tile64.

Bits per Element	**8**	**16**	**32**	**64**	**128**
Maximum MIP Dimensions (in texels)	32x32x32	16x32x32	16x32x16	16x16x16	8x16x16

3D Alignment Requirements

The vertical and horizontal alignment fields in the RENDER_SURFACE_STATE are ignored for standard tiling formats

(TRMODE != NONE). In the case of standard tiling formats (TileYs and TileYf) the alignment requirements are fixed and are provided for by the tables below for 3D (volumetric) surfaces.

Tile Mode	Bits per Element	Horizontal Alignment	Vertical Alignment	Depth Alignment
TileYS	128	16	16	16
	64	32	16	16
	32	32	32	16
	16	32	32	32
	8	64	32	32
TileYF	128	4	8	8
	64	8	8	8
	32	8	16	8
	16	8	16	16
	8	16	16	16

Surface Padding Requirements

This section covers the requirements for padding around surfaces stored in memory, as there are cases where the device will overfetch beyond the bounds of the surface due to implementation of caches and other hardware structures.

Alignment Unit Size

This section documents the alignment (in texels) that the Surface Pitch and Surface Height must be programmed. For most surface formats it is defined by HAlign and Valign

Alignment Parameters

Surface Defined By	Surface Format	Alignment Unit Width "*i*"	Alignment Unit Height "*j*"
3DSTATE_DEPTH_BUFFER	D16_UNORM	8	4
	not D16_UNORM	4	4
3DSTATE_STENCIL_BUFFER	N/A	8	8
SURFACE_STATE	BC*, ETC*, EAC*	4	4
	FXT1	8	4
	all others	set by **Surface Horizontal Alignment**	set by **Surface Vertical Alignment**

Surface Defined By	Surface Format	Alignment Unit Width "*i*"	Alignment Unit Height "*j*"
SURFACE_STATE	ASTC	Value of ASTC_2DBlockWidth (4, 5, 6, 8, 10, or 12)	ASTC_2DBlockHeight*4

Sampling Engine Surfaces

The sampling engine accesses texels outside of the surface if they are contained in the same cache line as texels that are within the surface. These texels will not participate in any calculation performed by the sampling engine and will not affect the result of any sampling engine operation, however if these texels lie outside of defined pages in the GTT, a GTT error will result when the cache line is accessed. In order to avoid these GTT errors, "padding" at the bottom and right side of a sampling engine surface is sometimes necessary.

It is possible that a cache line will straddle a page boundary if the base address or pitch is not aligned. All pages included in the cache lines that are part of the surface must map to valid GTT entries to avoid errors. To determine the necessary padding on the bottom and right side of the surface, refer to the table in Alignment Unit Size section for the i and j parameters for the surface format in use. The surface must then be extended to the next multiple of the alignment unit size in each dimension, and all texels contained in this extended surface must have valid GTT entries.

For example, suppose the surface size is 15 texels by 10 texels and the alignment parameters are i=4 and j=2. In this case, the extended surface would be 16 by 10. Note that these calculations are done in texels, and must be converted to bytes based on the surface format being used to determine whether additional pages need to be defined.

For compressed textures (BC*, FXT1, ETC*, and EAC* surface formats), padding at the bottom of the surface is to an even compressed row. This is equivalent to a multiple of 2*q*, where *q* is the compression block height in texels. Thus, for padding purposes, these surfaces behave as if $j = 2q$ only for surface padding purposes. The value of *j* is still equal to *q* for mip level alignment and QPitch calculation. For cube surfaces, an additional two rows of padding are required at the bottom of the surface. This must be ensured regardless of whether the surface is stored tiled or linear. This is due to the potential rotation of cache line orientation from memory to cache.

The above comments also apply to the ASTC* surface format.

For packed YUV, 96 bpt, 48 bpt, and 24 bpt surface formats, additional padding is required. These surfaces require an extra row plus 16 bytes of padding at the bottom in addition to the general padding requirements.

For linear surfaces, additional padding of 64 bytes is required at the bottom of the surface. This is in addition to the padding required above.

Programming Note	
Context:	Sampling Engine Surfaces.
For SURFTYPE_BUFFER, SURFTYPE_1D, and SURFTYPE_2D non-array, non-MSAA, non-mip-mapped surfaces in linear memory, the only padding requirement is to the next aligned 64-byte boundary beyond the end of the surface. The rest of the padding requirements documented above do not apply to these surfaces.	

Programming Note	
Context:	Sampling Engine Surfaces
For all surface types other than non-mipmapped non-arrayed 2D, 1D, and Buffer, when using linear mode and surface **Height**%4 != 0, the surface must be padded with 4-(**Height**%4)*__Surface_Pitch__ bytes to avoid fetching outside of allocated memory.	

Render Target and Media Surfaces

The data port accesses data (pixels) outside of the surface if they are contained in the same cache request as pixels that are within the surface. These pixels will not be returned by the requesting message, however if these pixels lie outside of defined pages in the GTT, a GTT error will result when the cache request is processed. In order to avoid these GTT errors, “padding” at the bottom of the surface is sometimes necessary.

Address Tiling Function Introduction

When dealing with memory operands (e.g., graphics surfaces) that are inherently rectangular in nature, certain functions within the graphics device support the storage/access of the operands using alternative (tiled) memory formats to increase performance. This section describes these memory storage formats, why and when they should be used, and the behavioral mechanisms within the device to support them.

Legacy Tiling Modes:

- **TileY**: Used for most tiled surfaces when **TR_MODE**=TR_NONE.
- **TileX** : Used primarily for display surfaces.
- **TileW:** Used for Stencil surfaces.

Tiled Resource Tiling Modes

- **TileYF:** 4KB tiling mode based on TileY
- **TileYS:** 64KB tiling mode based on TileY

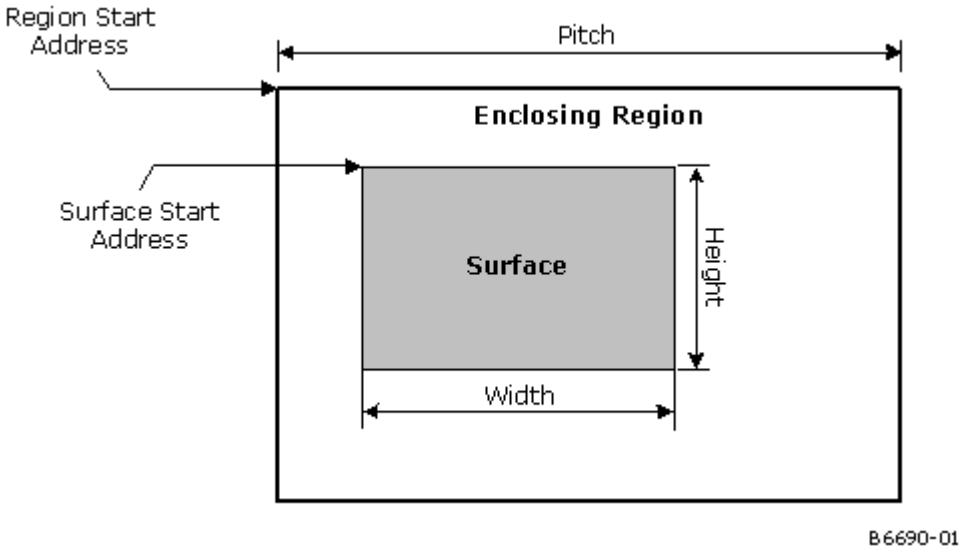
These modes are based on 4KB and 64KB tiles. The 64KB tile is made up of a 4x4 matrix of 4KB tiles. The 4KB tiles in general have a different layout as compared to the legacy modes, with the sub-mode defining the layout within the 4KB tile. The sub-modes are determined by the bits per element of the surface format. The Tiled Resource Mode field in SURFACE_STATE is used to select the new modes.

Tiled surface base addresses must be tile aligned (64KB aligned for TileYS, 4KB aligned for all other tile modes). For 1D surfaces, the base address must be 64KB aligned if **Tiled Resource Mode** is TRMODE_64KB, and 4KB aligned if **Tiled Resource Mode** is TRMODE_4KB. An exception to this tile alignment is when a SURFACE_STATE describes a single MIP within the MIP Tail of another surface, using a 64-bit or 128-bit **Surface Format**—then **Surface Base Address** can refer directly to the given MIP (e.g. to write to a non-renderable **Surface Format** by re-describing as an alternative surface).

Linear vs Tiled Storage

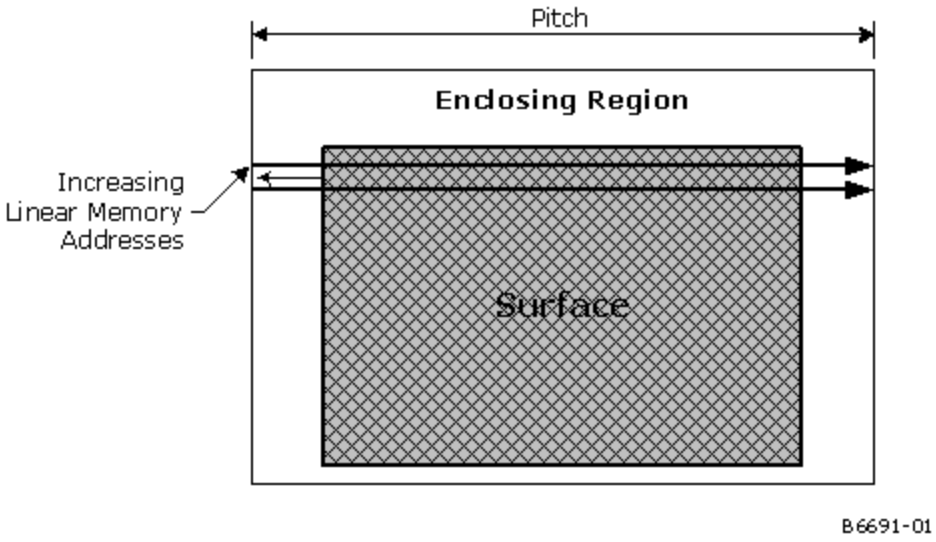
Regardless of the memory storage format, "rectangular" memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region must have widths less than or equal to the region pitch (indeed the enclosing region may coincide exactly with the surface). *Rectangular Memory Operand Parameters* shows these parameters.

Rectangular Memory Operand Parameters



The simplest storage format is the *linear* format (see *Linear Surface Layout*), where each row of the operand is stored in sequentially increasing memory locations. If the surface width is less than the enclosing region's pitch, there will be additional memory storage between rows to accommodate the region's pitch. The pitch of the enclosing region determines the distance (in the memory address space) between vertically-adjacent operand elements (e.g., pixels, texels).

Linear Surface Layout

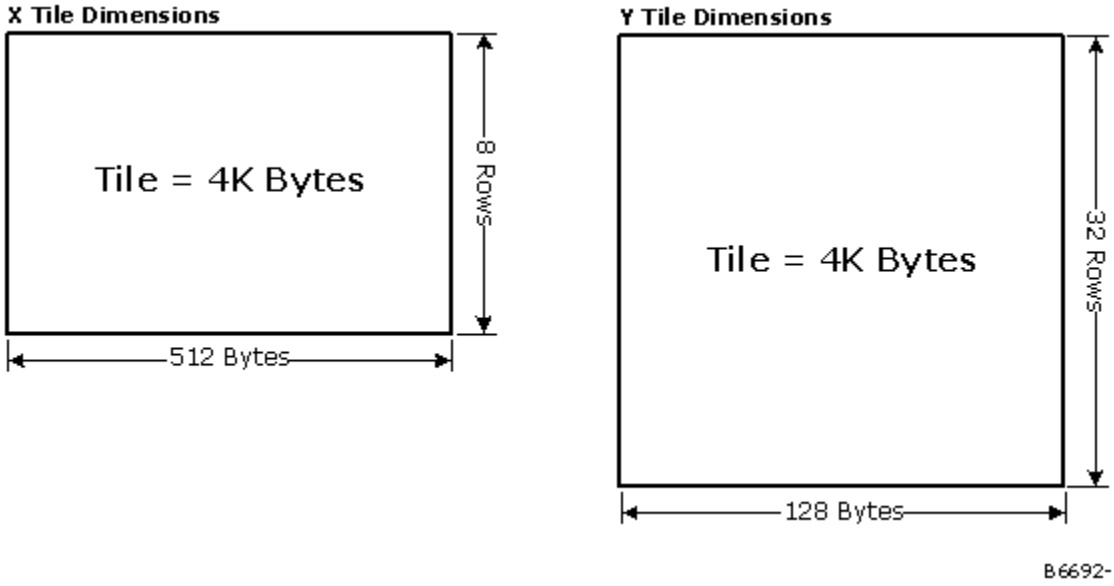


The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters is that horizontally-adjacent elements are stored contiguously. However, when a device function needs to access a 2D subregion within an operand (e.g., a read or write of a 4x4 pixel span by the 3D renderer, a read of a 2x2 texel block for bilinear filtering), having vertically-adjacent elements fall within different memory pages is to be avoided, as the page crossings required to complete the access typically incur increased memory latencies (and therefore lower performance).

One solution to this problem is to divide the enclosing region into an array of smaller rectangular regions, called memory *tiles*. Surface elements falling within a given tile will all be stored in the same physical memory page, thus eliminating page-crossing penalties for 2D subregion accesses within a tile and thereby increasing performance.

Tiles have a fixed 4KB size and are aligned to physical DRAM page boundaries. They are either 8 rows high by 512 bytes wide or 32 rows high by 128 bytes wide (see *Memory Tile Dimensions*). Note that the dimensions of tiles are irrespective of the data contained within – e.g., a tile can hold twice as many 16-bit pixels (256 pixels/row x 8 rows = 2K pixels) than 32-bit pixels (128 pixels/row x 8 rows = 1K pixels).

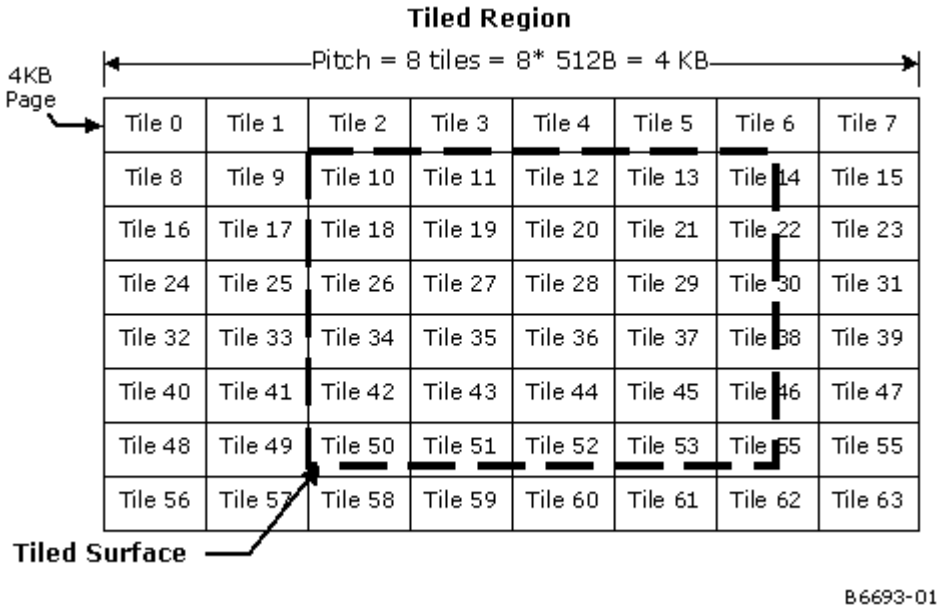
Memory Tile Dimensions



The pitch of a tiled enclosing region must be an integral number of tile widths. The 4KB tiles within a tiled region are stored sequentially in memory in row-major order.

The *Tiled Surface Layout* figure shows an example of a tiled surface located within a tiled region with a pitch of 8 tile widths (512 bytes * 8 = 4KB). Note that it is the *enclosing region* that is divided into tiles – the surface is not necessarily aligned or dimensioned to tile boundaries.

Tiled Surface Layout



Auxiliary Surfaces For Sampled Tiled Resources

For surfaces which are defined as Tiled Resources (TileYs or TileYf format), there may be auxiliary surfaces which are associated with the surface (e.g. HiZ, CCS or MCS). These auxiliary surfaces, while actually not defined as TileYs or TileYf will behave like tiled resources from the hardware perspective. It is possible for software to map and unmap tiles of auxiliary surfaces as tiles of the associated surface are mapped and unmapped. Below is a description how sampling to the mapped/unmapped tile resources is handled for the associated auxiliary surface. Normally, sampling unmapped tiles will return a NULL response to the requesting agen.

HiZ

A tile of HiZ data must be mapped to memory whenever any depth surface (Z) pixels associated with the HiZ tile are mapped. When all Z pixels associated with a HiZ tile are unmapped, the HiZ tile may be mapped or unmapped. Below is a table showing the responses for sampling to mapped and unmapped depth surfaces.

Table of Responses for Sampling to A Depth-Surface Tiled Resource

Depth Surface Mapping	HiZ Surface Mapping	Sample Response
Mapped	Mapped	Normal Sample Response
Mapped	Unmapped	Undefined
Unmapped	Mapped	NULL Response
Unmapped	Unmapped	NULL Response

A "NULL Response" means that the sample returned will be all 0's and the **Null Pixel Mask** (if requested) will indicate the depth pixel is Null.

CCS

A tile of CCS (Color Control Surface) must be mapped to memory whenever color surface pixels associated with the CCS tile are mapped. When all color pixels associated with a CCS tile are unmapped, the CCS may be mapped or unmapped. CCS is used to indicate that the color surface is losslessly compressed. Below is a table showing the responses for sampling to mapped and unmapped.

Table of Responses for Sampling to a Losslessly Compressed Color Surface That is a Tiled Resource

Color Surface Mapping	CCS Surface Mapping	Sample Response
Mapped	Mapped	Normal Response
Mapped	Unmapped	Undefined
Unmapped	Mapped	NULL Response
Unmapped	Unmapped	NULL Response

A "NULL Response" means that the sample returned will be all 0's and the **Null Pixel Mask** (if requested) will indicate the depth pixel is Null.

MCS

A tile of MCS (Multi-Sample Control Surface) must be mapped to memory whenever MSAA surface pixels associated with the CCS tile are mapped. When all MSAA pixels associated with a MCS tile are unmapped, the MCS may be mapped or unmapped. Below is a table showing the responses for sampling to mapped and unmapped.

Table of Responses for Sampling to MSAA Tiled Resources

MSAA Surface Mapping	MCS Mapping	Sample Response
Mapped	Mapped	Normal Response
Mapped	Unmapped	Undefined Response
Unmapped	Mapped	NULL Response
Unmapped	Unmapped	NULL Response

A "NULL Response" means that the sample returned will be all 0's and the **Null Pixel Mask** (if requested) will indicate the depth pixel is Null.

Tile Formats

Multiple tile formats are supported. The following sections define and describe these formats.

Tiling formats are controlled by programming the fields Tile_Mode and Tiled_Resource_Mode in the RENDER_SURFACE_STATE.

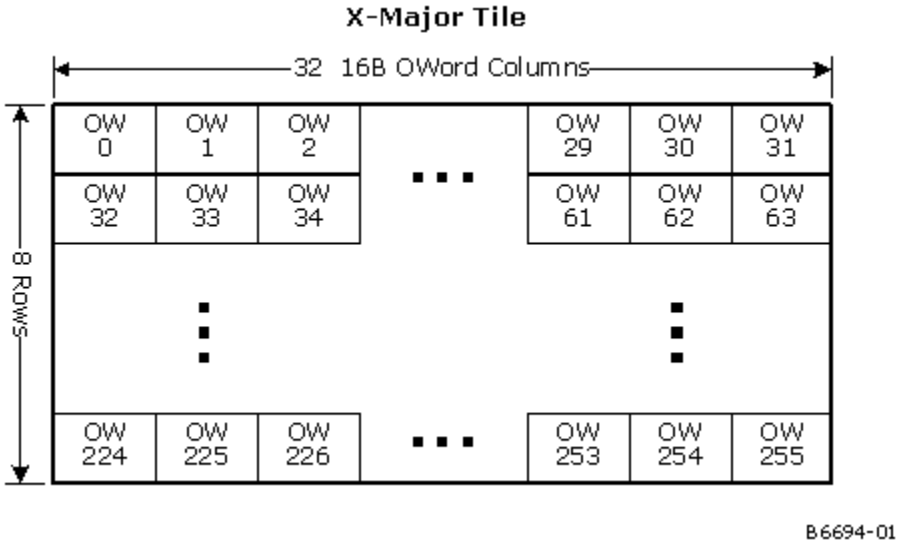
Tile-X Legacy Format

The legacy format Tile-X is a *X-Major* (row-major) storage of tile data units, as shown in the following figure. It is a 4KB tile which is subdivided into an 8-high by 32-wide array of 16-byte OWords . The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. Note that an X-major tiled region with a tile pitch of 1 tile is actually stored in a linear fashion.

Tile-X format is selected for a surface by programming the Tiled_Mode field in RENDER_SURFACE_STATE to XMAJOR.

For 3D sampling operation, a surface using Tile-X layout is generally lower performance the organization of texels in memory.

Tile X-Tile (X-Major) Layout



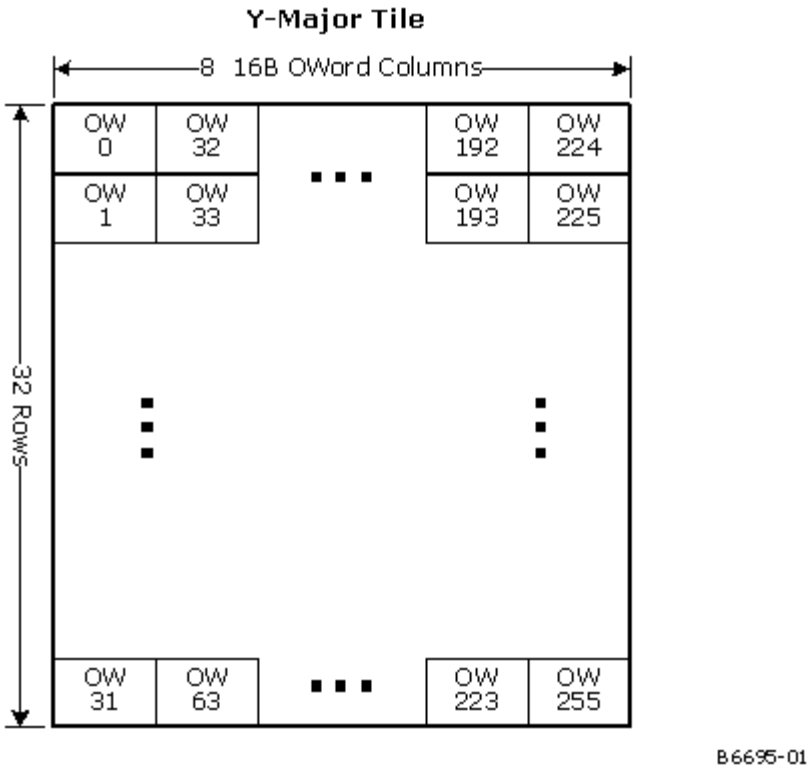
Tile-Y Legacy Format

The device supports Tile-Y legacy format which is *Y-Major* (column major) storage of tile data units, as shown in the following figure. A 4KB tile is subdivided 32-high by 8-wide array of OWords. The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles.

Tile-Y surface format is selected by programming the **Tile Mode** field in RENDER_SURFACE_STATE to YMAJOR.

Note that 3D sampling of a surface in Tile-Y format is usually has higher performance due to the layout of pixels.

Y-Major Tile Layout



W-Major Tile Format

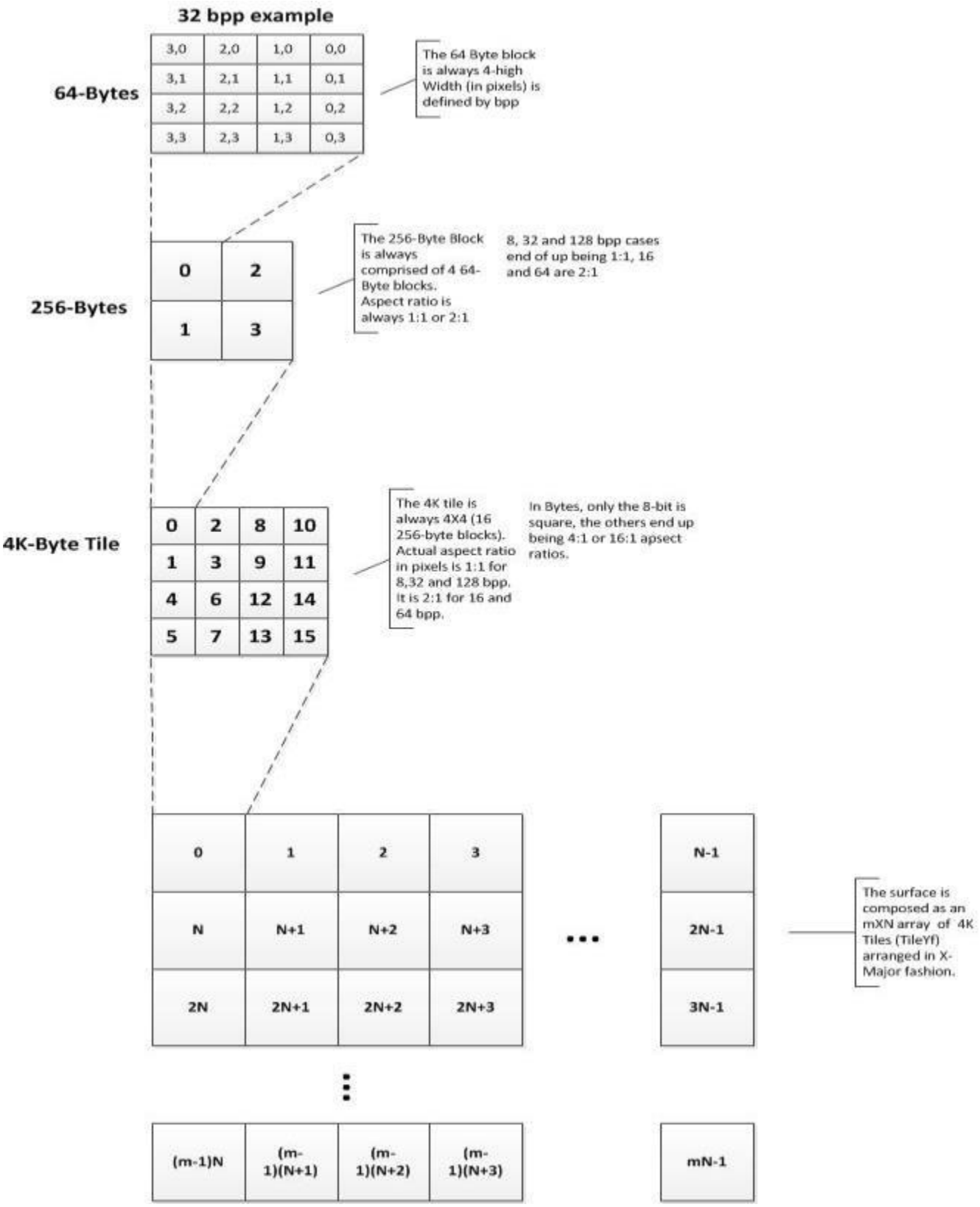
The device supports additional format *W-Major* storage of tile data units, as shown in the following figures. A 4KB tile is subdivided into 8-high by 8-wide array of Blocks for W-Major Tiles (W Tiles). Each Block is 8 rows by 8 bytes. The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. W-Major Tile Format is used for separate stencil.

Tile-W surface format is selected by programming the Tile_Mode field in the RENDER_SURFACE_STATE to WMAJOR.

Tile-Yf Format

Tile-Yf is a 4K-Byte tile format (similar to Tile-Y), but organized in a different manner. Tile-Yf is selected by programming the Tile_Mode field in the RENDER_SURFACE_STATE to YMAJOR and the Tiled_Resource_Mode to TILEYF. The diagram below shows how pixels are mapped into the TileYf format for 2D surfaces, and it uses 32Bpp (bits per pixel) surface format as an example on a 2D surface which is N tiles wide and m tiles high. The exact aspect ratio will be dependent on the Bpp of the surface. Note that the TileYf format is identical to the TileYs up to the 4K-Byte tile size.

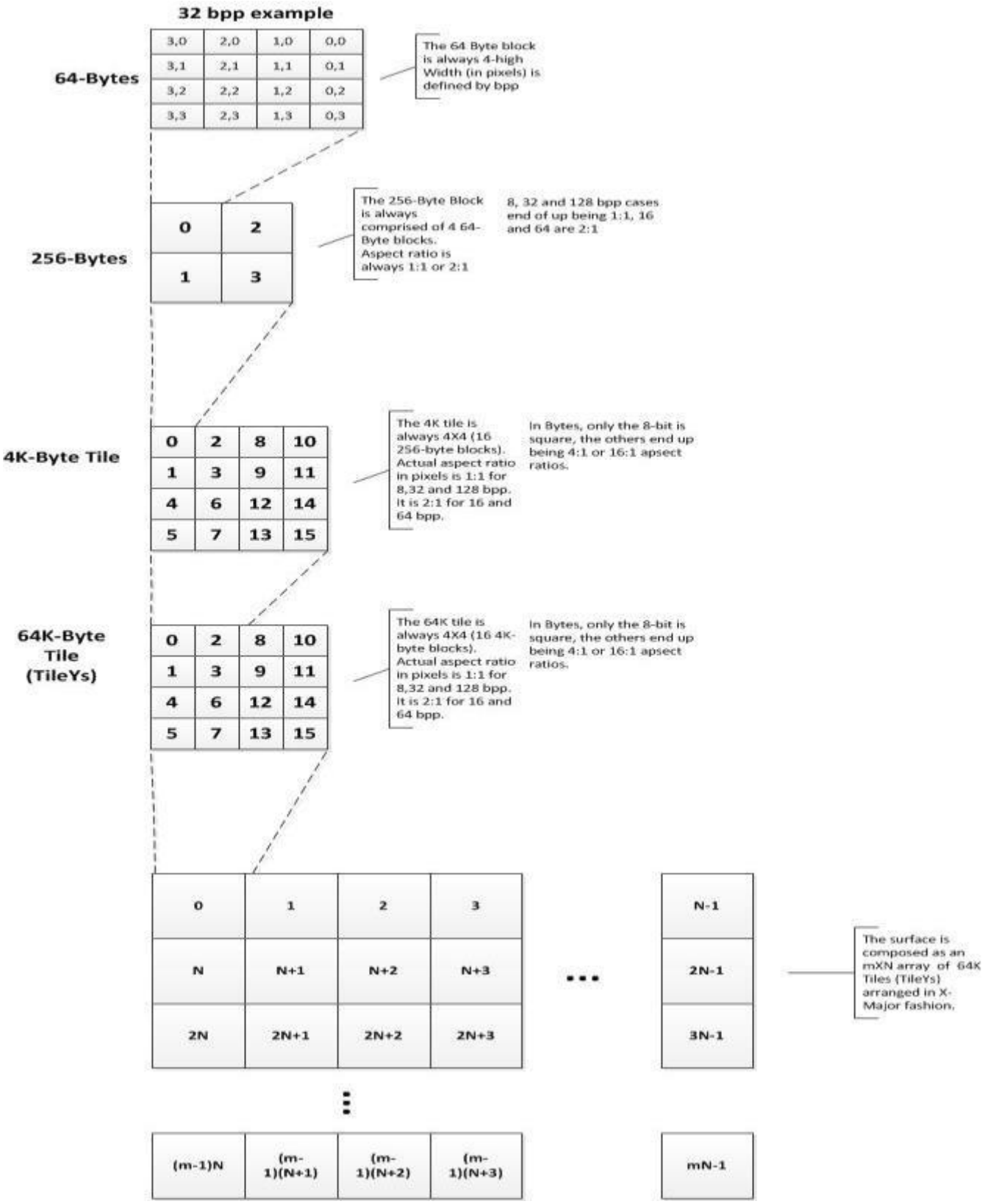
2D Tile Layout for TileYf



Tile-Ys Format

TileYs is a 64K-Byte tile size. It is enabled by programming the Tile_Mode field (in RENDER_SURFACE_STATE) to YMAJOR, and programming the Tiled_Resource_Mode to TILEYS. It is organized as shown below, and is composed of 4KByte blocks which have identical layout to the TileYf format. The diagram below shows how pixels are mapped into the TileYs format, and it uses 32Bpp (bits per pixel) surface format as an example on a 2D surface which is N tiles wide and m tiles high. The exact aspect ratio will be dependent on the Bpp of the surface.

Tile-Ys Layout



Tiling Algorithm

The following pseudo-code describes the algorithm for translating a tiled memory surface in graphics memory to an address in logical space.

The following new modes are supported for Tiled Resources (**TR_MODE** != TR_NONE) defined to enable tiled resources.

For more details about Mip Tails, see **Surface Layout and Tiling** in the Common Surface Formats section.

- **TileYF:** 4KB tiling mode based on TileY
- **TileYS:** 64KB tiling mode based on TileY

```
Inputs:
      LinearAddress(offset into regular or LT aperture in terms of bytes),
    Pitch(in terms of tiles),
            WalkY (1 for Y and 0 for X)
            WalkW (1 for W and 0 for the rest)

Static Parameters:
      TileH (Height of tile, 8 for X, 32 for Y and 64 for W),
      TileW (Width of Tile in bytes, 512 for X, 128 for Y and 64 for W)
      TileSize = TileH * TileW;
      RowSize = Pitch * TileSize;

If (Fenced) {
     LinearAddress = LinearAddress – FenceBaseAddress
     LinearAddrInTileW = LinearAddress div TileW;
     Xoffset_inTile = LinearAddress mod TileW;
       Y = LinearAddrInTileW div Pitch;
       X = LinearAddrInTileW mod Pitch + Xoffset_inTile;
}

// Internal graphics clients that access tiled memory already have the X, Y
// coordinates and can start here
YOff_Within_Tile = Y mod TileH;
XOff_Within_Tile = X mod TileW;
TileNumber_InY = Y div TileH;
TileNumber_InX = X div TileW;

      TiledOffsetY = RowSize * TileNumber_InY + TileSize * TileNumber_InX + TileH * 16 *
(XOff_Within_Tile div 16) +
             YOff_Within_Tile * 16 +
             (XOff_Within_Tile mod 16);

      TiledOffsetW = RowSize * TileNumber_InY +
             TileSize * TileNumber_InX +
             TileH * 8 * (XOff_Within_Tile div 8) +
             64 * (YOff_Within_Tile div 8) +
             32 * ((YOff_Within_Tile div 4) mod 2) +
             16 * ((XOff_Within_Tile div 4) mod 2) +
              8 * ((YOff_Within_Tile div 2) mod 2) +
              4 * ((XOff_Within_Tile div 2) mod 2) +
              2 * (YOff_Within_Tile mod 2) +
             (XOff_Within_Tile mod 2);

     TiledOffsetX = RowSize * TileNumber_InY + TileSize * TileNumber_InX + TileW *
YOff_Within_Tile + XOff_Within_Tile;

     TiledOffset = WalkW? TiledOffsetW : (WalkY? TiledOffsetY : TiledOffsetX);
```

```
        TiledAddress = Tiled? (BaseAddress + TiledOffset): (BaseAddress + Y*LinearPitch +
X);TiledAddress = (Tiled &&
              (Address Swizzling for Tiled-Surfaces == 01)) ?
              (WalkW || WalkY) ?
              (TiledAddress div 128) * 128 +
              (((TiledAddress div 64) mod 2) ^
              ((TiledAddress div 512) mod 2)) +
              (TiledAddress mod 32)
              :
              (TiledAddress div 128) * 128 +
              (((TiledAddress div 64) mod 2) ^
              ((TiledAddress div 512) mod 2)
              ((TiledAddress Div 1024) mod2) +
              (TiledAddress mod 32)
              :
             TiledAddress;
       }
```

Address Swizzling for Tiled-Surfaces is no longer used because the main memory controller has a more effective address swizzling algorithm.

For Address Swizzling for Tiled-Surfaces see ARB_MODE – Arbiter Mode Control register, ARB_CTL—Display Arbitration Control 1 and TILECTL - Tile Control register

The Y-Major tile formats have the characteristic that a surface element in an even row is located in the same aligned 64-byte cacheline as the surface element immediately below it (in the odd row). This spatial locality can be exploited to increase performance when reading 2x2 texel squares for bilinear texture filtering, or reading and writing aligned 4x4 pixel spans from the 3D Render pipeline.

On the other hand, the X-Major tile format has the characteristic that horizontally-adjacent elements are stored in sequential memory addresses. This spatial locality is advantageous when the surface is scanned in row-major order for operations like display refresh. For this reason, the Display and Overlay memory streams only support linear or X-Major tiled surfaces (Y-Major tiling is not supported by these functions). This has the side effect that 2D- or 3D-rendered surfaces must be stored in linear or X-Major tiled formats if they are to be displayed. Non-displayed surfaces, e.g., “rendered textures”, can also be stored in Y-Major order.

The following Psuedo Code Describes the algorithm for mapping TileYs and TileYf Tile Address to Byte Offset within a Tile. It describes the support for 2D for both TileYs and TileYf as well as MSAA 2D For TileYs.

```
/*****************************************************************************\
   BitMask
      Used for masking single bits of x, y, z, ss# when _pdep32 instruction is
    not available
\*****************************************************************************/
enum BitMask
{
    BIT0 = 1,
    BIT1 = (1 « 1),
    BIT2 = (1 « 2),
    BIT3 = (1 « 3),
    BIT4 = (1 « 4),
    BIT5 = (1 « 5),
    BIT6 = (1 « 6),
    BIT7 = (1 « 7),
    BIT8 = (1 « 8),
    BIT9 = (1 « 9),
```

```
    BIT10 = (1 « 10),
    BIT11 = (1 « 11),
    BIT12 = (1 « 12),
    BIT13 = (1 « 13),
    BIT14 = (1 « 14),
    BIT15 = (1 « 15)
};
/*****************************************************************************\
    TileYS/TileYF constant swizzle masks w/o _pdep32 instruction

    Used to mask contiguous x/y/z/sample bit groupings before being shifted into
    their final swizzled bit positions
\*****************************************************************************/
// used for fallback 'manual' bit shifting
static const UINT16 xMaskBits5_4  = 0x0030;
static const UINT16 xMaskBits3_0  = 0x000F;
static const UINT16 yMaskBits4_0  = 0x001F;
static const UINT16 yMaskBits3_0  = 0x000F;
static const UINT16 yMaskBits2_0  = 0x0007;
static const UINT16 yMaskBits1_0  = 0x0003;
static const UINT16 SampleMask3_0 = 0x000F;
static const UINT16 SampleMask2_0 = 0x0007;
static const UINT16 SampleMask1_0 = 0x0003;
static const UINT16 SampleMask0   = 0x0001;

/*****************************************************************************\
    TileYS 2D Tile address swizzling functions w/o _pdep32
\*****************************************************************************/
/*
 ____________________________________________________________________________
|    Num    | Bits per element |           Tiled element offset bits           |
|  Samples  |                  |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|___________|__________________|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|
|     1x    |     64 & 128     |x9|y5|x8|y4|x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|           |     16 & 32      |x8|y6|x7|y5|x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|           |        8         |x7|y7|x6|y6|x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYS2dElementOffset64_128bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;

    // shift bits in x and y to their respective TileYS swizzled bit locations
    xSwizzle = ((BIT9 & x) « 6) |
               ((BIT8 & x) « 5) |
               ((BIT7 & x) « 4) |
               ((BIT6 & x) « 3) |
               ((xMaskBits5_4 & x) « 2) | // shift to bit positions 7..6
               (xMaskBits3_0 & x);

    ySwizzle = ((BIT5 & y) « 9) |
               ((BIT4 & y) « 8) |
               ((BIT3 & y) « 7) |
               ((BIT2 & y) « 6) |
               ((yMaskBits1_0 & y) « 4);   // shift to bit positions 5..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}

UINT16 TileYS2dElementOffset16_32bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
```

```
    // shift bits in x and y to their respective TileYS swizzled bit locations
    xSwizzle = ((BIT8 & x) « 7) |
               ((BIT7 & x) « 6) |
               ((BIT6 & x) « 5) |
               ((BIT5 & x) « 4) |
               ((BIT4 & x) « 3) |
               (xMaskBits3_0 & x);

    ySwizzle = ((BIT6 & y) « 8) |
               ((BIT5 & y) « 7) |
               ((BIT4 & y) « 6) |
               ((BIT3 & y) « 5) |
               ((yMaskBits2_0 & y) « 4); // shift to bit positions 6..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}

UINT16 TileYS2dElementOffset8bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;

    // shift bits in x and y to their respective TileYS swizzled bit locations
    xSwizzle = ((BIT7 & x) « 8) |
               ((BIT6 & x) « 7) |
               ((BIT5 & x) « 6) |
               ((BIT4 & x) « 5) |
               (xMaskBits3_0 & x);

    ySwizzle = ((BIT7 & y) « 7) |
               ((BIT6 & y) « 6) |
               ((BIT5 & y) « 5) |
               ((yMaskBits4_0 & y) « 4); // shift to bit positions 8..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}

/*******************************************************************************\
    TileYS 2D MSAA Tile address swizzling functions w/o _pdep32
\*******************************************************************************/
/*
 ____________________________________________________________________________
|    Num    | Bits per element |           Tiled element offset bits           |
|  Samples  |                  |15 |14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|___________|__________________|___|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|
|     2x    |     64 & 128     |ss0|y5|x8|y4|x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|           |     16 & 32      |ss0|y6|x7|y5|x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|           |        8         |ss0|y7|x6|y6|x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYS2xMsaaElementOffset64_128bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT8 & x) « 5) |                  // shift to bit position 13
               ((BIT7 & x) « 4) |                  // shift to bit position 11
               ((BIT6 & x) « 3) |                  // shift to bit position 9
               ((xMaskBits5_4 & x) « 2) |          // shift to bit positions 7..6
               (xMaskBits3_0 & x);                  // leave in bits 3..0
```

```
    ySwizzle = ((BIT5 & y) « 9) |                  // shift to bit position 14
               ((BIT4 & y) « 8) |                  // shift to bit position 12
               ((BIT3 & y) « 7) |                  // shift to bit position 10
               ((BIT2 & y) « 6) |                  // shift to bit position 8
               ((yMaskBits1_0 & y) « 4);           // shift to bit positions 5..4

    SampleSwizzle = (sample && SampleMask0) « 15;  // shift to bit position 15

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS2xMsaaElementOffset16_32bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT7 & x) « 6) |                  // shift to bit position 13
               ((BIT6 & x) « 7) |                  // shift to bit position 11
               ((BIT5 & x) « 6) |                  // shift to bit position 9
               ((BIT4 & x) « 5) |                  // shift to bit position 7
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT6 & y) « 8) |                  // shift to bit position 14
               ((BIT5 & y) « 7) |                  // shift to bit position 12
               ((BIT4 & y) « 6) |                  // shift to bit position 10
               ((BIT3 & y) « 5) |                  // shift to bit position 8
               ((yMaskBits2_0 & y) « 4);           // shift to bit positions 6..4

    SampleSwizzle = (sample && SampleMask0) « 15;  // shift to bit position 15

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS2xMsaaElementOffset8bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT6 & x) « 7) |                  // shift to bit position 13
               ((BIT5 & x) « 6) |                  // shift to bit position 11
               ((BIT4 & x) « 5) |                  // shift to bit position 9
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT7 & y) « 7) |                  // shift to bit position 14
               ((BIT6 & y) « 6) |                  // shift to bit position 12
               ((BIT5 & y) « 5) |                  // shift to bit position 10
               ((yMaskBits4_0 & y) « 4);           // shift to bit positions 8..4

    SampleSwizzle = (sample && SampleMask0) « 15;  // shift to bit position 15

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

/*
 ____________________________________________________________________________
|    Num    | Bits per element |          Tiled element offset bits           |
```

```
|  Samples  |                  |15 |14 |13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|___________|__________________|___|___|__|__|__|__|__|__|__|__|__|__|__|__|__|__|
|     4x    |     64 & 128     |ss1|ss0|x8|y4|x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|           |     16 & 32      |ss1|ss0|x7|y5|x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|           |        8         |ss1|ss0|x6|y6|x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYS4xMsaaElementOffset64_128bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT8 & x) « 5) |                  // shift to bit position 13
               ((BIT7 & x) « 4) |                  // shift to bit position 11
               ((BIT6 & x) « 3) |                  // shift to bit position 9
               ((xMaskBits5_4 & x) « 2) |          // shift to bit positions 7..6
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT4 & y) « 8) |                  // shift to bit position 12
               ((BIT3 & y) « 7) |                  // shift to bit position 10
               ((BIT2 & y) « 6) |                  // shift to bit position 8
               ((yMaskBits1_0 & y) « 4);           // shift to bit positions 5..4

    SampleSwizzle = (sample && SampleMask1_0) « 14;// shift to bit positions 15..14

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS4xMsaaElementOffset16_32bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT7 & x) « 6) |                  // shift to bit position 13
               ((BIT6 & x) « 7) |                  // shift to bit position 11
               ((BIT5 & x) « 6) |                  // shift to bit position 9
               ((BIT4 & x) « 5) |                  // shift to bit position 7
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT5 & y) « 7) |                  // shift to bit position 12
               ((BIT4 & y) « 6) |                  // shift to bit position 10
               ((BIT3 & y) « 5) |                  // shift to bit position 8
               ((yMaskBits2_0 & y) « 4);           // shift to bit positions 6..4

    SampleSwizzle = (sample && SampleMask1_0) « 14;// shift to bit positions 15..14

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS4xMsaaElementOffset8bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT6 & x) « 7) |                  // shift to bit position 13
               ((BIT5 & x) « 6) |                  // shift to bit position 11
               ((BIT4 & x) « 5) |                  // shift to bit position 9
```

```
                (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT6 & y) « 6) |                   // shift to bit position 12
               ((BIT5 & y) « 5) |                   // shift to bit position 10
               ((yMaskBits4_0 & y) « 4);            // shift to bit positions 8..4

    SampleSwizzle = (sample && SampleMask1_0) « 14;// shift to bit positions 15..14

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

/*
 ____________________________________________________________________________________
|    Num     | Bits per element |           Tiled element offset bits                |
|  Samples   |                  |15 |14 |13 |12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|____________________________________|___|___|___|__|__|__|__|__|__|__|__|__|__|__|__|
|     8x     |     64 & 128     |ss2|ss1|ss0|y4|x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|            |     16 & 32      |ss2|ss1|ss0|y5|x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|            |        8         |ss2|ss1|ss0|y6|x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYS8xMsaaElementOffset64_128bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT7 & x) « 4) |                   // shift to bit position 11
               ((BIT6 & x) « 3) |                   // shift to bit position 9
               ((xMaskBits5_4 & x) « 2) |           // shift to bit positions 7..6
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT4 & y) « 8) |                   // shift to bit position 12
               ((BIT3 & y) « 7) |                   // shift to bit position 10
               ((BIT2 & y) « 6) |                   // shift to bit position 8
               ((yMaskBits1_0 & y) « 4);            // shift to bit positions 5..4

    SampleSwizzle = (sample && SampleMask2_0) « 13;// shift to bit positions 15..13

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS8xMsaaElementOffset16_32bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT6 & x) « 7) |                   // shift to bit position 11
               ((BIT5 & x) « 6) |                   // shift to bit position 9
               ((BIT4 & x) « 5) |                   // shift to bit position 7
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT5 & y) « 7) |                   // shift to bit position 12
               ((BIT4 & y) « 6) |                   // shift to bit position 10
               ((BIT3 & y) « 5) |                   // shift to bit position 8
               ((yMaskBits2_0 & y) « 4);            // shift to bit positions 6..4

    SampleSwizzle = (sample && SampleMask2_0) « 13;// shift to bit positions 15..13

    // OR the swizzled bit positions for final offset within a tile
```

```
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS8xMsaaElementOffset8bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT5 & x) « 6) |                  // shift to bit position 11
               ((BIT4 & x) « 5) |                  // shift to bit position 9
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT6 & y) « 6) |                  // shift to bit position 12
               ((BIT5 & y) « 5) |                  // shift to bit position 10
               ((yMaskBits4_0 & y) « 4);           // shift to bit positions 8..4

    SampleSwizzle = (sample && SampleMask2_0) « 13;// shift to bit positions 15..13

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

/*
 ____________________________________________________________________________________
|    Num    | Bits per element |           Tiled element offset bits                |
|  Samples  |                  |15 |14 |13 |12 |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|___________|__________________|___|___|___|___|__|__|__|__|__|__|__|__|__|__|__|__|
|    16x    |     64 & 128     |ss3|ss2|ss1|ss0|x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|           |     16 & 32      |ss3|ss2|ss1|ss0|x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|           |        8         |ss3|ss2|ss1|ss0|x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYS16xMsaaElementOffset64_128bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT7 & x) « 4) |                  // shift to bit position 11
               ((BIT6 & x) « 3) |                  // shift to bit position 9
               ((xMaskBits5_4 & x) « 2) |          // shift to bit positions 7..6
               (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT3 & y) « 7) |                  // shift to bit position 10
               ((BIT2 & y) « 6) |                  // shift to bit position 8
               ((yMaskBits1_0 & y) « 4);           // shift to bit positions 5..4

    SampleSwizzle = (sample && SampleMask3_0) « 12;// shift to bit positions 15..12

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS16xMsaaElementOffset16_32bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT6 & x) « 7) |                  // shift to bit position 11
               ((BIT5 & x) « 6) |                  // shift to bit position 9
```

```
                ((BIT4 & x) « 5) |                  // shift to bit position 7
                (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT4 & y) « 6) |                   // shift to bit position 10
                ((BIT3 & y) « 5) |                  // shift to bit position 8
                ((yMaskBits2_0 & y) « 4);           // shift to bit positions 6..4

    SampleSwizzle = (sample && SampleMask3_0) « 12;// shift to bit positions 15..12

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}

UINT16 TileYS16xMsaaElementOffset8bpe(UINT16 x, UINT16 y, UINT16 sample)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;
    UINT16 SampleSwizzle;

    // shift bits in x, y, and sample to their respective TileYS MSAA swizzled bit locations
    xSwizzle = ((BIT5 & x) « 6) |                   // shift to bit position 11
                ((BIT4 & x) « 5) |                  // shift to bit position 9
                (xMaskBits3_0 & x);                  // leave in bits 3..0

    ySwizzle = ((BIT5 & y) « 5) |                   // shift to bit position 10
                ((yMaskBits4_0 & y) « 4);           // shift to bit positions 8..4

    SampleSwizzle = (sample && SampleMask3_0) « 12;// shift to bit positions 15..12

    // OR the swizzled bit positions for final offset within a tile
    return SampleSwizzle | xSwizzle | ySwizzle;
}


/*****************************************************************************\
    TileYF 2D Tile address swizzling functions w/o _pdep32
\*****************************************************************************/
/*
 ___________________________________________________________________________
|     Num     | Bits per element |           Tiled element offset bits          |
|  Samples    |                  |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
|_____________|__________________|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|
|     1x      |     64 & 128     |  |  |  |  |x7|y3|x6|y2|x5|x4|y1|y0|x3|x2|x1|x0|
|             |     16 & 32      |  |  |  |  |x6|y4|x5|y3|x4|y2|y1|y0|x3|x2|x1|x0|
|             |        8         |  |  |  |  |x5|y5|x4|y4|y3|y2|y1|y0|x3|x2|x1|x0|
*/
UINT16 TileYF2dElementOffset64_128bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;

    // shift bits in x and y to their respective TileYF swizzled bit locations
    xSwizzle = ((BIT7 & x) « 4) |
                ((BIT6 & x) « 3) |
                ((xMaskBits5_4 & x) « 2) | // shift to bit positions 7..6
                (xMaskBits3_0 & x);

    ySwizzle = ((BIT3 & y) « 7) |
                ((BIT2 & y) « 6) |
                ((yMaskBits1_0 & y) « 4);   // shift to bit positions 5..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}
```

```
UINT16 TileYF2dElementOffset16_32bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;

    // shift bits in x and y to their respective TileYF swizzled bit locations
    xSwizzle = ((BIT6 & x) « 5) |
               ((BIT5 & x) « 4) |
               ((BIT4 & x) « 3) |
               (xMaskBits3_0 & x);

    ySwizzle = ((BIT4 & y) « 6) |
               ((BIT3 & y) « 5) |
               ((yMaskBits2_0 & y) « 4); // shift to bit positions 6..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}

UINT16 TileYF2dElementOffset8bpe(UINT16 x, UINT16 y)
{
    UINT16 xSwizzle;
    UINT16 ySwizzle;

    // shift bits in x and y to their respective TileYF swizzled bit locations
    xSwizzle = ((BIT5 & x) « 6) |
               ((BIT4 & x) « 5) |
               (xMaskBits3_0 & x);

    ySwizzle = ((BIT5 & y) « 5) |
               ((yMaskBits4_0 & y) « 4); // shift to bit positions 8..4

    // OR the swizzled bit positions for final offset within a tile
    return xSwizzle | ySwizzle;
}
```

Tiled Channel Select Decision

Previously, there was a historical configuration control field to swizzle address bit[6] for in X/Y tiling modes. This was set in three different places: TILECTL[1:0], ARB_MODE[5:4], and DISP_ARB_CTL[14:13].

The swizzle fields are all reserved, and the CPU's memory controller performs all address swizzling modifications.

Tiling Support

The rearrangement of the surface elements in memory must be accounted for in device functions operating upon tiled surfaces. (Note that not all device functions that access memory support tiled formats). This requires either the modification of an element's linear memory address or an alternate formula to convert an element's X,Y coordinates into a tiled memory address.

However, before tiled-address generation can take place, some mechanism must be used to determine whether the surface elements accessed fall in a linear or tiled region of memory, and if tiled, what the tile region pitch is, and whether the tiled region uses X-Major or Y-Major format. There are two mechanisms by which this detection takes place: (a) an implicit method by detecting that the pre-tiled (linear) address falls within a "fenced" tiled region, or (b) by an explicit specification of tiling parameters for surface operands (i.e., parameters included in surface-defining instructions).

The following table identifies the tiling-detection mechanisms that are supported by the various memory streams.

Access Path	Tiling-Detection Mechanisms Supported
Processor access through the Graphics Memory Aperture	Fenced Regions
3D Render (Color/Depth Buffer access)	Explicit Surface Parameters
Sampled Surfaces	Explicit Surface Parameters
Blt operands	Explicit Surface Parameters
Display and Overlay Surfaces	Explicit Surface Parameters

Tiled (Fenced) Regions

The only mechanism to support the access of surfaces in tiled format by the host or external graphics client is to place them within “fenced” tiled regions within Graphics Memory. A fenced region is a block of Graphics Memory specified using one of the sixteen FENCE device registers. (See *Memory Interface Registers* for details). Surfaces contained within a fenced region are considered tiled from an external access point of view. Note that fences cannot be used to untile surfaces in the PGM_Address space since external devices cannot access PGM_Address space. Even if these surfaces (or any surfaces accessed by an internal graphics client) fall within a region covered by an enabled fence register, that enable will be effectively masked during the internal graphics client access. Only the explicit surface parameters described in the next section can be used to tile surfaces being accessed by the internal graphics clients.

Tiled Surface Parameters

Internal device functions require explicit specification of surface tiling parameters via information passed in commands and state. This capability is provided to limit the reliance on the fixed number of fence regions.

The following table lists the surface tiling parameters that can be specified for 3D Render surfaces (Color Buffer, Depth Buffer, Textures, etc.) via SURFACE_STATE.

Surface Parameter	Description
Tiled Surface	If ENABLED, the surface is stored in a tiled format. If DISABLED, the surface is stored in a linear format.
Tile Walk	If Tiled Surface is ENABLED, this parameter specifies whether the tiled surface is stored in Y-Major or X-Major tile format.
Base Address	Additional restrictions apply to the base address of a Tiled Surface vs. that of a linear surface.
Pitch	Pitch of the surface. Note that, if the surface is tiled, this pitch must be a multiple of the tile width.

Tiled Surface Restrictions

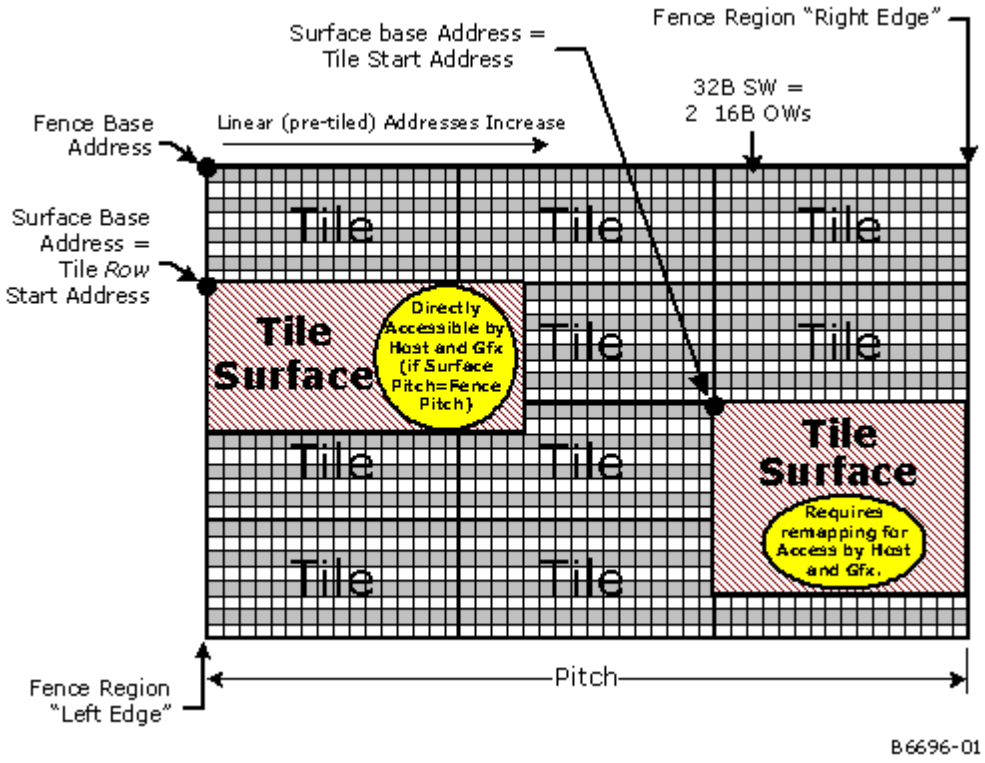
Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by internal device functions. An example of such a surface is a tiled texture that is initialized by the CPU and then sampled by the device.

The tiling algorithm for internal device functions is different from that of fence regions. Internal device functions always specify tiling in terms of a surface. The surface must have a base address, and this base address is not subject to the tiling algorithm. Only *offsets* from the base address (as calculated by X, Y addressing within the surface) are transformed through tiling. The base address of the surface must therefore be 4KB-aligned. This forces the 4KB tiles of the tiling algorithm to exactly align with 4KB device pages once the tiling algorithm has been applied to the offset. The width of a surface must be less than or equal to the surface pitch. There are additional considerations for surfaces that are also accessed by the host (via a fence region).

Fence regions have no base address per se. Host linear addresses that fall in a fence region are translated in their entirety by the tiling algorithm. It is as if the surface being tiled by the fence region has a base address in graphics memory equal to the fence base address, and all accesses of the surfaces are (possibly quite large) offsets from the fence base address. Fence regions have a virtual “left edge” aligned with the fence base address, and a “right edge” that results from adding the fence pitch to the “left edge”. Surfaces in the fence region must not straddle these boundaries.

Base addresses of surfaces that are to be accessed both by an internal graphics client and by the host have the tightest restrictions. In order for the surface to be accessed without GTT re-mapping, the surface base address (as set in SURFACE_STATE) must be a "Tile Row Start Address" (TRSA). The first address in each tile row of the fence region is a Tile Row Start Address. The first TRSA is the fence base address. Each TRSA can be generated by adding an integral multiple of the row size to the fence base address. The row size is simply the fence pitch in tiles multiplied by 4KB (the size of a tile.)

Tiled Surface Placement



The pitch in SURFACE_STATE must be set equal to the pitch of the fence that will be used by the host to access the surface if the same GTT mapping will be used for each access. If the pitches differ, a different GTT mapping must be used to eliminate the "extra" tiles (4KB memory pages) that exist in the excess rows at the right side of the larger pitch. Obviously no part of the surface that will be accessed can lie in pages that exist only in one mapping but not the other. The new GTT mapping can be done manually by SW between the time the host writes the surface and the device reads it, or it can be accomplished by arranging for the client to use a different GTT than the host (the PPGTT -- see *Logical Memory Mapping* below).

The width of the surface (as set in SURFACE_STATE) must be less than or equal to both the surface pitch and the fence pitch in any scenario where a surface will be accessed by both the host and an internal graphics client. Changing the GTT mapping will not help if this restriction is violated.

Surface Access	Base Address	Pitch	Width	Tile "Walk"
Host only	No restriction	Integral multiple of tile size <= 256KB	Must be <= Fence Pitch	No restriction
Client only	4KB-aligned	Integral multiple of tile size <= 256KB	Must be <= Surface Pitch	Restrictions imposed by the client (see Per Stream Tile Format Support)
Host and Client, No GTT Remapping	Must be TRSA	Fence Pitch = Surface Pitch = integral multiple of tile size <= 256KB	Width <= Pitch	Surface Walk must meet client restriction, Fence Walk = Surface Walk
Host and Client, GTT Remapping	4KB-aligned for client (will be tile-aligned for host)	Both must be Integral multiple of tile size <=128KB, but not necessarily the same	Width <= Min(Surface Pitch, Fence Pitch)	Surface Walk must meet client restriction, Fence Walk = Surface Walk

Per-Stream Tile Format Support

<table>
<tr><th>MI Client</th><th>Tile Formats Supported</th></tr>
<tr><td>CPU Read/Write</td><td>All</td></tr>
<tr><td>Display/Overlay</td><td>Y-Major not supported
X-Major required for Async Flips</td></tr>
<tr><td>Blt</td><td>Linear and X-Major only
No Y-Major support</td></tr>
<tr><td>3D Sampler</td><td>All Combinations of TileY, TileX and Linear are supported. TileY is the fastest, Linear is the slowest.</td></tr>
<tr><td>3D Color,Depth</td><td>
<table>
<tr><th>Rendering Mode
Color-vs-Depth bpp</th><th>Buffer Tiling Supported</th></tr>
<tr><td>Classical
Same Bpp</td><td>Both Linear
Both TileX
Both TileY
Linear & TileX
Linear & TileY
TileX & TileY</td></tr>
<tr><td>Classical
Mixed Bpp</td><td>Both Linear
Both TileX
Both TileY
Linear & TileX
Linear & TileY
TileX & TileY</td></tr>
</table>
</td></tr>
</table>

Memory Compression

Media Memory Compression

The software requirement when using media memory compression is to allocate each compressible surface one memory tile wider than is required based on the surface width plus normal byte padding (this approach is called “pitch+1”). The reason is each compressible surface needs an “extra” tile to the right edge of surface to store important compression control information. For example, if the surface is 1920x1088, this would normally be allocated by the driver to be 2048 bytes wide, or 16 tiles (for NV12 8bpp). Using this “pitch + 1”, the pitch would be set to 17 instead of 16 (and the surface width remains unchanged, only pitch is increased).

The largest supported width will be 4K pixels for 2D RGBA 8bpp surfaces and 2x2K for S3D surfaces (for 4KB pages). E.g. the pitch would be set to 129 in these cases (128+1). NV12 4K would be 33 (28+1). The case of 64KB pages is the same: the driver will allocate 1 extra page to the right (“pitch + 1”), however now the 4K wide restriction is relaxed. With 64KB pages, the widest surface that supports memory compression is 16K for 2D RGBA 8bpp or 2x8K for S3D. E.g. the pitch would be set to 129 in these cases (128+1).

Memory Object Overview

Any memory data accessed by the device is considered part of a *memory object* of some memory object type.

The following table lists the various memory objects types and an indication of their role in the system.

Memory Object Type	Role
Graphics Translation Table (GTT)	Contains PTEs used to translate "graphics addresses" into physical memory addresses.
Hardware Status Page	Cached page of sysmem used to provide fast driver synchronization.
Logical Context Buffer	Memory areas used to store (save/restore) images of hardware rendering contexts. Logical contexts are referenced via a pointer to the corresponding Logical Context Buffer.
Ring Buffers	Buffers used to transfer (DMA) instruction data to the device. Primary means of controlling rendering operations.
Batch Buffers	Buffers of instructions invoked indirectly from Ring Buffers.
State Descriptors	Contains state information in a prescribed layout format to be read by hardware. Many different state descriptor formats are supported.
Vertex Buffers	Buffers of 3D vertex data indirectly referenced through "indexed" 3D primitive instructions.
VGA Buffer (Must be mapped UC on PCI)	Graphics memory buffer used to drive the display output while in legacy VGA mode.
Display Surface	Memory buffer used to display images on display devices.
Overlay Surface	Memory buffer used to display overlaid images on display devices.

Memory Object Type	Role
Overlay Register, Filter Coefficients	Memory area used to provide double-buffer for Overlay register and filter coefficient loading.
Cursor Surface	Hardware cursor pattern in memory.
2D Render Source	Surface used as primary input to 2D rendering operations.
2D Render R-M-W Destination	2D rendering output surface that is read in order to be combined in the rendering function. Destination surfaces that accessed via this Read-Modify-Write mode have somewhat different restrictions than Write-Only Destination surfaces.
2D Render Write-Only Destination	2D rendering output surface that is written but not read by the 2D rendering function. Destination surfaces that accessed via a Write-Only mode have somewhat different restrictions than Read-Modify-Write Destination surfaces.
2D Monochrome Source	1 bpp surfaces used as inputs to 2D rendering after being converted to foreground/background colors.
2D Color Pattern	8x8 pixel array used to supply the "pattern" input to 2D rendering functions.
DIB	"Device Independent Bitmap" surface containing "logical" pixel values that are converted (via LUTs) to physical colors.
3D Color Buffer	Surface receiving color output of 3D rendering operations. May also be accessed via R-M-W (aka blending). Also referred to as a Render Target.
3D Depth Buffer	Surface used to hold per-pixel depth and stencil values used in 3D rendering operations. Accessed via RMW.
3D Texture Map	Color surface (or collection of surfaces) which provide texture data in 3D rendering operations.
"Non-3D" Texture	Surface read by Texture Samplers, though not in normal 3D rendering operations (for example, in video color conversion functions).
Motion Comp Surfaces	These are the Motion Comp reference pictures.
Motion Comp Correction Data Buffer	This is Motion Comp intra-coded or inter-coded correction data.