Intel® Open Source HD Graphics Programmers' Reference Manual (PRM)

Volume 5: Memory Views

For the 2014-2015 Intel Atom™ Processors, Celeron™ Processors and Pentium™ Processors based on the "Cherry Trail/Braswell" Platform (Cherryview/Braswell graphics)

June 2015, Revision 1.0

Creative Commons License

Notices and Disclaimers

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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.

The products described in this document 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.

Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Table of Contents

Introduction

The hardware supports three engines:

- The Render command streamer interfaces to 3D/IE and display streams.
- The Media command streamer interfaces to the fixed function media.
- The Blitter command streamer interfaces to the blit commands.

Software interfaces of all three engines are very similar and should only differ on engine-specific functionality.

Memory Views Glossary

Term	Definition
CHV, BSW	CherryView CPU/GFX platform. 8th generation processor graphics (Gen8).
IOMMU	I/O Memory Mapping unit
SVM	Shared Virtual Memory, implies the same virtual memory view between the IA cores and processor graphics.
Page Walker (GAM)	GFX page walker which handles page level translations between GFX virtual memory to physical memory domain.

GPU Memory Interface

GPU memory interface functions are divided into 4 different major sections:

- Global Arbitration
- Memory Interface Functions
- Page Translations (GFX Page Walker)
- Ring Interface Functions (GTI)

GT Interface functions are covered at a different chapter/HAS and not part of this documentation. The following documentation is meant for GFX arbitration paths in accessing to memory/cache interfaces and page translations and page walker functions.

Global Arbitration

The global memory arbitration fabric is meant to be a hierarchal memory fabric where memory accesses from different stages of the pipeline are consolidated to a single interface towards GT's connection to CPU's ring interface.

The arbitration on the fabric is programmable via a simple per pipeline stage priority levels.

Programming Note	
Context:	Global Memory Arbitration
Gen9 arbitration allows 4 levels of arbitration where each pipeline level can be put into these 4 levels. Each consolidation stage simply follows the 4-level arbitration with grace periods to allow ahead of the pipeline to get a higher share of the memory bandwidth.	

The final arbitration takes places in GAM between parallel compute engines. Each engine (in some cases major pipeline stages are also separated, i.e. Z vs Color vs L3 vs Fixed Functions) gets a count in a grace period where its accesses are counted against a global pool. If a particular engine (or pipeline stage) exhausts its max allowed, it is dropped to a lower priority and goes to fixed pipeline based prioritization. Once all counts are expired, the grace period completes and resets.

The count values are programmable via MMIO (i.e. *_MAX_REQ_COUNT) registers with defaults favoring the pipeline order.

Graphics Memory Interface Functions

The major role of an integrated graphics device's Memory Interface (MI) function is to provide various client functions access to "graphics" memory used to store commands, surfaces, and other information used by the graphics device. This chapter describes the basic mechanisms and paths by which graphics memory is accessed.

Information not presented in this chapter includes:

- Microarchitectural and implementation-dependent features (e.g., internal buffering, caching, and arbitration policies).
- MI functions and paths specific to the operation of external (discrete) devices attached via external connections.
- MI functions essentially unrelated to the operation of the internal graphics devices, .e.g., traditional "chipset functions"
- GFX Page Walker and GT interface functions are covered in different chapters.

Graphics Memory Clients

The MI function provides memory access functionality to a number of external and internal graphics memory *clients*, as described in the table below.

Graphics Memory Clients

MI Client	Access Modes
Host Processor	Read/Write of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses
External PEG Graphics Device	Write-Only of Graphics Operands located in Main Memory via the Graphics Aperture. (This client is not described in this chapter).
Peer PCI Device	Write-Only of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses (i.e., mapped by GTT). Note that DMI access to Graphics registers is not supported.
Coherent Read/Write (internal)	Internally-generated snooped reads/writes.
Command Stream (internal)	DMA Read of graphics commands and related graphics data.
Vertex Stream (internal)	DMA Read of indexed vertex data from Vertex Buffers by the 3D Vertex Fetch (VF) Fixed Function.
Instruction/State Cache (internal)	Read of pipelined 3D rendering state used by the 3D/Media Functions and instructions executed by the EUs.
Render Cache (internal)	Read/Write of graphics data operated upon by the graphics rendering engines (Blt, 3D, MPEG, etc.) Read of render surface state.
Sampler Cache (internal)	Read of texture (and other sampled surface) data stored in graphics memory.
Display/Overlay Engines (internal)	Read of display, overlay, cursor and VGA data.
Media Engines	Read and write of media content and media processing.
uController	Read/Write (DMA) functions for u-controller and scheduler.

Graphics Memory Addressing Overview

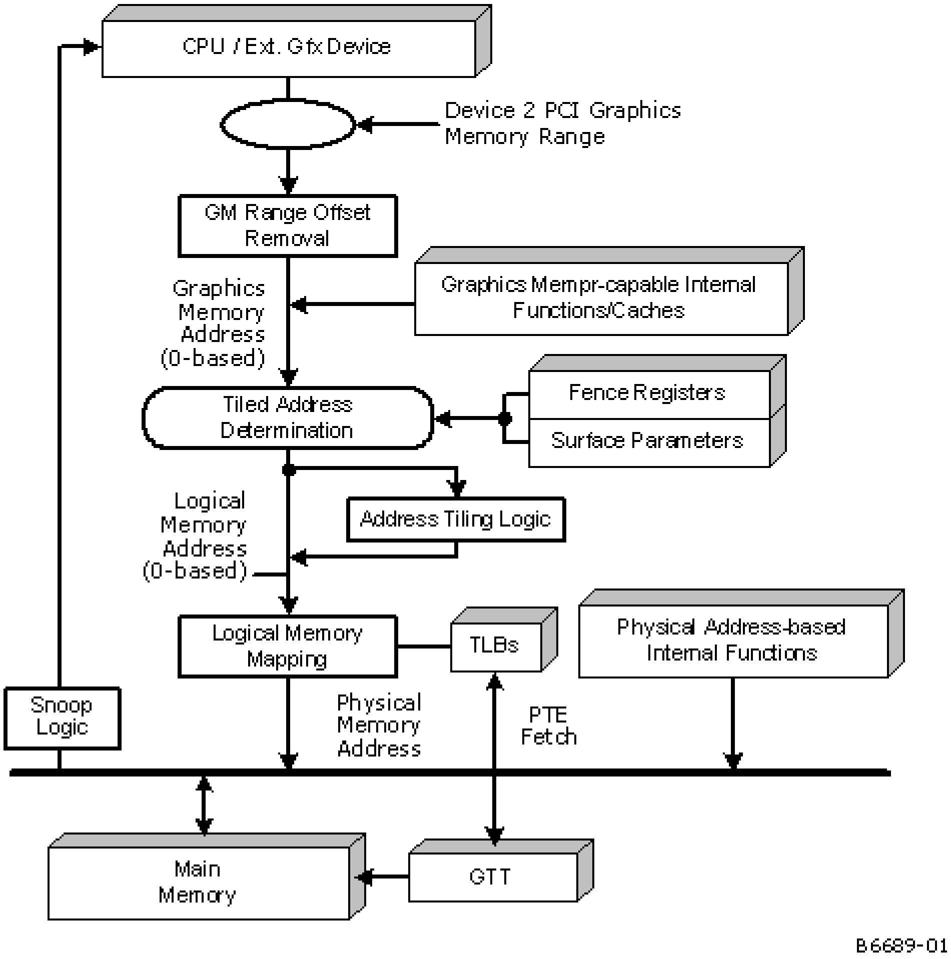
The Memory Interface function provides access to graphics memory (GM) clients. It accepts memory addresses of various types, performs a number of optional operations along *address paths*, and eventually performs reads and writes of graphics memory data using the resultant addresses. The remainder of this subsection will provide an overview of the graphics memory clients and address operations.

Graphics Address Path

Graphics Address Path shows the internal graphics memory address path, connection points, and optional operations performed on addresses. Externally-supplied addresses are normalized to zero-based *Graphics Memory* (*GM) addresses* (GM_Address). If the GM address is determined to be a tiled address (based on inclusion in a fenced region or via explicit surface parameters), *address tiling* is performed. At this point the address is considered a *Logical Memory address*, and is translated into a *Physical Memory address* via the GTT and associated TLBs. The physical memory location is then accessed.

CPU accesses to graphics memory are sent back on the ring to snoop. Hence pages that are mapped cacheable in the GTT will be coherent with the CPU cache if accessed through graphics memory aperture.

Graphics Memory Paths


CPU / Ext. Gfx Device
Device 2 PCI Graphics Memory Range
GM Range Offset Removal
Graphics Memory Address (0-based)
Graphics Mempr-capable Internal Functions/Caches
Tiled Address Determination
Fence Registers
Surface Parameters
Logical Memory Address (0-based)
Address Tiling Logic
Logical Memory Mapping
TLBs
Physical Address-based Internal Functions
Snoop Logic
Physical Memory Address
PTE Fetch
Main Memory
GTT
B6689-01

The remainder of this chapter describes the basic features of the graphics memory address pipeline, namely Address Tiling, Logical Address Mapping, and Physical Memory types and allocation considerations.

Graphics Memory Address Spaces

The *Graphics Memory Address Spaces* table lists the five supported Graphics Memory Address Spaces. Note that the Graphics Memory Range Removal function is automatically performed to transform system addresses to internal, zero-based Graphics Addresses.

Due to a workaround, first 4KB of DSM has to be reserved for GFX hardware use during render engine execution.

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.

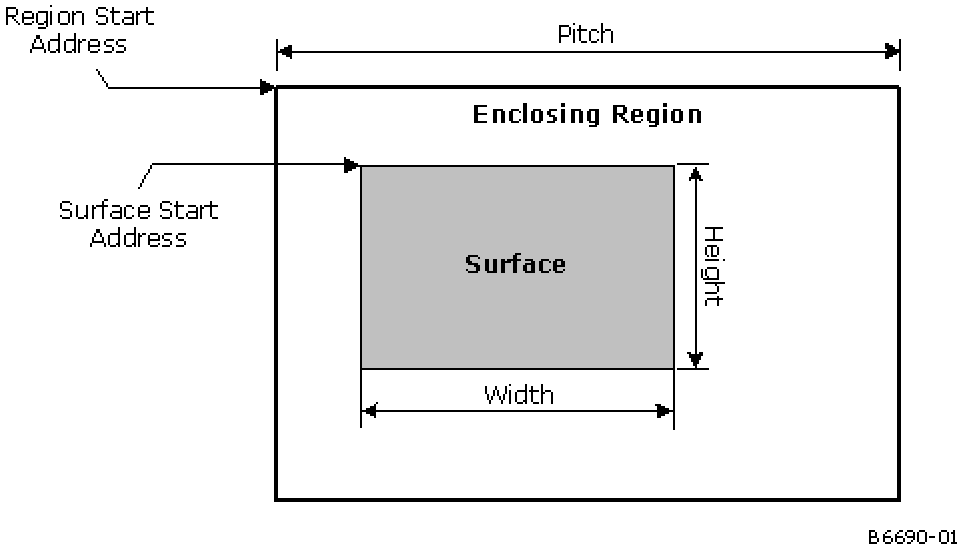
Legacy Tiling Modes:

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

Linear vs Tiled Storage

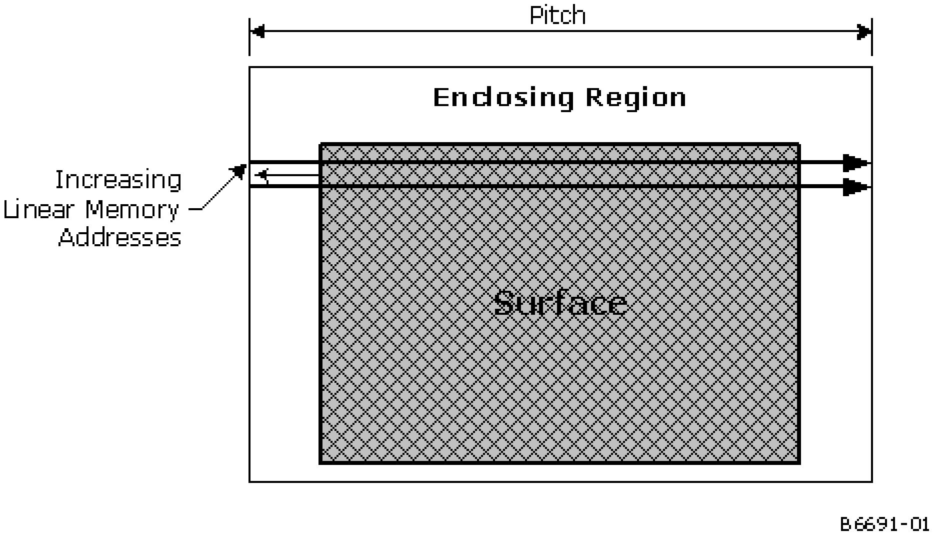
Regardless of the memory storage format, “rectangular” memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region 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

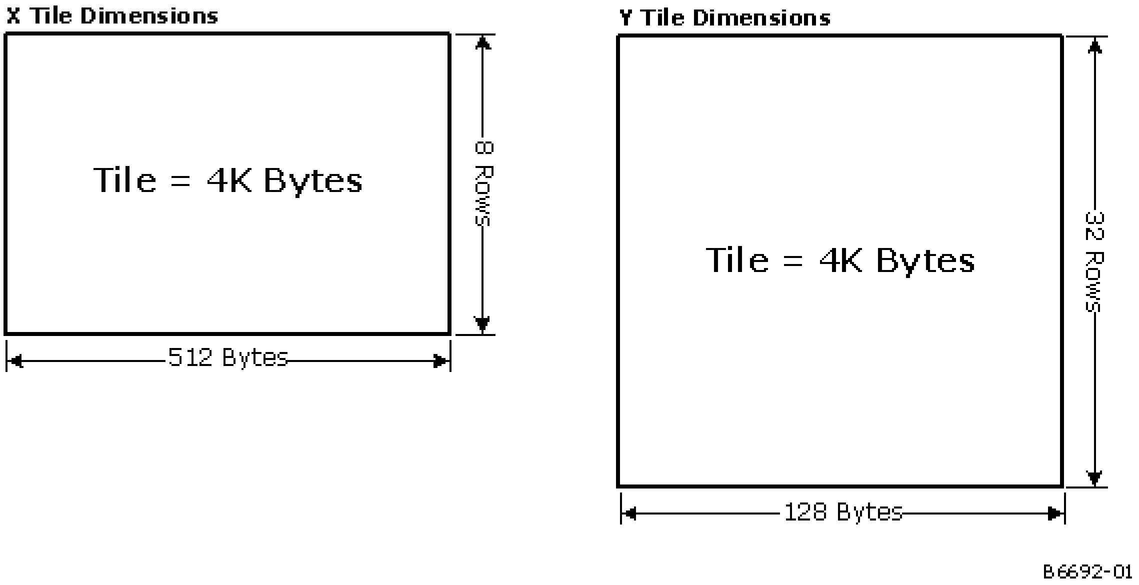


The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters 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).

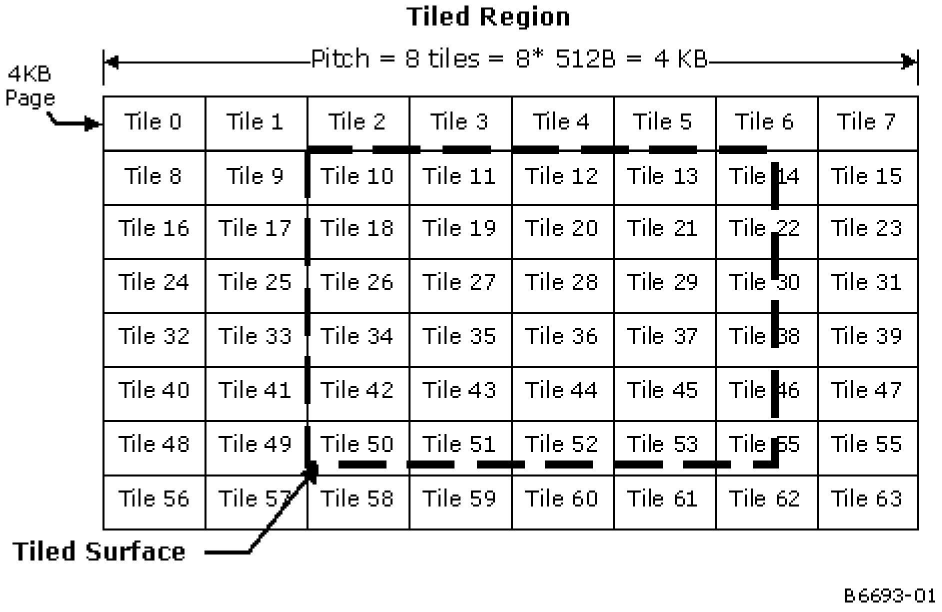
Memory Tile Dimensions



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


Tiled Region
Pitch = 8 tiles = 8* 512B = 4 KB
4KB Page
Tile 0 Tile 1 Tile 2 Tile 3 Tile 4 Tile 5 Tile 6 Tile 7
Tile 8 Tile 9 Tile 10 Tile 11 Tile 12 Tile 13 Tile 14 Tile 15
Tile 16 Tile 17 Tile 18 Tile 19 Tile 20 Tile 21 Tile 22 Tile 23
Tile 24 Tile 25 Tile 26 Tile 27 Tile 28 Tile 29 Tile 30 Tile 31
Tile 32 Tile 33 Tile 34 Tile 35 Tile 36 Tile 37 Tile 38 Tile 39
Tile 40 Tile 41 Tile 42 Tile 43 Tile 44 Tile 45 Tile 46 Tile 47
Tile 48 Tile 49 Tile 50 Tile 51 Tile 52 Tile 53 Tile 55 Tile 55
Tile 56 Tile 57 Tile 58 Tile 59 Tile 60 Tile 61 Tile 62 Tile 63
Tiled Surface
B6693-01

Tile Formats

Multiple tile formats are supported by the Gen Core. 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.

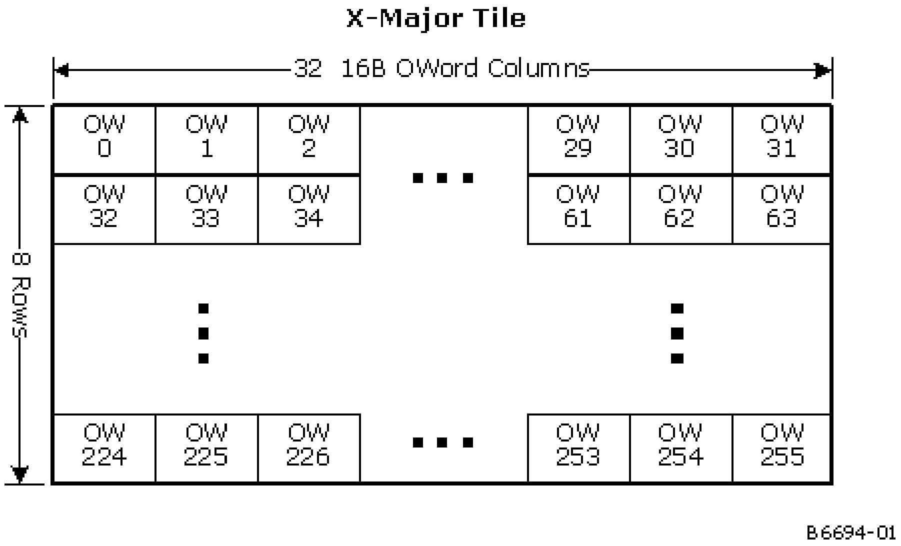
Tile-X Legacy Format

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



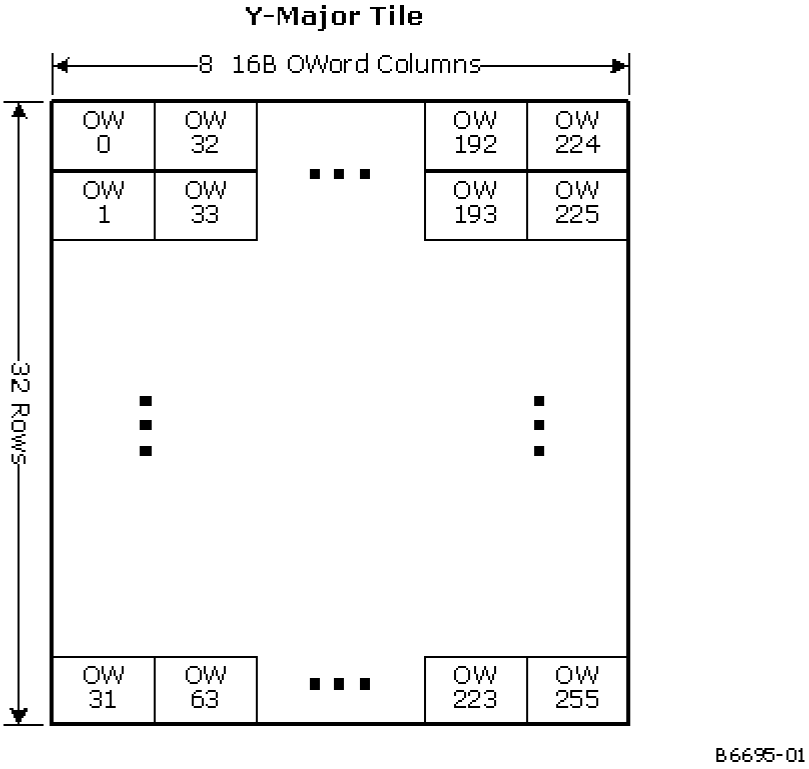
Tile-Y Legacy Format

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



Tiling Algorithm

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

```
 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.

Tiled Channel Select Decision

Before Gen8, 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].

For Gen8 and subsequent generations, 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.

Restriction: Each FENCE register (if its Fence Valid bit is set) defines a Graphics Memory region ranging from 4KB to the aperture size. The region is considered rectangular, with a pitch in tile widths from 1 tile width (128B or 512B) to 512 tile X widths (512 * 512B = 256KB) and 2048 tile Y widths (2048 * 128B = 256KB). Note that fenced regions must not overlap, or operation is UNDEFINED.

Context: Tiled (Fenced) Regions

Restriction: Also included in the FENCE register is a Tile Walk field that specifies which tile format applies to the fenced region.

Context: Tiled (Fenced) Regions

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

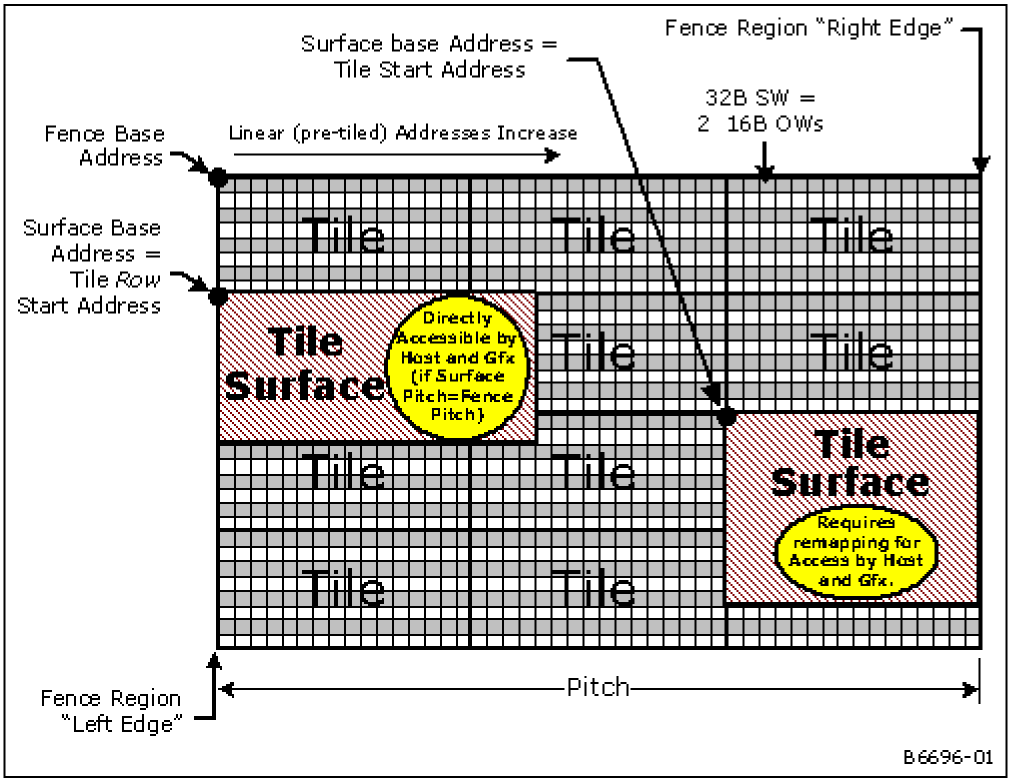
Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by 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>

Main Memory

The integrated graphics device is capable of using 4KB pages of physical main (system) memory for graphics functions. Some of this main memory can be "stolen" from the top of system memory during initialization (e.g., for a VGA buffer). However, most graphics operands are dynamically allocated to satisfy application demands. To this end the graphics driver will frequently need to allocate locked-down (i.e., non-swappable) physical system memory pages – typically from a cacheable non-paged pool. The locked pages required to back large surfaces are typically non-contiguous. Therefore a means to support "logically-contiguous" surfaces backed by discontiguous physical pages is required. The Graphics Translation Table (GTT) that was described in previous sections provides the means.

Optimizing Main Memory Allocation

This section includes information for software developers on how to allocate SDRAM Main Memory (SM) for optimal performance in certain configurations. The general idea is that these memories are divided into some number of page types, and careful arrangement of page types both within and between surfaces (e.g., between color and depth surfaces) will result in fewer page crossings and therefore yield somewhat higher performance.

The algorithm for allocating physical SDRAM Main Memory pages to logical graphics surfaces is somewhat complicated by (1) permutations of memory device technologies (which determine page sizes and therefore the number of pages per device row), (2) memory device row population options, and (3) limitations on the allocation of physical memory (as imposed by the OS).

However, the theory to optimize allocation by limiting page crossing penalties is simple: (a) switching between open pages is optimal (again, the pages do not need to be sequential), (b) switching between memory device rows does not in itself incur a penalty, and (c) switching between pages within a particular bank of a row incurs a page miss and should therefore be avoided.

Application of the Theory (Page Coloring)

This section provides some scenarios of how Main Memory page allocation can be optimized.

3D Color and Depth Buffers

Here we want to minimize the impact of page crossings (a) between corresponding pages (1-4 tiles) in the Color and Depth buffers, and (b) when moving from a page to a neighboring page within a Color or Depth buffer. Therefore corresponding pages in the Color and Depth Buffers, and adjacent pages within a Color or Depth Buffer should be mapped to different page types (where a page's "type" or "color" refers to the row and bank it's in).

Media/Video

The Y surfaces can be allocated using 4 page types in a similar fashion to the Color Buffer diagram. The U and V surfaces would split the same 4 page types as used in the Y surface.

Physical Graphics Address Types

The Physical Memory Address Types table lists the various *physical* address types supported by the integrated graphics device. Physical Graphics Addresses are either generated by Logical Memory mappings or are directly specified by graphics device functions. *These physical addresses are not subject to tiling or GTT re-mappings.*

Physical Memory Address Types

Address Type	Description	Range
MM_Address	Main Memory Address. Offset into physical, *unsnooped* Main Memory.	[0,TopOfMemory-1]
SM_Address	System Memory Address. Accesses are snooped in processor cache, allowing shared graphics/ processor access to (locked) cacheable memory data.	[0,512GB]

Graphics Translation Tables

The Graphics Translation Tables GTT (Graphics Translation Table, sometimes known as the global GTT) and PPGTT (Per-Process Graphics Translation Table) are memory-resident page tables containing an array of DWord Page Translation Entries (PTEs) used in mapping logical Graphics Memory addresses to physical memory addresses, and sometimes snooped system memory “PCI” addresses.

The base address (MM offset) of the GTT and the PPGTT are programmed via the PGTBL_CTL and PGTBL_CTL2 MI registers, respectively. The translation table base addresses must be 4KB aligned. The GTT size can be either 128KB, 256KB, or 512KB (mapping to 128MB, 256MB, and 512MB aperture sizes respectively) and is physically contiguous. The global GTT should only be programmed via the range defined by GTTMMADR. The PPGTT is programmed directly in memory. The per-process GTT (PPGTT) size is controlled by the PGTBL_CTL2 register. The PPGTT can, in addition to the above sizes, also be 64KB in size (corresponding to a 64MB aperture). Refer to the GTT Range chapter for a bit definition of the PTE entries.

Virtual Memory

This section describes the different paging models, their behaviors, and the page table formats.

GFX Page Tables

GPU supports three page table mechanisms

- PPGTT – per process GTT (private GFX)
- GGTT- global GTT

All page tables have the same PTE format, the difference was how to reach the final physical page and which fields with PTE are used.

Page Table Modes

The GFX Aperture and Display accesses are always mapped thru Global GTT. This is done to keep the walk simple (i.e. 1-level), however GT accesses to memory can be mapped via Global GTT and/or ppGTT with various addressing modes.

The walk modes are listed as following:

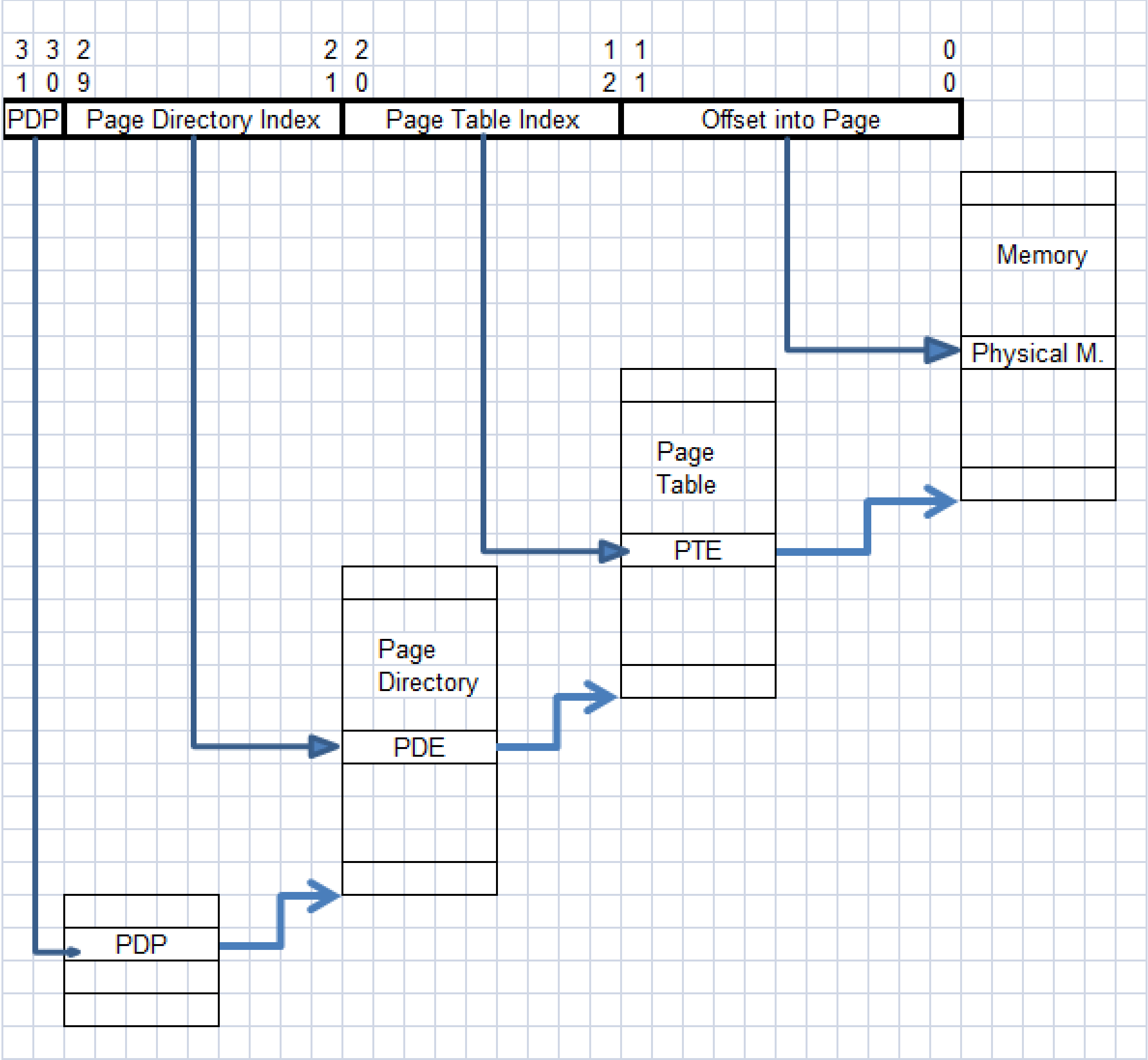
1. **Global GTT with 32b virtual addressing**: Global GTT usage is similar to pre-CHV, BSW behavior with extended capability to increase the VA to 4GB (from 2GB) and use a similar 64b PTE as ppGTT. The breakdown of the PTE for global GTT is given in later sections but fundamentally allows 1-level pagewalk where the 20b index is used to select the 64b PTE from stolen memory.
2. **Legacy 32b VA with ppGTT**: This is a mode where ppGTT page tables are managed via GFX s/w (driver) and context is tagged as Legacy 32b VA. Given each page walk is managed via 9b of the virtual address, 20b index is broken into 3 parts. However to optimize the walks and make it look like pre-CHV, BSW, s/w provides 4 pointers to page tables (called 4 PDP entries) – GPA. GFX h/w uses the four pointers and fetches the 4x4KB into h/w (for render and media) before the context execution starts. The optimization limits the dynamic (on demand) page walks to 1-level only.

Per Process GTT

Per process GTT mechanism has multiple hooks and mechanisms for s/w to prepare the page walks on hardware. The listed mechanisms here are selectable per-context and descriptors are delivered to hardware as part of context descriptor.

PPGTT for 32b Virtual Address

This page walk mechanism will be used for traditional 3D, Media type context. There is going to be a descriptor in the context header which will define the per process GTT walk that is required. For the standard context with 32bit virtual addressing, there is a possibility to take short cuts to reduce the overhead of the walk.



With 32-bit addressing the only entries that are needed for page directory pointers are 4x64bit locations (PDPE). For any standard context scheduling, it is required for s/w to provide 4 PDPEs as part of the context which would prevent h/w to do additional walks.

Hardware will do the remaining walks for PD and PTE similar to legacy behavior. In order to reduce the overhead of walks, hardware implements large caches for PDs.

Hardware does the remaining walks for PD and PTE similar to legacy behavior. To reduce the overhead of walks, hardware implements large caches for PDs:

- 4x4KB for 3D context
- 2x4x4KB for Media Context
- 4KB for VEBOX
- 4KB for Blitter

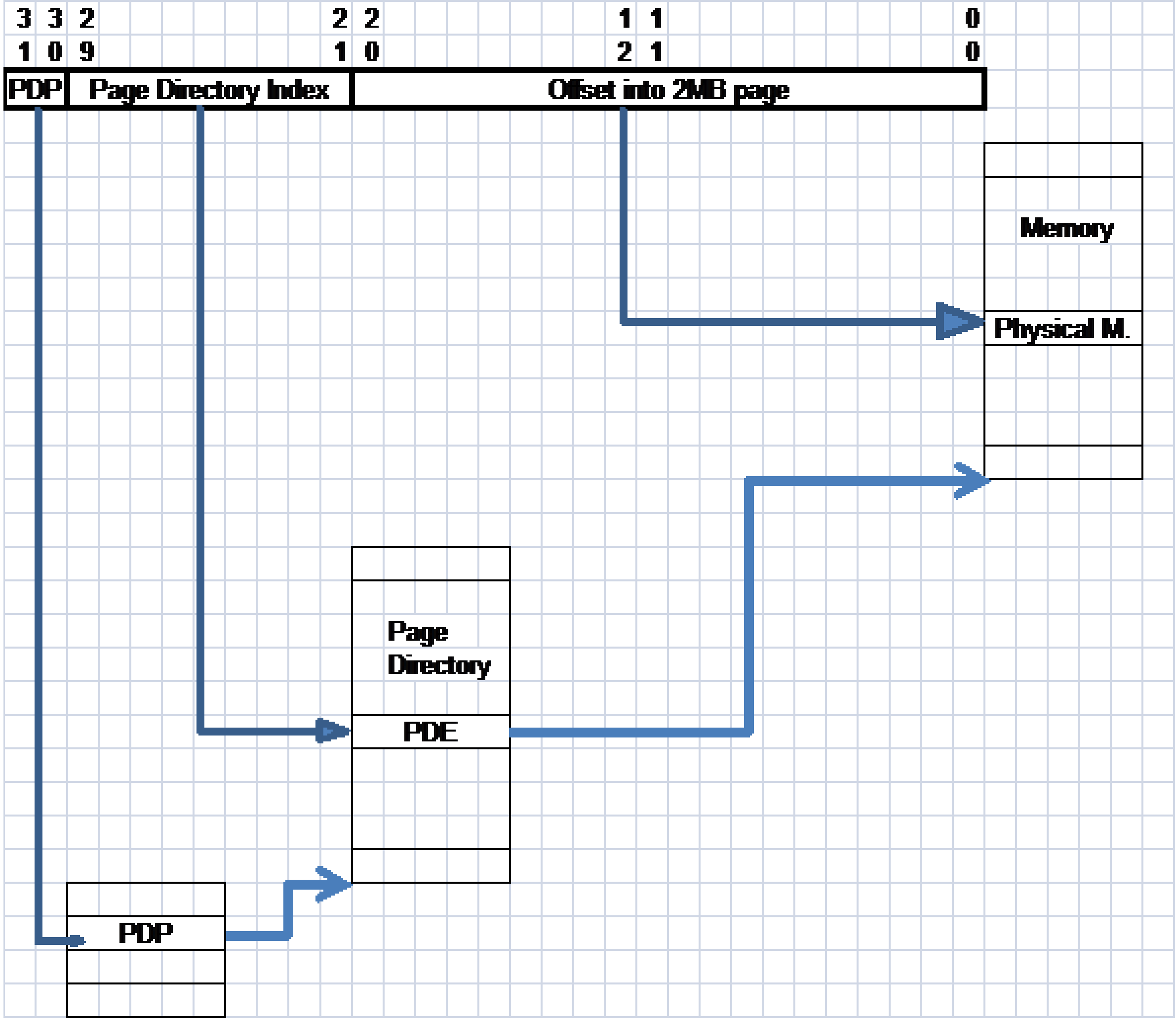
For Media and 3D context, the 16KB caches are preloaded for the entire page directory set up which limits the walk to 1-level before the final access. For remaining clients the PD cache is loaded on demand and can contain up to 512 entries.

Walk with 64KB Page

64KB Page size has a slightly different usage for how PTEs are selected for the corresponding 64KB page. In page table every 16th entry (PTE#0, PTE#16, PTE#32....PTE#496) should be used to index. This is calculated using address[21:16]& "0000". Note that hardware should not make any assumptions for any other PTEs.

Walk with 2MB Page

There is an option in the page walk to work with bigger page sizes, one of those sizes is 2MB pages. If allocated the page directory entry will indicate the page size and walk can be shortened as follows:

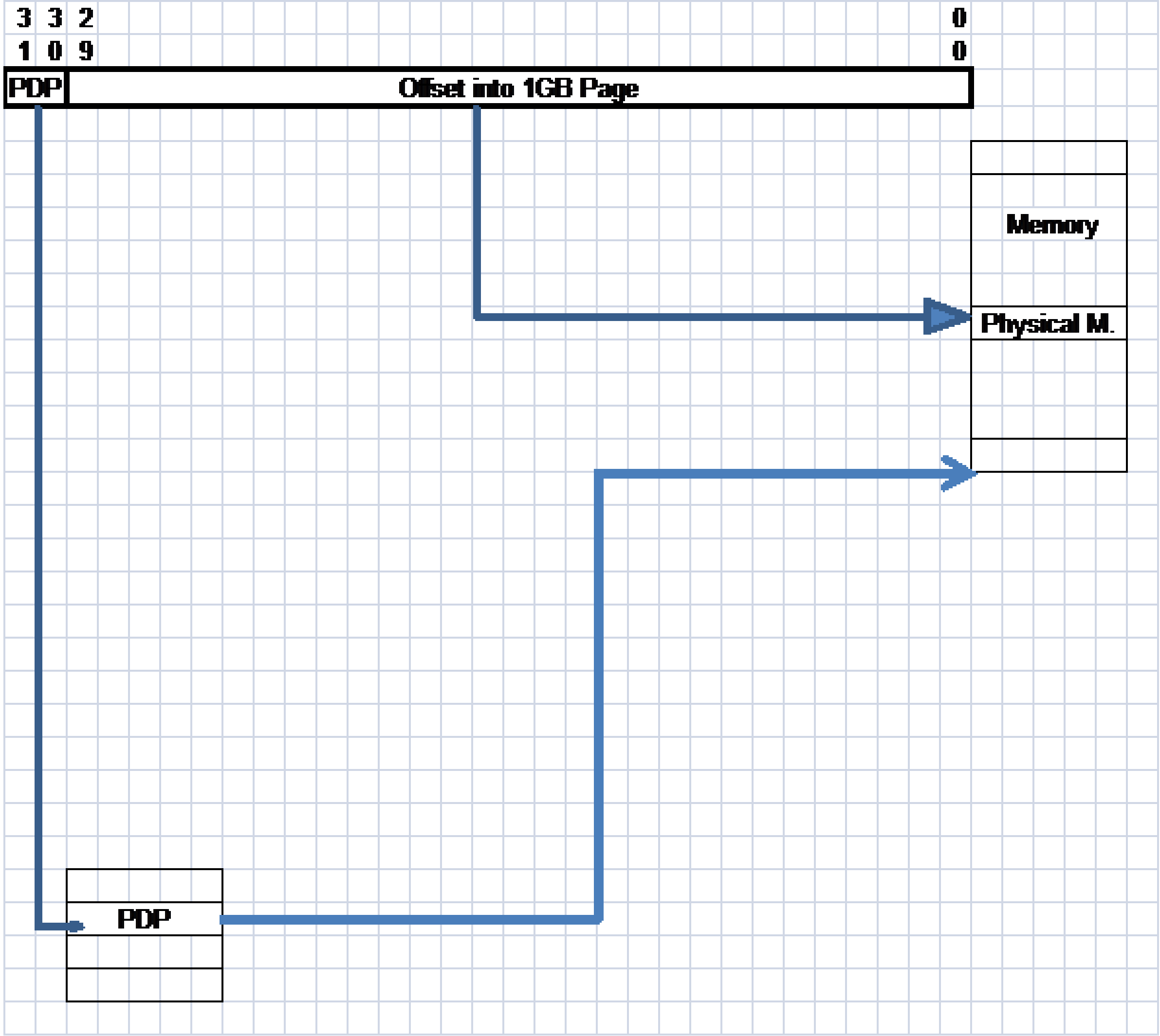


In this case there is no need to walk the page table after directory. And page directory has a pointer to 2MB range is physical memory.

Programming Note	
Context:	Walk with 2MB Page.
PPGTT32 is not going to support 2MB pages.	

Walk with 1GB Page

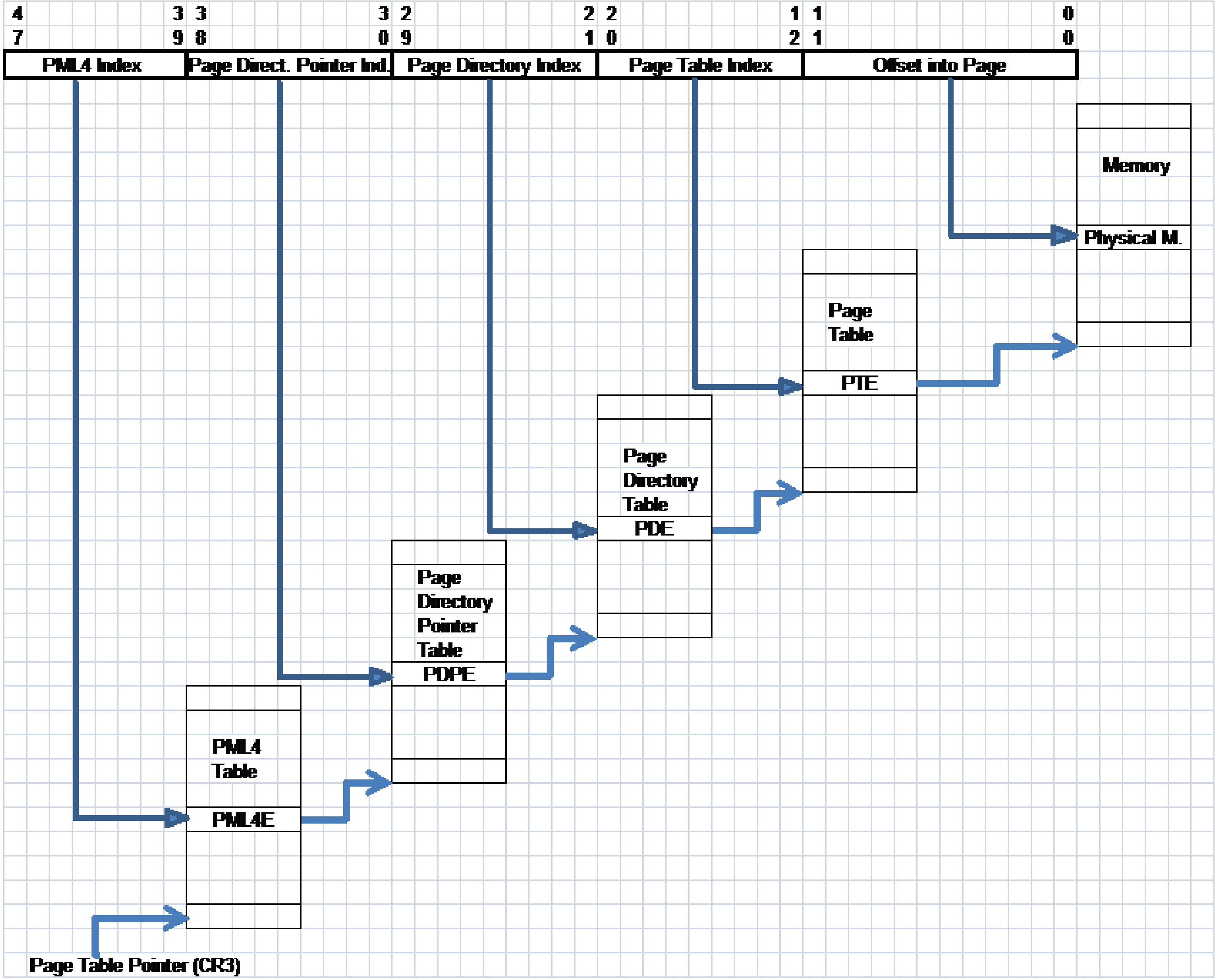
The same page walk is possible with 1GB page support as well.



Programming Note	
Context:	Walk with 1BG Page.
PPGTT32 is not going to support 1GB pages	

PPGTT for Standard Context (64b VA)

For advanced virtual addressing with legacy context, the full page walk mechanism needs to be exercised based on 48-bit canonical addressing.



64-bit (48b canonical) address requires 4-levels of page table format where the context carries a pointer to highest level page table (PML4 pointer or CR3). The rest of the walk is normal page walk thru various levels.

To repurpose the caches the following mechanism will be used:

- 3D: 4KB to store PML4, 4KB as PDP cache, 2x4PD cache.
- Media: 4KB to store PML4, 4KB as PDP cache, 2x4PD cache.
- VEBOX, Blitter: each with 4KB acting as PML4, PDP, PD cache.

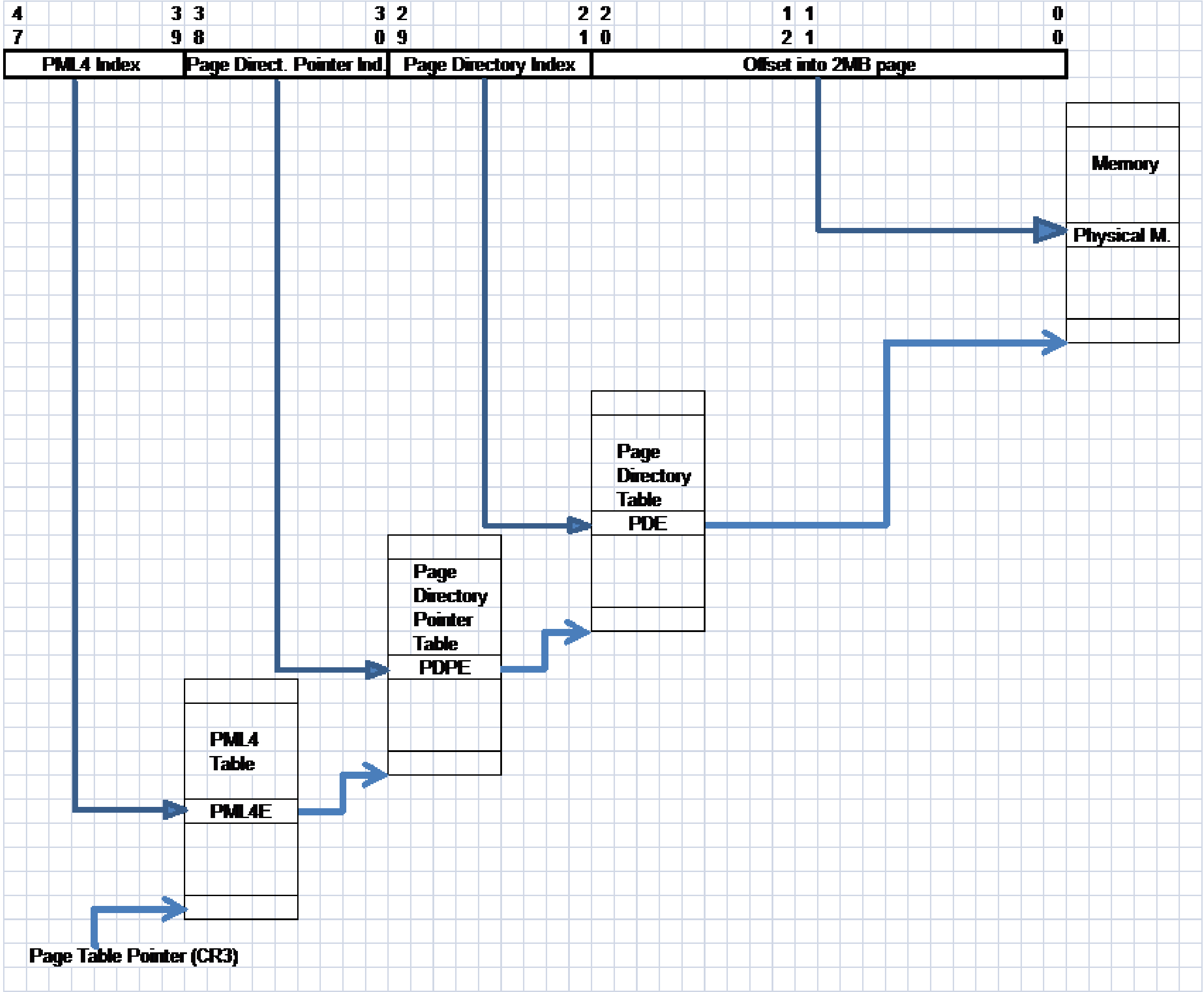
Programming Note	
Context:	PPGTT for Standard Context (64b VA)
Design can section the 512 entries within 4KB to separate areas for PML4, PDP, and PD.	

Walk with 64KB Page

64KB Page size has a slightly different usage for how PTEs are selected for the corresponding 64KB page. In page table every 16th entry (PTE#0, PTE#16, PTE#32....PTE#496) should be used to index. This is calculated using address[21:16]& "0000". Note that hardware should not make any assumptions for any other PTEs.

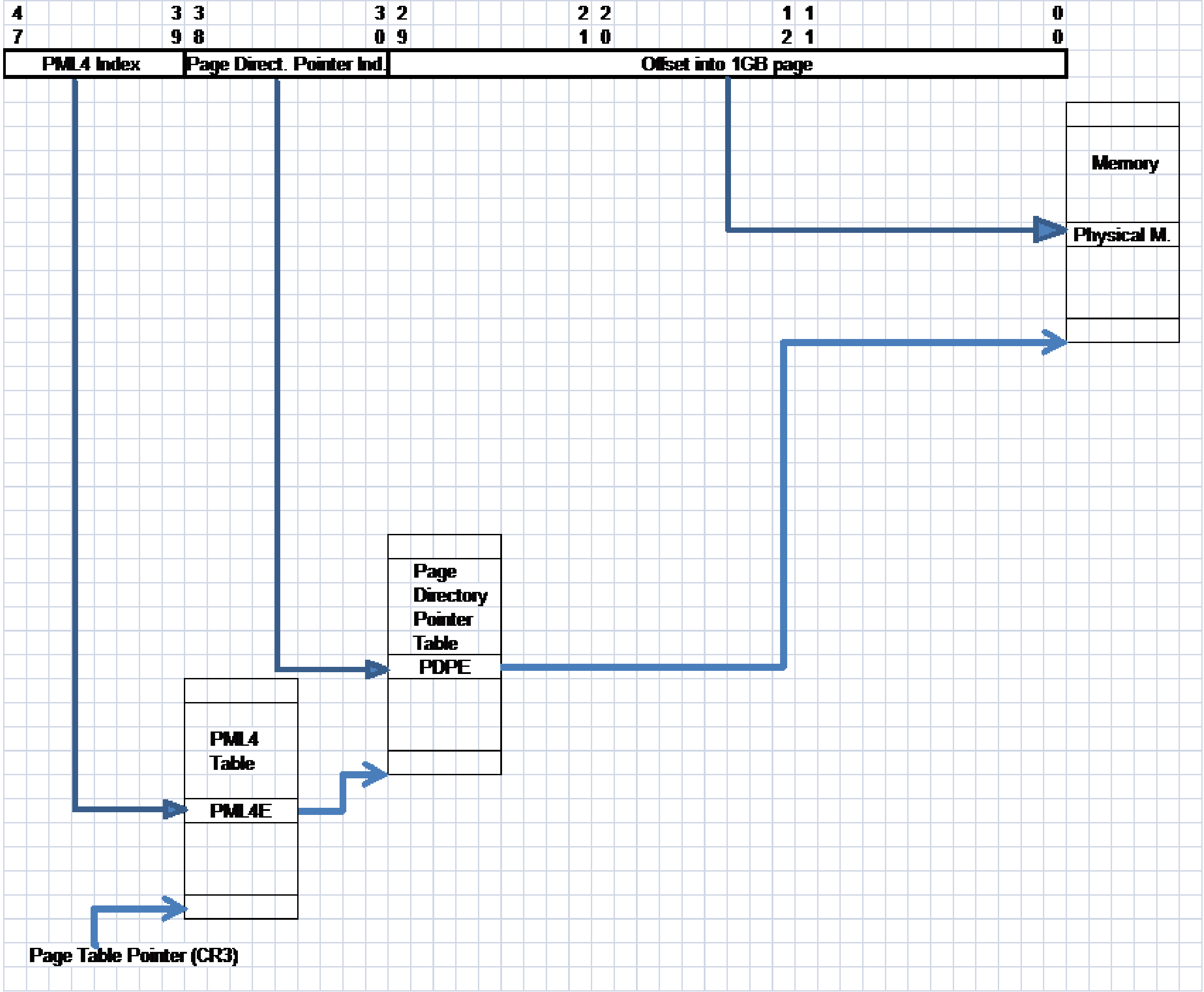
Walk with 2MB Page

Similar to the 32b VA walk, there is a support for larger pages where one of the sizes supported is 2MB.



Walk with 1GB Page

For the support for 1GB page size, the following mechanism is needed.



Programming Note	
Context:	Walk with 1 GB Page
PPGTT32 is not going to support 1GB pages.	

Global GTT

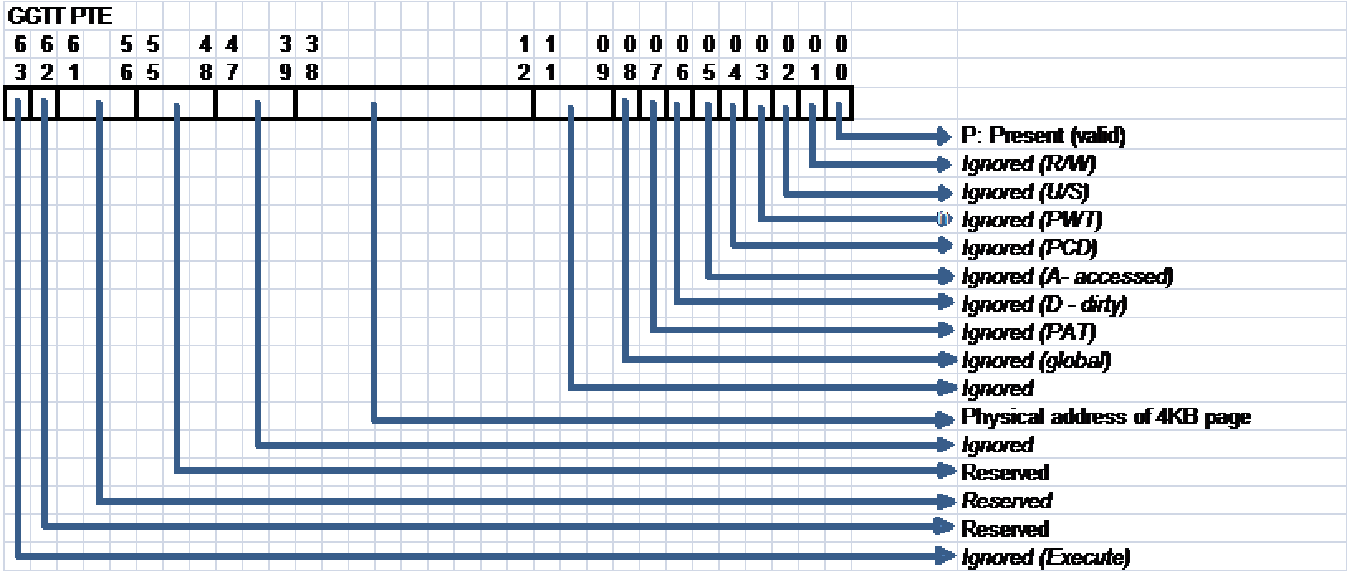
The Global GTT mechanism in CHV, BSW looks very similar to pre-CHV, BSW with the distinction of page table entry. Aperture and display will still use the global GTT even if GT core is mapped via per-process GTT.

The PTE format for CHV, BSW is updated to match per process GTT definitions and GSM is now expanded in size (2MB=>8MB) to cover for the entire 4GB (32b virtual addressing) space. Each entry corresponding to a 4KB page with 2^20 entries in GSM (each with 8B content)

For "*MI_update_GTT*", the page address provided 31:12 need to be shifted down to 22:3 for the correct QW position within the GGTT.

Page Table Entry

The following page table entry will be used for Global GTT:



- Present (Valid): The pointed PTE is valid
- *Ignored* - R/W (Read/Write): Are writes allowed to the region defined by this 4KB page. For GFX, in order 4KB memory to be usable it has to be both present and should also be write-able.
- *Ignored* - U/S (User/Supervisor access rights) : iGFX does not use these fields
- PWT/PCD/PAT bits are used as indexes into a PAT register which defines the cache attributes for the entire context.

PAT field is used to do the look up in private PAT for memory typing.

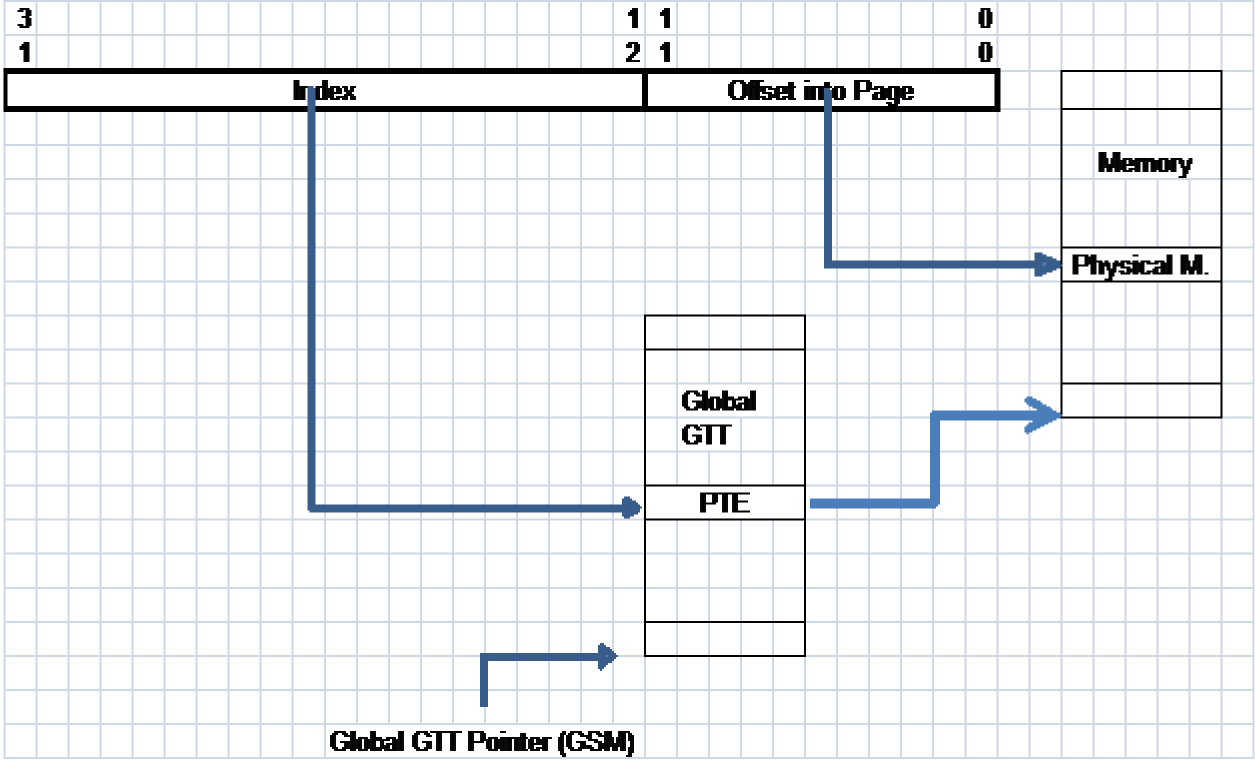
- *Ignored* - A (Accessed): It needs to be managed as the page table being accessed. Hardware needs to write this bit for the first access to the 4KB region defined with this PT entry.
- *Ignored* - D (Dirty): Hardware needs to set the dirty bit in page table if accessing this particular 4KB region in memory with the intention to modify it.
- *Ignored* - Global: this is not used by iGFX hardware, the field is used to identify global context where invalidation may not be required.
- Physical address of 4KB page

For the treatment of the page bit0 AND bit1 defines the validity of the page, the rest of the information is not relevant for Aperture and Display usage.

GGTT table entries are always read as uncacheable.

Page Walk

The global GTT page walk is identical to what it was before CHV, BSW. The only difference would be that each entry is 8B (instead of 4B) hence the entry selection needs to be updated once the corresponding Page Table miss read is returned.



GTT Cache

Processor graphics page walker implements a GTT cache which holds the remaining entries that are read as a cacheline but not used for the immediate page walk. This is only applicable in case of leaf walks and not including the 2MB/1GB page sizes. When s/w enables the use of 2MB/1GB page sizes, it will have to disable the GTT cache in CHV, BSW.

GFX Page Walker (GAM)

GPU supports various engines behind the same page walker. These streams/contexts are identified Client level IDs which are carried via the arbitration pipeline. Page walker using look-up tables does the correct selection for the page tables in case of concurrent context are running at the same time.

There are two different types of page table types:

Global graphics translation table (GGTT) is a single common translation table used for all processes. There can be many Per-process graphics translation table (PPGTT). This requires an additional lookup for translation. The actual location is not accessible directly via software since they're both located in graphics stolen memory (see graphics memory interface chapter for more detail).

Virtual Memory Structure	Memory Location
Global (GGTT)	GSM Only
Per-Process (PPGTT) – private	2 to4-level, Page Tables anywhere
Per-Process (IA32e) – shared	4 levels, Page Tables anywhere

IA32e compatible PPGTT is added to CHV, BSW to enable SVM (shared virtual memory) functions.

Context Definition for GFX Page Walker

Page Walker blocks need details about the context to decide on what type of page tables will be used, what would be the error handling cases would be and many other details to operate. The information will be passed to Page Walker (GAM) by the respective command streamer/DMA.

GAM supports the following engines:

- Render
- Media (VDBox) x2
- Blit
- VEBOX x2
- WDBox

The following fields will be sent to GAM:

- Context type (4 bits)
    - **Legacy vs Advanced Context**: Defines the context type and qualifies the rest of the fields. Same field may mean something else between the *Legacy* vs *Advanced* context. There is no restriction for what type of context can run in either combination.
        - *Requests without address-space-identifier (Legacy Context)*: These are the normal memory requests from endpoint devices. These requests typically specify the type of access (read/write/atomics), targeted DMA address/size, and identity of the device originating the request.
        - *Requests with address-space-identifier (Advanced Context)*: These are memory requests with additional information identifying the targeted process address space from endpoint devices supporting virtual memory capabilities. Beyond attributes in normal requests, these requests specify the targeted process address space identifier (PASID), and extended attributes such as Execute-Requested (ER) flag (to indicate reads that are instruction fetches), and Privileged-mode-Requested (PR) flag (to distinguish user versus supervisor access). For details, refer to the Process Address Space ID (PASID) Capability in the PCI-Express specifications.

Note CHV, BSW LP only supports Legacy PPGTT with 32b virtual addressing.

-
    - **A/D Support Enable**: Access and Dirty bits are used when OS managing the page tables and has been added to IA32e compatible page walk. Context will define whether A/D bits need to be managed via GPU. (only applicable in Advanced Context)
    - **Privileged Context Support**: Enables GPU to be able to run a privileged context which will translate into page table accesses regardless of user vs supervisor privileges. (only applicable in Advanced Context).
    - **32b vs 48b VA Support**: Enables 48b VA in page tables for the page walks. The rest of the h/w is seamless to 32b vs 48b VA address walks, however GAM will do the check and properly align the page walk to address bits.
        - *Note: Only applicable in Legacy Context, Advanced context is always 48b.*
        - Note: CHV, BSW LP only supports 32b VA
- **Function Number**: 3 bit field that defines the function number of the device. GFX device is always on BUS=0 and DEVICE=2. If we are not virtualized, our FUNCTION#=0 however if virtualized function number can be any 8 possible values (i.e. 0-7). The BUS/DEVICE/FUNCTION numbers are used to the initial walk for ROOT and CONTEXT tables.
- **PASID** – Process Address Space IDentifier: Use to identify the context that is submitted to h/w. We use the PASID in many places where during the page walk (i.e. PASID table look up) or while communicating with s/w on page faults. Each engine could be running an independent context

with different PASID. The page walker should have a mechanism to be able to cache at least some number of PASID table entries (matching to the engine count) for faster walk.

- **Context ID** (Queue ID, Bell ID) – Context ID is used to further qualify the running context beyond the PASID. PASID is given per process, and same process may allocate multiple queues to communicate with h/w. The only way to further identify the process is to use an additional ID. For GFX h/w Context ID could be same as the bell number assigned to it. GAM h/w will use the context ID to populate the queue ID field while communicating page faults to s/w.
- **Page Table Pointers** – The field could be up to 256 bits (i.e. 4x64bits) to identify the page table pointers associated with the context. For legacy 32b context, the entire 256b is valid representing the 4 PDPTR table entries. For 48b legacy context only the lower 64b is relevant pointing to base of PML4. In case of advanced context, PASID is given in the context definition.

Context Definition Delivery

Context Definition is supposed to be delivered from the corresponding command streamer to GAM and GAM has independent storage for each engine present. WDBOX is an exception. It does not have a command streamer. Its context definition is default.

Context Definition will be given by *CS to GAM via a new message:

Message: "Context Available"

GAM prepares for new context, cleans up internal state and does the proper fencing. Most of these steps should have been performed when context switch request was done for the previous context, but added here for completeness.

Message: "Context Receive Ready"

GAM is ready for the context. *CS writes all new context values into the descriptor registers. To push all context descriptors CS sends the following message to GAM also indicating new context descriptor is downloaded.

Message: "Context Launched"

GAM does the context requirements and sends the following message to CS to resume its command parser.

Message: Context Confirmed

GAM should send context confirmed message only after PD restore is done. CS waiting for context confirmed message will be treated as PD restore busy. Since all clients memory interface are blocked during PD restore it doesn't make any difference if the context confirmed message is send by GAM immediately or after PD restore.

Element Descriptor Register

General Description	Element Information: The register is populated by command streamer and consumed by GAM
Register Offset	See per engine list below

Bits	Access	Default	Field
63:32	RO	Xh	**Context ID:** Context identification number assigned to separate this context from others. Context IDs needs to be recycled in such a way that there could not be two active context with the same ID. This is a unique identification number by which a context is identified and referenced
31:12	RO	Xh	**LRCA:** Command Streamer Only
11:9	RO	Xh	**Function Number:** GFX device is considered to be on Bus0 with device number of 2. Function number is normally assigned as "0" however for gfx virtualization; there would be different function numbers which needs to be attached to context. Not used in CHV, BSW.
8	RO	Xh	**Privileged Context / GGTT vs PPGTT mode:** Differs in legacy vs advanced context modes: **In Legacy Context**: Defines the page tables to be used. This is how page walker come to know PPGTT vs GGTT selection for the entire context. "0": Use Global GTT "1": Use Per-Process GTT **In Advanced Context**: Defines the privilege level for the context "0": user mode context "1": supervisor mode context.
7:6	RO	Xh	**Fault Model:** "00": Fault & Hang (chicken bit to survive). Same mode as gen7.5

Bits	Access	Default	Field
			"01": Fault & Halt/Wait. Same mode as gen7.5 "10": Fault & Stream & Switch "11": Fault & Continue: does not generate a page request to IOMMU.
5	RO	Xh	**Deeper IA coherency Support:** **In Advanced Context:** Defines the level of IA coherency "0": IA coherency is provided at LLC level for all streams of GPU (i.e. gen7.5 like mode) "1": IA coherency is provided at L3 level for EU data accesses of GPU
4	RO	Xh	**A&D Support / 32&64b Address Support:** Differs in legacy vs advanced context modes: **In Legacy Context**: Defines 32b vs 64b (48b canonical) addressing format "0": 32b addressing format "1": 64b (48b canonical) addressing format **In Advanced Context**: Defines A&D bit support "0": A&D bit management in page tables is NOT supported "1": A&D bit management in page tables is supported.
3	RO	Xh	**Context Type: Legacy vs Advanced** Defines the context type. "0": Advanced Context: Defines the rest of the advanced capabilities (i.e. OS page table support, fault models...). Note that advanced context is not bounded to GPGPU. "1": Legacy Context: Defines the context as legacy mode which is similar to prior generations of CHV, BSW. *Note that: Bits [8:4] differs in functions when legacy vs advanced context modes are selected.*
2	RO	Xh	**FR:** Command streamer specific
1	RO	Xh	**Scheduling Mode:** "1": Indicates execlist mode of scheduling. "0": Indicates Ring Buffer mode of scheduling.
0	RO	Xh	**Valid:** Indicates that element descriptor is valid. If GAM is programmed with an invalid descriptor, it will continue but flag an error.

PDP0/PML4/PASID Descriptor Register

General Description	PDP0/PML4/PASID: The register is populated by command streamer and consumed by GAM. It contains one of the 3 values which is determined by looking at the element descriptor.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:0	RO	Xh	**PDP0/PML4/PASID:** This register can contain three values which depend on the element descriptor definition. **PASID[19:0]**: Populated in the first 20bits of the register and selected when Advanced Context flag is set. **PML4[38:12]:** Pointer to base address of PML4 and selected when Legacy Context flag is set and 64b address support is selected **PDP0[38:12]:** Pointer to one of the four page directory pointer (lowest) and defines the first 0-1GB of memory mapping *Note: This is a guest physical address* *(unused bits need to be populated as 0's)*

PDP1 Descriptor Register

General Description	PDP1: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP1:** Pointer to one of the four page directory pointer (lowest+1) and defines the first 1-2GB of memory mapping *Note: This is a guest physical address* *(unused bits need to be populated as 0's)*

PDP2 Descriptor Register

General Description	PDP2: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP2:** Pointer to one of the four page directory pointer (lowest+2) and defines the first 2-3GB of memory mapping *Note: This is a guest physical address* *(unused bits need to be populated as 0's)*

PDP3 Descriptor Register

General Description	PDP3: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP3:** Pointer to one of the four page directory pointer (lowest+3) and defines the first 3-4GB of memory mapping *Note: This is a guest physical address* *(unused bits need to be populated as 0's)*

List of Registers and Command Streamers

The following registers are message registers and not written directly by software.

Engine	Offset	Description
Render	x4400h	Element Descriptor Register
	x4408h	PDP0/PML4/PASID Descriptor Register
	x4410h	PDP1 Descriptor Register
	x4418h	PDP2 Descriptor Register
	x4420h	PDP3 Descriptor Register
Media0 (VDBOX0)	x4440h	Element Descriptor Register
	x4448h	PDP0/PML4/PASID Descriptor Register
	x4450h	PDP1 Descriptor Register
	x4458h	PDP2 Descriptor Register
	x4460h	PDP3 Descriptor Register
Media1 (VDBOX1)	x4480h	Element Descriptor Register
	x4488h	PDP0/PML4/PASID Descriptor Register
	x4490h	PDP1 Descriptor Register
	x4498h	PDP2 Descriptor Register
	x44A0h	PDP3 Descriptor Register
VEBOX	x44C0h	Element Descriptor Register
	x44C8h	PDP0/PML4/PASID Descriptor Register
	x44D0h	PDP1 Descriptor Register
	x44D8h	PDP2 Descriptor Register
	x44E0h	PDP3 Descriptor Register
Blitter	x4500h	Element Descriptor Register
	x4508h	PDP0/PML4/PASID Descriptor Register
	x4510h	PDP1 Descriptor Register
	x4518h	PDP2 Descriptor Register
	x4520h	PDP3 Descriptor Register

Messages

Message Name	Source	Destination	Category	Address (Hex)	Bit	Mask Bit	Value	Description
Context Available	CS (GT)	GAM (GT)	self-clear	4004	0	16	1	Signal request from CS to GAM as new context is about to be submitted.
Context Receive Ready	GAM (GT)	CS(GT)	self-clear	3438	0	16	1	Signal ack from GAM to CS in response to Context Available message from CS to GAM.
Context Launched	CS (GT)	GAM (GT)	self-clear	4004	1	17	1	Signal indicator to GAM that context descriptor is pushed.
Context Confirmed	GAM (GT)	CS(GT)	self-clear	3438	1	17	1	Signal ack from GAM to CS in response to Context Launched message from CS to GAM.
Context Available	BCS (GT)	GAM (GT)	self-clear	4014	0	16	1	Signal request from CS to GAM as new context is about to be submitted.
Context Receive Ready	GAM (GT)	BCS(GT)	self-clear	23438	0	16	1	Signal ack from GAM to BCS in response to Context Available message from BCS to GAM.
Context Launched	BCS (GT)	GAM (GT)	self-clear	4014	1	17	1	Signal indicator to GAM that context descriptor is pushed.
Context Confirmed	GAM (GT)	BCS(GT)	self-clear	23438	1	17	1	Signal ack from GAM to BCS in response to Context Launched message from BCS to GAM.
Context Available	VECS (GT)	GAM (GT)	self-clear	4010	0	16	1	Signal request from CS to GAM as new context is about to be submitted.
Context Receive Ready	GAM (GT)	VECS(GT)	self-clear	1B438	0	16	1	Signal ack from GAM to VECS in response to Context Available message from VECS to GAM.
Context Launched	VECS (GT)	GAM (GT)	self-clear	4010	1	17	1	Signal indicator to GAM that context descriptor is pushed.
Context Confirmed	GAM (GT)	VECS(GT)	self-clear	1B438	1	17	1	Signal ack from GAM to VECS in response to Context Launched message from VECS to GAM.
Context Available	VCS0 (GT)	GAM (GT)	self-clear	4008	0	16	1	Signal request from CS to GAM as new context is about to be submitted.

Message Name	Source	Destination	Category	Address (Hex)	Bit	Mask Bit	Value	Description
Context Receive Ready	GAM (GT)	VCS0(GT)	self-clear	13438	0	16	1	Signal ack from GAM to VCS in response to Context Available message from VCS to GAM.
Context Launched	VCS0 (GT)	GAM (GT)	self-clear	4008	1	17	1	Signal indicator to GAM that context descriptor is pushed.
Context Confirmed	GAM (GT)	VCS0(GT)	self-clear	13438	1	17	1	Signal ack from GAM to VCS in response to Context Launched message from VCS to GAM.
Context Available	VCS1 (GT)	GAM (GT)	self-clear	400C	0	16	1	Signal request from CS to GAM as new context is about to be submitted.
Context Receive Ready	GAM (GT)	VCS1(GT)	self-clear	1D438	0	16	1	Signal ack from GAM to VCS in response to Context Available message from VCS to GAM.
Context Launched	VCS1 (GT)	GAM (GT)	self-clear	400C	1	17	1	Signal indicator to GAM that context descriptor is pushed.
Context Confirmed	GAM (GT)	VCS1(GT)	self-clear	1D438	1	17	1	Signal ack from GAM to VCS in response to Context Launched message from VCS to GAM.

Updating Page Table Pointers (aka PD Load)

In case of legacy context, driver is allowed to add/remove pages as long as it is ensured that h/w is not using these entries. Pre-CHV, BSW flow allowed a mid-context PD load to update the PD entries and directed h/w to reload updated entries. CHV, BSW legacy context will require a similar mechanism.

Instead of a PD load, the new mechanism will let the driver update the page table pointers via sending a reload command. Mechanism will be overlapped to same sort messaging between CS and GAM to deliver the context header. CS will send the following message to GAM:

Message: Context Reload *(TBD – for address offset and data)*

GAM will respond immediately to CS with the following message

Message: Context Confirmed *(TBD – for address offset and data)*

Meanwhile GAM will block the related interfaces and updates the PDs or PML4.

Page Walker (GAM) Reset

GAM gets all the engine specific resets as well as device and bus resets to manage its internal logic domains. It is the expectation of SW when a particular GPU engine (i.e. Render, Media, ...) gets reset, all its related HW is cleared and comes out fresh for reprogramming. That is true for most of the logic with the exception of some shared HW blocks. The following blocks require additional steps (post-reset) from SW to further clean-up the HW:

- **Hardware TLBs:** The caching structures for the page walks are often considered shared resources. The expectation is for GFX driver to clear the TLBs via "TLB Invalidate" prior to reusing the engine post reset. This is the same process that was followed on previous GPU generations.
- **Page Requests:** At the time of the reset HW may have outstanding page requests to SW for page faulted accesses. These requests could be at any level hence it is required for SW to clear these paging requests pre/post-engine reset. Engine reset ensures no new page requests would be sent from HW. Page requests could be at the "page request queue" in memory where they could be mapped to a dummy page post engine reset completion. Or they could be at the MMIO registers which will block the completion of the reset; it is up to SW to service paging request interrupts without waiting for the completion of reset request.

Device reset (FLR) covers most of the page walker, however there are exceptions where IOMMU structures and all messaging towards rest of the system (system agent) should not be impacted by it. All external interactions and IOMMU related blocks are kept under bus (system) reset. GAM will keep the following blocks outside the device reset:

- IOMMU registers and content.
- All system agent messaging structures (including translation enable flows, root pointer structures, DMA fault reporting pieces.).

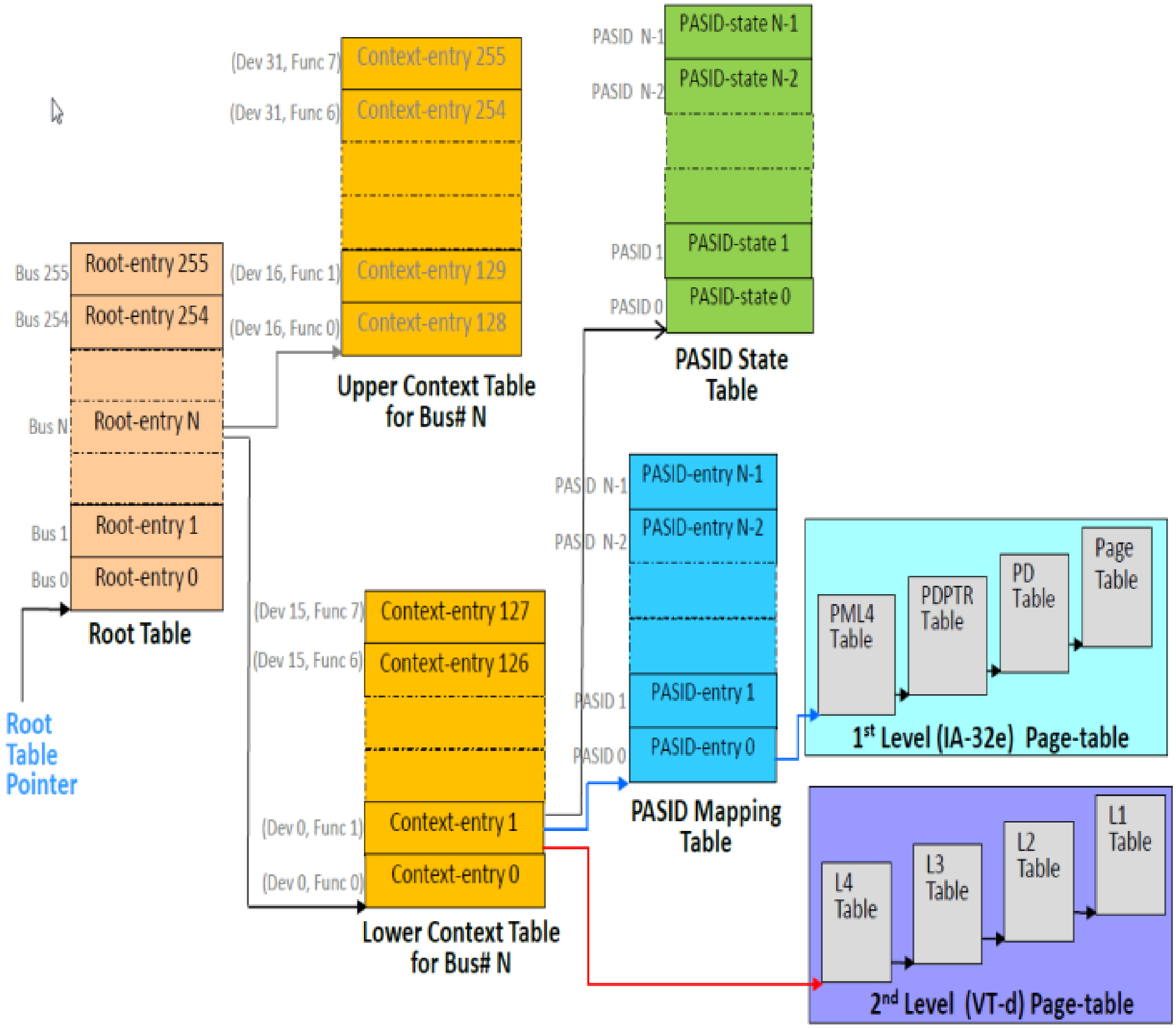
An engine being reset also means the particular context that engine is running, is complete or taken out. That will require GAM to decrement the PASID_State Counter if the engine was running a PASID based (advanced) context. For FLR (device reset), a similar requirement holds. In case of device reset, GAM would need to decrement all the PASID state counters that are active on the GPU before completing the sequence.

Legacy Context

Legacy context could use either Global GTT or Per Process GTT which is given to page walker as part of the context descriptor. Even under PPGTT, there could be accesses from Command Streamers that would require to use Global GTT which requires to treat the walk requirement per transaction.

For Legacy context indicator command streamer is going to pass the "context type" information along with other parameters that defines how certain behavior for paging needs to be.

Full Walk

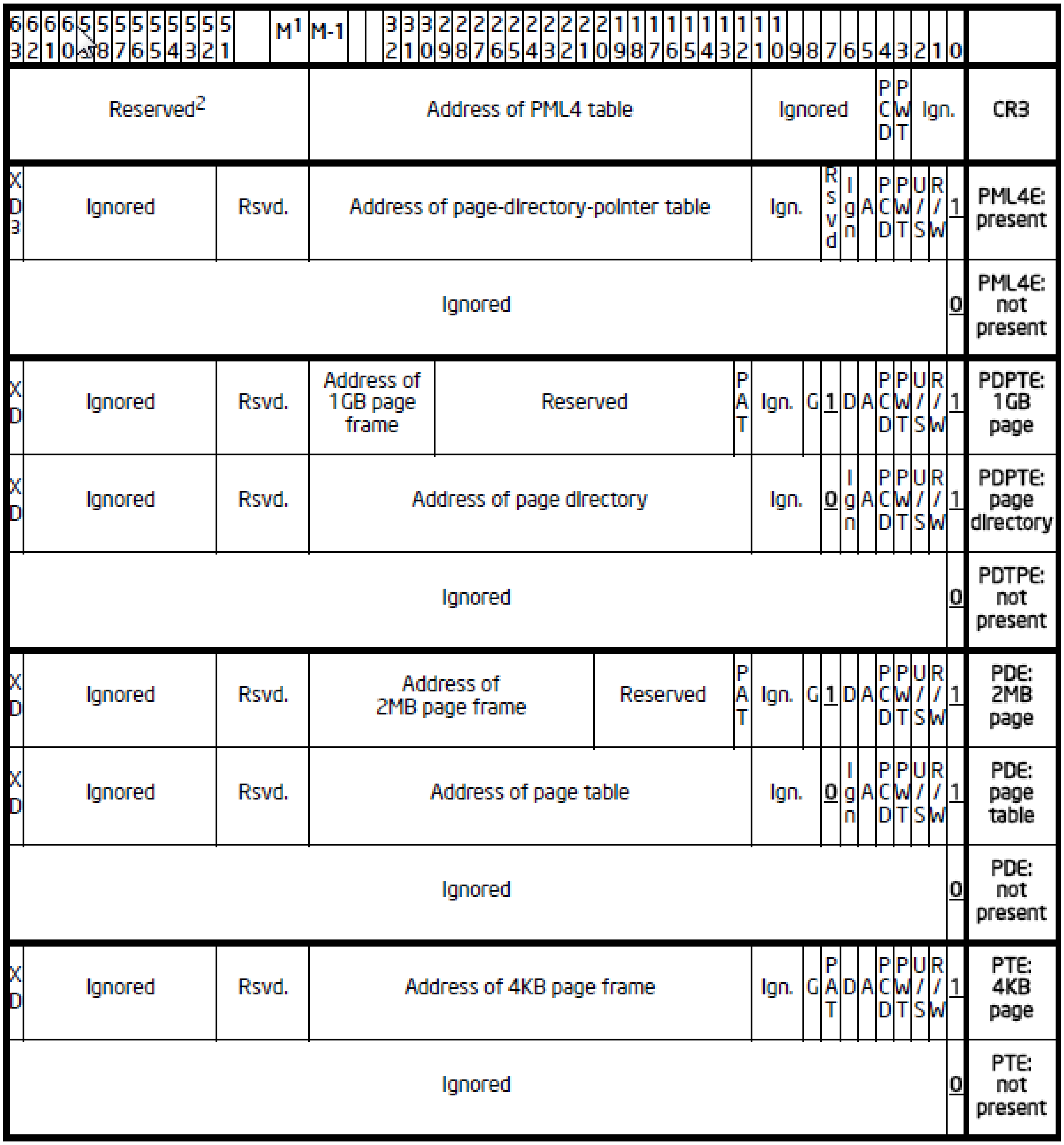


In the full walk case (i.e. advanced context), the root of the 1st and 2nd level page tables share a common source. Both the root table and context table is walked with the assumption of GFX device is always on Bus#0 and it is always Device#2.

Function number however is part of the context and it can be non-Zero only for virtualized modes. GAM will receive the function number (for Context table look-up) as part of the context. Both Root entry and Context Entry should be fetched along with PASID Table entry prior to running the context accesses.

TLB Caching and Management

As compared to previous generation of TLB entry, IA32e page translation entry is quite different. At every stage of the page different bits need to be taken into account and proper treatment is required. Regardless of PPGTT vs GGTT usage, the paging entry has the same format. Linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses. Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time. IA-32e paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.



63	62 61 60 59 58 57 56 55 54 53 52	51 M[1]	M-1		12	11 10 9	8	7	6	5	4	3	2	1	0	
Reserved[2]			Address of PML4 table			Ignored					PCD	PWT	Ign.			CR3
XD[3]	Ignored	Rsvd.	Address of page-directory-pointer table			Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored															0	PML4E: not present
XD	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.	Address of page directory			Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored															0	PDTPE: not present
XD	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.	Address of page table			Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored															0	PDE: not present
XD	Ignored	Rsvd.	Address of 4KB page frame			Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored															0	PTE: not present

The following rules apply:

1. M is an abbreviation for MAXPHYSICAL ADDRESS.
2. Reserved fields must be "0".
3. Ignored field must be ignored. (There could be private information.)

All ignore options are part of the context entry and come from the IOMMU definition.

TLB Caches

For CHV, BSW the caching structures are separated as following with the architectural view, this is also applicable to s/w view of these caches when it comes to invalidations.

Context Cache - CC

This is the storage for context table entry which is achieved as part of root/context table walk.

Context cache can also be invalidated with directed invalidations, where HW needs to invalidate the content of the context cache along with all low level caches.

PASID Cache - PC

This is where the HW copy of the PASID table entry is kept and it is per context. This makes it unique for every HW engine that could be running an independent context (per GAM):

- Render/GPGPU
- MFX (VDBOX) – 1
- MFX (VDBOX) – 2
- Video Enhancement (VEBOX) – 1
- Video Enhancement (VEBOX) – 2
- Blitter

The cache content is updated if the corresponding engine is running an advanced context where its page table pointers are accessible via PASID table. In case of legacy context running engine, corresponding PASID Cache entry is not valid. Recommendation is to keep ONE physical storage per engine which is filled/invalidated during the context switch time.

PASID Cache can also be invalidated with the directed invalidations along with low level caches and needs to be re-filled prior to context resuming.

Intermediate Page Walk Caches (PML4, PDP, PD) – PWC

These are the stages where intermediate page walk entries are cached to speed-up/shorten the page walk when final TLB is missed. Each level can be cached separately or along with different levels, the cacheability structures will have programmability to move the boundary of different levels to accommodate more/less on each page walk level. However as a concept, for legacy 32b addressing mode, requirement is to cache 4PDPs along with 4x4KB PDs for certain engines, at least for render and media. The others will use cache concept.

TLB – Final Page Entry

The size of the TLBs has been increased over the previous generation and should be targeting the below table:

- L3 TLB: 768 TLB entries – This is where all HDC, I$, Constant, State, and Sampler streams are stored.
- MFX: 512 TLB entries – All Media streams (split 256/256 between two media engines).
- BLT: 32 entries.
- Z: 512 TLB entries – All depth accesses.
- C: 256 (256 TLB entries) – All color accesses.
- FF: 128 (128 TLB entries) – All FF accesses to memory.
- VLF: 32 (32 TLB entries) – Media surface.
- GAV: 192 (192 TLB entries) – Video enhancement. Increased compared to other Gen8 projects.
- WiDi: 64 (64 TLB entries) – Wireless Display.

All TLB entries are increased to 48b to contain larger address as well as the page attributes attached to it.

The max size of a single TLB is 256 entries, larger quantities have to be handled as set-associative storages. Set associativity is managed by low order page bits (i.e. address#12, address#13, ...).

TLB Entry Content

When a page walk entry is cached (or loaded prior to context start), certain bits need to be cached as well along with the physical address bits. The treatment on these bits would be considered when a HIT vs MISS decision needs to be made during a look up.

The purpose of caching is to accelerate the paging process by caching individual translations in **translation look-aside buffers** (**TLBs**). Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number:
    - The logical-AND of the R/W flags.
    - The logical-AND of the U/S flags.
    - The logical-OR of the XD flags.
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):

- The dirty flag.
- The memory type.

PRESENT: This is the same VALID bit description we had in previous page table designs. The lack of present bit (i.e. bit[0]=0) points that rest of the information in the page table entry is being invalid. For some fault models, even NOT PRESENT entries are cached to filter further page faults (*see fault models on caching page faulting entries*). If such entry is cached, there are couple ways that it can be removed from the page tables:

1. LRA selection where the entry becomes a victim for replacement
2. Global or Selective invalidation
3. Page fault response stating the faulting page is now fixed.

R/W Privilege: Certain pages can be allocated as read-only and write operations are not allowed. To make this check work, TLB has to keep the R/W bit. This bit has no affect on read operations; however for write operation privilege needs to be checked. If there is mis-match, the result of the TLB look-up should be a MISS. This does not mean a page fault immediately; the walk has to be re-done as for any TLB MISS result. There are cases OS may change page table privileges without invalidating pages in TLB (*note: all downgrades result in invalidation of the TLB, however upgrades can be done silently hence re-walk is required*). In case where the TLB Miss is due to privilege mis-match, the existing entry from TLB has to be invalidated and page walk will bring in the most up-to-date copy from memory.

The R/W privilege on final frame is generated as a logical-AND process of all upper page walks pointing to this location.

User vs Supervisor Privilege: The GPU typically operates in user mode when it comes to page tables. So the GTT walk can be treated as faulted when GPU encounters a page with supervisor privileges and the context is marked as user mode. The faulted entry can be cached back into TLB with “P” bit off indicating a faulted entry. However the page fault report should carry the correct reason why h/w detected the fault in the first place which was the user vs supervisor privilege. There is an option in context header to define the context as supervisor, than it legal to access supervisor pages.

- This is not stored in TLB

The U/S privilege on final frame is generated as a logical-AND process of all upper page walks pointing to this location.

Accessed Bit: This where a stage of the page walk cannot be used if the accessed bit is not set for that level in the page walk. This is true for both storage into TLB as well as to make progress on the page walk. In order to achieve the process of Accessed bit, every stage of the ppGTT read is done via a new semantics between the GAM and GTI such that GTI can atomically process A-bit w/o running into access violations. The details of the semantics are defined as part of the following sections. The “A” bit does not need to be stored as part of the TLB, just the fact that a valid page table entry is present in the TLB does mean that h/w took care off the “A” bit at the time the page was brought up to TLB. Note that TLB prefetching is disabled when A-bit management is enabled.

IA32e mode page tables cannot co-exist with TLB pre-fetching due to lack of A-bit management for all entries of the line.

- This is not stored in TLB

Dirty Bit: Similar to accessed bit, dirty bit needs to be managed. It is only applicable for “write” accesses. Given there are cases where a TLB entry was acquired as part of a read operation, the presence of D-bit should be maintained with the TLB. This gives us the capability to declare a TLB miss for a write access when the D-bit is not set even though TLB has a valid translation. In such case, The TLB entry needs to invalidated and the final stage of the walk needs to be re-done to ensure most up-to-date copy of GTT entry is brought into h/w. The operation of Dirty bit update is also atomic similar to A-bit management.

Execute (XD) Bit: XD bit is also present on every stage of the walk and applicable to executable code that GT would be fetching. In the first pass, instruction cache accesses are not allowed to proceed if the corresponding page does not have the execute credentials set properly. Similar treatment of the TLB entry as privilege bits is expected. A page entry that was already cached in TLB and later accessed for instruction space will have to check the XD bit which is also stored in TLB. If mis-match, the end result is a TLB miss and walk has to be re-done replacing the different stages of the walk.

The XD privilege on final frame is generated as a logical-OR process of all upper page walks pointing to this location.

Faulted Bit: There are usage models where the faulted entries are cached in TLB. This is to filter further faults to the same page as opportunistic way to prevent fault storms. When faulted bit is set the address is included in the TLB look up but final treatment is fault filtering. The rest of the bits are used to define what would be the reason for the fault. If the look-up conflicts with the original faulted reason, a re-walk is required. As a basic case, take a read access bringing up a PTE with W-flag cleared. A subsequent write access has a conflict on privilege, and it will perform a re-walk. If the result of the re-walk is W-flag set, than TLB is upgraded and write makes progress. However if the result is still W-flag cleared, the write access will fault and TLB entry will be tagged as a faulted entry with only read-allowed. Subsequent write accesses will be filtered as fault but read accesses should cause a re-walk of the page and if successful, the TLB can be updated with PTE as valid with read-only attribute.

TLB Accessed and Dirty Flags

For any paging-structure entry that is used during linear-address translation, bit 5 is the **accessed** flag. For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the **dirty** flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor and/or GPU uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor and/or GPU sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are “sticky,” meaning that, once set, the processor and/or GPU does not clear them; only software can clear them.

A processor and/or GPU may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the GPU might not set the corresponding bit in memory on a subsequent access using an affected linear address

Accessed bit is applicable to every stage of the page walk, however the dirty bit is only applicable to final stage of the walk.

The rule states that a particular access cannot be committed until the Accessed and/or Dirty bits are not visible to page management s/w. In order for GPU to follow the rule, GTT accesses (when A/D bits are supported) are going to be done via a special cycle definition between GAM and GTI.

Updating A/D Bits

New atomic operations are added to GAM to GPU interface (GTI) to handle paging entries. GAM has to set the correct atomic opcodes based on the access type and context entry controls as well as level of access.

Requires setting for opcodes are given in the table below. The steps of operations in the atomic ALUs are given later in the document.

<table>
<tr><td colspan="6"><b>The Following Atomics are only applicable in GTI and used for Page Walks</b>
<b>R/W => Bit[0]</b>
<b>Extended Access required => Bit[1]</b>
<b>Write Protect Enable => Bit[2]</b>
<b>Intermediate Entry => Bit[3]</b></td></tr>
<tr><th>Atomic Operation</th><th>Opcode</th><th>Description</th><th>New Destination Value</th><th>Applicable</th><th>Return Value (optional)</th></tr>
<tr><td>Atomic_Page_update_0000</td><td>1100_0000</td><td>Read Access
Extended Access bit is disabled
Write Protection is disabled
Final PTE</td><td>Set bit[5] if not set</td><td></td><td>new_dst</td></tr>
</table>

Atomic_Page_update_0001	1100_0001	Write Access Extended Access bit is disabled Write Protection is disabled Final PTE	Set bit[5,6] if not set		new_dst
Atomic_Page_update_0000	1100_0010	Read Access Extended Access bit is enabled Write Protection is disabled Final PTE	Set bit[5,10] if not set		new_dst
Atomic_Page_update_0001	1100_0011	Write Access Extended Access bit is enabled Write Protection is disabled Final PTE	Set bit[5,6,10] if not set		new_dst
Atomic_Page_update_0100	1100_0100	Read Access Extended Access bit is disabled Write Protection is enabled Final PTE	Set bit[5] if not set		new_dst
Atomic_Page_update_0101	1100_0101	Write Access Extended Access bit is disabled Write Protection is enabled Final PTE	Set bit[5,6] if not set		new_dst
Atomic_Page_update_0100	1100_0110	Read Access Extended Access bit is enabled Write Protection is	Set bit[5,10] if not set		new_dst

		enabled Final PTE			
Atomic_Page_update_0101	1100_0111	Write Access Extended Access bit is enabled Write Protection is enabled Final PTE	Set bit[5,6,10] if not set		new_dst
Atomic_Page_update_0000	1100_1000	Read Access Extended Access bit is disabled Write Protection is disabled Intermediate Paging Entry	Set bit[5] if not set		new_dst
Atomic_Page_update_0001	1100_1001	Write Access Extended Access bit is disabled Write Protection is disabled Intermediate Paging Entry	Set bit[5,6] if not set		new_dst
Atomic_Page_update_0000	1100_1010	Read Access Extended Access bit is enabled Write Protection is disabled Intermediate Paging Entry	Set bit[5,10] if not set		new_dst
Atomic_Page_update_0001	1100_1011	Write Access Extended Access bit is enabled Write Protection is disabled	Set bit[5,6,10] if not set		new_dst

		Intermediate Paging Entry			
Atomic_Page_update_0100	1100_1100	Read Access Extended Access bit is disabled Write Protection is enabled Intermediate Paging Entry	Set bit[5] if not set		new_dst
Atomic_Page_update_0101	1100_1101	Write Access Extended Access bit is disabled Write Protection is enabled Intermediate Paging Entry	Set bit[5,6] if not set		new_dst
Atomic_Page_update_0100	1100_1110	Read Access Extended Access bit is enabled Write Protection is enabled Intermediate Paging Entry	Set bit[5,10] if not set		new_dst
Atomic_Page_update_0101	1100_1111	Write Access Extended Access bit is enabled Write Protection is enabled Intermediate Paging Entry	Set bit[5,6,10] if not set		new_dst

Atomic updates are only possible for cacheable memory types. There could be cases where the PTE could be in WT/WC/UC space where atomic update is not possible via WB space. Those are the cases where IA cores use bus lock to update the A/D bits in PTE.

GT core is not capable of supporting bus locks and has the following options. These options will be enabled/disabled via register space.

Option#1: Ignore the PAT/MTRR setting of the PTE and update the space as WB with atomic ops. This is the place GAM will decide to go forward with atomic updates assuming WB space works

Option#2: Once the memory type is determined and the end result of the page is WC/UC/WT space, we can not guarantee an atomic update. GAM will report an application error (catastrophic) to the scheduler and handle the case as error.

Bit	Access	Default Value	Description
1	R/W	0b	**A/D Bit Update on non-WB Space:** A/D bit updates are only possible via atomic operations which are required to be on WB space to work properly. On non-WB spaces, the A/D bit updates are done via bus locks which are not supported for GT. **"1":** Ignore the page level cacheability and do atomic updates for A/D bit management **"0":** Detect the page level cacheability as part of the atomic operation and throw a catastrophic error when non-WB space is seen for A/D bit updates.

PAT (IA32e)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified.

The PAT is a 64-bit data structure defined in context entry when advanced context is chosen and for legacy context and internal 64b register is defined to keep the page table based cacheability. It is compromising eight (8) 8-bit entries (entry *i* comprises bits 8*i*+7:8*i* of the register).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Specifically, it comes from entry *i* of the PAT, where *i* is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with IA-32e paging):

i = 2*PCD+PWT, where the PCD and PWT values come from CR3.

- For an access to a paging-structure entry X whose address is in another paging structure

  entry Y, *i* = 2*PCD+PWT, where the PCD and PWT values come from Y.

- For an access to the physical address that is the translation of a linear address, *i* = 4*PAT+2*PCD+PWT, where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).

PAT in Context Table Entry

PAT definition is embedded inside the context entry and already defined as part of the context entry definition (see related section). It allows 8 different settings which can be indexed using the following encodings listed as part of memory types. PAT in context entry is used for advanced context usage.

Memory Types and Applicability to GFX

The Memory Types defined for IA are listed below as:

Memory Type	Encoding in MTRR/PAT
Uncacheable (UC)	00h
Write Combining (WC)	01h
Write Through (WT)	04h
Write Protected (WP)	05h
WriteBack (WB)	06h
Uncached (UC-)	07h
Reserved*	02, 03, 08h-FFh

Note: * use of any reserved encodings will result in a FAULT and reported into fault registers.

- **Uncacheable (UC):** IA semantics for a UC cycle is slightly different than traditional UC concept that was adapted by GFX as part of integration into CPU. When UC type is selected from the MTRR table, GAM will enforce the request to be uncacheable in LLC/eLCC (turn-off the cacheability flags) and also force the fence semantics in GTI.

  Note: This behavior is not followed with a fence in case of GPUs.
- **Uncached (UC-)**: Same concept as UC from behavior perspective however the precedence can be overridden by WC unlike UC.
- **Write Combining (WC):** Write combining follows a streaming model in IA terms which is not cached in uncore. Semantically the existing GT use of UC concept overlaps with WC memory type defined by IA. GFX will treat the WC memory type as a streaming uncacheable memory type in GFX pipelines.
- **Write Through (WT):** Write through concept is already introduced as part of the gen7.5 design, the IA version of the WT overlaps with the same concept.
- **Write Protected (WP):** GFX has no concept of write protected; however this is simply a combination of two modes distributed over different access types:
  - **Reads**: Acts as WB.
  - **Writes**: Acts as WC.
- **Write Back (WB):** WB memory type is traditional memory type used where accesses are cached in uncore as per the directives provided. This is the main cacheable mode that will be used.

Basically, GPU will support all memory-types that CPU supports (as below), with the same meaning with respect to "**Cacheable**", "**Writeback Cacheable**", and "**Serialization**". Because the GPU has its own device-specific notion of what "*speculative processor ordering*" means, those are not specified/attached to memory-types.

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering
Uncacheable (UC-)	No	No	No	Strong Ordering. Can only be selected through the PAT. Can be overridden by WC in MTRRs.
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.
Write Protected (WP)	Yes for reads; no for writes	No	Yes	Speculative Processor Ordering. Available by programming MTRRs.

The key similarities with CPU memory types are:

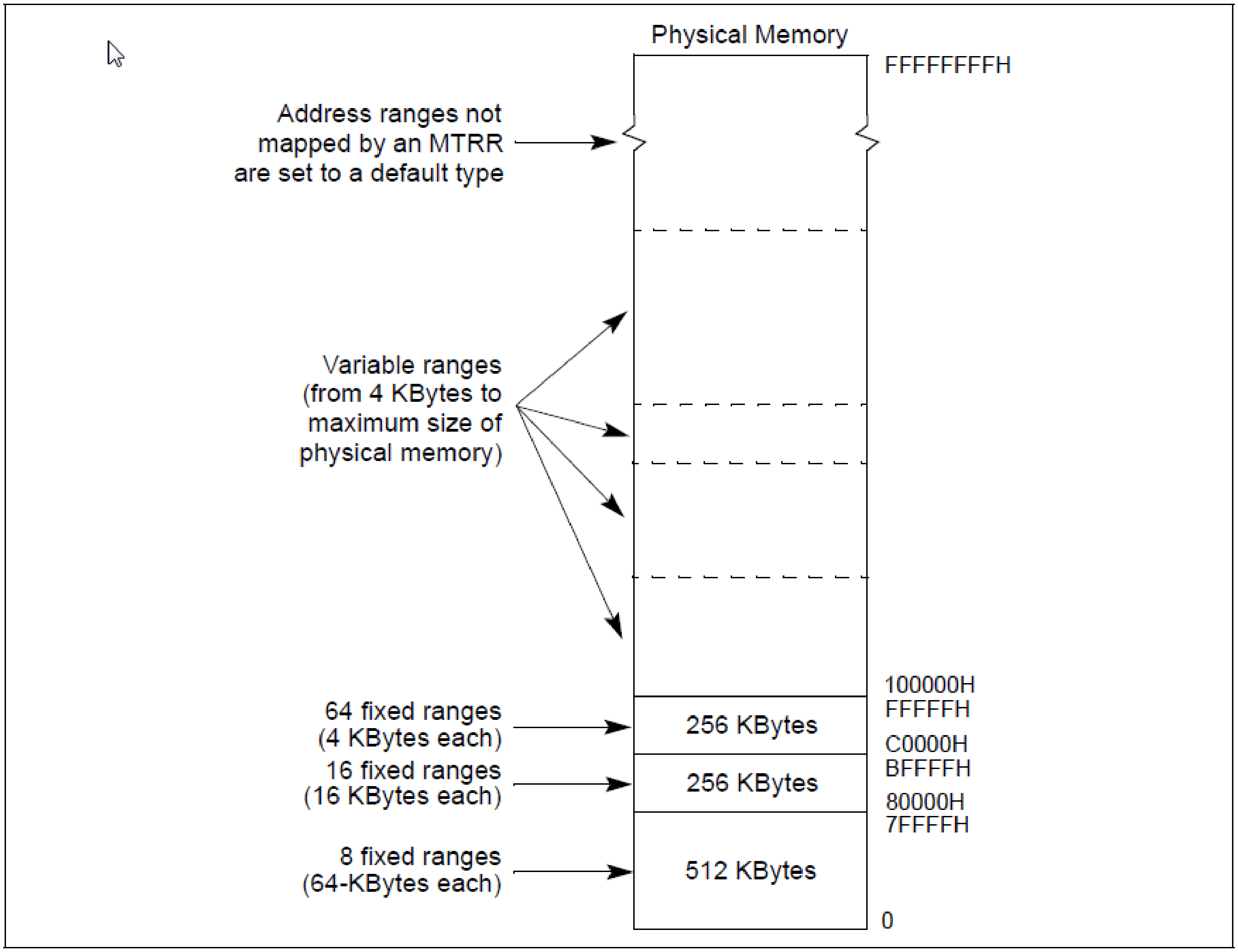
- GPU's cache (L3 cache) will be made coherent (works similar to MLC on CPUs).
- All GPU accesses (regardless of memory type) will snoop LLC.
- All GPU accesses will self-snoop.
- Outside of virtualization, GPU computes effective memory-type same as CPU (MTRR, PAT, PCD, PWT, etc.), and caches follow normal versus non-allocating mode per CD bit (similar to CPUs).
- UC, UC- and WC accesses do not allocate to caches. Will invalidate if line already exists in caches.
- UC- works same ways as CPU (i.e., unlike UC, UC- allows override by WC).
- WB accesses will allocate to cache without updating memory.
- WT behaves same as WB, except writes, updates memory along with allocate to cache.
- WP behaves same as WT, except writes always propagate to memory (invalidating any cache-line that hits).

The key difference with CPU memory types are:

- Speculative processor ordering not specified
- For GPU, only difference between UC/UC- and WC is that, UC has stronger ordering.

MTRR Ranges

Memory Type Range Registers are defined to cover the entire physical memory. The following table shows how each region is defined and how they map over the physical memory.


Physical Memory
FFFFFFFFH
Address ranges not mapped by an MTRR are set to a default type
Variable ranges (from 4 KBytes to maximum size of physical memory)
64 fixed ranges (4 KBytes each)
16 fixed ranges (16 KBytes each)
8 fixed ranges (64-KBytes each)
256 KBytes
256 KBytes
512 KBytes
100000H
FFFFFH
C0000H
BFFFFH
80000H
7FFFFH
0

Memory Type Selection and Priority

Memory typing determination is split up to two categories:

- **Architectural Options**: This is the traditional memory typing defined via external specifications and controls the cacheability of various surfaces.
- **Design Specific Options**: To target product specific caches and cache optimizations

Design-Specific Memory Types

The following are the design specific memory types for CHV, BSW.

LRU Age: Both LLC and eDRAM uses LRU-like replacement algorithm with Age based determination

00: Age is 0

01: Age is 1

10: Age is 2

11: Age is 3

Target Cache: CHV, BSW has two large caches in Uncore where they could be separately targeted.

00: reserved

01: eLLC only

10: LLC only

11: eLLC/LLC

Both LRU and target cache selections can only be managed via non-architectural solutions.

Legacy Context: Selection is based on whether target cache field programmed in surface state is “00” or non-“00”. If non-“00”, than both the Age and Target cache parameters are picked from surface state and page table controls are ignored. If target cache parameter in surface state is “00” than, only page table controls are used via private PAT programming (see PAT calculation) and indexed into “PAT in MMIO Register Space”.

Advanced Context: Decision is based on surface state only. Advanced context uses architectural definition of PAT via IA32e page tables which do not carry design specific information. The only mechanism to control cache LRU and targets is to program the surfaces state accordingly.

Memory Object Control State (Surface)

Bit	Description
6:5	**Memory Type: LLC/eLLC Cacheability Control (LeLLCCC)** This is the field used in GT interface block to determine what type of access need to be generated to uncore. For the cases where the LeLLCCC is set, cacheable transaction are generated to enable LLC usage for particular stream. 00: Use Cacheability Controls from page table / UC with Fence (if coherent cycle) 01/10: non-snooped 11: snooped **For CHV, BSW, GFX driver should use snooped type for only surfaces that are prepared by the driver in IA WB space. All other surfaces should be tagged as nonsnooped**
4:3	**Target Cache (TC)** This field allows the choice of LLC vs eLLC for caching

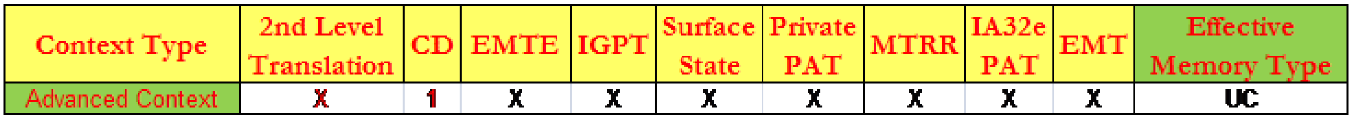
	00: eLLC Only – not snooped in GT 01: LLC Only 10: LLC/eLLC Allowed 11: L3, LLC, eLLC Allowed
2	**Encrypted Data** This field controls whether data is decrypted while being read. This field is ignored for writes. Format = Enable
1:0	**Age for QUADLRU (AGE)** This field allows the selection of AGE parameter for a given surface in LLC or eLLC. . If a particular allocation is done at youngest age ("3") it tends to stay longer in the cache as compared to older age allocations ("2", "1", or "0"). This option is given to driver to be able to decide which surfaces are more likely to generate HITs, hence need to be replaced least often in caches. 11: Good chance of generating hits. 10: Next good chance of generating hits 01: Decent chance of generating hits 00: Poor chance of generating hits

Architectural Memory Types

Memory typing is decided via several levels of checks and comparing different priority levels. The following table shows a visual mapping between these selections:

Context Type	2nd Level Translation	CD	EMTE	IGPT	Surface State	Private PAT	MTRR	IA32e PAT	EMT	Effective Memory Type
Legacy Context	X	X	X	X	UC	UC	X	X	X	UC
	X	X	X	X		WC	X	X	X	WC
	X	X	X	X		WT	X	X	X	WT
	X	X	X	X		WB	X	X	X	WB
	X	X	X	X	WC	UC	X	X	X	WC
	X	X	X	X		WC	X	X	X	
	X	X	X	X		WT	X	X	X	
	X	X	X	X		WB	X	X	X	
	X	X	X	X	WT	UC	X	X	X	WT
	X	X	X	X		WC	X	X	X	
	X	X	X	X		WT	X	X	X	
	X	X	X	X		WB	X	X	X	
	X	X	X	X	WB	UC	X	X	X	WB
	X	X	X	X		WC	X	X	X	
	X	X	X	X		WT	X	X	X	
	X	X	X	X		WB	X	X	X	
Advanced Context	0 (disabled)	0	X	X	X	X	UC	UC	X	UC
			X	X	X	X		UC-	X	UC
			X	X	X	X		WC	X	WC
			X	X	X	X		WT	X	UC
			X	X	X	X		WB	X	UC
			X	X	X	X		WP	X	UC
			X	X	X	X	WC	UC	X	UC
			X	X	X	X		UC-	X	WC
			X	X	X	X		WC	X	WC
			X	X	X	X		WT	X	UC
			X	X	X	X		WB	X	WC
			X	X	X	X		WP	X	UC
			X	X	X	X	WT	UC	X	UC
			X	X	X	X		UC-	X	UC
			X	X	X	X		WC	X	WC
			X	X	X	X		WT	X	WT
			X	X	X	X		WB	X	WT
			X	X	X	X		WP	X	WP
			X	X	X	X	WB	UC	X	UC
			X	X	X	X		UC-	X	UC
			X	X	X	X		WC	X	WC
			X	X	X	X		WT	X	WT
			X	X	X	X		WB	X	WB
			X	X	X	X		WP	X	WP
			X	X	X	X	WP	UC	X	UC
			X	X	X	X		UC-	X	WC
			X	X	X	X		WC	X	WC
			X	X	X	X		WT	X	WT
			X	X	X	X		WB	X	WP
			X	X	X	X		WP	X	WP

Context Type	2nd Level Translation	CD	EMTE	IGPT	Surface State	Private PAT	MTRR	IA32e PAT	EMT	Effective Memory Type
Advanced Context	1 (enabled)	0	0	X	X	X	UC	UC	X	UC
				X	X	X		UC-	X	UC
				X	X	X		WC	X	WC
				X	X	X		WT	X	UC
				X	X	X		WB	X	UC
				X	X	X		WP	X	UC
				X	X	X	WC	UC	X	UC
				X	X	X		UC-	X	WC
				X	X	X		WC	X	WC
				X	X	X		WT	X	UC
				X	X	X		WB	X	WC
				X	X	X		WP	X	UC
				X	X	X	WT	UC	X	UC
				X	X	X		UC-	X	UC
				X	X	X		WC	X	WC
				X	X	X		WT	X	WT
				X	X	X		WB	X	WT
				X	X	X		WP	X	WP
				X	X	X	WB	UC	X	UC
				X	X	X		UC-	X	UC
				X	X	X		WC	X	WC
				X	X	X		WT	X	WT
				X	X	X		WB	X	WB
				X	X	X		WP	X	WP
				X	X	X	WP	UC	X	UC
				X	X	X		UC-	X	WC
				X	X	X		WC	X	WC
				X	X	X		WT	X	WT
				X	X	X		WB	X	WP
				X	X	X		WP	X	WP
Advanced Context	1 (enabled)	0	1	0	X	X	X	UC	UC	UC
					X	X	X	UC-		UC
					X	X	X	WC		WC
					X	X	X	WT		UC
					X	X	X	WB		UC
					X	X	X	WP		UC
					X	X	X	UC	WC	UC
					X	X	X	UC-		WC
					X	X	X	WC		WC
					X	X	X	WT		UC
					X	X	X	WB		WC
					X	X	X	WP		UC
					X	X	X	UC	WT	UC
					X	X	X	UC-		UC
					X	X	X	WC		WC
					X	X	X	WT		WT
					X	X	X	WB		WT
					X	X	X	WP		WP
					X	X	X	UC	WB	UC
					X	X	X	UC-		UC
					X	X	X	WC		WC
					X	X	X	WT		WT
					X	X	X	WB		WB
					X	X	X	WP		WP
					X	X	X	UC	WP	UC
					X	X	X	UC-		WC
					X	X	X	WC		WC
					X	X	X	WT		WT
					X	X	X	WB		WP
					X	X	X	WP		WP
Advanced Context	1 (enabled)	0	1	1	X	X	X	X	UC	UC
					X	X	X	X	WC	WC
					X	X	X	X	WT	WT
					X	X	X	X	WB	WB
					X	X	X	X	WP	WP



Context Type	2nd Level Translation	CD	EMTE	IGPT	Surface State	Private PAT	MTRR	IA32e PAT	EMT	Effective Memory Type
Advanced Context	X	1	X	X	X	X	X	X	X	UC

Page Walker Access and Memory Types

Most of these notes are further explained in the document; however summarized as part of the page table behavior:

Page Walker Memory Types

1. Legacy Contexts
   a. GT access to root/extended-root table and context/extended-context table
   b. GTT access to private paging (PPGTT) entries
   c. GT access to GPA-to-HPA paging entries
   d. GT access to the translated page
2. Advanced context (without nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
3. Advanced context (with nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
   e. GT access to GPA-to-HPA paging entries to translate address of PASID-entry and PASID-state entry
   f. GT access to GPA-to-HPA paging entries to translate address of IA-32e paging entries
   g. GT access to GPA-to-HPA paging entries to translate address of page

For gen8, the following behavior is defined:

Error Cases

- A/D bit update attempt for paging entry in non-WB memory, cause page-walk to be aborted; Error reported to device in Translation Response, gets reported to driver as GPGPU context in error – catastrophic error case.

- Locked/Atomic operations to pages in non-WB memory aborted; gets reported to driver as GPGPU context in error (catastrophic error)
- CD=1 treated same as non-WB memory, for above lock behavior

Replacement

TLB replacements during runtime are based on LRA algorithm; in addition, invalidations and page responses will have to invalidate the TLB entries.

Invalidations of TLB

There are various ways to invalidate TLBs:

1. **Traditional invalidation from command streamer**: Could be part of any fence accesses including newly added atomics.
2. **SVM based invalidations**: Listed as part of the new SVM related invalidations, various stages of TLBs including intermediate stages can be invalidated selectively and/or as a whole.
3. **Context Switch**: A context switch has to invalidate caches to make sure we have no residual value of the TLBs across multiple PASIDs. GAM will treat the context reload message from CS as a form of TLB invalidation.
4. **A page response**: should invalidate faulted recordings. It should be done via address matching to kick the faulted entries within the matching PASID.

Invalidation response “Invalidation Wait Descriptor” should also be a fence for both READs and WRITEs that used the previous TLB entries. Gam can only respond to “invalidation wait descriptor” after getting a GTI EMPTY indication.

Optional Invalidations

The following cases are listed as page table updates which software may choose not to invalidate the TLBs.

- If a paging-structure is modified to change the Present (Valid) flag from 0 to 1, s/w may choose not to invalidate TLBs. This affects only the case where GPU keeps the faulted page in its TLB to filter out future faults. Regardless of s/w does invalidation or not, for the cases where h/w cares, there will be a page response from s/w which will be used to shootdown the faulted record from the TLB.

  GAM will only put faulted entries to its TLBs if there has been page request for it. This would mean only faultable surfaces can be stored in GAM TLBs as a faulted entry.

- If a paging-structure entry is modified to change the accessed flag from 0 to 1,no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging structure entry in which the accessed flag is 0.
- If a paging-structure entry is modified to change the R/W or U/S or XD flag from 0 to 1, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address

Faulting

Page Faults

CHV, BSW does not support page faulting

Memory Types and Cache Interface

This section has additional information on the types of memory which are accessible via the various GT mechanisms. It includes discussion on how the various paging models are used and accessed. See the Graphics Translation Tables for more detailed discussions on paging models.

This section also includes descriptions of how different surface types (MOCS) can be cached in the L3 and the different behaviors which can be enabled.

Memory Object Control State (MOCS)

The memory object control state defines the behavior of memory accesses beyond the graphics core, including encryption, graphics data types that allow selective flushing of data from outer caches, and controlling cacheability in the outer caches.

This control uses several mechanisms. Control state for all memory accesses can be defined page by page in the GTT entries. Memory objects that are defined by state per surface generally have additional memory object control state in the state structure that defines the other surface attributes. Memory objects without state defining them have memory object state control defined per class in the STATE_BASE_ADDRESS command, with class divisions the same as the base addresses. Finally, some memory objects only have the GTT entry mechanism for defining this control. The table below enumerates the memory objects and the location of the control state for each:

Memory Object	Location of Control State
surfaces defined by SURFACE_STATE: sampling engine surfaces, render targets, media surfaces, pull constant buffers, streamed vertex buffers	SURFACE_STATE
depth, stencil, and hierarchical depth buffers	corresponding state command that defined the buffer attributes
stateless buffers accessed by data port	STATE_BASE_ADDRESS
indirect state objects	STATE_BASE_ADDRESS
kernel instructions	STATE_BASE_ADDRESS
push constant buffers	3DSTATE_CONSTANT_(VS | GS | PS)
index buffers	3DSTATE_INDEX_BUFFER
vertex buffers	3DSTATE_VERTEX_BUFFERS
indirect media object	STATE_BASE_ADDRESS
generic state prefetch	GTT control only
ring/batch buffers	GTT control only
context save buffers	GTT control only
store DWord	GTT control only

MOCS Registers

These registers provide the detailed format of the MOCS table entries that need to be programmed to define each surface state.

MEMORY_OBJECT_CONTROL_STATE

MEMORY_OBJECT_CONTROL_STATE

Page Walker Access and Memory Types

Most of these notes are further explained in the document however summarized as part of the page table behavior:

Page Walker Memory Types

1. Legacy Contexts
    a. GT access to root/extended-root table and context/extended-context table
    b. GTT access to private paging (PPGTT) entries
    c. GT access to GPA-to-HPA paging entries
    d. GT access to the translated page
2. Advanced context (without nesting)
    a. GT access to extended-root table and extended-context table
    b. GT access to PASID-entry & PASID-state entry
    c. GT access to IA-32e paging entries
    d. GT access to the translated page
3. Advanced context (with nesting)
    a. GT access to extended-root table and extended-context table
    b. GT access to PASID-entry & PASID-state entry
    c. GT access to IA-32e paging entries
    d. GT access to the translated page
    e. GT access to GPA-to-HPA paging entries to translate address of PASID-entry and PASID-state entry
    f. GT access to GPA-to-HPA paging entries to translate address of IA-32e paging entries
    g. GT access to GPA-to-HPA paging entries to translate address of page

Gen8 Memory Typing for Paging

The following information is duplicated in the Page Walker Memory Types topic:

1. Legacy Contexts
   a. GT access to root/extended-root table and context/extended-context table
   b. GTT access to private paging (PPGTT) entries
   c. GT access to GPA-to-HPA paging entries
   d. GT access to the translated page
2. Advanced context (without nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
3. Advanced context (with nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
   e. GT access to GPA-to-HPA paging entries to translate address of PASID-entry and PASID-state entry
   f. GT access to GPA-to-HPA paging entries to translate address of IA-32e paging entries
   g. GT access to GPA-to-HPA paging entries to translate address of page

This information is new in this topic and references the cases and subcases enumerated above:

For case [1]:

- [1.a] is always covered as a non-cacheable access
- [1.b] & [1.c] is covered with MMIO register where PPGTT entries can be forced to be cached in LLC (default option is cached).
- [1.d] is defined via private PAT (MMIO based) and surface state.

For case [2]:

- [2.a] is always covered as a non-cacheable access
- [2.b] is always cached & PASID state table entry is always accessed "atomically"
- [2.c] is accessed as cached
- [2.d] use memory-type as evaluated through MTRR, CD, and PCD/PWT/PAT bits in leaf IA-32e paging entry

For case [3]:

- [3.a] is always covered as a non-cacheable access
- [3.b] is always cached & PASID state table entry is always accessed "atomically"
- [3.c] is accessed as cached
- [3.d] use memory-type as follows (this section is further described in detail in memory typing section)
    - If CD=1, memory-type is UC
    - If CD=0:
        - If EMTE=0 in extended-context-entry, it is handled same as [2.d]
        - If EMTE=1 in extended-context-entry:
            - If IGMT=1 in leaf GPA-to-HPA entry, memory type used is the EMT field in this GPA-to-HPA entry.
            - If IGMT=0 in leaf GPA-to-HPA entry, memory type from [2.d] is combined with EMT field in this GPA-to-HPA entry.
- [3.e] is always cached & PASID state table entry is always accessed "atomically"
- [3.f] &[3.g] is accessed as cached

Error Cases

- A/D bit update attempt for paging entry in non-WB memory, causes page-walk to be aborted; Error reported to device in Translation Response; For Gen, gets reported to driver as GPGPU context in error – catastrophic error case.
- Locked/Atomic operations to pages in non-WB memory aborted; For Gen, gets reported to driver as GPGPU context in error (catastrophic error).
- CD=1 treated same as non-WB memory, for above lock behavior.

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 (same as R1_UINT) and MONO8

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

MONO8 format is identical to R1_UNORM but has different semantics for filtering. MONO8 is the only supported format for the MAPFILTER_MONO filter. See the *Sampling Engine* chapter.

Palette Formats

Palette formats are supported by the sampling engine. These formats include an index into the palette (Px) that selects the actual channel values from the palette, which is loaded via the 3DSTATE_SAMPLER_PALETTE_LOAD0 command.

P4A4_UNORM

This surface format contains a 4-bit Alpha value (in the high nibble) and a 4-bit Palette Index value (in the low nibble).

7			4	3			0
Alpha				Palette Index			

Bit	Description
7:4	**Alpha** Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by 255 to yield a [0.0,1.0] Alpha value. Format: U4
3:0	**Palette Index** A 4-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx) Format: U4

A4P4_UNORM

This surface format contains a 4-bit Alpha value (in the low nibble) and a 4-bit Color Index value (in the high nibble).

7			4	3			0
Palette Index				Alpha			

Bit	Description
7:4	**Palette Index** A 4-bit color index which is used to lookup a 24-bit RGB value in the texture palette. Format: U4
3:0	**Alpha** Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by

Bit	Description
	255 to yield a [0.0,1.0] alpha value. Format: U4

P8A8_UNORM

This surface format contains an 8-bit Alpha value (in the high byte) and an 8-bit Palette Index value (in the low byte).

15			8	7			0
Alpha				Palette Index			

Bit	Description
15:8	**Alpha** Alpha value which will be divided by 255 to yield a [0.0,1.0] Alpha value. Format: U8
7:0	**Palette Index** An 8-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx) Format: U8

A8P8_UNORM

This surface format contains an 8-bit Alpha value (in the low byte) and an 8-bit Color Index value (in the high byte).

15			8	7			0
Palette Index				Alpha			

Bit	Description
15:8	**Palette Index** An 8-bit color index which is used to lookup a 24-bit RGB value in the texture palette. Format: U8
7:0	**Alpha** Alpha value which will be divided by 255 to yield a [0.0,1.0] alpha value. Format: U8

P8_UNORM

This surface format contains only an 8-bit Color Index value.

Bit	Description
7:0	**Palette Index** An 8-bit color index which is used to lookup a 32-bit ARGB value in the texture palette. Format: U8

P2_UNORM

This surface format contains only a 2-bit Color Index value.

Bit	Description
1:0	**Palette Index** A 2-bit color index which is used to lookup a 32-bit ARGB value in the texture palette. Format: U2

Compressed Surface Formats

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

ETC1_RGB8

CHV, BSW: 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
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

Bits	Description
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]
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

Bits	Description
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]

Bits	Description
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]
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
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)` R_i `= R0*8 + 4 +` im_i else R_i `= R0*8 + 4 + (`im_i `* m * 8)`

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

For EAC_SIGNED_R11:

`if (m == 0)` R_i `= R0*8 +` im_i else R_i `= R0*8 + (`im_i `* 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 of these 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
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

Bits	Description
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

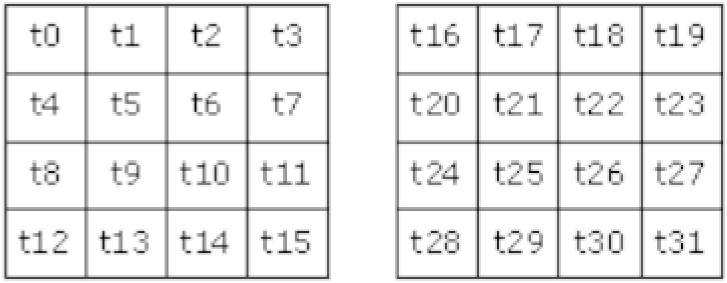
These compression formats are identical to the EAC_R11 and EAC_SIGNED_R11 formats, except that they supply two channels of output data, both red and green, from two independent 8-byte portions of the compression block. The low half of the compression block contains the red information, and the high half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

FXT Texture Formats

There are four different FXT1 compressed texture formats. Each of the formats compress two 4x4 texel blocks into 128 bits. In each compression format, the 32 texels in the two 4x4 blocks are arranged according to the following diagram:

FXT1 Encoded Blocks



t0	t1	t2	t3
t4	t5	t6	t7
t8	t9	t10	t11
t12	t13	t14	t15

t16	t17	t18	t19
t20	t21	t22	t23
t24	t25	t26	t27
t28	t29	t30	t31

B6682-01

Overview of FXT1 Formats

During the compression phase, the encoder selects one of the four formats for each block based on which encoding scheme results in best overall visual quality. The following table lists the four different modes and their encodings:

FXT1 Format Summary

Bit 127	Bit 126	Bit 125	Block Compression Mode	Summary Description
0	0	X	**CC_HI**	2 R5G5B5 colors supplied. Single LUT with 7 interpolated color values and transparent black
0	1	0	**CC_CHROMA**	4 R5G5B5 colors used directly as 4-entry LUT.
0	1	1	**CC_ALPHA**	3 A5R5G5B5 colors supplied. LERP bit selects between 1 LUT with 3 discrete colors + transparent black and 2 LUTs using interpolated values of Color 0,1 (t0-15) and Color 1,2 (t16-31).
1	x	x	**CC_MIXED**	4 R5G5B5 colors supplied, where Color0,1 LUT is used for t0-t15, and Color2,3 LUT used for t16-31. Alpha bit selects between LUTs with 4 interpolated colors or 3 interpolated colors + transparent black.

FXT1 CC_HI Format

In the CC_HI encoding format, two base 15-bit R5G5B5 colors (Color 0, Color 1) are included in the encoded block. These base colors are then expanded (using high-order bit replication) to 24-bit RGB colors, and used to define an 8-entry lookup table of interpolated color values (the 8th entry is transparent black). The encoded block contains a 3-bit index value per texel that is used to lookup a color from the table.

CC_HI Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_HI block format:

FXT CC_HI Block Encoding

Bit	Description
127:126	Mode = '00'b (CC_HI)
125:121	Color 1 Red
120:116	Color 1 Green
115:111	Color 1 Blue
110:106	Color 0 Red
105:101	Color 0 Green
100:96	Color 0 Blue
95:93	Texel 31 Select
...	...
50:48	Texel 16 Select
47:45	Texel 15 Select
...	...
2:0	Texel 0 Select

CC_HI Block Decoding

The two base colors, Color 0 and Color 1 are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

FXT CC_HI Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [23:19]	Color 1 Red [7:3]	[125:121]
Color 1 [18:16]	Color 1 Red [2:0]	[125:123]
Color 1 [15:11]	Color 1 Green [7:3]	[120:116]
Color 1 [10:08]	Color 1 Green [2:0]	[120:118]
Color 1 [07:03]	Color 1 Blue [7:3]	[115:111]
Color 1 [02:00]	Color 1 Blue [2:0]	[115:113]
Color 0 [23:19]	Color 0 Red [7:3]	[110:106]
Color 0 [18:16]	Color 0 Red [2:0]	[110:108]
Color 0 [15:11]	Color 0 Green [7:3]	[105:101]
Color 0 [10:08]	Color 0 Green [2:0]	[105:103]
Color 0 [07:03]	Color 0 Blue [7:3]	[100:96]
Color 0 [02:00]	Color 0 Blue [2:0]	[100:98]

These two 24-bit colors (Color 0, Color 1) are then used to create a table of seven interpolated colors (with Alpha = 0FFh), along with an eight entry equal to RGBA = 0,0,0,0, as shown in the following table:

FXT CC_HI Interpolated Color Table

Interpolated Color	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(5 * Color0.RGB + 1 * Color1.RGB + 3) / 6	0FFh
2	(4 * Color0.RGB + 2 * Color1.RGB + 3) / 6	0FFh
3	(3 * Color0.RGB + 3 * Color1.RGB + 3) / 6	0FFh
4	(2 * Color0.RGB + 4 * Color1.RGB + 3) / 6	0FFh
5	(1 * Color0.RGB + 5 * Color1.RGB + 3) / 6	0FFh
6	Color1.RGB	0FFh
7	RGB = 0,0,0	0

This table is then used as an 8-entry Lookup Table, where each 3-bit Texel n Select field of the encoded CC_HI block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_HI block.

FXT1 CC_CHROMA Format

In the CC_CHROMA encoding format, four 15-bit R5B5G5 colors are included in the encoded block. These colors are then expanded (using high-order bit replication) to form a 4-entry table of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

CC_CHROMA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_CHROMA block format:

FXT CC_CHROMA Block Encoding

Bit	Description
127:125	Mode = ‘010’b (CC_CHROMA)
124	Unused
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red

Bit	Description
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
…	
33:32	Texel 16 Select
31:30	Texel 15 Select
…	
1:0	Texel 0 Select

CC_CHROMA Block Decoding

The four colors (Color 0-3) are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

FXT CC_CHROMA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10:08]	Color 3 Green [2:0]	[118:116]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

This table is then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_CHROMA block is used to index into a 32-bit A8R8G8B8 color from the table (Alpha defaults to 0FFh) completing the decode of the CC_CHROMA block.

FXT CC_CHROMA Interpolated Color Table

Texel Select	Color ARGB
0	Color0.ARGB
1	Color1.ARGB
2	Color2.ARGB
3	Color3.ARGB

FXT1 CC_MIXED Format

In the CC_MIXED encoding format, four 15-bit R5G5B5 colors are included in the encoded block: Color 0 and Color 1 are used for Texels 0-15, and Color 2 and Color 3 are used for Texels 16-31.

Each pair of colors are then expanded (using high-order bit replication) to form 4-entry tables of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

CC_MIXED Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_MIXED block format:

FXT CC_MIXED Block Encoding

Bit	Description
127	Mode = ‘1’b (CC_MIXED)
126	Color 3 Green [0]
125	Color 1 Green [0]
124	Alpha [0]
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red

Bit	Description
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	...
33:32	Texel 16 Select
31:30	Texel 15 Select
...	...
1:0	Texel 0 Select

CC_MIXED Block Decoding

The decode of the CC_MIXED block is modified by Bit 124 (Alpha [0]) of the encoded block.

Alpha[0] = 0 Decoding

When Alpha[0] = 0 the four colors are encoded as 16-bit R5G6B5 values, with the Green LSB defined as per the following table:

FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 2 Green [0]	Encoded Bit [33] XOR Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]
Color 0 Green [0]	Encoded Bit [1] XOR Encoded Bit [125]

The four colors (Color 0-3) are then converted from R5G5B6 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

FXT CC_MIXED Decoded Colors (Alpha[0] = 0)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10]	Color 2 Green [2]	[33] XOR [126]]
Color 2 [09:08]	Color 2 Green [1:0]	[103:100]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]
Color 1 [09:08]	Color 1 Green [1:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10]	Color 0 Green [2]	[1] XOR [125]
Color 0 [09:08]	Color 0 Green [1:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four interpolated colors (with Alpha = 0FFh). The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices, as shown in the following figures:

FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(2*Color0.RGB + Color1.RGB + 1) /3	0FFh
2	(Color0.RGB + 2*Color1.RGB + 1) /3	0FFh
3	Color1.RGB	0FFh

FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	(2/3) * Color2.RGB + (1/3) * Color3.RGB	0FFh
2	(1/3) * Color2.RGB + (2/3) * Color3.RGB	0FFh
3	Color3.RGB	0FFh

Alpha[0] = 1 Decoding

When Alpha[0] = 1, Color0 and Color2 are encoded as 15-bit R5G5B5 values. Color1 and Color3 are encoded as RGB565 colors, with the Green LSB obtained as shown in the following table:

FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]

All four colors are then expanded to 24-bit R8G8B8 colors by bit replication, as show in the following diagram.

FXT CC_MIXED Decoded Colors (Alpha[0] = 1)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:19]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]
Color 1 [09:08]	Color 1 Green [1:0]	[88:87]

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:19]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices. The color at index 1 is the linear interpolation of the base colors, while the color at index 3 is defined as Black (0,0,0) with Alpha = 0, as shown in the following figures:

FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(Color0.RGB + Color1.RGB) /2	0FFh
2	Color1.RGB	0FFh
3	Black (0,0,0)	0

FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	(Color2.RGB + Color3.RGB) /2	0FFh
2	Color3.RGB	0FFh
3	Black (0,0,0)	0

These tables are then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_MIXED block is used to index into the appropriate 32-bit A8R8G8B8 color from the table, completing the decode of the CC_CMIXED block.

FXT1 CC_ALPHA Format

In the CC_ALPHA encoding format, three A5R5G5B5 colors are provided in the encoded block. A control bit (LERP) is used to define the lookup table (or tables) used to dereference the 2-bit Texel Selects.

CC_ALPHA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_ALPHA block format:

FXT CC_ALPHA Block Encoding

Bit	Description
127:125	Mode = ‘011’b (CC_ALPHA)
124	LERP
123:119	Color 2 Alpha
118:114	Color 1 Alpha
113:109	Color 0 Alpha
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
…	…
33:32	Texel 16 Select
31:30	Texel 15 Select
…	…
1:0	Texel 0 Select

CC_ALPHA Block Decoding

Each of the three colors (Color 0-2) are converted from A5R5G5B5 to A8R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

FXT CC_ALPHA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 2 [31:27]	Color 2 Alpha [7:3]	[123:119]
Color 2 [26:24]	Color 2 Alpha [2:0]	[123:121]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [31:27]	Color 1 Alpha [7:3]	[118:114]
Color 1 [26:24]	Color 1 Alpha [2:0]	[118:116]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [31:27]	Color 0 Alpha [7:3]	[113:109]
Color 0 [26:24]	Color 0 Alpha [2:0]	[113:111]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

LERP = 0 Decoding

When LERP = 0, a single 4-entry lookup table is formed using the three expanded colors, with the 4th entry defined as transparent black (ARGB=0,0,0,0). Each 2-bit Texel n Select field of the encoded CC_ALPHA block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_ALPHA block.

FXT CC_ALPHA Interpolated Color Table (LERP=0)

Texel Select	Color	Alpha
0	Color0.RGB	Color0.Alpha
1	Color1.RGB	Color1.Alpha
2	Color2.RGB	Color2.Alpha
3	Black (RGB=0,0,0)	0

LERP = 1 Decoding

When LERP = 1, the three expanded colors are used to create two tables of four interpolated colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color1,2 table used for texels 16-31 indices, as shown in the following figures:

FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 0-15)

Texel 0-15 Select	Color ARGB
0	Color0.ARGB
1	(2*Color0.ARGB + Color1.ARGB + 1) /3
2	(Color0.ARGB + 2*Color1.ARGB + 1) /3
3	Color1.ARGB

FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 16-31)

Texel 16-31 Select	Color ARGB
0	Color2.ARGB
1	(2*Color2.ARGB + Color1.ARGB + 1) /3
2	(Color2.ARGB + 2*Color1.ARGB + 1) /3
3	Color1.ARGB

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

Word Address	16-bit Word
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]
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
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

Bit	Description
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

For CHV, BSW, 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]
39:35	rx[4:0]

Bit	Description
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

Bit	Description
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]
44:41	gy[3:0]
48:45	gx[3:0]
49	gw[10]

Bit	Description
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[3:0]
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

Bit	Description
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]

Bit	Description
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]
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]

Bit	Description
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]
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]

Bit	Description
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]

Bit	Description
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]
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
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

Bit	Description
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
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

Bit	Description
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
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

Bit	Description
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

Bit	Description
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
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

Bit	Description
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

Bit	Description
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

Bit	Description
	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	1	0	0	0	0	0	0	0	1	1
	1	1	1	0	0	0	0	0	[1]	0	0	0	0	0	0	1
	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.

Project	Supported Formats
CHV, BSW	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 the existing BC and ETC, and can reduce footprint & BW 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. ASTC also has a support of high dynamic range (HDR) image and 3D textures. 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.

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.

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
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

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)
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		001 110 001	ASTC_LDR_2D_10x5_FLT16	2.56
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

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)
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
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
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	ASTC_FULL_3D_3x3x3_FLT16	4.74

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)
					000		
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

Content for this topic is currently under development.

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:

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

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

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

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$_0$
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

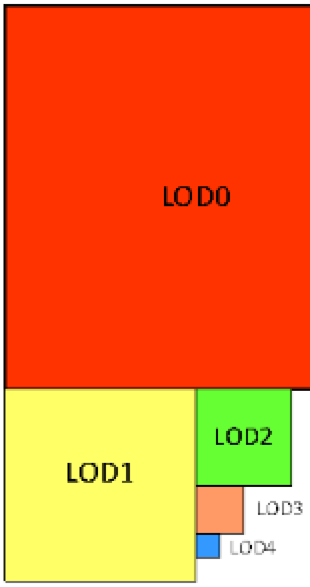
Block Size	ASTC Block Height (in line)	ASTC Block Width (in Byte)	ASTC Block Depth/slab (in slice)
3	((Ceil(HL /3) +3)/4) *4	Ceil(WL /3) * 16	Ceil(DL /3)
4	((Ceil(HL /4) +3)/4) *4	Ceil(WL /4) * 16	Ceil(DL /4)
5	((Ceil(HL /5) +3)/4) *4	Ceil(WL /5) * 16	Ceil(DL /5)
6	((Ceil(HL /6) +3)/4) *4	Ceil(WL /6)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

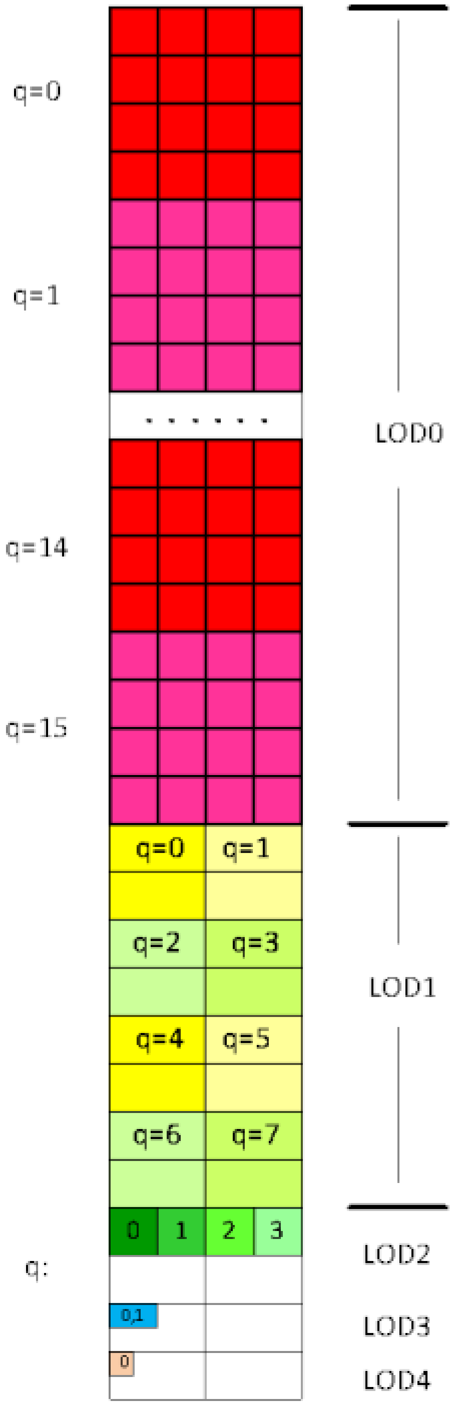
For example, an image of 64x64 with 5x5 block coding in LOD0 will have:

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.





ASTC Data Structure

Content for this topic is currently under development.

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

<table>
<tr><td>Partition Disable</td><td colspan="6">127:19</td><td colspan="2">18:17</td><td>16:13</td><td>12:11</td><td>10:0</td></tr>
<tr><td>(1 Plane)</td><td colspan="4">Index Data</td><td colspan="4">Color Endpoint Data</td><td>CEM</td><td>00</td><td>Index Mode</td></tr>
<tr><td>(2 Planes)</td><td colspan="4">Index Data</td><td colspan="2">Color Endpoint Data</td><td colspan="2">CCS</td><td>CEM</td><td>00</td><td>Index Mode</td></tr>
<tr><td colspan="12"></td></tr>
<tr><td>Multi-partitions</td><td colspan="5">127:29</td><td colspan="2">28:23</td><td colspan="2">22:13</td><td>12:11</td><td>10:0</td></tr>
<tr><td>(1 Plane)</td><td>Index Data</td><td colspan="2">Rest of CEM</td><td colspan="2">Color Endpoint Data</td><td colspan="2">CEM (Initial 6 bits)</td><td colspan="2">Partition Index</td><td>Part</td><td>Index Mode</td></tr>
<tr><td>(2 Planes)</td><td>Index Data</td><td>CCS</td><td>Rest of CEM</td><td colspan="2">Color Endpoint Data</td><td colspan="2">CEM (Initial 6 bits)</td><td colspan="2">Partition Index</td><td>Part</td><td>Index Mode</td></tr>
</table>

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.

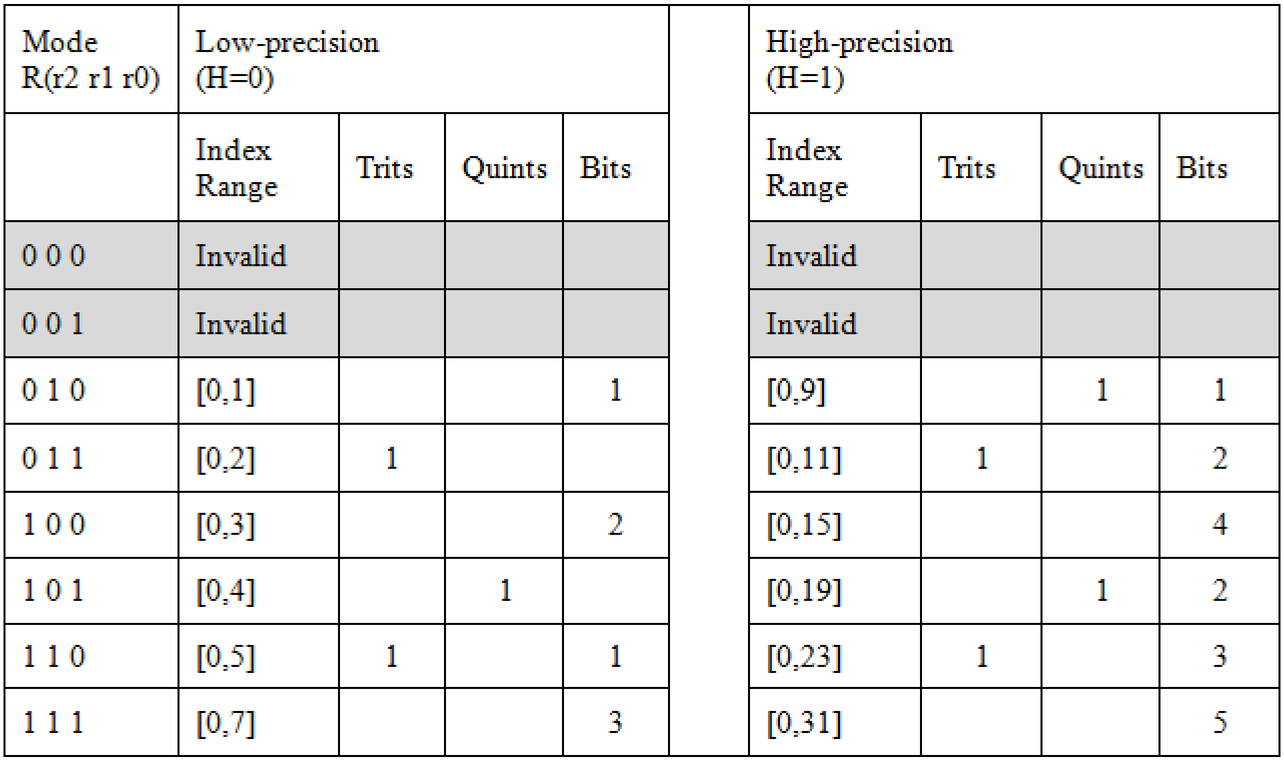
Partitioning

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	

Bits									Width	Height	Notes
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

CEM	Description	Class	# of integers to represent each pair of color end points
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
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

Project	Description
CHV, BSW	LDR modes are supported in ASTC LDR profile, which is enabled since CHV, BSW.

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 since CHV, BSW.

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

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

Content for this topic is currently under development.

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:

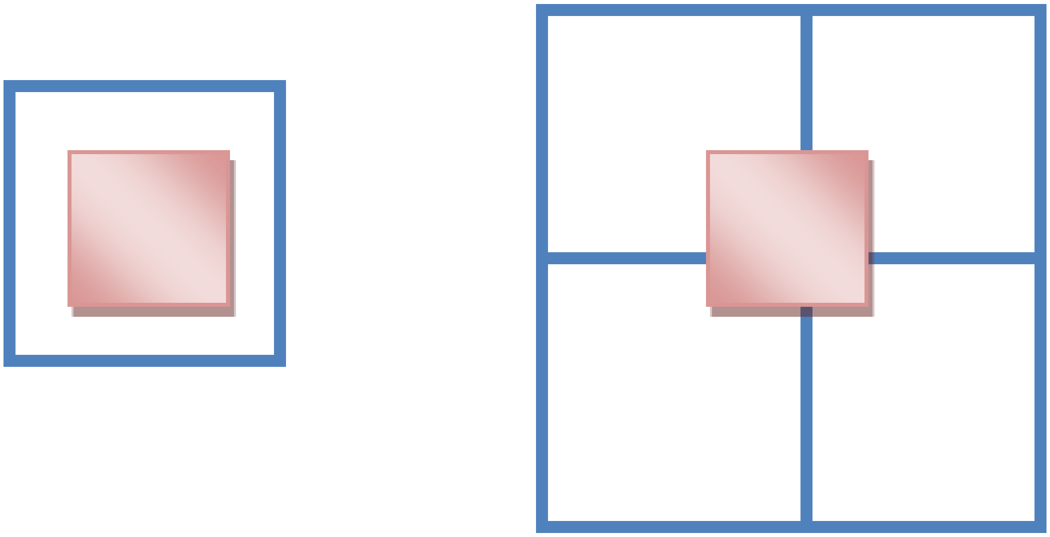
```
bw = MAX ( 2, (w+ tsize -1)/ tsize )
bh  = MAX (2, (h+ tsize -1)/ tsize)
```

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:

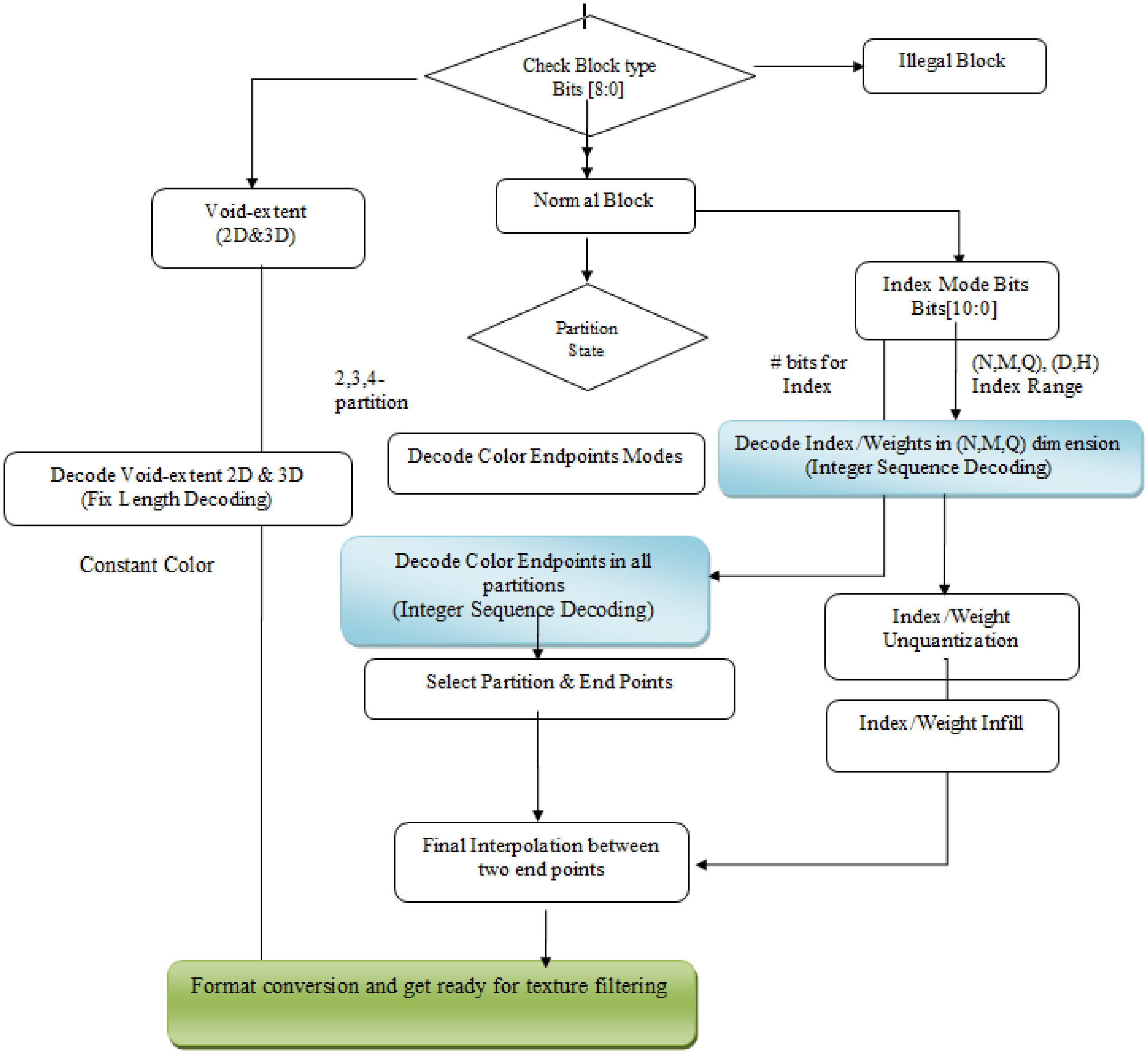
Front End Decoding Processing:

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:


Check Block type
Bits [8:0]
Illegal Block
Void-extent
(2D&3D)
Normal Block
Index Mode Bits
Bits[10:0]
Partition
State
2,3,4-
partition
# bits for
Index
(N,M,Q), (D,H)
Index Range
Decode Void-extent 2D & 3D
(Fix Length Decoding)
Decode Color Endpoints Modes
Decode Index/Weights in (N,M,Q) dimension
(Integer Sequence Decoding)
Constant Color
Decode Color Endpoints in all
partitions
(Integer Sequence Decoding)
Index/Weight
Unquantization
Select Partition & End Points
Index/Weight Infill
Final Interpolation between
two end points
Format conversion and get ready for texture filtering

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

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); }

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); }

CEM	Range	Description	# of end points	Endpoints Reconstruction
				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

	Number of bits				Description of Extra Bits						
Mode	**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[4]	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[4]	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 216 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.

Delete?

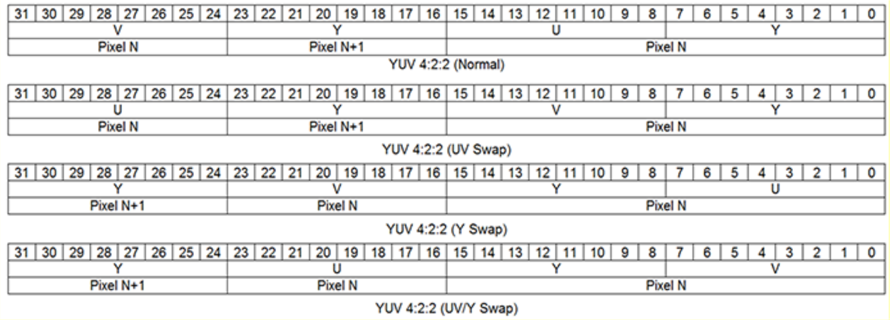
The following four YUV 4:2:2 surface formats are supported, listed with alternate names:

- 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



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V								Y								U								Y							
Pixel N								Pixel N+1								Pixel N															

YUV 4:2:2 (Normal)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U								Y								V								Y							
Pixel N								Pixel N+1								Pixel N															

YUV 4:2:2 (UV Swap)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y								V								Y								U							
Pixel N+1								Pixel N								Pixel N															

YUV 4:2:2 (Y Swap)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y								U								Y								V							
Pixel N+1								Pixel N								Pixel N															

YUV 4:2:2 (UV/Y Swap)

Planar Memory Organization

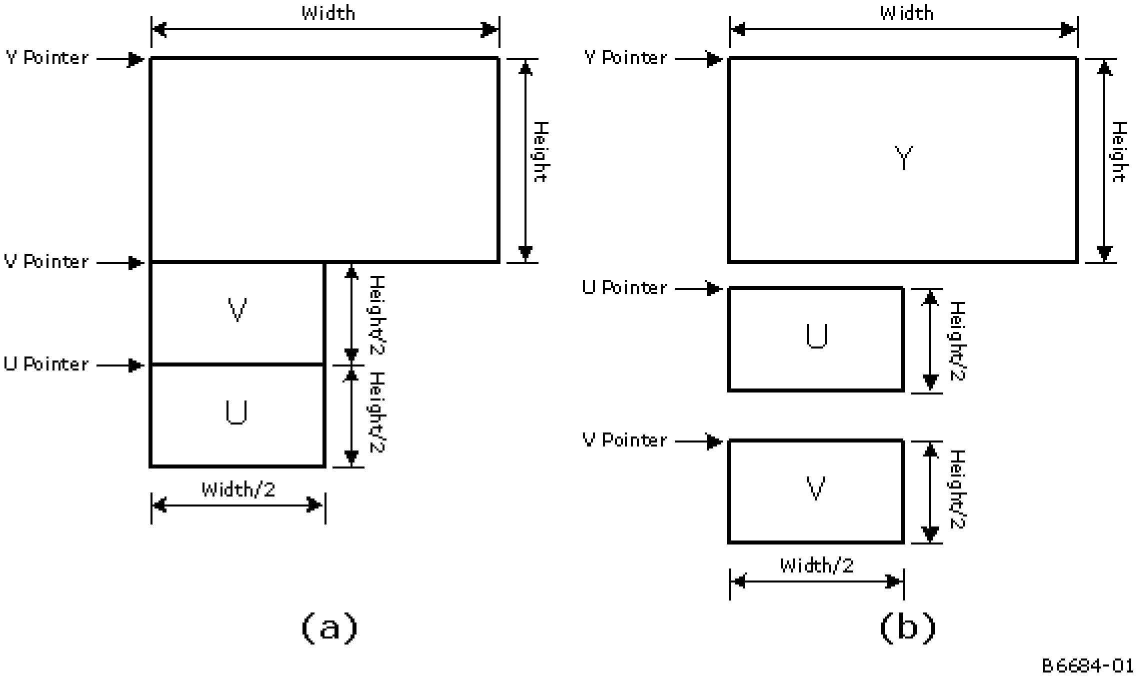
Planar formats use what could be thought of as separate buffers for the three color components. Because there is a separate stride for the Y and U/V data buffers, several memory footprints can be supported.

Note: There is no direct support for use of planar video surfaces as textures. The sampling engine can be used to operate on each of the 8bpp buffers separately (via a single-channel 8-bit format such as I8_UNORM). The U and V buffers can be written concurrently by using multiple render targets from the pixel shader. The Y buffer must be written in a separate pass due to its different size.

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.

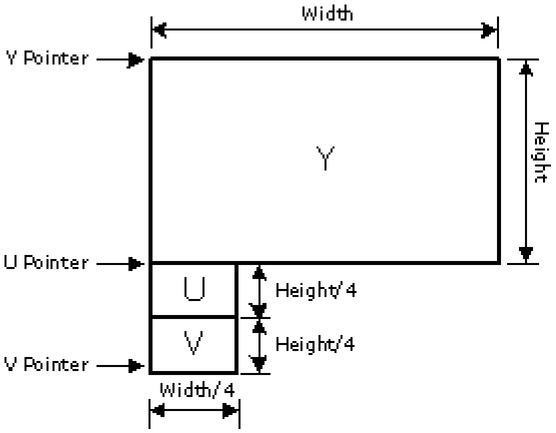
YUV 4:2:0 Format Memory Organization



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. In general, YV12 is supported only in linear format because separate planes cannot be supported correctly with a tiled format.

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 planes in memory when Y Height is a multiple of 4 or when Interleaved Chroma (e.g. NV21) is used.

Case	Interleave Chroma	Pitch	Vertical U/V Offset	Restrictions

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
				(Y Height)%4 = 0
				(U Height)%2 = 0
				(V Height)%2 = 0
				Vertical for Y surface must be 0

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
				(Y Height)%2 = 0
				(U Height)%2 = 0
				(V Height)%2 = 0

YUV for Media Sampling	Yes	Always Full	Same as 3D full pitch	Same as 3D full pitch

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.

UNRESOLVED CROSS REFERENCE, 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

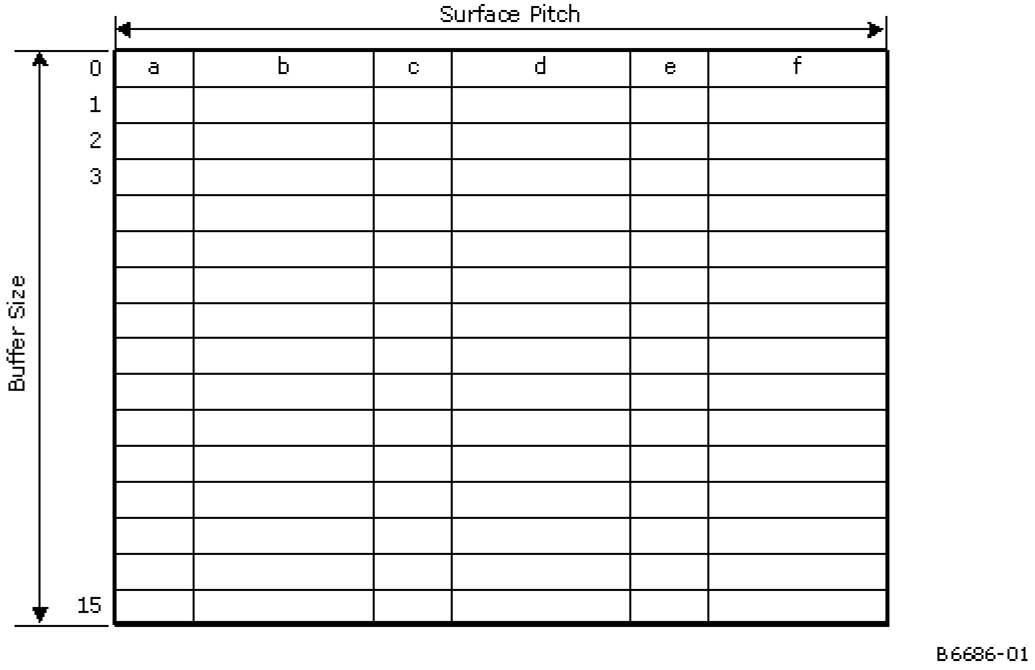
Surface Layout

In addition to restrictions on maximum height, width, and depth, surfaces are also restricted to a maximum size in bytes. This maximum is 2 GB for all products and all surface types.

Buffers

A 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.



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**.

1D Surfaces

One-dimensional surfaces are identical to 2D surfaces with height of one. Arrays of 1D surfaces are also supported. Please refer to the 2D Surfaces section for details on how these surfaces are stored.

2D Surfaces

Surfaces that comprise texture mip-maps are stored in a fixed “monolithic” format and referenced by a single base address. The base map and associated mipmaps are located within a single rectangular area of memory identified by the base address of the upper left corner and a pitch. The base address references the upper left corner of the base map. The pitch must be specified at least as large as the widest mip-map. In some cases it must be wider; see the section on Minimum Pitch below.

These surfaces may be overlapped in memory and must adhere to the following memory organization rules:

- For non-compressed texture formats, each mipmap must start on an even row within the monolithic rectangular area. For 1-texel-high mipmaps, this may require a row of padding below the previous mipmap. This restriction does not apply to any compressed texture formats; each subsequent (lower-res) compressed mipmap is positioned directly below the previous mipmap.
- Vertical alignment restrictions vary with memory tiling type: 1 DWord for linear, 16-byte (DQWord) for tiled. (Note that tiled mipmaps are *not* required to start at the left edge of a tile row.)

Computing MIP Level Sizes

Map width and height specify the size of the largest MIP level (LOD 0). Less detailed LOD level (i+1) sizes are determined by dividing the width and height of the current (i) LOD level by 2 and truncating to an integer (floor). This is equivalent to shifting the width/height by 1 bit to the right and discarding the bit shifted off. The map height and width are clamped on the low side at 1.

In equations, the width and height of an LOD "*L*" can be expressed as:

$W_L = ((\mathit{width} \gg L) > 0?\ \mathit{width} \gg L:1)$

$H_L = ((\mathit{height} \gg L) > 0?\ \mathit{height} \gg L:1)$

If the surface is multisampled and it is a depth or stencil surface or **Multisampled Surface StorageFormat** in SURFACE_STATE is MSFMT_DEPTH_STENCIL, W_L and H_L must be adjusted as follows before proceeding:

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) * 4d
16	ceiling(W_L / 2) * 8	ceiling(H_L / 2) * 8

Base Address for LOD Calculation

It is conceptually easier to think of the space that the map uses in Cartesian space (x, y), where x and y are in units of texels, with the upper left corner of the base map at (0, 0). The final step is to convert from Cartesian coordinates to linear addresses as documented at the bottom of this section.

It is useful to think of the concept of "stepping" when considering where the next MIP level will be stored. We either step down or step right when moving to the next higher LOD.

- for MIPLAYOUT_RIGHT maps:
    - step right when moving from LOD 0 to LOD 1
    - step down for all of the other MIPs
- for MIPLAYOUT_BELOW maps:
    - step down when moving from LOD 0 to LOD 1
    - step right when moving from LOD 1 to LOD 2
    - step down for all of the other MIPs

To account for the cache line alignment required, we define *i* and *j* as the width and height, respectively, of an *alignment unit*. This alignment unit is defined below. We then define lower-case w_L and h_L as the padded width and height of LOD "*L*" as follows:

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

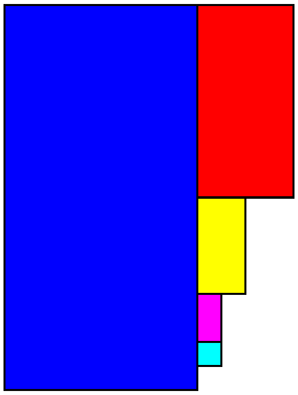
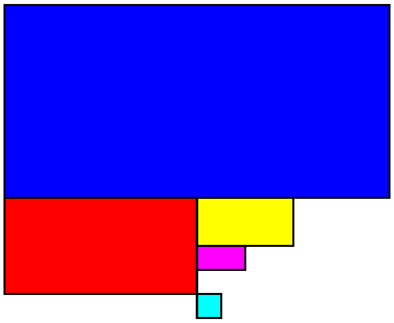
$$h_L = j * ceil\left(\frac{H_L}{j}\right)$$

For separate stencil buffer, the width must be mutiplied by 2 and height divided by 2 as follows:

$$w_l = 2 * i * ceil\left(\frac{W_L}{i}\right)$$

$$h_L = 1/2 * j * ceil\left(\frac{H_L}{j}\right)$$

Equations to compute the upper left corner of each MIP level are then as follows:

for *MIPLAYOUT_RIGHT* maps: $LOD_0 = (0,0)$ $LOD_1 = (w_0,0)$ $LOD_2 = (w_0,h_1)$ $LOD_3 = (w_0,h_1 + h_2)$ $LOD_4 = (w_0,h_1 + h_2 + h_3)$ …	
for *MIPLAYOUT_BELOW* maps: $LOD_0 = (0,0)$ $LOD_1 = (0,h_0)$ $LOD_2 = (w_1,h_0)$ $LOD_3 = (w_1,h_0 + h_2)$ $LOD_4 = (w_1,h_0 + h_2 + h_3)$ …	

Minimum Pitch for MIPLAYOUT_RIGHT and Other Maps

For MIPLAYOUT_RIGHT maps, the minimum pitch must be calculated before choosing a fence to place the map within. This is approximately equal to 1.5x the pitch required by the base map, with possible adjustments made for cache line alignment. For MIPLAYOUT_BELOW and MIPLAYOUT_LEGACY maps, the minimum pitch required is equal to that required by the base (LOD 0) map.

A safe but simple calculation of minimum pitch is equal to 2x the pitch required by the base map for MIPLAYOUT_RIGHT maps. This ensures that enough pitch is available, and since it is restricted to MIPLAYOUT_RIGHT maps, not much memory is wasted. It is up to the driver (hardware independent) whether to use this simple determination of pitch or a more complex one.

Cartesian to Linear Address Conversion

A set of variables are defined in addition to the i and j defined above.

- b = bytes per texel of the native map format (0.5 for DXT1, FXT1, and 4-bit surface format, 2.0 for YUV 4:2:2, others aligned to surface format)
- t = texel rows / memory row (4 for DXT1-5 and FXT1, 1 for all other formats)
- p = pitch in bytes (equal to pitch in dwords * 4)
- B = base address in bytes (address of texel 0,0 of the base map)
- x, y = cartestian coordinates from the above calculations in units of texels (assumed that x is always a multiple of i and y is a multiple of j)
- A = linear address in bytes

$$A = B + \frac{yp}{t} + xbt$$

This calculation gives the linear address in bytes for a given MIP level (taking into account L1 cache line alignment requirements).

Compressed Mipmap Layout

Mipmaps of textures using compressed (DXTn, FXT) texel formats are also stored in a monolithic format. The compressed mipmaps are stored in a similar fashion to uncompressed mipmaps, with each block of source (uncompressed) texels represented by a 1 or 2 QWord compressed block. The compressed blocks occupy the same logical positions as the texels they represent, where each row of compressed blocks represent a 4-high row of uncompressed texels. The format of the blocks is preserved, i.e., there is no “intermediate” format as required on some other devices.

The following exceptions apply to the layout of compressed (vs. uncompressed) mipmaps:

- Mipmaps are not required to start on even rows, therefore each successive mip level is located on the texel row immediately below the last row of the previous mip level. Pad rows are neither required nor allowed.

- The dimensions of the mip maps are first determined by applying the sizing algorithm presented in Non-Power-of-Two Mipmaps above. Then, if necessary, they are padded out to compression block boundaries.

Surface Arrays

Arrays of 1D and 2D surfaces can be treated as a single surface. This section covers the layout of these composite surfaces.

For All Surfaces

Both 1D and 2D surfaces can be specified as an array. An array surface is indicated by enabling the **Surface Array** field in SURFACE_STATE. 2D multisampled surfaces with Multisampled **Surface Storage Format** set to MSFMT_MSS also are stored like an array in memory.

A value QPitch is defined which indicates the worst-case height for one slice in the texture array. This QPitch is multiplied by the array index to and added to the vertical component of the address to determine the vertical component of the address for that slice. Within the slice, the map is stored identically to a 2D surface.

Since cube surfaces are stored identically to 2D arrays, QPitch is used to determine the spacing between faces of the cube.

For CMS/UMS multisampled surfaces, QPitch is used to determine the spacing between sample slices. For IMS multisampled surfaces, QPitch must account for the additional slice size due to sample storage.

For surfaces defined with SURFACE_STATE, The QPitch field in this state defines the value of QPitch. Software must ensure that QPitch is sufficiently large to avoid overlap between array slices in the memory layout. If an auxiliary surface is defined, a separate QPitch must be set for that surface.

For *depth* and stencil surfaces, QPitch is set in the corresponding state command.

Multisampled Surfaces

Multisampled render targets and sampling engine surfaces are supported. 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.

Compressed Multisampled Surfaces

Multisampled render targets can be compressed. If **MCS Enable** is enabled in SURFACE_STATE, 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).

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

For CHV, BSW, the 2x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

2x MCS [CHV, BSW]

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.

For CHV, BSW, the 4x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

4x MCS [CHV, BSW]

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.

For CHV, BSW, 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.

8x MCS [CHV, BSW]

31:24	23:21	20:18	17:15	14:12	11:9	8:6	5:3	2:0
reserved	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 8x algorithm is the same as the 4x algorithm. The MCS value indicating clear state is 0x00ffffff.

Physical 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.

Uncompressed Multisampled Surfaces

UMS surfaces similar to CMS, except that the MCS is disabled, and 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.

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.

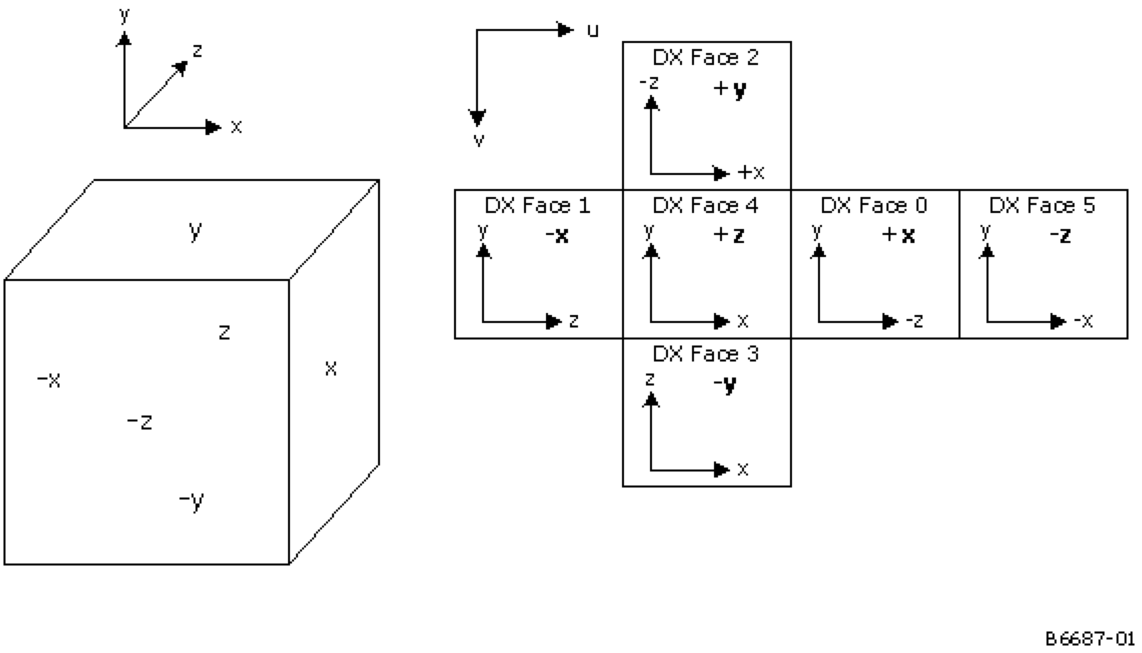
DirectX API Definition

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 origin of the U,V texel grid is at the top left corner of each face.

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

DirectX Cube Map Definition



Hardware Cube Map Layout

The cube face textures are stored in the same way as 2D array surfaces are stored (see section *2D Surfaces*for details). For cube surfaces, the depth (array instances) is equal to 6. The array index “q” corresponds to the face according to the following table:

“q” coordinate	face
0	+x
1	-x
2	+y
3	-y
4	+z
5	-z

Restrictions

- The cube map memory layout is the same whether or not the cube map is mip-mapped, and whether or not all six faces are “enabled”, though the memory backing disabled faces or non-supplied levels can be used by software for other purposes.
- The cube map faces all share the same **Surface Format**

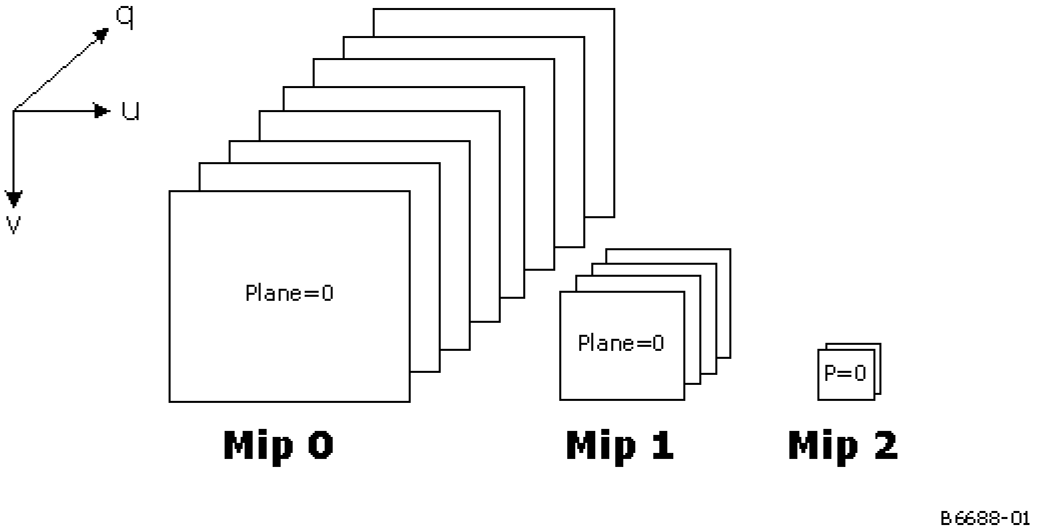
Cube Arrays

Cube arrays are stored identically to 2D surface arrays. A group of 6 consecutive array elements makes up a single cube map. A cube array with N array elements is stored identically to a 2D array with 6N array elements.

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



The number of planes defined at each successive mip level is halved. Volumetric texture maps are stored as follows. All of the LOD=0 q-planes are stacked vertically, then below that, the LOD=1 q-planes are stacked two-wide, then the LOD=2 q-planes are stacked four-wide below that, and so on.

The width, height, and depth of LOD "L" are as follows:

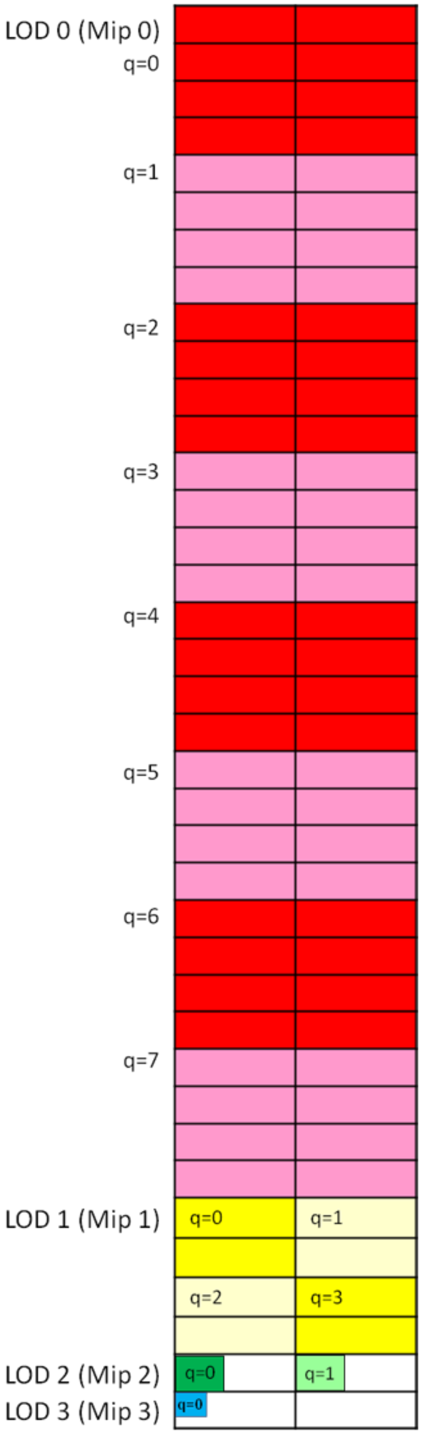
$W_L = ((width \gg L) > 0\ ?\ width \gg L : 1)$

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

This is the same as for a regular texture. For volume textures we add:

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

Cache-line aligned width and height are as follows, with i and j being a function of the map format as shown in Alignment Unit Size.


LOD 0 (Mip 0)
q=0
q=1
q=2
q=3
q=4
q=5
q=6
q=7
LOD 1 (Mip 1)
q=0
q=1
q=2
q=3
LOD 2 (Mip 2)
q=0
q=1
LOD 3 (Mip 3)
q=0

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

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

It is not necessary to cache-line align in the "depth" dimension (i.e. lowercase "*d*").

The following equations for $LOD_{L,q}$ give the base address Cartesian coordinates for the map at LOD L and depth q.

$$LOD_{0,q} = (0, q * h_0)$$

$$LOD_{1,q} = ((q\%2) * w_1, D_0 * h_0 + (q >> 1) * h_1)$$

$$LOD_{2,q} = ((q\%4) * w_2, D_0 * h_0 + ceil\left(\frac{D_1}{2}\right) * h_1 + (q >> 2) * h_2)$$

$$LOD_{3,q} = ((q\%8) * w_3, D_0 * h_0 + ceil\left(\frac{D_1}{2}\right) * h_1 + ceil\left(\frac{D_2}{4}\right) * h_2 + (q >> 3) * h_3)$$

These values are then used as "base addresses" and the 2D MIP Map equations are used to compute the location within each LOD/q map.

Minimum Pitch

The minimum pitch required to store the 3D map may in some cases be greater than the minimum pitch required by the LOD=0 map. This is due to cache line alignment requirements that may impact some of the MIP levels requiring additional spacing in the horizontal direction.

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 parameters *i* and *j* that are used depending on the surface.

Alignment Parameters, CHV, BSW

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
	ASTC	Value of ASTC_2DBlockWidth (4, 5, 6, 8, 10, or 12)	Value of ASTC_2DBlockHeight (4, 5, 6, 8, 10, or 12)
	all others	set by **Surface Horizontal Alignment**	set by **Surface Vertical Alignment**

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.

Buffer Padding Requirements

For compressed textures (BC*, FXT1, ETC*, EAC*, and ASTC* 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.

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 SURFTYPE_2D and SURFTYPE_3D with linear mode and height % 4 != 0, the surface must be padded with 4-(height % 4)***Surface Pitch** # of bytes.	

Programming Note	
Context:	Sampling Engine Surfaces
For SURFTYPE_1D with linear mode, the surface must be padded to 4 times the Surface Pitch # of bytes	

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.