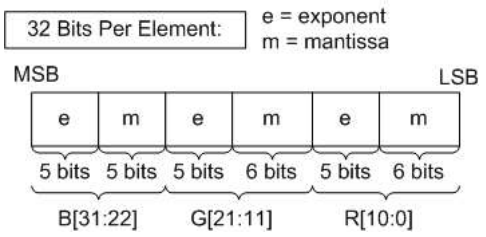
This format consists of 3 independent, reduced-mantissa floating point channels. See the 11-bit and 10-bit Floating Point(3.1.6) section for a description of the mechanics of these reduced precision numbers.



19.3.4 Blending with compressed HDR Formats:

- Only the R11G11B10_FLOAT supports blending (RGBE cannot be a RenderTarget).
- Blending with R11G11B10_FLOAT is defined to occur exactly as if the data is converted to 16f per channel before the blend, and then back afterward.
- As a result, all rules that apply to 16-bit floating point blending also applies to B10G11R11_FLOAT.
- Since R11G11B10_FLOAT does not store alpha, it is always implied to be 1.0f on read into the blender.

19.4 Sub-Sampled Formats

The sub-sampled formats (such as R8G8_B8G8) are reconstructed via replication to per-pixel RGB values prior to use.

The G component is taken from the currently addressed pixel value. The R component is taken from the current pixel value for even x resource addresses, and from the previous ('-1th) x dimension pixel value for odd x resource addresses. The B component is taken from the next ('+1th) x dimension pixel value for even x resource addresses, and from the current pixel value for odd x resource addresses.

Resources in these formats are required to be a multiple of 2 in the x dimension, rounding up to an x dimension of 2 for the smallest mipmap levels. For mipmaps, the sizing and sampling hardware behavior is similar to Block Compressed Formats(19.5), where the top level map must be a multiple of 2 size in the x dimension, and for smaller maps the virtual x dimension size may be odd while the physical size is always even.

The regions being sourced and modified by the Resource Manipulation(5.6) operations are required to be a multiple of 2 in the x dimension.

19.5 Block Compression Formats

Section Contents

(back to chapter)

19.5.1 Overview
19.5.2 Error Tolerance
19.5.3 Promotion to wider UNORM values:
19.5.4 Promotion to wider SNORM values:
19.5.5 Memory Layout
19.5.6 BC1{U|G}: 2(+2 Derived) Opaque Colors or 2(+1 Derived) Opaque Colors + Transparent Black
19.5.7 BC2{U|G}: 2(+2 Derived) Colors, 16 Alphas
19.5.8 BC3{U|G}: 2(+2 Derived) Colors, 2(+6 Derived) Alphas or 2(+4 Derived + Transparent + Opaque) Alphas
19.5.9 BC4U: 2(+6 Derived) Single Component UNORM Values
19.5.10 BC4S: 2(+6 Derived) Single Component SNORM Values
19.5.11 BC5U: 2(+6 Derived) Dual (Independent) Component UNORM Values
19.5.12 BC5S: 2(+6 Derived) Dual (Independent) Component SNORM Values
19.5.13 BC6H / DXGI_FORMAT_BC6H

- 19.5.13.1 BC6H Implementation
- 19.5.13.2 BC6H Decoding
- 19.5.13.3 Per-Block Memory Encoding of BC6H
- 19.5.13.4 BC6H Partition Set
- 19.5.13.5 BC6H Compressed Endpoint Format
- 19.5.13.6 When to Sign_extend
- 19.5.13.7 Transform_inverse
- 19.5.13.8 Generate_palette_unquantized
- 19.5.13.9 Unquantize
- 19.5.13.10 Finish_unquantize

19.5.14 BC7U / DXGI_FORMAT_BC7_UNORM

- 19.5.14.1 BC7 Implementation
- 19.5.14.2 BC7 Decoding
- 19.5.14.3 BC7 Endpoint Decoding, Value Interpolation, Index Extraction, and Bitcount Extraction
- 19.5.14.4 Per-Block Memory Encoding of BC7
  - 19.5.14.4.1 Mode 0
  - 19.5.14.4.2 Mode 1
  - 19.5.14.4.3 Mode 2
  - 19.5.14.4.4 Mode 3
  - 19.5.14.4.5 Mode 4
  - 19.5.14.4.6 Mode 5
  - 19.5.14.4.7 Mode 6
  - 19.5.14.4.8 Mode 7

19.5.1 Overview

This section describes various block-based compression formats. A surface is divided into 4x4 texel blocks, and each 16-texel block is encoded in a particular manner as an atomic unit. Each distinct encoding method is given a unique format name (identified by a four-character code and matching DXGI_FORMAT_BC* name).

Block Compressed formats can be used for Texture2D (including arrays), Texture3D or TextureCube (including arrays), including mipmap surfaces in these Resources.

BC format surfaces are always multiples of full blocks, each block representing 4x4 pixels. For mipmaps, the top level map is required to be a multiple of 4 size in all dimensions. The sizes for the lower level maps are computed as they are for all mipmapped surfaces, and thus may not be a multiple of 4, for example a top level map of 20 results in a second level map size of 10. For these cases, there is a differing 'physical' size and a 'virtual' size. The virtual size is that computed for each mip level without adjustment, which is 10 for the example. The physical size is the virtual size rounded up to the next multiple of 4, which is 12 for the example, and this represents the actual memory size. The sampling hardware will apply texture address processing based on the virtual size (using, for example, border color if specified for accesses beyond 10), and thus for the example case will not access the 11th and 12th row of the resource. So for mipmap chains when an axis becomes < 4 in size, only texels 'a','b','e','f' (see diagram below) are used for a 2x2 map, and texel 'a' is used for 1x1. Note that this is similar to, but distinct from, the surface pitch, which can encompass additional padding beyond the physical surface size.

The regions of BC formats being sourced and/or modified by the Resource Manipulation(5.6) operations are required to be a multiple of 4.

Decompression always occurs before filtering.

19.5.2 Error Tolerance

Valid implementations of BC formats other than BC6H and BC7 may optionally promote or do round-to-nearest division, so long as they meet the following equation for all channels of all texels:

```
| generated - reference | < absolute_error + 0.03
                            *MAX( | endpoint_0 - endpoint_1 |,
                                  | endpoint_0_promoted - endpoint_1_promoted | )
```

absolute_error is defined in the description of each format.

endpoint_0, endpoint_1, and their promoted counterparts have been converted to float from either UNORM or SNORM as specified in the Integer Conversion(3.2.3) rules. Values that the reference decodes to 0.0, 1.0 or -1.0 must always be exact.

For BC6H and BC7, decompression hardware is required to be bit accurate; the hardware must give results that are identical to the decoder described in this specification.

19.5.3 Promotion to wider UNORM values:

Promotion is defined to utilize MSB extension to define the new LSBs as follows.

```
int UNORMPromote(int input, int baseBitCount, int targetBitCount)
{
    int numBits = targetBitCount-baseBitCount;
    input <<= numBits;
    int outval = input;
    do {
        input >>= baseBitCount;
        outval |= input;
        numbits -= baseBitCount;
    } while(numBits > 0);
    return outval;
}
```

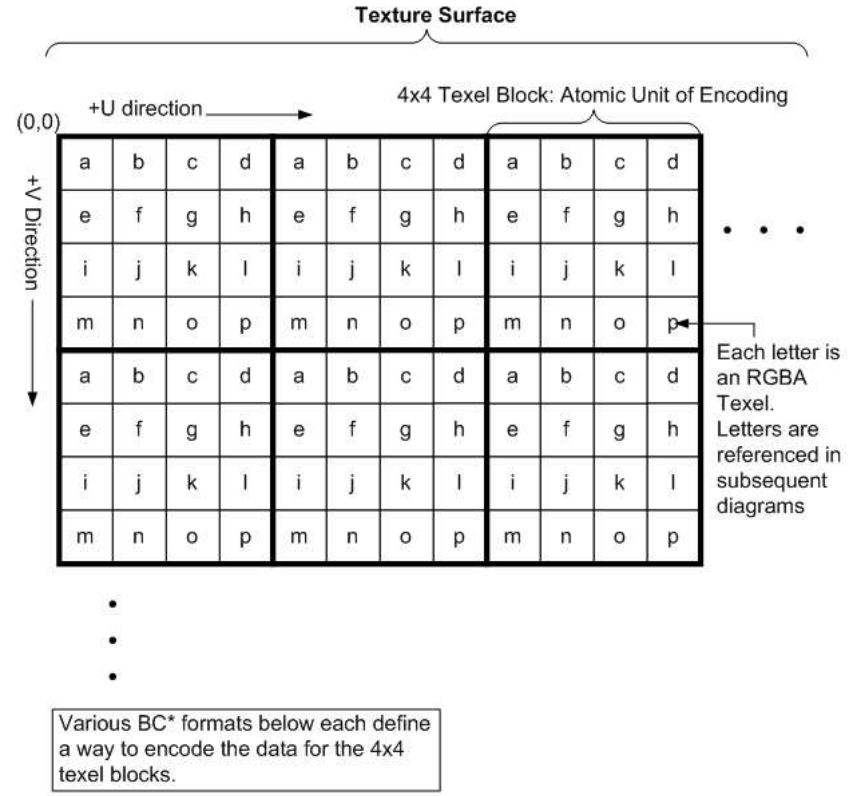
19.5.4 Promotion to wider SNORM values:

```
int SNORMPromote(int input, int base, int target)
{
    if (input<0)
        return -UNORMPromote(-input, baseBitCount-1, targetBitCount-1);
    return UNORMPromote(input, baseBitCount-1, targetBitCount-1);
}
```

19.5.5 Memory Layout

The following diagram depicts the overall layout of data in a Block Compressed surface. After that, the per-block memory encoding for each BC* format is individually illustrated.

Block Compression (BC*) Format Layout

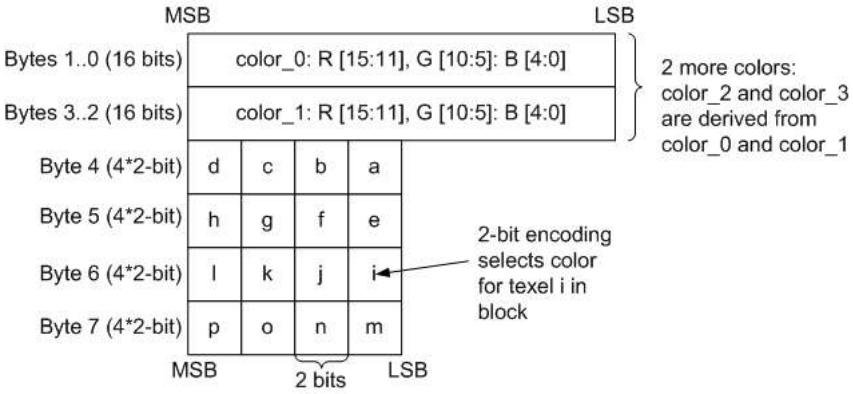


19.5.6 BC1{U|G}: 2(+2 Derived) Opaque Colors or 2(+1 Derived) Opaque Colors + Transparent Black

BC1U/BC1G (DXGI_FORMAT_BC1_UNORM[_SRGB]) is known in older APIs as DXGI_FORMAT_DXT1.

BC1U / DXGI_FORMAT_BC1_UNORM:

8 Bytes per 4x4 Texel Block




```
color_0_p = promoteToUNORM8(color_0)
color_1_p = promoteToUNORM8(color_1)
if (color_0 > color_1) // unsigned compare
{
    // Four-color block:
    color_2 = (2 * color_0_p + color_1_p) / 3;
    color_3 = (color_0_p + 2 * color_1_p) / 3;
    alpha_3 = 1.0f
} else {
    // Three-color block:
    color_2 = (color_0_p + color_1_p) / 2;
    color_3 = (0.0f, 0.0f, 0.0f); alpha_3 = 0.0f
}
// color_*: Actually 3 independent calculations for R,G,B.

// The following 2-bit codes select
// a UNORM8 color for each texel:
// (MSB)00(LSB) = color_0_p, alpha=1.0f
//      01      = color_1_p, alpha=1.0f
//      10      = color_2,   alpha=1.0f
//      11      = color_3,   alpha=alpha_3
```

- Filtering must occur with at least UNORM8 precision.
- Values specified as 0.0f or 1.0f must be exact.
- **absolute_error = 1.0 / 255.0**

BC1G / DXGI_FORMAT_BC1_UNORM_SRGB:

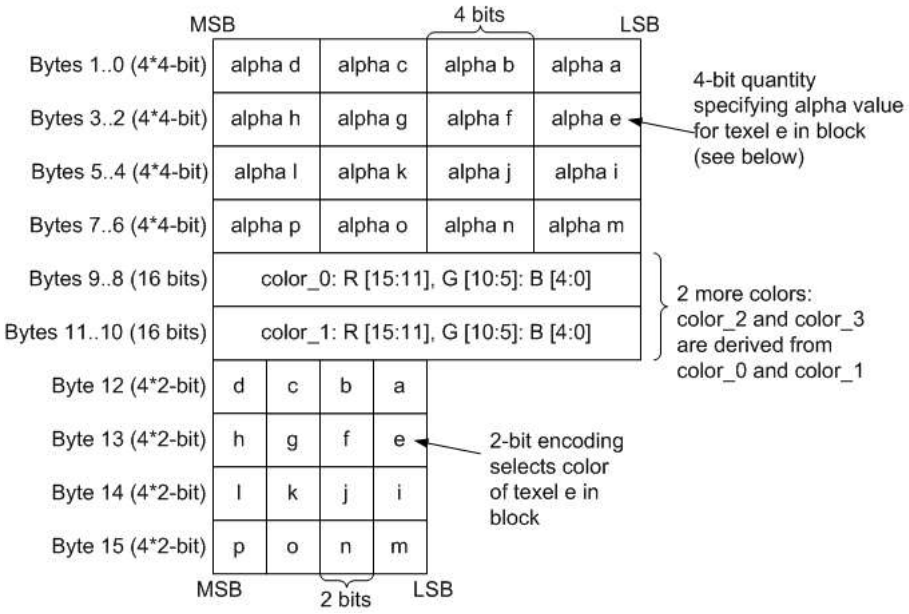
Same as BC1U, but colors are in sRGB space, linearized pre-filter on read. sRGB conversion should occur the same as with uncompressed UNORM8 formats. If an implementation provides more precise palette entries than it can linearize, it may have up to 1 UNORM8 ULP error in conversion on input to linearization.

19.5.7 BC2{U|G}: 2(+2 Derived) Colors, 16 Alphas

BC2U/BC2G (DXGI_FORMAT_BC2_UNORM[_SRGB]) is known in older APIs as both DXGI_FORMAT_DXT2 and DXGI_FORMAT_DXT3, where DXT2 is the same as DXT3 except whether or not the color data is assumed to be pre-multiplied by alpha. This pre-multiplied alpha distinction is meaningless to the graphics system, as the hardware doesn't care about pre-multiplied alpha. It is up to application to change Shader code if appropriate for handling the distinction. Therefore, the use of separate format names to distinguish pre-multiplied alpha vs. non-pre-multiplied alpha was removed for D3D11.3. If applications want to keep track of whether a format contains pre-multiplied alpha, that can be done by other means (such as storing private data for resources), which would work equally well for all formats, and not just the Block Compression formats. Note that in contrast to the pre-multiplied alpha property, the distinction of whether the resource contains SRGB data or not is indeed important for hardware, so in D3D11.3 separate formats are used for linear vs SRGB data where appropriate.

BC2U / DXGI_FORMAT_BC2_UNORM:

16 Bytes per 4x4 Texel Block



```
// Four-color block: derive the other two colors
color_0_p = promoteToUNORM8(color_0)
color_1_p = promoteToUNORM8(color_1)
color_2 = (2 * color_0 + color_1) / 3;
color_3 = (color_0 + 2 * color_1) / 3;
// color_*: Actually 3 independent calculations for R,G,B.

// The following 2-bit codes select
// a UNORM8 color for each texel:
// (MSB)00(LSB) = color_0_p,
//      01      = color_1_p,
//      10      = color_2,
//      11      = color_3

// Derive alpha value for texel t:
alpha = alpha[t]/15.0f
```

- Implementations must follow the same precision rules as BC1 for color palette entries.
- Alpha values must be filtered with at least UNORM8 precision.
- **absolute_error = 1.0/255.0**

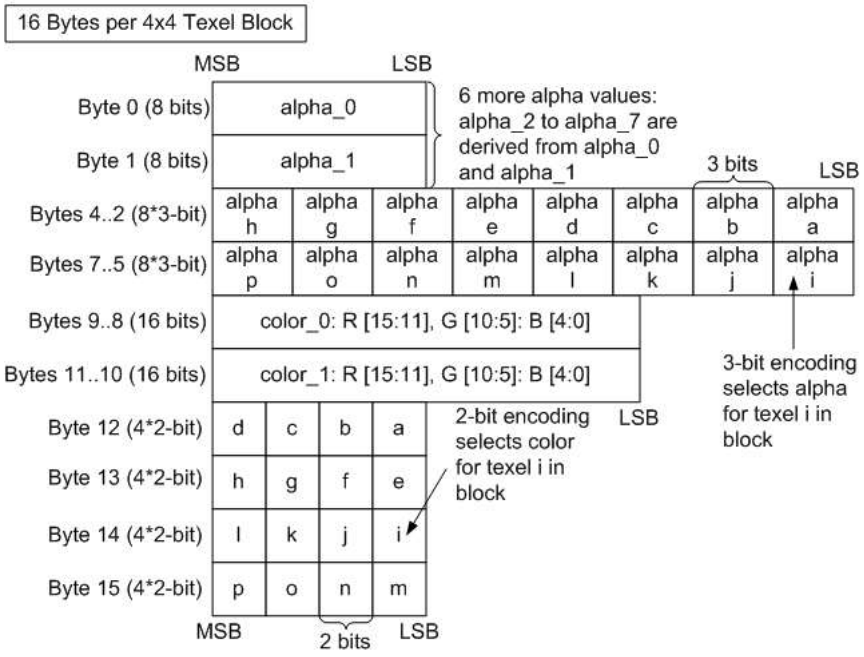
BC2G / DXGI_FORMAT_BC2_UNORM_SRGB:

- Color is decoded prior to gamma conversion the same as BC2U.
- Gamma conversion then occurs the same way as BC1G.
- Alpha decodes the same as BC2U.

19.5.8 BC3{U|G}: 2(+2 Derived) Colors, 2(+6 Derived) Alphas or 2(+4 Derived + Transparent + Opaque) Alphas

BC3U/BC3G (DXGI_FORMAT_BC3_UNORM[_SRGB]) is known in older APIs as both DXGI_FORMAT_DXT4 and DXGI_FORMAT_DXT5, where DXT4 is the same as DXT5 except whether or not the color data is assumed to be pre-multiplied by alpha. This pre-multiplied alpha distinction is meaningless to the graphics system, as the hardware doesn't care about pre-multiplied alpha. It is up to the application to change Shader code if appropriate for handling the distinction. Therefore, the use of separate format names to distinguish pre-multiplied alpha vs. non-pre-multiplied alpha was removed for D3D11.3. If applications want to keep track of whether a format contains pre-multiplied alpha, that can be done by other means (such as storing private data for resources), which would work equally well for all formats, and not just the Block Compression formats. Note that in contrast to the pre-multiplied alpha property, the distinction of whether the resource contains SRGB data or not is indeed important for hardware, so in D3D11.3 separate format names distinguish linear vs SRGB data where appropriate.

BC3U / DXGI_FORMAT_BC3_UNORM:




```
// Four-color block: derive the other two colors
color_0_p = promoteToUNORM8(color_0)
color_1_p = promoteToUNORM8(color_1)
color_2 = (2 * color_0 + color_1) / 3;
color_3 = (color_0 + 2 * color_1) / 3;
// color_*: Actually 3 independent calculations for R,G,B.

// The following 2-bit codes select
// a UNORM8 color for each texel:
// (MSB)00(LSB) = color_0_p,
//       01     = color_1_p,
//       10     = color_2,
//       11     = color_3
```

- Implementations must follow the same guidelines as BC1 for color palette entries.
- Alpha values decode as with BC4U
- Filtering must occur with at least UNORM8 precision.
- **absolute_error = 1.0/255.0**

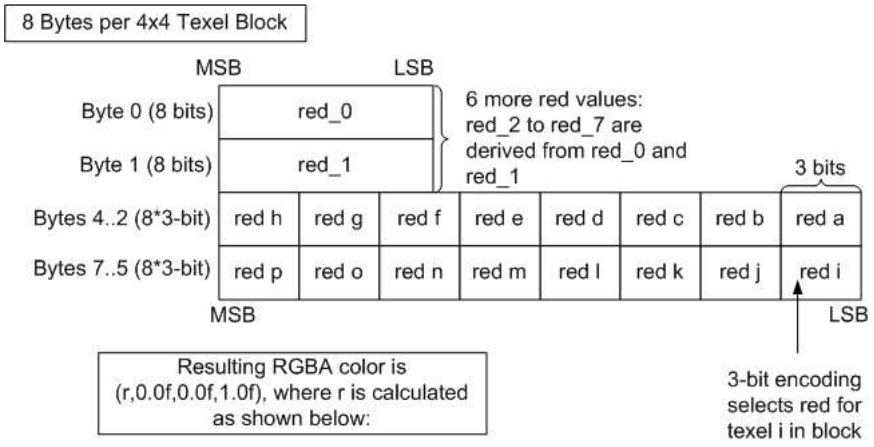
BC3G / DXGI_FORMAT_BC3_UNORM_SRGB:

- Color is decoded prior to gamma conversion the same as BC3U.
- Gamma conversion then occurs the same way as BC1G.
- Alpha decodes the same as BC3U.

19.5.9 BC4U: 2(+6 Derived) Single Component UNORM Values

This general purpose format compresses single-component UNORM data.

BC4U / DXGI_FORMAT_BC4_UNORM:



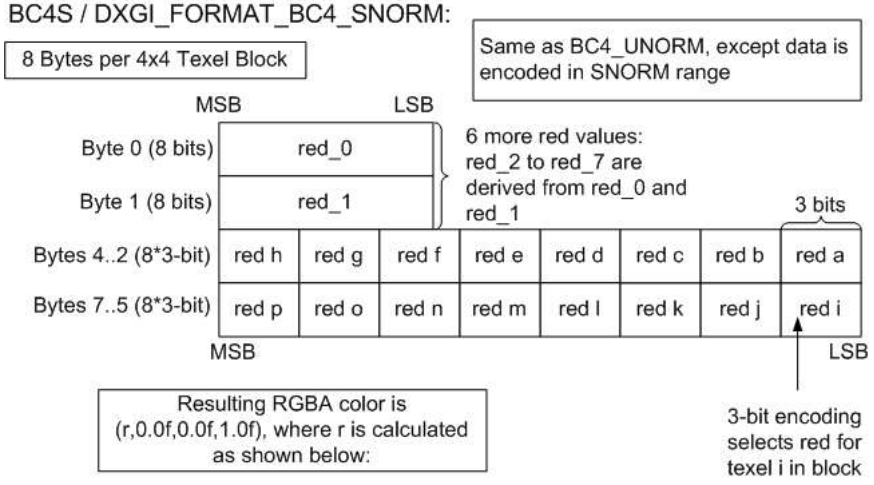

```
redf_0 = UNORM8ToFloat(red_0)
redf_1 = UNORM8ToFloat(red_1)
if (red_0 > red_1) // unsigned compare
{
    // 8-red block
    // Bit code 000 = redf_0, 001 = redf_1, others are interpolated.
    redf_2 = (6 * redf_0 + 1 * redf_1) / 7.0f;   // bit code 010
    redf_3 = (5 * redf_0 + 2 * redf_1) / 7.0f;   // bit code 011
    redf_4 = (4 * redf_0 + 3 * redf_1) / 7.0f;   // bit code 100
    redf_5 = (3 * redf_0 + 4 * redf_1) / 7.0f;   // bit code 101
    redf_6 = (2 * redf_0 + 5 * redf_1) / 7.0f;   // bit code 110
    redf_7 = (1 * redf_0 + 6 * redf_1) / 7.0f;   // bit code 111
} else {
    // 6-red block.
    // Bit code 000 = redf_0, 001 = redf_1, others are interpolated.
    redf_2 = (4 * redf_0 + 1 * redf_1) / 5.0f; // bit code 010
    redf_3 = (3 * redf_0 + 2 * redf_1) / 5.0f; // bit code 011
    redf_4 = (2 * redf_0 + 3 * redf_1) / 5.0f; // bit code 100
    redf_5 = (1 * redf_0 + 4 * redf_1) / 5.0f; // bit code 101
    redf_6 = 0.0f;                             // bit code 110
    redf_7 = 1.0f;                             // bit code 111
}
```

- Filtering must occur with at least UNORM16 precision.
- red_0, red_1, 0.0f and 1.0f must be exact.
- **absolute_error = 1.0/65535.0**

19.5.10 BC4S: 2(+6 Derived) Single Component SNORM Values

This general purpose format compresses single-component SNORM data.

BC4S / DXGI_FORMAT_BC4_SNORM:




```
redf_0 = SNORM8ToFloat(red_0)
redf_1 = SNORM8ToFloat(red_1)
if (red_0 > red_1) // signed compare.
{
    // 8-red block
    // Bit code 000 = redf_0, 001 = redf_1, others are interpolated.
    redf_2 = (6 * redf_0 + 1 * redf_1) / 7.0f;   // bit code 010
    redf_3 = (5 * redf_0 + 2 * redf_1) / 7.0f;   // bit code 011
    redf_4 = (4 * redf_0 + 3 * redf_1) / 7.0f;   // bit code 100
    redf_5 = (3 * redf_0 + 4 * redf_1) / 7.0f;   // bit code 101
    redf_6 = (2 * redf_0 + 5 * redf_1) / 7.0f;   // bit code 110
    redf_7 = (1 * redf_0 + 6 * redf_1) / 7.0f;   // bit code 111
} else {
```

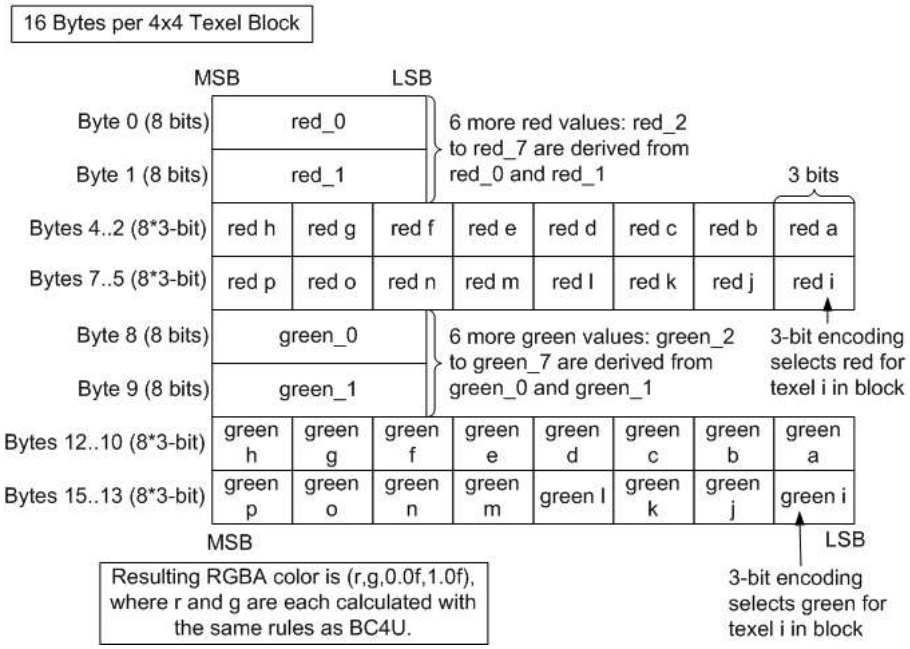
```
    // 6-red block.
    // Bit code 000 = redf_0, 001 = redf_1, others are interpolated.
    redf_2 = (4 * redf_0 + 1 * redf_1) / 5.0f; // bit code 010
    redf_3 = (3 * redf_0 + 2 * redf_1) / 5.0f; // bit code 011
    redf_4 = (2 * redf_0 + 3 * redf_1) / 5.0f; // bit code 100
    redf_5 = (1 * redf_0 + 4 * redf_1) / 5.0f; // bit code 101
    redf_6 = -1.0f;                            // bit code 110
    redf_7 = 1.0f;                             // bit code 111
}
```

- Filtering must occur with at least SNORM16 precision.
- red_0, red_1, -1.0f and 1.0f must be exact.
- **absolute_error = 1.0/32767.0**

19.5.11 BC5U: 2(+6 Derived) Dual (Independent) Component UNORM Values

This general purpose format compresses dual-component UNORM data.

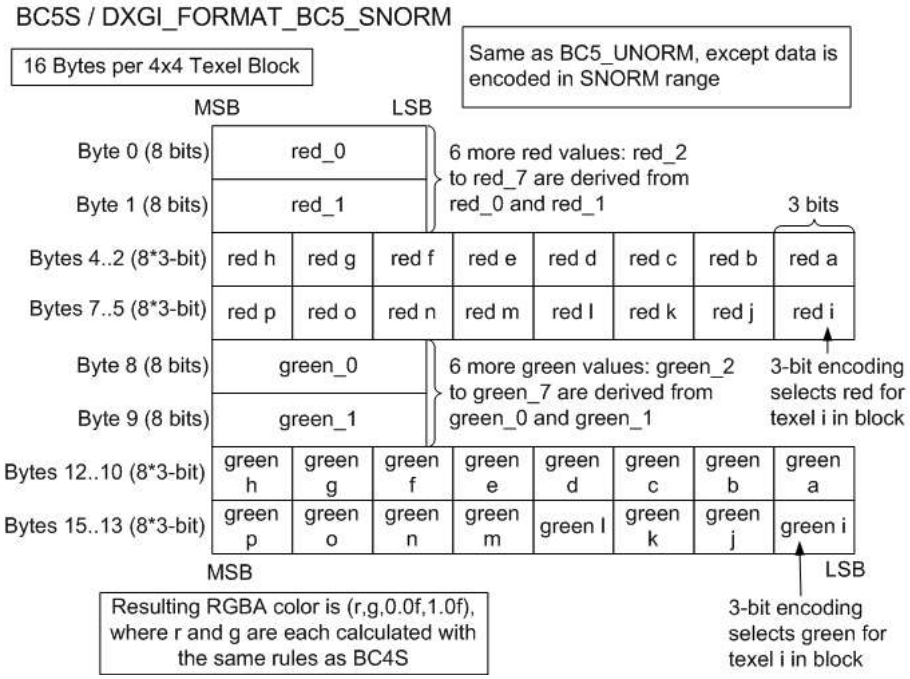
BC5U / DXGI_FORMAT_BC5_UNORM:



absolute_error = 1.0/65535.0

19.5.12 BC5S: 2(+6 Derived) Dual (Independent) Component SNORM Values

---



absolute_error = 1.0/32767.0

19.5.13 BC6H / DXGI_FORMAT_BC6H

The following DXGI_FORMATs are in this category: DXGI_FORMAT_BC6H_TYPELESS, DXGI_FORMAT_BC6H_UF16, and DXGI_FORMAT_BC6H_SF16.

The BC6H format can be used for Texture2D (including arrays), Texture3D or TextureCube (incl. arrays). All of these uses include mipmap surfaces in these resources.

BC6H uses a fixed block size of 16 bytes and a fixed tile size of 4x4 pixels. Just as with previous BC formats, images larger than BC6H's tile size are compressed using multiple blocks. The same addressing identity also applies to three-dimensional images as well as mip-maps, cubemaps, and texture arrays.

BC6H compresses three-channel images that have high dynamic range greater than 8 bits per channel. The supported per-channel formats are:

- Unsigned 16 bit floating point (DXGI_FORMAT_BC6H_UF16)
- Signed 16 bit floating point (DXGI_FORMAT_BC6H_SF16)

All image tiles must be of the same format.

Note that the 16 bit floating point format is often referred to as "half" format, containing 1 sign bit, 5 exponent bits, and 10 mantissa bits.

BC6H supports floating point denorms, but INF and NaN are not supported. The exception is the signed mode of BC6H, which can represent ±INF. While this ±INF "support" was unintentional, it is baked into the format. So it is valid for encoders to intentionally use ±INF, but they also have the option to clamp during encode to avoid it. In general, faced with ±INF or NaN input data to deal with, encoders are loosely encouraged to clamp ±INFs to the corresponding maximum non-INF representable value, and map NaN to 0 prior to compression.

BC6H does not store any alpha data.

The BC6H decoder decompresses to the specified format prior to filtering.

BC6H decompression hardware is required to be bit accurate; the hardware must give results that are identical to the decoder described in this specification.

19.5.13.1 BC6H Implementation

A BC6H block consists of mode bits, compressed endpoints, sometimes a partition index, and compressed indices.

BC6H uses 14 different modes.

BC6H stores endpoint colors as a red, green, and blue (RGB) triplet, defining a palette of colors on an approximate line between two endpoints. Depending upon the mode, a tile is divided into one or two regions, each having its own pair of endpoints. BC6H stores one palette index per pixel.

In the two region case (hereafter referred to as TWO), there are 32 possible partitions. (The one region case will hereafter be referred to as ONE.)

19.5.13.2 BC6H Decoding

The pseudocode below outlines the steps to decompress the pixel at *(x,y)* given the 16-byte BC6H block.

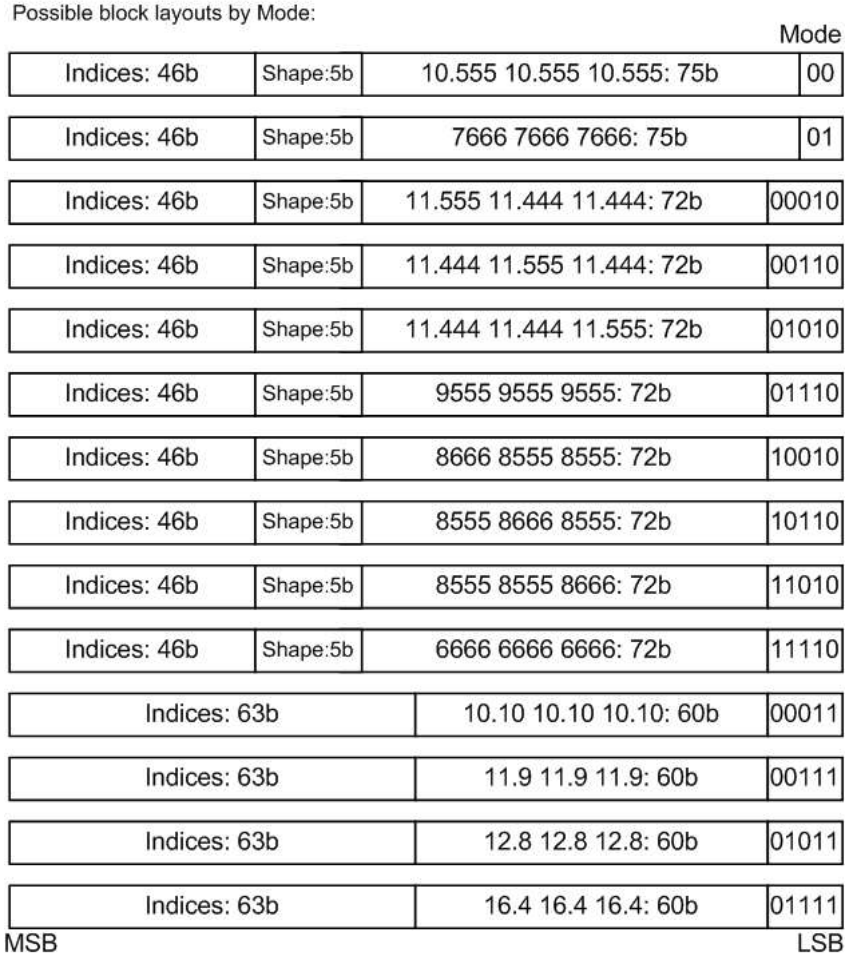
```
decompress(x, y, block)
{
    mode = extract_mode(block);
    endpoints;
    index;

    if(mode.type == ONE)
    {
        endpoints = extract_compressed_endpoints(mode, block);
        index = extract_index_ONE(x, y, block);
    }
    else //mode.type == TWO
    {
        partition = extract_partition(block);
        region = get_region(partition, x, y);
        endpoints = extract_compressed_endpoints(mode, region, block);
        index = extract_index_TWO(x, y, partition, block);
    }

    unquantize(endpoints);
    color = interpolate(index, endpoints);
    finish_unquantize(color);
}
```

19.5.13.3 Per-Block Memory Encoding of BC6H

BC6H / DXGI_FORMAT_BC6H

16 Bytes per 4x4 Texel Block

Possible block layouts by Mode:



The diagram above shows the 14 possible formats for BC6H blocks. The formats can be uniquely identified by the Mode bits.The first ten modes are used by TWO, and the mode field can be either 2 or 5 bits long. These blocks also have fields for the compressed endpoints (75 bits), partition (5 bits), and indices (46 bits). As an example, the code "11.555 11.444 11.444" indicates both the precision of the red, green, and blue endpoints stored (11), as well as the number of bits used to store the delta values for the transformed endpoints (5, 4, and 4 bits for red, green, and blue, respectively, for 3 delta values.) The "6666" mode handles the case when the endpoints cannot be transformed; only the quantized endpoints are stored.

The last four modes are used by ONE, and the mode field is 5 bits. These blocks have fields for the endpoints (60 bits) and indices (63 bits). For ONE, the example endpoint code "11.9 11.9 11.9" indicates both the precision of the red, green, and blue endpoints stored (11), as well as the number of bits used to store the delta values for the transformed endpoints (9 bits for red, green, and blue, respectively, for 1 delta value.) The "10.10" mode handles the case when the endpoints cannot be transformed; only the quantized endpoints are stored.

Modes 10011, 10111, 11011, and 11111 are reserved and should not be used by the encoder. If hardware is given these modes, the resulting decompressed block must contain zeroes in all channels except the alpha channel. For BC6H, the alpha channel should always return 1.0 regardless of the mode.

19.5.13.4 BC6H Partition Set

There are 32 partition sets for TWO, which are defined by Table 1 below. Each 4x4 block represents a single shape. Note that this table is equivalent to the first 32 entries of BC7's 2 subset partition table.

```
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,

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,

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,

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,

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,

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,

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,

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,
```

Table 1: Partition Sets for TWO

In the table of partitions above, the bolded and underlined entry is the location of the fix-up index for subset 1 which is specified with one less bit. The fix-up index for subset 0 is always index 0 (i.e. the partitioning is arranged so that index 0 is always in subset 0). Partition order goes from top-left to bottom right, walking left-to-right, then top-to-bottom.

19.5.13.5 BC6H Compressed Endpoint Format

Header Bit	10 5 5 5	7 6 6 6	11 5 4 4	11 4 5 4	11 4 4 5	9 5 5 5	8 6 5 5	8 5 6 5	8 5 5 6	6 6 6 6	10 10	11 9	12 8	16 4
0														
1	m[1:0]	m[1:0]												
2	gy[4]	gy[5]												
3	by[4]	gz[4]												
4	bz[4]	gz[5]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]	m[4:0]
5														
6														
7														
8														
9														
10										rw[5:0]				
11		rw[6:0]								gz[4]				
12		bz[0]					rw[7:0]	rw[7:0]	rw[7:0]	bz[0]				
13		bz[1]				rw[8:0]	gz[4]	bz[0]	bz[1]	bz[1]				
14	rw[9:0]	by[4]	rw[9:0]	rw[9:0]	rw[9:0]	by[4]	by[4]	by[4]	by[4]	by[4]	rw[9:0]	rw[9:0]	rw[9:0]	rw[9:0]
15														
16														
17														
18														
19														
20										gw[5:0]				
21		gw[6:0]								gy[5]				
22		by[5]					gw[7:0]	gw[7:0]	gw[7:0]	by[5]				
23		bz[2]				gw[8:0]	bz[2]	gy[5]	by[5]	bz[2]				
24	gw[9:0]	gy[4]	gw[9:0]	gw[9:0]	gw[9:0]	gy[4]	gy[4]	gy[4]	gy[4]	gy[4]	gw[9:0]	gw[9:0]	gw[9:0]	gw[9:0]
25														
26														
27														
28														
29														
30										bw[5:0]				
31		bw[6:0]								gz[5]				
32		bz[3]					bw[7:0]	bw[7:0]	bw[7:0]	bz[3]				
33		bz[5]				bw[8:0]	bz[3]	gz[5]	bz[5]	bz[5]				
34	bw[9:0]	bz[4]	bw[9:0]	bw[9:0]	bw[9:0]	bz[4]	bz[4]	bz[4]	bz[4]	bz[4]	bw[9:0]	bw[9:0]	bw[9:0]	bw[9:0]
35														
36														
37														
38				rx[3:0]	rx[3:0]									rx[3:0]
39	rx[4:0]		rx[4:0]	rw[10]	rw[10]	rx[4:0]		rx[4:0]	rx[4:0]					
40	gz[4]	rx[5:0]	rw[10]	gz[4]	by[4]	gz[4]	rx[5:0]	gz[4]	gz[4]	rx[5:0]				
41														
42													rx[7:0]	
43												rx[8:0]		
44	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	gy[3:0]	rx[9:0]	rw[10]	rw[10:11]	rw[10:15]
45														
46														
47														
48			gx[3:0]		gx[3:0]									gx[3:0]
49	gx[4:0]		gw[10]	gx[4:0]	gw[10]	gx[4:0]	gx[4:0]		gx[4:0]					
50	bz[0]	gx[5:0]	bz[0]	gw[10]	bz[0]	bz[0]	bz[0]	gx[5:0]	bz[0]	gx[5:0]				
51														
52														
53												gx[8:0]		
54	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gz[3:0]	gx[9:0]	gw[10]	gw[10:11]	gw[10:15]
55														
56														
57														
58			bx[3:0]	bx[3:0]										bx[3:0]
59	bx[4:0]		bw[10]	bw[10]	bx[4:0]	bx[4:0]	bx[4:0]	bx[4:0]						
60	bz[1]	bx[5:0]	bz[1]	bz[1]	bw[10]	bz[1]	bz[1]	bz[1]	bx[5:0]	bx[5:0]				
61														
62													bx[7:0]	
63												bx[8:0]		
64	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	by[3:0]	bx[9:0]	bw[10]	bw[10:11]	bw[10:15]
65														
66														
67														
68				ry[3:0]	ry[3:0]									
69	ry[4:0]		ry[4:0]	bz[0]	bz[1]	ry[4:0]		ry[4:0]	ry[4:0]					
70	bz[2]	ry[5:0]	bz[2]	bz[2]	bz[2]	bz[2]	ry[5:0]	bz[2]	bz[2]	ry[5:0]				
71														
72														
73														
74				rz[3:0]	rz[3:0]									
75	rz[4:0]		rz[4:0]	gy[4]	bz[4]	rz[4:0]		rz[4:0]	rz[4:0]					
76	bz[3]	rz[5:0]	bz[3]	bz[3]	bz[3]	bz[3]	rz[5:0]	bz[3]	bz[3]	rz[5:0]				
77														
78														
79														
80														
81	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]	d[4:0]				
	10 5 5 5	7 6 6 6	11 5 4 4	11 4 5 4	11 4 4 5	9 5 5 5	8 6 5 5	8 5 6 5	8 5 5 6	6 6 6 6	10 10	11 9	12 8	16 4

Table 2: Compressed Endpoint Formats

Table 2 above shows the bit fields for the packed compressed endpoints as a function of the endpoint format. This takes up 82 bits for TWO and 65 bits for ONE. As an example, the first 5 bits of the header for the last encoding above (i.e. the right-most column) are bits m[4:0], the next 10 bits of the header are the bits rw[9:0], and so forth.

The field names are defined by the following table

```
FIELD   VARIABLE    FIELD   VARIABLE        FIELD    VARIABLE        FIELD    VARIABLE
m       mode        rw        endpt[0].A[0]   gw         endpt[0].A[1]        bw          endpt[0].A[2]
d       shape index rx        endpt[0].B[0]   gx         endpt[0].B[1]        bx          endpt[0].B[2]
                    ry        endpt[1].A[0]   gy         endpt[1].A[1]        by            endpt[1].A[2]
                    rz        endpt[1].B[0]   gz         endpt[1].B[1]        bz            endpt[1].B[2]
```

Endpt[i] refers to the 0th or 1st pair of endpoints. A is one endpoint of 3 channels A[0]..A[2], and similarly B is the other endpoint of 3 channels.

19.5.13.6 When to Sign_extend

For TWO, there are four endpoint values to possibly sign-extend. *endpts[0].A* is signed only if the format is a signed format. The other endpoints are signed only if the endpoint was transformed, or the format is a signed format.

```
static void sign_extend(Pattern &p, IntEndpts endpts[NREGIONS_TWO])
{
    for (int i=0; i<NCHANNELS; ++i)
    {
    if (BC6H::FORMAT == SIGNED_F16)
        endpts[0].A[i] = SIGN_EXTEND(endpts[0].A[i], p.chan[i].prec);
    if (p.transformed || BC6H::FORMAT == SIGNED_F16)
    {
        endpts[0].B[i] = SIGN_EXTEND(endpts[0].B[i], p.chan[i].delta[0]);
        endpts[1].A[i] = SIGN_EXTEND(endpts[1].A[i], p.chan[i].delta[1]);
        endpts[1].B[i] = SIGN_EXTEND(endpts[1].B[i], p.chan[i].delta[2]);
    }
    }
}
```

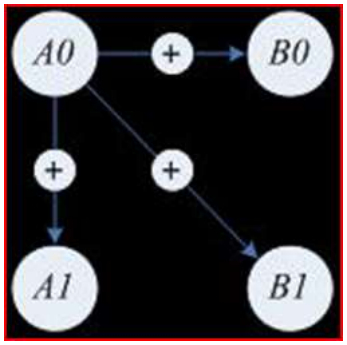
The code for ONE is similar and just removes *endpts[1]*.

```
static void sign_extend(Pattern &p, IntEndpts endpts[NREGIONS_ONE])
{
    for (int i=0; i<NCHANNELS; ++i)
    {
    if (BC6H::FORMAT == SIGNED_F16)
        endpts[0].A[i] = SIGN_EXTEND(endpts[0].A[i], p.chan[i].prec);
    if (p.transformed || BC6H::FORMAT == SIGNED_F16)
        endpts[0].B[i] = SIGN_EXTEND(endpts[0].B[i], p.chan[i].delta[0]);
    }
}
```

There is also sign extending for signed formats in the transform_inverse step shown below.

19.5.13.7 Transform_inverse

For TWO, the transform applies the inverse of the difference encoding, adding the base value at *endpt[0].A* to the other three entries, for a total of 9 adds. In the diagram below, the base value is represented as *A0* and has the highest precision. *A1, B0, and B1* are all deltas off of the anchor value, and these deltas are represented with lower precision. (A0 corresponds to endpt[0].A, B0 to endpt[0].B, and similarly for A1 and B1.)



The ONE case is similar, except there is only 1 delta offset, and thus a total of only 3 adds.

The decompressor should ensure that the results of the inverse transform will not overflow the precision of *endpt[0].A*. In the case of overflow, the values resulting from the inverse transform should wrap within the same number of bits. If the precision of A0 is 'p' bits, the transform is:

```
B0 = (B0+A0) & ((1 << p) - 1)
```

and similarly for the other cases.

For signed formats the results of the delta arithmetic must be sign extended as well. If the sign extend operation is thought of as extending both signs: 1 (negative) and 0 (positive), then the sign extending of 0 takes care of the clamp above. Or equivalently after the clamp above, only 1 (negative) needs to be extended.

19.5.13.8 Generate_palette_unquantized

Given the uncompressed endpoints, the next steps are to perform an initial unquantization step, interpolate, and then do a final unquantize. Seperating the unquantize step into two substeps reduces the number of multiplications required compared to doing a full unquantize before interpolating.

The code below illustrates the unquantizing process to retrieve estimates of the original 16 bit value, and then using the specified weights to get 6 additional values into the palette. The same operation is performed on each channel.

Since the full range of the unquantize function is -32768 to 65535, the interpolator is implemented using 17 bit signed arithmetic.

After interpolation, the values are passed to the finish_unquantize function, which applies the final scaling.

All hardware decompressors are required to return bit accurate results with this functon.

```
int aWeight3[] = {0, 9, 18, 27, 37, 46, 55, 64};
int aWeight4[] = {0, 4, 9, 13, 17, 21, 26, 30, 34, 38, 43, 47, 51, 55, 60, 64};

// c1, c2: endpoints of a component
void generate_palette_unquantized(UINT8 uNumIndices, int c1, int c2, int prec, UINT16 palette[NINDICES])
{
    int* aWeights;
    if(uNumIndices == 8)
        aWeights = aWeight3;
    else  // uNumIndices == 16
        aWeights = aWeight4;

    int a = unquantize(c1, prec);
    int b = unquantize(c2, prec);

    // interpolate
    for(int i = 0; i < uNumIndices; ++i)
        palette[i] = finish_unquantize((a * (64 - aWeights[i]) + b * aWeights[i] + 32) >> 6);
}
```

19.5.13.9 Unquantize

The following describes how unquantize works. For UF16, 'comp' is unquantized into 0x0000 ~ 0xFFFF range to maximize the usage of bits.

```
int unquantize(int comp, int uBitsPerComp)
{
    int unq, s = 0;
    switch(BC6H::FORMAT)
    {
    case UNSIGNED_F16:
        if(uBitsPerComp >= 15)
            unq = comp;
        else if(comp == 0)
            unq = 0;
        else if(comp == ((1 << uBitsPerComp) - 1))
            unq = 0xFFFF;
        else
            unq = ((comp << 16) + 0x8000) >> uBitsPerComp;
        break;

    case SIGNED_F16:
        if(uBitsPerComp >= 16)
            unq = comp;
        else
        {
            if(comp < 0)
            {
                s = 1;
                comp = -comp;
            }

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

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

19.5.13.10 Finish_unquantize

finish_unquantize is called after palette interpolation. The *unquantize* function postpones the scaling by 31/32 for signed, 31/64 for unsigned. This is needed to get the final value into valid half range(-0x7BFF ~ 0x7BFF) after the palette interpolation is completed to reduce the number of necessary multiplications. *finish_unquantize* applies the final scaling and returns an *unsigned short* value that gets reinterpreted into *half*.

```
unsigned short finish_unquantize(int comp)
{
    if(BC6H::FORMAT == UNSIGNED_F16)
    {
        comp = (comp * 31) >> 6;                                    // scale the magnitude by 31/64
        return (unsigned short) comp;
    }
    else // (BC6H::FORMAT == SIGNED_F16)
    {
        comp = (comp < 0) ? -(((-comp) * 31) >> 5) : (comp * 31) >> 5;   // scale the magnitude by 31/32
        int s = 0;
        if(comp < 0)
        {
            s = 0x8000;
            comp = -comp;
        }
        return (unsigned short) (s | comp);
    }
}
```

19.5.14 BC7U / DXGI_FORMAT_BC7_UNORM

The following DXGI_FORMATs are in this category: DXGI_FORMAT_BC7_TYPELESS, DXGI_FORMAT_BC7_UNORM, and DXGI_FORMAT_BC7_UNORM_SRGB

The BC7 format can be used for Texture2D (including arrays), Texture3D or TextureCube (incl. arrays). All of these uses include mipmap surfaces in these resources.

BC7 uses a fixed block size of 16 bytes and a fixed tile size of 4x4 pixels. As with other BC formats, images larger than BC7's tile size are compressed using multiple blocks. The same addressing identity also applies to three-dimensional images as well as mip-maps, cubemaps, and texture arrays.

BC7 compresses both three-channel and four-channel fixed-point data images. Typically source data will be 8-bits per component fixed point, although the format is capable of encoding source data with higher bits per component. All image tiles must be of the same format.

The BC7 decoder decompresses to the specified format prior to filtering.

BC7 decompression hardware is required to be bit accurate; the hardware must give results that are identical to the decoder described in this specification.

19.5.14.1 BC7 Implementation

A BC7 block can take one of 8 modes, and the block mode is always stored in the LSBs of the 128-bit block. The block mode is encoded by zero or more "0"'s followed by a "1". This mode string starts from the block LSB.

A BC7 block may contain multiple endpoint pairs. For the purposes of this document, the set of indices that correspond to an endpoint pair may be referred to as a subset.

In some block modes the endpoint representation is encoded in a form that for the purposes of this document will be called RGBP – in these cases the P bit represents a shared LSB for the components of the endpoint. For example, if the endpoint representation for the format was RGBP 5.5.5.1 then the endpoint would be interpreted as an RGB 6.6.6 value, with the LSB of each component being taken from the state of the P bit. If the representation was RGBAP 5.5.5.5.1 then the endpoint would be interpreted as an RGBA 6.6.6.6 value. Depending on the block mode the shared LSB may either be specified for both endpoints of a subset individually (2 P-bits per subset), or shared between the endpoints of the subset (1 P-bit per subset)

For BC7 blocks that do not explicity encode alpha, a BC7 block consists of mode bits, partition bits, compressed endpoints, sometimes a P-bit, and compressed indices. In these blocks the endpoints have an R.G.B-only representation and alpha is decoded as 1.0 for all texels

For BC7 blocks that encode combined color and alpha, a block consists of mode bits, sometimes partition bits, compressed endpoints, and compressed indices. In these blocks the endpoint color values are specified in an R.G.B.A format, and alpha values are interpolated along with the color values.

For BC7 blocks that separately encode color and alpha, a block consists of mode bits, rotation bits, sometimes an index selector bit, compressed endpoints, and compressed indices. These blocks effectively have a vector channel (R.G.B) and a scalar channel (A) separately encoded.

BC7 uses 8 different modes.

BC7 defines a palette of colors on an approximate line between two endpoints. The mode specifies the number of interpolating endpoint pairs per block. BC7 stores one palette index per pixel.

For each subset of indices that corresponds to a pair of endpoints, the encoder fixes the state of one bit of the compressed index data for that subset. This is done by choosing an endpoint order that allows the index for the designated fixup index to have 0 as its MSB, which can therefore be discarded saving one bit per subset. The indices with the "fix-up" bit are noted in the partition tables for 2 subsets(19.5.14.5) and 3 subsets(19.5.14.6) below. For block modes with only a single subset, the fix-up index is always index 0.

19.5.14.2 BC7 Decoding

The pseudocode below outlines the steps to decompress the pixel at *(x,y)* given the 16-byte BC7 block.

```
decompress(x, y, block)
{
    mode = extract_mode(block);

    //decode partition data from explicit partition bits
    subset_index = 0;
    num_subsets = 1;

    if (mode.type == 0 OR == 1 OR == 2 OR == 3 OR == 7)
    {
        num_subsets = get_num_subsets(mode.type);
        partition_set_id = extract_partition_set_id(mode, block);
        subset_index = get_partition_index(num_subsets, partition_set_id, x, y);
    }

    //extract raw, compressed endpoint bits
    UINT8 endpoint_array[num_subsets][4] = extract_endpoints(mode, block);

    //decode endpoint color and alpha for each subset
    fully_decode_endpoints(endpoint_array, mode, block);

    //endpoints are now complete.
    UINT8 endpoint_start[4] = endpoint_array[2 * subset_index];
    UINT8 endpoint_end[4]   = endpoint_array[2 * subset_index + 1];

    //Determine the palette index for this pixel
    alpha_index     = get_alpha_index(block, mode, x, y);
    alpha_bitcount  = get_alpha_bitcount(block, mode);
    color_index     = get_color_index(block, mode, x, y);
    color_bitcount  = get_color_bitcount(block, mode);

    //determine output
    UINT8 output[4];
    output.rgb = interpolate(endpoint_start.rgb, endpoint_end.rgb, color_index, color_bitcount);
    output.a   = interpolate(endpoint_start.a,   endpoint_end.a,   alpha_index, alpha_bitcount);

    if (mode.type == 4 OR == 5)
    {
```

```
        //Decode the 2 color rotation bits as follows:
        // 00 - Block format is Scalar(A) Vector(RGB) - no swapping
        // 01 - Block format is Scalar(R) Vector(AGB) - swap A and R
        // 10 - Block format is Scalar(G) Vector(RAB) - swap A and G
        // 11 - Block format is Scalar(B) Vector(RGA) - swap A and B
        rotation = extract_rot_bits(mode, block);
        output = swap_channels(output, rotation);
    }

}
```

19.5.14.3 BC7 Endpoint Decoding, Value Interpolation, Index Extraction, and Bitcount Extraction

The pseudocode below outlines the steps to fully decode endpoint color and alpha for each subset given the 16-byte BC7 block.

```
fully_decode_endpoints(endpoint_array, mode, block)
{
    //first handle modes that have P-bits
    if (mode.type == 0 OR == 1 OR == 3 OR == 6 OR == 7)
    {
        for each endpoint i
        {
            //component-wise left-shift
            endpoint_array[i].rgba = endpoint_array[i].rgba << 1;
        }

        //if P-bit is shared
        if (mode.type == 1)
        {
            pbit_zero = extract_pbit_zero(mode, block);
            pbit_one = extract_pbit_one(mode, block);

            //rgb component-wise insert pbits
            endpoint_array[0].rgb |= pbit_zero;
            endpoint_array[1].rgb |= pbit_zero;
            endpoint_array[2].rgb |= pbit_one;
            endpoint_array[3].rgb |= pbit_one;
        }
        else //unique P-bit per endpoint
        {
            pbit_array = extract_pbit_array(mode, block);
            for each endpoint i
            {
                endpoint_array[i].rgba |= pbit_array[i];
            }
        }
    }

    for each endpoint i
    {
        // Color_component_precision & alpha_component_precision includes pbit
        // left shift endpoint components so that their MSB lies in bit 7
        endpoint_array[i].rgb = endpoint_array[i].rgb << (8 - color_component_precision(mode));
        endpoint_array[i].a = endpoint_array[i].a << (8 - alpha_component_precision(mode));

        // Replicate each component's MSB into the LSBs revealed by the left-shift operation above
        endpoint_array[i].rgb = endpoint_array[i].rgb | (endpoint_array[i].rgb >> color_component_precision(mode));
        endpoint_array[i].a = endpoint_array[i].a | (endpoint_array[i].a >> alpha_component_precision(mode));
    }

    //If this mode does not explicitly define the alpha component
    //set alpha equal to 1.0
    if (mode.type == 0 OR == 1 OR == 2 OR == 3)
    {
        for each endpoint i
        {
            endpoint_array[i].a = 255; //i.e. alpha = 1.0f
        }
    }
}
```

In order to generate each interpolated component for each subset the following algorithm is used: Let "c" be the component being generated, "e0" be that component of endpoint 0 of the subset, "e1" be that component of endpoint 1 of the subset.

```
UINT16 aWeight2[] = {0, 21, 43, 64};
UINT16 aWeight3[] = {0, 9, 18, 27, 37, 46, 55, 64};
UINT16 aWeight4[] = {0, 4, 9, 13, 17, 21, 26, 30, 34, 38, 43, 47, 51, 55, 60, 64};

UINT8 interpolate(UINT8 e0, UINT8 e1, UINT8 index, UINT8 indexprecision)
{
    if(indexprecision == 2)
        return (UINT8) (((64 - aWeights2[index])*UINT16(e0) + aWeights2[index]*UINT16(e1) + 32) >> 6);
    else if(indexprecision == 3)
        return (UINT8) (((64 - aWeights3[index])*UINT16(e0) + aWeights3[index]*UINT16(e1) + 32) >> 6);
    else // indexprecision == 4
        return (UINT8) (((64 - aWeights4[index])*UINT16(e0) + aWeights4[index]*UINT16(e1) + 32) >> 6);
}
```

The following pseudocode illustrates how to extract indices and bitcounts for color and alpha components. Blocks with separate color and alpha also have two sets of index data – one for the vector channel and one for the scalar channel. For Mode 4, these indices are of differing widths (3 or 2 bits) and there is a one-bit selector which chooses whether the vector or scalar data uses the 3-bit indices. (Extracting the alpha bitcount is similar to extracting color bitcount but with inverse behavior based on the idxMode bit.)

```
bitcount get_color_bitcount(block, mode)
{
    if (mode.type == 0 OR == 1)
        return 3;

    if (mode.type == 2 OR == 3 OR == 5 OR == 7)
        return 2;

    if (mode.type == 6)
```

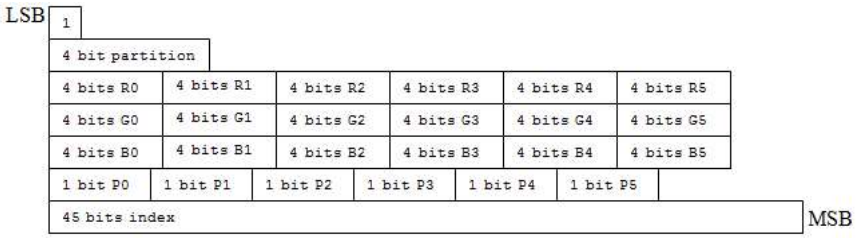
```
        return 4;

    //Only remaining case is Mode 4 with 1-bit index selector
    idxMode = extract_idxMode(block);
    if (idxMode == 0)
        return 2;
    else
        return 3;
}
```

19.5.14.4 Per-Block Memory Encoding of BC7

Below is a list of the 8 block modes and bit allocations for the 8 possible BC7 blocks. The colors for each subset within a block are represented using two explicit endpoint colors and a set of interpolated colors between them. Depending on the block's index precision, each subset may have 4, 8 or 16 possible colors.

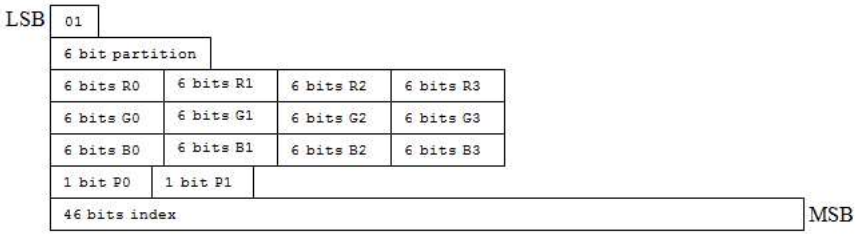
19.5.14.4.1 Mode 0

- Color Only
- 3 Subsets
- R.G.B 4.4.4.1 endpoints (unique P-bit per endpoint)
- 3-bit indices
- 16 partitions



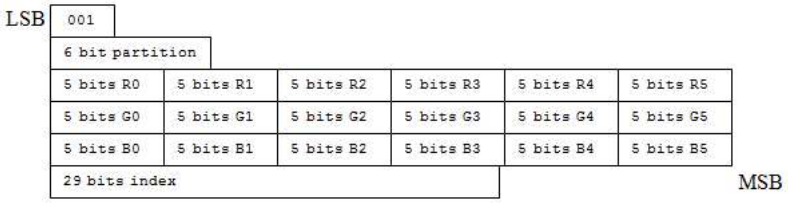
19.5.14.4.2 Mode 1

- Color Only
- 2 Subsets
- R.G.B.P 6.6.6.1 endpoints (shared P-bit per subset)
- 3-bit indices
- 64 partitions



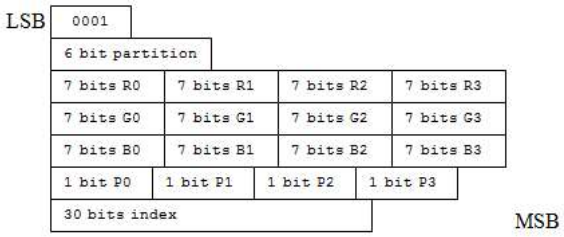
19.5.14.4.3 Mode 2

- Color Only
- 3 Subsets
- R.G.B 5.5.5 endpoints
- 2-bit indices
- 64 partitions



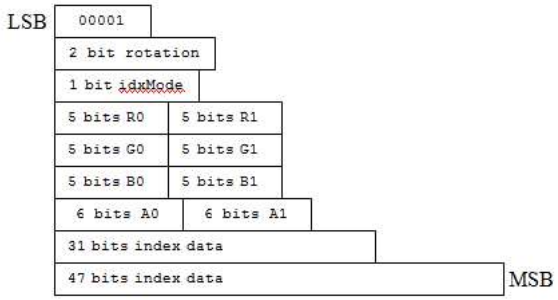
19.5.14.4.4 Mode 3

- Color Only
- 2 Subsets
- R.G.B.P. 7.7.7.1 Endpoints (unique P-bit per endpoint)
- 2-bit indices
- 64 partitions



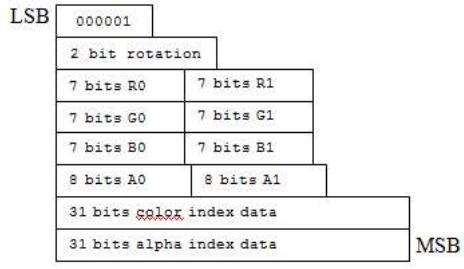
19.5.14.4.5 Mode 4

- Color with Separate Alpha
- One Subset
- R.G.B 5.5.5 Color endpoints
- 6-bit Alpha endpoints
- 16x2-bit indices
- 16x3-bit indice
- 2-bit component rotation
- 1-bit index selector



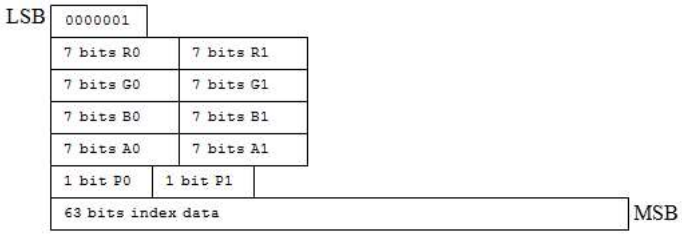
19.5.14.4.6 Mode 5

- Color with Separate Alpha
- One Subset
- R.G.B 7.7.7 Color endpoints
- 8-bit Alpha endpoints
- 16x2-bit color indices
- 16x2-bit alpha indices
- 2-bit component rotation



19.5.14.4.7 Mode 6

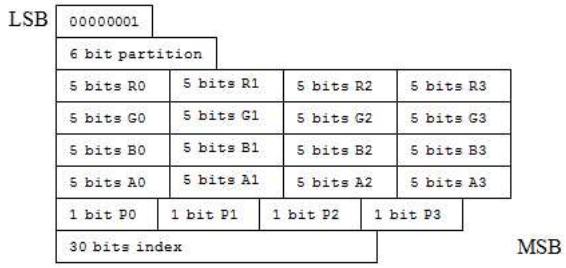
- Combined Color and Alpha
- One Subset
- R.G.B.A.P 7.7.7.7.1 endpoints (unique P bit per endpoint)
- 16x4-bit indices



19.5.14.4.8 Mode 7

- Combined Color and Alpha
- 2 Subsets
- R.G.B.A.P. 5.5.5.5.1 Endpoints (unique P-bit per endpoint)
- 2-bit indices

- 64 partitions



Mode 8 (LSB 0x00) is reserved and should not be used by the encoder. If this mode is given to the hardware, an all 0 block will be returned.

As previously discussed, in some block modes the endpoint representation is encoded in a form called RGBP – in these cases the P bit represents a shared LSB for the components of the endpoint. For example, if the endpoint representation for the format was RGBP 5.5.5.1 then the endpoint would be interpreted as an RGB 6.6.6 value, with the LSB of each component being taken from the state of the P bit. If the representation was RGBAP 5.5.5.5.1 then the endpoint would be interpreted as an RGBA 6.6.6.6 value. Depending on the block mode the shared LSB may either be specified for both endpoints of a subset individually (2 P-bits per subset), or shared between the endpoints of the subset (1 P-bit per subset)

In BC7, alpha can be encoded in several different ways:

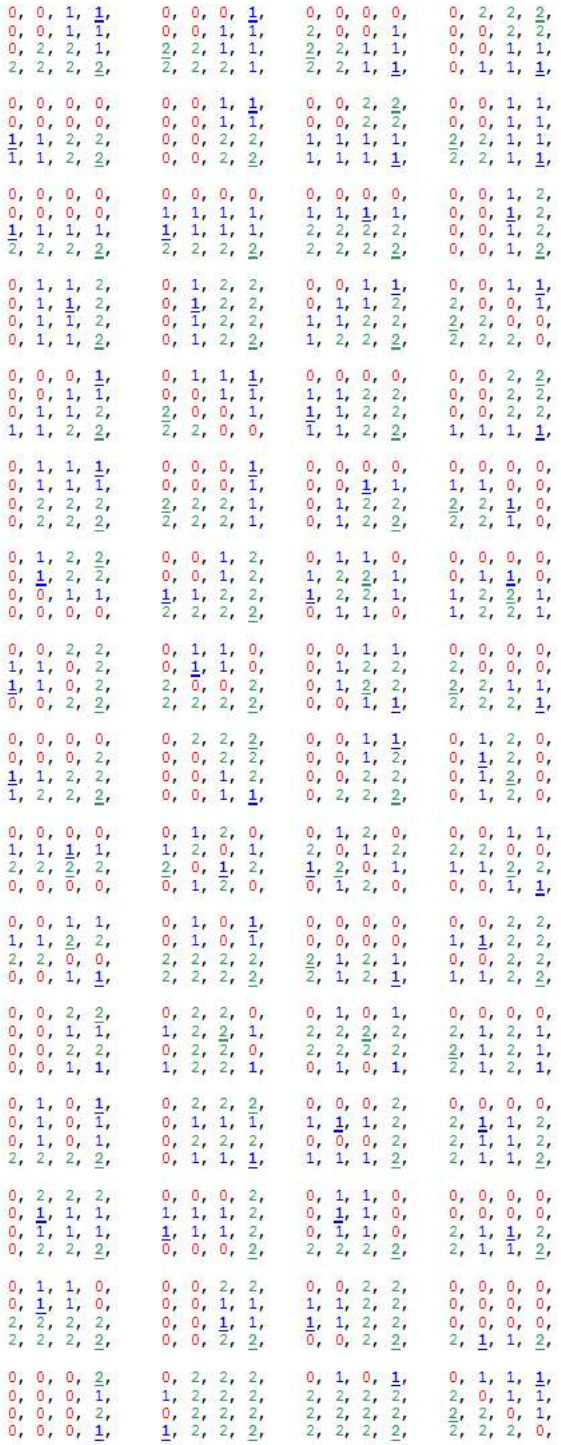
- **Block types without explicit alpha encoding:** In these blocks the endpoints have an R.G.B-only representation and alpha is decoded as 1.0 for all texels.
- **Block types with combined color and alpha:** In these blocks the endpoint color values are specified in an R.G.B.A format, and alpha values are interpolated along with the color values.
- **Block types with separated color and alpha:** In these blocks the alpha values and color values are specified separately, each with their own sets of indices. These blocks effectively have a vector channel (R.G.B) and a scalar channel (A) separately encoded. In these blocks a separate 2-bit field is also encoded that allows specification on a per-block basis of the channel that is encoded separately as a scalar, so the block can have 4 different representations – RGB|A, AGB|R, RAB|G, RGA|B. The channel order is swizzled back to RGBA after decoding, so the internal block format is transparent to the developer. Blocks with separate color and alpha also have two sets of index data – one for the vector channel and one for the scalar channel. In the case of Mode 4, these indices are of differing widths (3 or 2 bits). Mode 4 contains is a one-bit selector which chooses whether the vector or scalar data uses the 3-bit indices.

19.5.14.5 BC7 Partition Set for 2 Subsets

0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, **1**,	0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, **1**,	0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, **1**,	0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, **1**,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, **1**,	0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, **1**,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, **1**,	0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, **1**,
0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, **1**,	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, **1**,
0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, **1**,	0, 1, **1**, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,	0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 1, 1, 1, 0,	0, 1, **1**, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, **1**, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,	0, 0, 0, 0, 1, 0, 0, 0, **1**, 1, 0, 0, 1, 1, 1, 0,	0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 1, 1, 0, 0,	0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, **1**,
0, 0, **1**, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,	0, 0, 0, 0, 1, 0, 0, 0, **1**, 0, 0, 0, 1, 1, 0, 0,	0, 1, **1**, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0,	0, 0, **1**, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0,
0, 0, 0, 1, 0, 1, 1, 1, **1**, 1, 1, 0, 1, 0, 0, 0,	0, 0, 0, 0, 1, 1, 1, 1, **1**, 1, 1, 1, 0, 0, 0, 0,	0, 1, **1**, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0,	0, 0, **1**, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, **1**,	0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, **1**,	0, 1, 0, 1, 1, 0, **1**, 0, 0, 1, 0, 1, 1, 0, 1, 0,	0, 0, 1, 1, 0, 0, 1, 1, **1**, 1, 0, 0, 1, 1, 0, 0,
0, 0, **1**, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,	0, 1, 0, 1, 0, 1, 0, 1, **1**, 0, 1, 0, 1, 0, 1, 0,	0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, **1**,	0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, **1**,
0, 1, **1**, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0,	0, 0, 0, 1, 0, 0, 1, 1, **1**, 1, 0, 0, 1, 0, 0, 0,	0, 0, **1**, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0,	0, 0, **1**, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0,
0, 1, **1**, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,	0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, **1**,	0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, **1**,	0, 0, 0, 0, 0, 1, **1**, 0, 0, 1, 1, 0, 0, 0, 0, 0,
0, 1, 0, 0, 1, 1, **1**, 0, 0, 1, 0, 0, 0, 0, 0, 0,	0, 0, **1**, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0,	0, 0, 0, 0, 0, 0, **1**, 0, 0, 1, 1, 1, 0, 0, 1, 0,	0, 0, 0, 0, 0, 1, 0, 0, **1**, 1, 1, 0, 0, 1, 0, 0,
0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, **1**,	0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, **1**,	0, 1, **1**, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0,	0, 0, **1**, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0,
0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, **1**,	0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, **1**,	0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, **1**,	0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, **1**,
0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, **1**,	0, 0, **1**, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,	0, 0, **1**, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,	0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, **1**,

In the table of partitions above, the bolded, underlined entry is the location of the fix-up index for subset 1 which is specified with one less bit. The fix-up index for subset 0 is always index 0 (the partitioning is arranged so that index 0 is always in subset 0). Partition order goes from top-left to bottom right, walking left-to-right, then top-to-bottom.

19.5.14.6 BC7 Partition Set for 3 Subsets



For this table of partitions, underneath the entry in each subset, printed in bold and underlined, is the location of the fix-up index which is specified with one less bit. Index 0 always contains the fixed index bit for subset 0. Partition order goes from top-left to bottom right, walking left-to-right, then top-to-bottom.

19.6 Resurrected 16-Bit Formats From D3D9

Three formats were added back to D3D11 which all existing GPUs support:

1) DXGI_FORMAT_B5G6R5_UNORM
2) DXGI_FORMAT_B5G5R5A1_UNORM
3) DXGI_FORMAT_B4G4R4A4_UNORM

Required support for these formats depending on the hardware feature level:

Capability	Feature Level 9_x	Feature Level 10.0	Feature Level 10.1	Feature Level 11+
Typed Buffer	no	optional	optional	optional
Input Assembler Vertex Buffer	no	optional	optional	optional
Texture1D	no	req	req	req

Texture2D	req	req	req	req
Texture3D	no	req	req	req
TextureCube	req	req	req	req
Shader ld*	yes (point sample)	req	req	req
Shader sample* (with filtering)	req	req	req	req
Shader gather4	no	no	no	req
Mipmap	req	req	req	req
Mipmap Auto-Generation	req for 565, no for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551
RenderTarget	req for 565, no for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551
Blendable RenderTarget	req for 565, no for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551
UAV Typed Store	no	no	no	optional
CPU Lockable	req	req	req	req
4x MSAA	optional	optional	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551
8x MSAA	optional	optional	optional	req for 565, opt for 4444, 5551
Other MSAA Sample Count	optional	optional	optional	optional
Multisample Resolve	req (if MSAA supported) for 565, no for 4444, 5551	req (if MSAA supported) for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551
Multisample Load	no	req (if MSAA supported) for 565, opt for 4444, 5551)	req for 565, opt for 4444, 5551	req for 565, opt for 4444, 5551

19.7 ASTC Formats

TODO

20 Asynchronous Notification

Chapter Contents

(back to top)

20.1 Pipeline statistics
20.2 Predicated Primitive Rendering
20.3 Query Manipulation
20.4 Query Type Descriptions
20.5 Performance Monitoring and Counters

Summary of Changes in this Chapter from D3D10 to D3D11.3