# Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM)

## Volume 1 Part 1: Graphics Core™ (Ivy Bridge)

## For the 2012 Intel® Core™ Processor Family

**May 2012**

**Revision 1.0**

***NOTICE:***
This document contains information on products in the design phase of development, and Intel reserves the right to add or remove product features at any time, with or without changes to this open source documentation.

*Doc Ref #: IHD-OS-V1 Pt 1 – 05 12*

## Creative Commons License

**You are free to Share** — to copy, distribute, display, and perform the work

**Under the following conditions:**

**Attribution**. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**No Derivative Works**. You may not alter, transform, or build upon this work.

# Contents

# 1. Introduction

This ***Intel® HD Graphics Open Source Programmer's Reference Manual*** (PRM) describes the architectural behavior and programming environment of the Ivy Bridge chipset family. The Graphics Controller (GC) contains an extensive set of registers and instructions for configuration, 2D, 3D, and video systems. The PRM describes the register, instruction, and memory interfaces, and the device behaviors as controlled and observed through those interfaces. The PRM also describes the registers and instructions, and provides detailed bit/field descriptions.

The PRM is organized into four volumes, with each volume split into multiple parts for easy accessibility:

### PRM Volume 1: Intel Graphics Core™

This volume contains an introduction to the:

- Graphics Processing Unit (GPU)
- Graphics Processing Engine (GPE)

  The GPE is a collective term for 3D, Media, the subsystem, and the parts of the memory interface that are used by these units. Display, blitter, and their memory interfaces are *not* included in the GPE.
- Graphics Memory Interface Functions
- Device Programming Environment
- Command Streamers, including the Render, Blitter, and Video Codec command streamers
- GT Interface Registers
- L3 Cache and Unified Return Buffer

### PRM Volume 2: 3D / Media

This volume includes:

- An introduction to the 3D Pipeline and its stages, including Vertex Fetch, Vertex Shader, Hull Shader, Tesselation Engine, Domain Shader, Geometry Shader, Stream Output Logic, Clip, Strips and Fans, Windower and Color Calculator stages
- The single Media fixed function, Variable Length Decode (VLD), describing how to initiate generic threads using the thread spawner (TS). Generic threads are used for the majority of media functions. Programmable kernels handle the algorithms for media functions such as IDCT, Motion Compensation, and Motion Estimation (used for encoding MPEG streams).
- Details about the Media pipeline, which is positioned in parallel with the 3D fixed function pipeline.
- The Multi-Format Codec (MFX) engine, the hardware fixed-function popeline for decoding and encoding.

### PRM Volume 3: Display Registers

Volume 3 describes the control registers for the display. This information includes:

- VGA and Extended VGA Registers
- PCI Registers
- North Display Engine Registers
- South Display Engine Registers

### PRM Volume 4: Subsystem and Cores/ Shared Functions

The subsystem contains the programmable cores, or Executable Units (EUs), and the “shared functions” that are shared by more than one EU and perform I/O functions and complex math functions.

The shared functions consist of the sampler:

- Extended math unit
- Data port (the interface to memory for 3D and media)
- Unified Return Buffer (URB)
- The Message Gateway used by EU threads to signal each other

The EUs use messages to send data to and receive data from the subsystem; the messages are described with the shared functions. The generic message ‘send EU instruction’ is described with the rest of the instructions in the Instruction Set Architecture (ISA) chapters.

The latter part of this volume describes the GMHC core, or EU, and the associated instructions used to program it. The instruction descriptions make up an Instruction Set Architecture, or ISA. The ISA describes all of the instructions that the core can execute, along with the registers that are used to store local data.

## 1.1 Reserved Bits and Software Compatibility

In many register, instruction, and memory layout descriptions, certain bits are marked as “Reserved”. When bits are marked as reserved, it is essential for compatibility with future devices that the software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as undefined *and unpredictable*. Software should follow these guidelines in dealing with reserved bits:

1. Do not depend on the states of any reserved bits when testing values of registers that contain such bits.
2. Mask out the reserved bits before testing.
3. Do not depend on the states of any reserved bits when storing to an instruction or to a register.
4. When loading a register or formatting an instruction, always load the reserved bits with the values indicated in the documentation (if any), or reload them with the values previously read from the register.

## 1.2 Terminology

| Term | Abbr. | Definition |
|---|---|---|
| 3D Pipeline | -- | One of the two pipelines supported in the GPE. The 3D Pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See *3D Pipeline*. |
| Adjacency | -- | One can consider a single line object as existing in a strip of connected lines. The neighboring line objects are called “adjacent objects”, with the non-shared endpoints called the “adjacent vertices.” The same concept can be applied to a single triangle object, considering it as existing in a mesh of connected triangles. Each triangle shares edges with three other adjacent triangles, each defined by an non-shared adjacent vertex. Knowledge of these adjacent objects/vertices is required by some object processing algorithms (e.g., silhouette edge detection). See *3D Pipeline*. |
| Application IP | AIP | Application Instruction Pointer. This is part of the control registers for exception handling for a thread. Upon an exception, hardware moves the current IP into this register and then jumps to SIP. |
| Architectural Register File | ARF | A collection of architecturally visible registers for a thread such as address registers, accumulator, flags, notification registers, IP, null, etc. ARF should not be mistaken as just the address registers. |
| Array of Cores | -- | Refers to a group of EUs, which are physically organized in two or more rows. The fact that the EUs are arranged in an array is (to a great extent) transparent to CPU software or EU kernels. |
| Binding Table | -- | Memory-resident list of pointers to surface state blocks (also in memory). |
| Binding Table Pointer | BTP | Pointer to a binding table, specified as an offset from the Surface State Base Address register. |
| Bypass Mode | -- | Mode where a given fixed function unit is disabled and forwards data down the pipeline unchanged. Not supported by all FF units. |
| Byte | B | A numerical data type of 8 bits, B represents a signed byte integer. |
| Child Thread | | A branch-node or a leaf-node thread that is created by another thread. It is a kind of thread associated with the media fixed function pipeline. A child thread is originated from a thread (the parent) executing on an EU and forwarded to the Thread Dispatcher by the TS unit. A child thread may or may not have child threads depending on whether it is a branch-node or a leaf-node thread. All pre-allocated resources such as URB and scratch memory for a child thread are managed by its parent thread. |
| Clip Space | -- | A 4-dimensional coordinate system within which a clipping frustum is defined. Object positions are projected from Clip Space to NDC space via “perspecitive divide” by the W coordinate, and then viewport mapped into Screen Space |
| Clipper | -- | 3D fixed function unit that removes invisible portions of the drawing sequence by discarding (culling) primitives or by “replacing” primitives with one or more primitives that replicate only the visible portion of the original primitive. |
| Color Calculator | CC | Part of the Data Port shared function, the color calculator performs fixed-function pixel operations (e.g., blending) prior to writing a result pixel into the render cache. |
| Command | -- | Directive fetched from a ring buffer in memory by the Command Streamer and routed down a pipeline. Should not be confused with instructions which are fetched by the instruction cache subsystem and executed on an EU. |
| Command Streamer | CS or CSI | Functional unit of the Graphics Processing Engine that fetches commands, parses them and routes them to the appropriate pipeline. |
| Constant URB Entry | CURBE | A UE that contains “constant” data for use by various stages of the pipeline. |
| Control Register | CR | The read-write registers are used for thread mode control and exception handling for |

| Term | Abbr. | Definition |
|---|---|---|
| | | a thread. |
| Data Port | DP | Shared function unit that performs a majority of the memory access types on behalf of programs. The Data Port contains the render cache and the constant cache and performs all memory accesses requested by programs except those performed by the Sampler. See DataPort. |
| Degenerate Object | -- | Object that is invisible due to coincident vertices or because does not intersect any sample points (usually due to being tiny or a very thin sliver). |
| Destination | -- | Describes an output or write operand. |
| Destination Size | | The number of data elements in the destination of an SIMD instruction. |
| Destination Width | | The size of each of (possibly) many elements of the destination of an SIMD instruction. |
| Double Quad word (DQword) | DQ | A fundamental data type, DQ represents 16 bytes. |
| Double word (DWord) | D or DW | A fundamental data type, D or DW represents 4 bytes. |
| Drawing Rectangle | -- | A screen-space rectangle within which 3D primitives are rendered. An objects screen-space positions are relative to the Drawing Rectangle origin. See *Strips and Fans*. |
| End of Block | EOB | A 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer. |
| End Of Thread | EOT | a message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set in order to properly terminate. |
| Exception | -- | Type of (normally rare) interruption to EU execution of a thread's instructions. An exception occurrence causes the EU thread to begin executing the System Routine which is designed to handle exceptions. |
| Execution Channel | -- | |
| Execution Size | ExecSize | Execution Size indicates the number of data elements processed by an SIMD instruction. It is one of the instruction fields and can be changed per instruction. |
| Execution Unit | EU | Execution Unit. An EU is a multi-threaded processor within the multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a Core. |
| Execution Unit Identifier | EUID | The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU a thread is located. A thread can be uniquely identified by the EUID and TID. |
| Execution Width | ExecWidth | The width of each of several data elements that may be processed by a single SIMD instruction. |
| Extended Math Unit | EM | A Shared Function that performs more complex math operations on behalf of several EUs. |
| FF Unit | -- | A Fixed-Function Unit is the hardware component of a 3D Pipeline Stage. A FF Unit typically has a unique FF ID associated with it. |
| Fixed Function | FF | Function of the pipeline that is performed by dedicated (vs. programmable) hardware. |
| Fixed Function ID | FFID | Unique identifier for a fixed function unit. |
| FLT_MAX | fmax | The magnitude of the maximum representable single precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's. |
| Gateway | GW | See Message Gateway. |
| Core | | Alternative name for an EU in the multi-processor system. |
| General Register File | GRF | Large read/write register file shared by all the EUs for operand sources and destinations. This is the most commonly used read-write register space organized as |

| Term | Abbr. | Definition |
|---|---|---|
| | | an array of 256-bit registers for a thread. |
| General State Base Address | -- | The Graphics Address of a block of memory-resident “state data”, which includes state blocks, scratch space, constant buffers and kernel programs. The contents of this memory block are referenced via offsets from the contents of the General State Base Address register. See *Graphics Processing Engine*. |
| Geometry Shader | GS | Fixed-function unit between the vertex shader and the clipper that (if enabled) dispatches “geometry shader” threads on its input primitives. Application-supplied geometry shaders normally expand each input primitive into several output primitives in order to perform 3D modeling algorithms such as fur/fins. See *Geometry Shader*. |
| Graphics Address | | The GPE virtual address of some memory-resident object. This virtual address gets mapped by a GTT or PGTT to a physical memory address. Note that many memory-resident objects are referenced not with Graphics Addresses, but instead with offsets from a “base address register”. |
| Graphics Processing Engine | GPE | Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer. |
| Guardband | GB | Region that may be clipped against to make sure objects do not exceed the limitations of the renderer’s coordinate space. |
| Horizontal Stride | HorzStride | The distance in element-sized units between adjacent elements of a region-based GRF access. |
| Immediate floating point vector | VF | A numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction. |
| Immediate integer vector | V | A numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4-bit each. The 4-bit integer element is in 2’s compliment form. It may be used to specify the type of an immediate operand in an instruction. |
| Index Buffer | IB | Buffer in memory containing vertex indices. |
| In-loop Deblocking Filter | ILDB | The deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe. |
| Instance | | In the context of the VF unit, an instance is one of a sequence of sets of similar primitive data. Each set has identical vertex data but may have unique instance data that differentiates it from other sets in the sequence. |
| Instruction | -- | Data in memory directing an EU operation. Instructions are fetched from memory, stored in a cache and executed on one or more cores. Not to be confused with commands which are fetched and parsed by the command streamer and dispatched down the 3D or Media pipeline. |
| Instruction Pointer | IP | The address (really an offset) of the instruction currently being fetched by an EU. Each EU has its own IP. |
| Instruction Set Architecture | ISA | The ISA describes the instructions supported by an EU. |
| Instruction State Cache | ISC | On-chip memory that holds recently-used instructions and state variable values. |
| Interface Descriptor | -- | Media analog of a State Descriptor. |
| Intermediate Z | IZ | Completion of the Z (depth) test at the front end of the Windower/Masker unit when certain conditions are met (no alpha, no pixel-shader computed Z values, etc.) |
| Inverse Discrete Cosine Transform | IDCT | the stage in the video decoding pipe between IQ and MC |
| Inverse Quantization | IQ | A stage in the video decoding pipe between IS and IDCT. |

| Term | Abbr. | Definition |
|---|---|---|
| Inverse Scan | IS | A stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for both MPEG-2. |
| Jitter | | Just-in-time compiler. |
| Kernel | -- | A sequence of instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to instructions. |
| Least Significant Bit | LSB | |
| MathBox | -- | See Extended Math Unit |
| Media | -- | Term for operations such as video decode and encode that are normally performed by the Media pipeline. |
| Media Pipeline | -- | Fixed function stages dedicated to media and “generic” processing, sometimes referred to as the generic pipeline. |
| Message | -- | Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of architecture. |
| Message Gateway | -- | Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline. |
| Message Register File | MRF | Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction. |
| Most Significant Bit | MSB | |
| Motion Compensation | MC | Part of the video decoding pipe. |
| Motion Picture Expert Group | MPEG | MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc. |
| Motion Vector Field Selection | MVFS | A four-bit field selecting reference fields for the motion vectors of the current macroblock. |
| Multi Render Targets | MRT | Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state. |
| Normalized Device Coordinates | NDC | Clip Space Coordinates that have been divided by the Clip Space “W” component. |
| Object | -- | A single triangle, line or point. |
| Open GL | OGL | A Graphics API specification associated with Linux. |
| Parent Thread | -- | A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy. |
| Pipeline Stage | -- | A abstracted element of the 3D Pipeline, providing functions performed by a combination of the corresponding hardware FF unit and the threads spawned by that FF unit. |
| Pipelined State Pointers | PSP | Pointers to state blocks in memory that are passed down the pipeline. |
| Pixel Shader | PS | Shader that is supplied by the application, translated by the jitter and is dispatched to the EU by the Windower (conceptually) once per pixel. |
| Point | -- | A drawing object characterized only by position coordinates and width. |
| Primitive | -- | Synonym for object: triangle, rectangle, line or point. |
| Primitive Topology | -- | A composite primitive such as a triangle strip, or line list. Also includes the objects triangle, line and point as degenerate cases. |
| Provoking Vertex | -- | The vertex of a primitive topology from which vertex attributes that are constant across the primitive are taken. |
| Quad Quad word (QQword) | QQ | A fundamental data type, QQ represents 32 bytes. |

| Term | Abbr. | Definition |
|---|---|---|
| Quad Word (QWord) | QW | A fundamental data type, QW represents 8 bytes. |
| Rasterization | | Conversion of an object represented by vertices into the set of pixels that make up the object. |
| Region-based addressing | -- | Collective term for the register addressing modes available in the EU instruction set that permit discontiguous register data to be fetched and used as a single operand. |
| Render Cache | RC | Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test. |
| Render Target | RT | A destination surface in memory where render results are written. |
| Render Target Array Index | -- | Selector of which of several render targets the current operation is targeting. |
| Root Thread | -- | A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE. |
| Sampler | -- | Shared function that samples textures and reads data from buffers on behalf of EU programs. |
| Scratch Space | -- | Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc. |
| Shader | -- | A program that is supplied by the application in a high level shader language, and translated to instructions by the jitter. |
| Shared Function | SF | Function unit that is shared by EUs. EUs send messages to shared functions; they consume the data and may return a result. The Sampler, Data Port and Extended Math unit are all shared functions. |
| Shared Function ID | SFID | Unique identifier used by kernels and shaders to target shared functions and to identify their returned messages. |
| Single Instruction Multiple Data | SIMD | The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at instruction level. It can also be used to describe the instructions in such architecture. |
| Source | -- | Describes an input or read operand |
| Spawn | -- | To initiate a thread for execution on an EU. Done by the thread spawner as well as most FF units in the 3D Pipeline. |
| Sprite Point | -- | Point object using full range texture coordinates. Points that are not sprite points use the texture coordinates of the point's center across the entire point object. |
| State Descriptor | -- | Blocks in memory that describe the state associated with a particular FF, including its associated kernel pointer, kernel resource allowances, and a pointer to its surface state. |
| State Register | SR | The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP. |
| State Variable | SV | An individual state element that can be varied to change the way given primitives are rendered or media objects processed. On state variables persist only in memory and are cached as needed by rendering/processing operations except for a small amount of non-pipelined state. |
| Stream Output | -- | A term for writing the output of a FF unit directly to a memory buffer instead of, or in addition to, the output passing to the next FF unit in the pipeline. Currently only supported for the Geometry Shader (GS) FF unit. |
| Strips and Fans | SF | Fixed function unit whose main function is to decompose primitive topologies such as strips and fans into primitives or objects. |

| Term | Abbr. | Definition |
|---|---|---|
| Sub-Register | | Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, *r2*, is 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*. |
| Subsystem | -- | The name given to the resources shared by the FF units, including shared functions and EUs. |
| Surface | -- | A rendering operand or destination, including textures, buffers, and render targets. |
| Surface State | -- | State associated with a render surface including |
| Surface State Base Pointer | -- | Base address used when referencing binding table and surface state data. |
| Synchronized Root Thread | -- | A root thread that is dispatched by TS upon a 'dispatch root thread' message. |
| System IP | SIP | There is one global System IP register for all the threads. From a thread's point of view, this is a virtual read only register. Upon an exception, hardware performs some bookkeeping and then jumps to SIP. |
| System Routine | -- | Sequence of instructions that handles exceptions. SIP is programmed to point to this routine, and all threads encountering an exception will call it. |
| Thread | | An instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction – a send instruction with EOT that signals the thread termination. Threads in the system may be independent from each other or communicate with each other through Message Gateway share function. |
| Thread Dispatcher | TD | Functional unit that arbitrates thread initiation requests from Fixed Functions units and instantiates the threads on EUs. |
| Thread Identifier | TID | The field within a thread state register (SR0) that identifies which thread slots on an EU a thread occupies. A thread can be uniquely identified by the EUID and TID. |
| Thread Payload | | Prior to a thread starting execution, some amount of data will be pre-loaded in to the thread's GRF (starting at r0). This data is typically a combination of control information provided by the spawning entity (FF Unit) and data read from the URB. |
| Thread Spawner | TS | The second and the last fixed function stage of the media pipeline that initiates new threads on behalf of generic/media processing. |
| Topology | | See Primitive Topology. |
| Unified Return Buffer | URB | The on-chip memory managed/shared by Fixed Functions in order for a thread to return data that will be consumed either by a Fixed Function or other threads. |
| Unsigned Byte integer | UB | A numerical data type of 8 bits. |
| Unsigned Double Word integer | UD | A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction. |
| Unsigned Word integer | UW | A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction. |
| Unsynchronized Root Thread | -- | A root thread that is automatically dispatched by TS. |
| URB Dereference | -- | |
| URB Entry | UE | URB Entry: A logical entity stored in the URB (such as a vertex), referenced via a URB Handle. |
| URB Entry Allocation Size | -- | Number of URB entries allocated to a Fixed Function unit. |
| URB Fence | Fence | Virtual, movable boundaries between the URB regions owned by each FF unit. |
| URB Handle | -- | A unique identifier for a URB entry that is passed down a pipeline. |
| URB Reference | -- | |
| Variable Length Decode | VLD | The first stage of the video decoding pipe that consists mainly of bit-wide operations. The GPU supports hardware VLD acceleration in the VFE fixed function stage. |

| Term | Abbr. | Definition |
|---|---|---|
| VC-1 | VC-1 | VC-1 is the informal name of the SMPTE 421M video codec standard |
| Vertex Buffer | VB | Buffer in memory containing vertex attributes. |
| Vertex Cache | VC | Cache of Vertex URB Entry (VUE) handles tagged with vertex indices. See the VS chapter for details on this cache. |
| Vertex Fetcher | VF | The first FF unit in the 3D Pipeline responsible for fetching vertex data from memory. Sometimes referred to as the Vertex Formatter. |
| Vertex Header | -- | Vertex data required for every vertex appearing at the beginning of a Vertex URB Entry. |
| Vertex ID | -- | Unique ID for each vertex that can optionally be included in vertex attribute data sent down the pipeline and used by kernel/shader threads. |
| Vertex Index | -- | Offset (in vertex-sized units) of a given vertex in a vertex buffer. Not unique per vertex instance. |
| Vertex Sequence Number | -- | Unique ID for each vertex sent down the south bus . |
| Vertex Shader | VS | An API-supplied program that calculates vertex attributes. Also refers to the FF unit that dispatches threads to “shade” (calculate attributes for) vertices. |
| Vertex URB Entry | VUE | A URB entry that contains data for a specific vertex. |
| Vertical Stride | VertStride | The distance in element-sized units between 2 vertically-adjacent elements of a region-based GRF access. |
| Video Front End | VFE | The first fixed function in the generic pipeline; performs fixed-function media operations. |
| Viewport | VP | |
| Windower IZ | WIZ | Term for Windower/Masker that encapsulates its early (“intermediate”) depth test function. |
| Windower/Masker | WM | Fixed function triangle/line rasterizer. |
| Word | W | A numerical data type of 16 bits, W represents a signed word integer. |

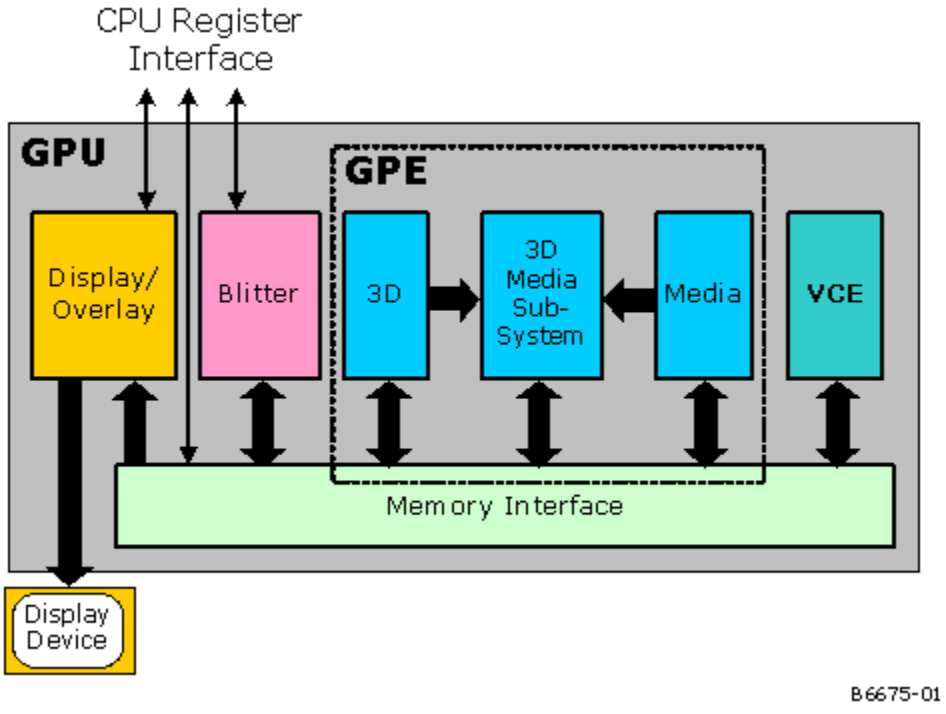
# 2. Graphics Device Overview

## 2.1 Graphics Processing Unit (GPU)

The Graphics Processing Unit is controlled by the CPU through a direct interface of memory-mapped IO registers, and indirectly by parsing commands that the CPU has placed in memory. The display interface and blitter (**bl**ock **i**mage **t**ransferr**er**) are controlled primarily by direct CPU register addresses, while the 3D and Media pipelines and the parallel Video Codec Engine (VCE) are controlled primarily through instruction lists in memory.

The subsystem contains an array of cores, or execution units, with a number of “shared functions”, which receive and process messages at the request of programs running on the cores. The shared functions perform critical tasks, such as sampling textures and updating the render target (usually the frame buffer). The cores themselves are described by an instruction set architecture, or ISA.

**Logical Block Diagram**



# 3. Graphics Processing Engine (GPE)

## 3.1 Introduction to the GPE

This chapter serves two purposes:

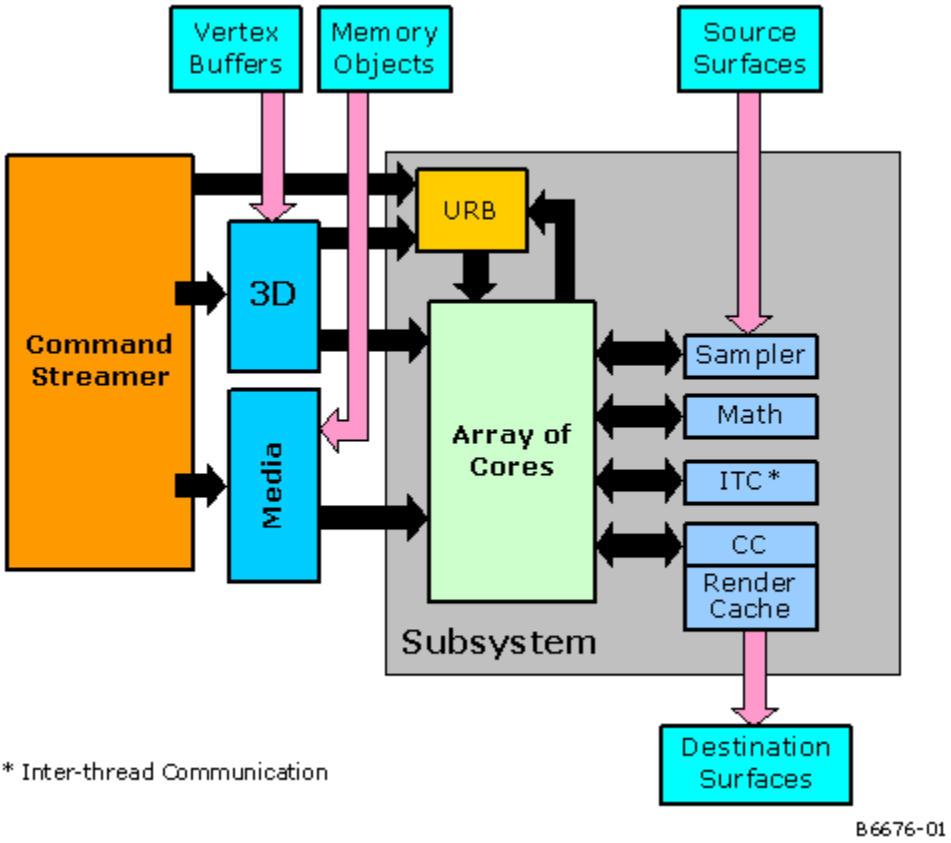
- It provides a high-level description of the Graphics Processing Engine (GPE) of the Graphics Processing Unit (GPU).
- It also specifies the programming and behaviors of the functions common to both pipelines (3D, Media) within the GPE. However, details specific to either pipeline are not addressed here.

## 3.2 Overview

The Graphics Processing Engine (GPE) performs the bulk of the graphics processing provided by the GPU. It consists of the 3D and Media fixed-function pipelines, the Command Streamer (CS) unit that feeds them, and the Subsystem that provides the bulk of the computations required by the pipelines.

### 3.2.1 Graphics Processing Engine Block Diagram

**The Graphics Processing Engine**




### 3.2.2 Command Stream (CS) Unit

The Command Stream (CS) unit manages the use of the 3D and Media pipelines; it performs switching between pipelines and forwarding command streams to the currently active pipeline. It manages allocation of the URB and helps support the Constant URB Entry (CURBE) function.

### 3.2.3 3D Pipeline

The 3D Pipeline provides specialized 3D primitive processing functions. These functions are provided by a pipeline of “fixed function” stages (units) and threads spawned by these units. See *3D Pipeline Overview*.

### 3.2.4 Media Pipeline

The Media pipeline provides both specialized media-related processing functions and the ability to perform more general (“generic”) functionality. These Media-specific functions are provided by a Video Front End (VFE) unit. A Thread Spawner (TS) unit is utilized to spawn threads requested by the VFE unit, or as required when the pipeline is used for general processing. See *Media Pipeline Overview*.

### 3.2.5 Subsystem

The Subsystem is the collective name for the programmable cores, the Shared Functions accessed by them (including the Sampler, Extended Math Unit (“MathBox”), the DataPort, and the Inter-Thread Communication (ITC) Gateway), and the Dispatcher that manages threads running on the cores.

### 3.2.6 Execution Units (EUs)

While the number of EU cores in the subsystem is almost entirely transparent to the programming model, there are a few areas where this parameter comes into play:

- The amount of scratch space required is a function of (#EUs * #Threads/EU)

| Device | # of EUs | #Threads/EU |
|---|---|---|
| Ivy Bridge | 16 | 8 |
| Ivy Bridge | 6 | 6 |

### 3.2.7 GPE Function IDs

The following table lists the assignments (encodings) of the Shared Function and Fixed Function IDs used within the GPE. A Shared Function is a valid target of a message initiated via a ‘send’ instruction. A Fixed Function is an identifiable unit of the 3D or Media pipeline. Note that the Thread Spawner is both a Shared Function and Fixed Function.

**Function IDs**

| ID[3:0] | SFID | Shared Function | FFID | Fixed Function |
|---|---|---|---|---|
| 0x0 | SFID_NULL | Null | FFID_NULL | Null |
| 0x1 | Reserved | --- | Reserved | --- |
| 0x2 | SFID_SAMPLER | Sampler | Reserved | --- |
| 0x3 | SFID_GATEWAY | Message Gateway | Reserved | --- |
| 0x4 | SFID_DP_SAMPLER | Sampler Cache Data Port | FFID_HS | Hull Shader |
| 0x5 | SFID_DP_RC | Render Cache Data Port | FFID_DS | Domain Shader |
| 0x6 | SFID_URB | URB | Reserved | --- |
| 0x7 | SFID_SPAWNER | Thread Spawner | FFID_SPAWNER | Thread Spawner |
| 0x8 | SFID_VME | Video Motion Estimation | FFID_VFE | Video Front End |
| 0x9 | SFID_DP_CC | Constant Cache Data Port | FFID_VS | Vertex Shader |
| 0xA | SFID_DP_DC | Data Cache Data Port | FFID_CS | Command Stream |
| 0xB | SFID_PI | Pixel Interpolator | FFID_VF | Vertex Fetch |
| 0xC | Reserved | --- | FFID_GS | Geometry Shader |
| 0xD | Reserved | --- | FFID_CLIP | Clipper Unit |
| 0xE | Reserved | --- | FFID_SF | Strip/Fan Unit |
| 0xF | Reserved | --- | FFID_WM | Windower/Masker Unit |

## 3.3 Pipeline Selection

The PIPELINE_SELECT command is used to specify which GPE pipeline (3D or Media orGPGPU ) is to be considered the “current” active pipeline. Issuing 3D-pipeline-specific commands when the Media pipeline is selected, or vice versa, is UNDEFINED.

This command causes the URB deallocation of the previously selected pipe. For example, switching from the 3D pipe to the Media pipe (either within or between contexts) will cause the CS to send a “Deallocating Flush” down the 3D pipe, and each 3D FF will start a URB deallocation sequence after the

current tasks are done. Then, the WM will de-reference the current Constant URB Entry, and all 3D URB entries will be deallocated (after some north bus delay) , which allows the CS to set the URB fences for the media pipe. The process relatively is the same for switching from Media to 3D pipes. The deallocating flush goes down the Media pipe, causing each Media function to start a URB deallocation sequence, and the WM will de-reference the current Constant URB entry and all media entries will be de-allocated to allow the CS to set the 3D pipe.

**Programming Restriction:**

Software must ensure the current pipeline is flushed via an MI_FLUSH or PIPE_CONTROL prior to the execution of PIPELINE_SELECT.

| DWord | Bit | Description |
|---|---|---|
| 0 | 31:29 | **Instruction Type =** GFXPIPE = 3h |
| | 28:16 | **3D Instruction Opcode =** PIPELINE_SELECT<br>GFXPIPE[28:27 = 1h, 26:24 = 1h, 23:16 = 04h] (Single DW, Non-pipelined) |
| | 15:2 | Reserved: MBZ |
| | 1: 0 | Pipeline Select<br>0: 3D Pipeline is selected<br>1: Media pipeline is selected (Includes and generic media workloads)<br>2: GPGPU pipeline is selected<br>3: Reserved |

The **Pipeline Select** state is contained within the logical context.

## 3.4 Memory Object Control State

The memory object control state defines behavior of memory accesses beyond the graphics core, graphics data type that allows selective flushing of data from outer caches, and ability to control 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 location the 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 |

| Memory Object | Location of Control State |
|---|---|
| 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 |

### 3.4.1 MEMORY_OBJECT_CONTROL_STATE

**MEMORY_OBJECT_CONTROL_STATE**

Default Value: 0x00000000

| DWord | Bit | Description |
|---|---|---|
| 0 | 2 | **Graphics Data Type (GFDT)**<br>Format: U1<br><br>This field contains the GFDT bit for this surface when writes occur. GFDT can also be set by the GTT. The effective GFDT is the logical OR of this field with the GFDT from the GTT entry. This field is ignored for reads.<br><br>The GFDT bit is stored in the LLC and selective cache flushing of lines with GFDT set is supported. It is intended to be set on displayable data, which enables efficient flushing of data to be displayed after rendering, since display engine does not snoop the rendering caches. Note that MLC would need to be completely flushed as it does not allow selective flushing. |
| | 1 | **LLC Cacheability Control (LLCCC)**<br><br>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 LLCCC is set, cacheable transaction are generated to enable LLC usage for particular stream.<br><br>0: use cacheability controls from GTT entry<br><br>1: Data is cached in LLC |
| | 0 | **L3 Cacheability Control (L3CC)**<br><br>This field is used to control the L3 cacheability (allocation) of the stream.<br><br>0: not cacheable within L3<br><br>1: cacheable in L3<br><br>*Note: even if the surface is not cacheable in L3, it is still kept coherent with L3 content.* |

## 3.5 Memory Access Indirection

The GPE supports the indirection of certain graphics (GTT-mapped) memory accesses. This support comes in the form of two *base address* state variables used in certain memory address computations with the GPE.

The intent of this functionality is to support the dynamic relocation of certain driver-generated memory structures after command buffers have been generated but prior to the their submittal for execution. For example, as the driver builds the command stream it could append pipeline state descriptors, kernel binaries, etc. to a general state buffer. References to the individual items would be inserting in the command buffers as offsets from the base address of the state buffer. The state buffer could then be freely relocated prior to command buffer execution, with the driver only needing to specify the final base address of the state buffer. Two base addresses are provided to permit surface-related state (binding tables, surface state tables) to be maintained in a state buffer separate from the general state buffer.

While the use of these base addresses is unconditional, the indirection can be effectively disabled by setting the base addresses to zero. The following table lists the various GPE memory access paths and which base address (if any) is relevant.

**Base Address Utilization**

| Base Address Used | Memory Accesses |
|---|---|
| General State Base Address | |
| | |
| | |
| | DataPort memory accesses resulting from **'stateless' DataPort Read/Write requests**. See *DataPort* for a definition of the 'stateless' form of requests. |
| Dynamic State Base Address only | Sampler reads of SAMPLER_STATE data and associated SAMPLER_BORDER_COLOR_STATE. |
| | **Viewport states** used by CLIP, SF, and WM/CC |
| | COLOR_CALC_STATE, DEPTH_STENCIL_STATE, and BLEND_STATE |
| | Push Constants (depeding on state of **INSTPM<CONSTANT_BUFFER Address Offset Disable>** ) |
| Instruction Base Address only | **Normal EU instruction stream** (non-system routine) |
| | **System routine** EU instruction stream (starting address = SIP) |
| Surface State Base Address | Sampler and DataPort reads of BINDING_TABLE_STATE, as referenced by BT pointers passed via 3DSTATE_BINDING_TABLE_POINTERS |
| | Sampler and DataPort reads of SURFACE_STATE data |
| Indirect Object Base Address | **MEDIA_OBJECT Indirect Data** accessed by the CS unit . |
| None | CS unit reads from **Ring Buffers**, **Batch Buffers** |
| | |
| | CS writes resulting from PIPE_CONTROL command |
| | All VF unit memory accesses (**Index Buffers**, **Vertex Buffers**) |
| | All Sampler **Surface Memory Data** accesses (texture fetch, etc.) |
| | All **DataPort memory accesses**except 'stateless' DataPort Read/Write requests (e.g., RT accesses.) See *DataPort* for a definition of the 'stateless' form of requests. |
| | Memory reads resulting from **STATE_PREFETCH** commands |
| | Any **physical memory access** by the device |
| | GTT-mapped accesses not included above (i.e., default) |
| | Push Constants (depeding on state of **INSTPM<CONSTANT_BUFFER Address Offset Disable>** ) |

The following notation is used in the BSpec to distinguish between addresses and offsets:

| Notation | Definition |
|---|---|
| PhysicalAddress[n:m] | Corresponding bits of a physical graphics memory byte address (not mapped by a GTT) |
| GraphicsAddress[n:m] | Corresponding bits of an absolute, virtual graphics memory byte address (mapped by a GTT) |
| GeneralStateOffset[n:m] | Corresponding bits of a relative byte offset added to the General State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT) |
| DynamicStateOffset[n:m] | Corresponding bits of a relative byte offset added to the Dynamic State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT) |
| InstructionBaseOffset[n:m] | Corresponding bits of a relative byte offset added to the Instruction Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT) |
| SurfaceStateOffset[n:m] | Corresponding bits of a relative byte offset added to the Surface State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT) |

### 3.5.1 STATE_BASE_ADDRESS

The STATE_BASE_ADDRESS command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPE. (See *Memory Access Indirection* for details)

**Programming Notes:**

- The following commands must be reissued following any change to the base addresses:
    - 3DSTATE_PIPELINE_POINTERS
    - 3DSTATE_BINDING_TABLE_POINTERS
    - MEDIA_STATE_POINTERS.
- Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance.

| STATE_BASE_ADDRESS | | |
|---|---|---|
| Length Bias: | | 2 |
| The STATE_BASE_ADDRESS command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPE. (See Table: Base Address Utilization for details) | | |
| **Programming Notes** | | |
| The following commands must be reissued following any change to the base addresses<br>• 3DSTATE_CC_POINTERS<br>• 3DSTATE_BINDING_TABLE_POINTERS<br>• 3DSTATE_SAMPLER_STATE_POINTERS<br>• 3DSTATE_VIEWPORT_STATE_POINTERS<br>• MEDIA_STATE_POINTERS<br>Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance | | |
| **DWord** | **Bit** | **Description** |
| 0 | 31:29 | **Command Type** |

# STATE_BASE_ADDRESS

| | | |
|---|---|---|
| | | Default Value: 3h GFXPIPE |
| | 28:27 | **Command SubType**<br>Default Value: 0h GFXPIPE_COMMON |
| | 26:24 | **3D Command Opcode**<br>Default Value: 1h GFXPIPE_NONPIPELINED |
| | 23:16 | **3D Command Sub Opcode**<br>Default Value: 01h STATE_BASE_ADDRESS |
| | 15:8 | **Reserved**<br>Format: MBZ |
| | 7:0 | **DWord Length**<br>Project: All<br>Format: =n Total Length - 2<br><br>**Value** / **Name** / **Description**<br>8h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1) |
| 1 | 31:12 | **General State Base Address**<br>Format: GraphicsAddress[31:12]<br>Specifies the 4K-byte aligned base address for general state accesses. See Table 4-3 for details on where this base address is used. |
| | 11:8 | **General State Memory Object Control State**<br>Format: MEMORY_OBJECT_CONTROL_STATE<br>Specifies the memory object control state for indirect state using the General State Base Address, with the exception of the stateless data port accesses. |
| | 7:4 | **Stateless Data Port Access Memory Object Control State**<br>Format: MEMORY_OBJECT_CONTROL_STATE<br>Specifies the memory object control state for stateless data port accesses. |
| | 3 | **Stateless Data Port Access Force Write Thru**<br>Format: U1<br>0: If the stateless data port access memory object control indicates L3 cachable the accesses will be write back cacheable.<br><br>1: If the stateless data port access memory object control indicates L3 cachable the accesses will be write thru cacheable. |
| | 2:1 | **Reserved**<br>Format: MBZ |

# STATE_BASE_ADDRESS

| | | |
|---|---|---|
| | 0 | **General State Base Address Modify Enable**<br>Format: Enable<br>The other fields in this dword are updated only when this bit is set.<br>0h – Disable – Ignore the updated address<br>1h – Enable – Modify the address |
| 2 | 31:12 | **Surface State Base Address**<br>Format: GraphicsAddress[31:12]<br>Specifies the 4K-byte aligned base address for binding table and surface state accesses. See Table 4-3 for details on where this base address is used. |
| | 11:8 | **Surface State Memory Object Control State**<br>Project: All<br>Format: MEMORY_OBJECT_CONTROL_STATE<br>Specifies the memory object control state for indirect state using the **Surface State Base Address**. |
| | 7:1 | **Reserved**<br>Format: MBZ |
| | 0 | **Surface State Base Address Modify Enable**<br>Format: Enable<br>The other fields in this dword are updated only when this bit is set.<br>0h – Disable – Ignore the updated address<br>1h – Enable – Modify the address |
| 3 | 31:12 | **Dynamic State Base Address**<br>Format: GraphicsAddress[31:12]<br>Specifies the 4K-byte aligned base address for sampler and viewport state accesses. See Table 4-3 for details on where this base address is used. |
| | 11:8 | **Dynamic State Memory Object Control State**<br>Project: All<br>Format: MEMORY_OBJECT_CONTROL_STATE<br>Specifies the memory object control state for indirect state using the **Dynamic State Base Address**. Push constants defined in 3DSTATE_CONSTANT_(VS \| GS \| PS) commands do not use this control state, although they can use the corresponding base address. The memory object control state for push constants is defined within the command. |
| | 7:1 | **Reserved**<br>Project: All<br>Format: MBZ |
| | 0 | **Dynamic State Base Address Modify Enable**<br>Project: All<br>Format: Enable<br>The other fields in this dword are updated only when this bit is set. |

## STATE_BASE_ADDRESS

| | | | | | |
|---|---|---|---|---|---|
| | | **Value** | **Name** | **Description** | **Project** |
| | | 0h | Disable | Ignore the updated address | All |
| | | 1h | Enable | Modify the address | All |
| 4 | 31:12 | **Indirect Object Base Address** | | | |
| | | Project: | All | | |
| | | Format: | GraphicsAddress[31:12] | | |
| | | Specifies the 4K-byte aligned base address for indirect object load in MEDIA_OBJECT command. See Table 4-3 for details on where this base address is used. | | | |
| | 11:8 | **Indirect Object Memory Object Control State** | | | |
| | | Project: | All | | |
| | | Format: | MEMORY_OBJECT_CONTROL_STATE | | |
| | | Specifies the memory object control state for indirect objects using the **Indirect Object Base Address**. | | | |
| | 7:1 | **Reserved** | | | |
| | | Project: | All | | |
| | | Format: | MBZ | | |
| | 0 | **Indirect Object Base Address Modify Enable** | | | |
| | | Project: | All | | |
| | | Format: | Enable | | |
| | | The other fields in this dword are updated only when this bit is set. | | | |
| | | **Value** | **Name** | **Description** | **Project** |
| | | 0h | Disable | Ignore the updated address | All |
| | | 1h | Enable | Modify the address | All |
| 5 | 31:12 | **Instruction Base Address** | | | |
| | | Project: | All | | |
| | | Format: | GraphicsAddress[31:12] | | |
| | | Specifies the 4K-byte aligned base address for all EU instruction accesses. | | | |
| | 11:8 | **Instruction Memory Object Control State** | | | |
| | | Project: | All | | |
| | | Format: | MEMORY_OBJECT_CONTROL_STATE | | |
| | | Specifies the memory object control state for EU instructions using the **Instruction Base Address**. | | | |
| | 7:1 | **Reserved** | | | |
| | | Project: | All | | |
| | | Format: | MBZ | | |
| | 0 | **Instruction Base Address Modify Enable** | | | |
| | | Project: | All | | |
| | | Format: | Enable | | |
| | | The other fields in this dword are updated only when this bit is set. | | | |
| | | **Value** | **Name** | **Description** | **Project** |
| | | 0h | Disable | Ignore the updated address | All |
| | | 1h | Enable | Modify the address | All |
| 6 | 31:12 | **General State Access Upper Bound** | | | |
| | | Project: | All | | |
| | | Format: | GraphicsAddress[31:12] | | |
| | | Specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address for | | | |

general state accesses. This includes all accesses that are offset from **General State Base Address** (see Table 4-3). Read accesses from this address and beyond will return UNDEFINED values. Data port writes to this address and beyond will be “dropped on the floor” (all data channels will be disabled so no writes occur). Setting this field to 0 will cause this range check to be ignored.

If non-zero, this address must be greater than the General State Base Address.

**11:1 Reserved**

| | |
|---|---|
| Project: | All |
| Format: | MBZ |

**0 General State Access Upper Bound Modify Enable**

| | |
|---|---|
| Project: | All |
| Format: | Enable |

The bound in this dword is updated only when this bit is set.

| Value | Name | Description | Project |
|---|---|---|---|
| 0h | Disable | Ignore the updated bound | All |
| 1h | Enable | Modify the bound | All |

**7 31:12 Dynamic State Access Upper Bound**

| | |
|---|---|
| Project: | All |
| Format: | GraphicsAddress[31:12] |

Specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address for dynamic state accesses. This includes all accesses that are offset from **Dynamic State Base Address** (see Table 4-3). Read accesses from this address and beyond will return UNDEFINED values. Data port writes to this address and beyond will be “dropped on the floor” (all data channels will be disabled so no writes occur). Setting this field to 0 will cause this range check to be ignored.

If non-zero, this address must be greater than the **Dynamic State Base Address**.

**11:1 Reserved**

| | |
|---|---|
| Project: | All |
| Format: | MBZ |

**0 Dynamic State Access Upper Bound Modify Enable**

| | |
|---|---|
| Project: | All |
| Format: | Enable |

The bound in this dword is updated only when this bit is set.

| Value | Name | Description | Project |
|---|---|---|---|
| 0h | Disable | Ignore the updated bound | All |
| 1h | Enable | Modify the bound | All |

**8 31:12 Indirect Object Access Upper Bound**

| | |
|---|---|
| Project: | All |
| Format: | GraphicsAddress[31:12] |

This field specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address access by an indirect object load in a MEDIA_OBJECT command. Indirect data accessed at this address and beyond will appear to be 0. Setting this field to 0 will cause this range check to be ignored.

If non-zero, this address must be greater than the **Indirect Object Base Address**.

Hardware ignores this field if indirect data is not present.

Setting this field to FFFFFh will cause this range check to be ignored.

**11:1 Reserved**

| | |
|---|---|
| Project: | All |
| Format: | MBZ |

**0 Indirect Object Access Upper Bound Modify Enable**

| | |
|---|---|
| Project: | All |
| Format: | Enable |

The bound in this dword is updated only when this bit is set.

| Value | Name | Description | Project |
|---|---|---|---|
| 0h | Disable | Ignore the updated bound | All |
| 1h | Enable | Modify the bound | All |

**9 31:12 Instruction Access Upper Bound**

| | |
|---|---|
| Project: | All |
| Format: | GraphicsAddress[31:12] |

This field specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address access by an EU instruction. Instruction data accessed at this address and beyond will return UNDEFINED values. Setting this field to 0 will cause this range check to be ignored.

If non-zero, this address must be greater than the **Instruction Base Address**.

**11:1 Reserved**

| | |
|---|---|
| Project: | All |
| Format: | MBZ |

**0 Instruction Access Upper Bound Modify Enable**

| | |
|---|---|
| Project: | All |
| Format: | Enable |

The bound in this dword is updated only when this bit is set.

| Value | Name | Description | Project |
|---|---|---|---|
| 0h | Disable | Ignore the updated bound | All |
| 1h | Enable | Modify the bound | All |

## 3.5.2 SWTESS_BASE_ADDRESS

The SWTESS_BASE_ADDRESS command sets the base pointers for SW Tessellation data read accesses by the TE unit.

### SWTESS_BASE_ADDRESS

Length Bias: 2

The SWTESS_BASE_ADDRESS command sets the base pointers for SW Tessellation data read access by the TE unit.

**Programming Notes**

This base address must also be comprehended in the SURFACE_STATE used by the HS kernel to write the SW tessellation data.

Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance.

| DWord | Bit | Description |
|---|---|---|
| 0 | 31:29 | **Command Type**<br>Default Value: 3h GFXPIPE |
| | 28:27 | **Command SubType**<br>Default Value: 0h GFXPIPE_COMMON |
| | 26:24 | **3D Command Opcode**<br>Default Value: 1h GFXPIPE_NONPIPELINED |
| | 23:16 | **3D Command Sub Opcode**<br>Default Value: 03h SWTESS_BASE_ADDRESS |
| | 15:8 | **Reserved**<br>Project: All<br>Format: MBZ |
| | 7:0 | **DWord Length**<br>Project: All<br>Format: =n Total Length - 2<br><br>Value / Name / Description<br>0h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1) |
| 1 | 31:12 | **SW Tessellation Base Address [31:12]**<br>Project: All<br>Format: GraphicsAddress[31:12]<br>Specifies the 4K-byte aligned base address for TE unit SW tessellation data read accesses. |
| | 11:8 | **SW Tessellation Memory Object Control State**<br>Project: All<br>Format: MEMORY_OBJECT_CONTROL_STATE<br>Specifies the memory object control state used by the TE unit to read SW tessellation data from memory. |
| | 7:0 | **Reserved**<br>Project: All |

| SWTESS_BASE_ADDRESS | | | |
|---|---|---|---|
| | | Format: | MBZ |

## 3.6 Instruction and State Prefetch

The STATE_PREFETCH command is provided strictly as an optional mechanism to possibly enhance pipeline performance by prefetching data into the GPE's Instruction and State Cache (ISC).

### 3.6.1 STATE_PREFETCH

**STATE_PREFETCH**

| | |
|---|---|
| Project: | All |
| Length Bias: | 2 |

(This command is provided strictly for performance optimization opportunities, and likely requires some experimentation to evaluate the overall impact of additional prefetching.)

The STATE_PREFETCH command causes the GPE to attempt to prefetch a sequence of 64-byte cache lines into the GPE-internal cache ("L2 ISC") used to access EU kernel instructions and fixed/shared function indirect state data. While state descriptors, surface state, and sampler state are automatically prefetched by the GPE, this command may be used to prefetch data not automatically prefetched, such as: 3D viewport state; Media pipeline Interface Descriptors; EU kernel instructions.

| DWord | Bit | Description |
|---|---|---|
| 0 | 31:29 | **Command Type**<br>Default Value: 3h GFXPIPE |
| | 28:27 | **Command SubType**<br>Default Value: 0h GFXPIPE_COMMON |
| | 26:24 | **3D Command Opcode**<br>Default Value: 0h GFXPIPE_PIPELINED |
| | 23:16 | **3D Command Sub Opcode**<br>Default Value: 03h STATE_PREFETCH |
| | 15:8 | **Reserved**<br>Project: All<br>Format: MBZ |
| | 7:0 | **DWord Length**<br>Project: All<br>Format: =n Total Length - 2<br><br>Value / Name / Description<br>0h / DWORD_COUNT_n **[Default]** / Excludes DWord (0,1) |
| 1 | 31:6 | **Prefetch Pointer**<br>Project: All<br>Format: GraphicsAddress[31:6]<br>Specifies the 64-byte aligned address to start the prefetch from. This pointer is an absolute virtual address, it is not relative to any base pointer. |
| | 5:3 | **Reserved**<br>Project: All<br>Format: MBZ |
| | 2:0 | **Prefetch Count** |

| STATE_PREFETCH | | | | |
|---|---|---|---|---|
| | | Project: | All | |
| | | Format: | U3-1 count of cache lines | |
| | | Indicates the number of contiguous 64-byte cache lines that will be prefetched. | | |
| | | **Value** | **Name** | **Description** |
| | | [0,7] | | indicating a count of [1,8] |

# 3.7 System Thread Configuration

## 3.7.1 STATE_SIP

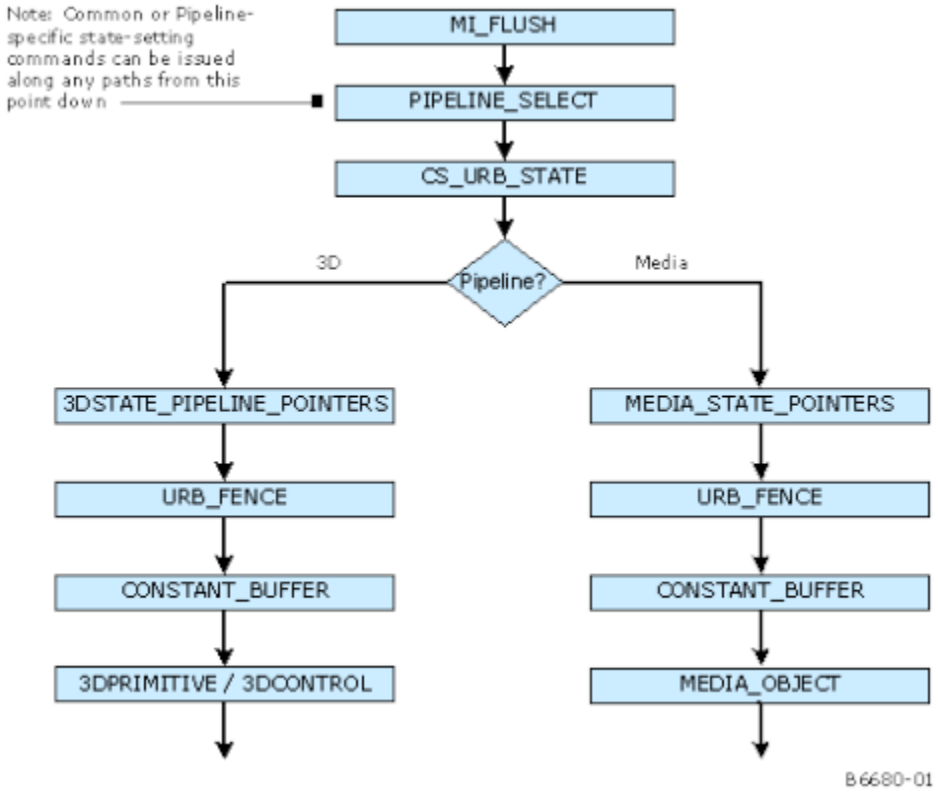
| STATE_SIP | | | | | |
|---|---|---|---|---|---|
| Project: | | | | All | |
| Length Bias: | | | | 2 | |
| The STATE_SIP command specifies the starting instruction location of the System Routine that is shared by all threads in execution. | | | | | |
| **DWord** | **Bit** | **Description** | | | |
| 0 | 31:29 | **Command Type** | | | |
| | | Default Value: | 3h GFXPIPE | | |
| | 28:27 | **Command SubType** | | | |
| | | Default Value: | 0h GFXPIPE_COMMON | | |
| | 26:24 | **3D Command Opcode** | | | |
| | | Default Value: | 1h GFXPIPE_NONPIPELINED | | |
| | 23:16 | **3D Command Sub Opcode** | | | |
| | | Default Value: | 02h STATE_SIP | | |
| | 15:8 | **Reserved** | | | |
| | | Project: | All | | |
| | | Format: | MBZ | | |
| | 7:0 | **DWord Length** | | | |
| | | Project: | All | | |
| | | Format: | =n Total Length - 2 | | |
| | | **Value** | **Name** | **Description** | **Project** |
| | | 0h | DWORD_COUNT_n **[Default]** | Excludes DWord (0,1) | |
| 1 | 31:4 | **System Instruction Pointer (SIP)** | | | |
| | | | | | |
| | | Format: | InstructionBaseOffset[31:4]Kernel | | |
| | | Specifies the instruction address of the system routine associated with the current context as a 128-bit granular offset from the **Instruction Base Address**. SIP is shared by all threads in execution. The address specifies the double quadword aligned instruction location. | | | |

| STATE_SIP | | | |
|---|---|---|---|
| | 3:0 | **Reserved** | |
| | | Project: | All |
| | | Format: | MBZ |

## 3.8 Command Ordering Rules

There are several restrictions regarding the ordering of commands issued to the GPE. This subsection describes these restrictions along with some explanation of why they exist. Refer to the various command descriptions for additional information.

The following flowchart illustrates an example ordering of commands which can be used to perform activity within the GPE.



## 3.8.1 PIPELINE_SELECT

The previously-active pipeline needs to be flushed via the MI_FLUSH command immediately before switching to a different pipeline via use of the PIPELINE_SELECT command. Refer to *GPE Function IDs* for details on the PIPELINE_SELECT command.

### PIPELINE_SELECT

Length Bias: 1

The PIPELINE_SELECT command is used to specify which GPE pipeline (3D or Media or GPGPU) is to be considered the "current" active pipeline. Issuing 3D-pipeline-specific commands when the Media pipeline is selected, or vice versa, is UNDEFINED. This command causes the URB deallocation of the previously selected pipe. For example, switching from the 3D pipe to the Media pipe (either within or between contexts) will cause the CS to send a "Deallocating Flush" down the 3D pipe, and each 3D FF will start a URB deallocation sequence after the current tasks are done. Then, the WM will de-reference the current Constant URB Entry, and all 3D URB entries will be deallocated (after some north bus delay) , which allows the CS to set the URB fences for the media pipe. The process relatively is the same for switching from Media to 3D pipes. The deallocating flush goes down the Media pipe, causing each Media function to start a URB deallocation sequence, and the WM will de-reference the current Constant URB entry and all media entries will be de-allocated to allow the CS to set the 3D pipe.

| Programming Notes | Project |
|---|---|
| Software must ensure the current pipeline is flushed via an MI_FLUSH or PIPE_CONTROL prior to the execution of PIPELINE_SELECT. | |
| Software must send a dummy DRAW after every MI_SET_CONTEXT and after any PIPELINE_SELECT that is enabling 3D mode. A dummy draw is a 3DPRIMITIVE command with Indirect Parameter Enable set to 0, UAV Coherency Required set to 0, Predicate Enable set to 0, End Offset Enable set to 0, and Vertex Count Per Instance set to 0. All other parameters are a don't care. | |

| DWord | Bit | Description |
|---|---|---|
| 0 | 31:29 | **Command Type**<br>Default Value: 3h GFXPIPE |
| | 28:27 | **Command SubType**<br>Default Value: 1h GFXPIPE_COMMON |
| | 26:24 | **3D Command Opcode**<br>Value: 1h — Name: GFXPIPE_NONPIPELINED **[Default]** — Project: |
| | 23:16 | **3D Command Sub Opcode**<br>Default Value: 04h GFXPIPE |
| | 15:2 | **Reserved** |
| | 1:0 | **Pipeline Select**<br>0 — 3D — 3D pipeline is selected<br>1 — Media — Media pipeline is selected (Includes HD Video playback and generic media workloads) |

## 3.8.2 PIPE_CONTROL

The PIPE_CONTROL command does not require URB fencing/allocation to have been performed, nor does it rely on any other pipeline state. It is intended to be used on both the 3D pipe and the Media pipe.

It has special optimizations to support the pipelining capability in the 3D pipe which do not apply to the Media pipe.

### 3.8.3 URB-Related State-Setting Commands

Several commands are used (among other things) to set state variables used in URB entry allocation --- specifically, the **Number of URB Entrie**s and the **URB Entry Allocation Size** state variables associated with various pipeline units. These state variables must be set-up prior to the issuing of a URB_FENCE command. (See the subsection on URB_FENCE.)

CS_URB_STATE (only) specifies these state variables for the common CS FF unit. 3DSTATE_PIPELINED_POINTERs sets the state variables for FF units in the 3D Pipeline, and MEDIA_STATE_POINTERS sets them for the Media pipeline. Depending on which pipeline is currently active, only one of these commands needs to be used. Note that these commands can also be reissued at a later time to change other state variables, though if a change is made to (a) any **Number of URB Entrie**s and the **URB Entry Allocation Size** state variables or (b) the **Maximum Number of Threads** state for the GS or CLIP FF units, a URB_FENCE command must follow.

### 3.8.4 Common Pipeline State-Setting Commands

The following commands are used to set state common to both the 3D and Media pipelines. This state is comprised of CS FF unit state, non-pipelined global state (EU, etc.), and Sampler shared-function state.

- STATE_BASE_ADDRESS
- STATE_SIP
- 3DSTATE_SAMPLER_PALETTE_LOAD
- 3DSTATE_CHROMA_KEY

The state variables associated with these commands must be set appropriately prior to initiating activity within a pipeline (i.e., 3DPRIMITIVE or MEDIA_OBJECT).

### 3.8.5 3D Pipeline-Specific State-Setting Commands

The following commands are used to set state specific to the 3D Pipeline.

- 3DSTATE_PIPELINED_POINTERS
- 3DSTATE_BINDING_TABLE_POINTERS
- 3DSTATE_VERTEX_BUFFERS
- 3DSTATE_VERTEX_ELEMENTS
- 3DSTATE_INDEX_BUFFERS
- 3DSTATE_VF_STATISTICS
- 3DSTATE_DRAWING_RECTANGLE
- 3DSTATE_CONSTANT_COLOR
- 3DSTATE_DEPTH_BUFFER
- 3DSTATE_POLY_STIPPLE_OFFSET
- 3DSTATE_POLY_STIPPLE_PATTERN
- 3DSTATE_LINE_STIPPLE
- 3DSTATE_GLOBAL_DEPTH_OFFSET

The state variables associated with these commands must be set appropriately prior to issuing 3DPRIMITIVE.

### 3.8.6 Media Pipeline-Specific State-Setting Commands

The following command is used to set state specific to the Media pipeline:

- MEDIA_STATE_POINTERS

The state variables associated with this command must be set appropriately prior to issuing MEDIA_OBJECT.

### 3.8.7 CONSTANT_BUFFER (CURBE Load)

The CONSTANT_BUFFER command is used to load constant data into the CURBE URB entries owned by the CS unit. In order to write into the URB, CS URB fencing and allocation must have been established. Therefore, CONSTANT_BUFFER can only be issued after CS_URB_STATE and URB_FENCE commands have been issued, and prior to any other pipeline processing (i.e., 3DPRIMITIVE or MEDIA_OBJECT). See the definition of CONSTANT_BUFFER for more details.

Modifying the CS URB allocation via URB_FENCE invalidates any previous CURBE entries. Therefore software must subsequently [re]issue a CONSTANT_BUFFER command before CURBE data can be used in the pipeline.

### 3.8.8 3DPRIMITIVE

Before issuing a 3DPRIMITIVE command, all state (with the exception of MEDIA_STATE_POINTERS) needs to be valid. Thus the commands used to assigned that state must be issued before issuing 3DPRIMITIVE.

### 3.8.9 MEDIA_OBJECT

Before issuing a MEDIA_OBJECT command, all state (with the exception of 3D-pipeline-specific state) needs to be valid. Therefore the commands used to set this state need to have been issued at some point prior to the issue of MEDIA_OBJECT.

# 4. Video Codec Engine

The parallel Video Codec Engine (VCE) is a fixed function video decoder and encoder engine. It is also referred to as the multi-format codec (MFX) engine, as a unified fixed function pipeline is implemented to support multiple video coding standards such as MPEG2, VC1 and AVC.

- VCS – VCE Command Streamer unit (also referred to as BCS)
- BSD – Bitstream Decoder unit
- VDS – Video Dispatcher unit
- VMC – Video Motion Compensation unit
- VIP – Video Intra Prediction unit
- VIT – Video Inverse Transform unit
- VLF – Video Loop Filter unit
- VFT – Video Forward Transform unit (encoder only)
- BSC – Bitstream Encoder unit (encoder only)

## 4.1 Video Command Streamer (VCS)

VCS (Video Command Streamer) unit is primarily served as the software programming interface between the O/S driver and the MFD Engine. It is responsible for fetching, decoding, and dispatching of data packets (Media Commands with the header DW removed) to the front end interface module of MFX Engine.

Its logic functions include

- MMIO register programming interface.
- Management of the Head pointer for the Ring Buffer.
- Decode of ring data and sending it to the appropriate destination; AVC, VC1 or MPEG2 engine
- Handling of user interrupts and ring context switch interrupt.
- Flushing the MFX Engine
- Handle NOP

The register programming (RM) bus is a dword interface bus that is driven by the Gx Command Streamer. The VCS unit will only claim memory mapped I/O cycles that are targeted to its range of 0x4000 to 0x4FFFF. The Gx and MFX Engines use semaphore to synchronize their operations.

Any interaction and control protocols between the VCS and Gx CS in IronLake will remain the same as in Cantiga. But in Gesher, VCS will operate completely independent of the Gx CS.

The simple sequence of events is as follows: a ring (say PRB0) is programmed by a memory-mapped register write cycle. The DMA inside VCS is kicked off. The DMA fetches commands from memory based on the starting address and head pointer. The DMA requests cache lines from memory (one cacheline CL at a time). There is guaranteed space in the DMA FIFO (16 CL deep) for data coming back from memory. The DMA control logic has copies of the head pointer and the tail pointer. The DMA increments the head pointer after making requests for ring commands. Once the DMA copy of the head pointer becomes equal to the tail pointer, the DMA stops requesting.

The parser starts executing once the DMA FIFO has valid commands. All the commands have a header dword packet. Based on the encoding in the header packet, the command may be targeted towards AVC/VC1/MPEG2 engine or the command parser. After execution of every command, the actual head pointer is updated. The ring is considered empty when the head pointer becomes equal to the tail pointer.

# 5. Graphics Command Formats

## 5.1 Command Formats

This section describes the general format of the graphics device commands.

Graphics commands are defined with various formats. The first DWord of all commands is called the *header* DWord. The header contains the only field common to all commands -- the *client* field that determines the device unit that will process the command data. The Command Parser examines the client field of each command to condition the further processing of the command and route the command data accordingly.

Some devices include two Command Parsers, each controlling an independent processing engine. These will be referred to in this document as the Render Command Parser (RCP) and the Video Codec Command Parser (VCCP).

Valid client values for the Render Command Parser are:

| Client # | Client |
|---|---|
| 0 | Memory Interface (MI_xxx) |
| 1 | Miscellaneous |
| 2 | 2D Rendering (xxx_BLT_xxx) |
| 3 | Graphics Pipeline (3D and Media) |
| 4-7 | Reserved |

Valid client values for the Video Codec Command Parser are:

| Client # | Client |
|---|---|
| 0 | Memory Interface (MI_xxx) |
| 1-2 | Reserved |
| 3 | AVC and VC1 State and Object Commands |
| 4-7 | Reserved |

Graphics commands vary in length, though are always multiples of DWords. The length of a command is either:

- Implied by the client/opcode
- Fixed by the client/opcode yet included in a header field (so the Command Parser explicitly knows how much data to copy/process)
- Variable, with a field in the header indicating the total length of the command

Note that command *sequences* require QWord alignment and padding to QWord length to be placed in Ring and Batch Buffers.

The following subsections provide a brief overview of the graphics commands by client type provides a diagram of the formats of the header DWords for all commands. Following that is a list of command mnemonics by client type.

### 5.1.1 Memory Interface Commands

Memory Interface (MI) commands are basically those commands which do not require processing by the 2D or 3D Rendering/Mapping engines. The functions performed by these commands include:

Control of the command stream (e.g., Batch Buffer commands, breakpoints, ARB On/Off, etc.)

Hardware synchronization (e.g., flush, wait-for-event)

Software synchronization (e.g., Store DWORD, report head)

Graphics buffer definition (e.g., Display buffer, Overlay buffer)

Miscellaneous functions

Refer to the *Memory Interface Commands* chapter for a description of these commands.

## 5.1.2 2D Commands

The 2D commands include various flavors of BLT operations, along with commands to set up BLT engine state without actually performing a BLT. Most commands are of fixed length, though there are a few commands that include a variable amount of "inline" data at the end of the command.

See the *2D Commands* chapter for a description of these commands.

## 5.1.3 3D/Media Commands

The 3D/Media commands are used to program the graphics pipelines for 3D or media operations.

Refer to the *3D* chapter for a description of the 3D state and primitive commands and the *Media* chapter for a description of the media-related state and object commands.

## 5.1.4 Video Codec Commands

### 5.1.4.1 MFX Commands

The MFX (MFD for decode and MFC for encode) commands are used to program the multi-format codec engine attached to the Video Codec Command Parser. See the *MFD and MFC* chapters for a description of these commands.

## 5.1.5 Command Header

**RCP Command Header Format**

| Bits | | | | | |
|---|---|---|---|---|---|
| **TYPE** | **31:29** | **28:24** | **23** | **22** | **21:0** |
| Memory Interface (MI) | 000 | Opcode<br>00h – NOP<br>0Xh – Single DWord Commands<br>1Xh – Two+ DWord Commands<br>2Xh – Store Data Commands<br>3Xh – Ring/Batch Buffer Cmds | | Identification No./DWord Count<br>Command Dependent Data<br>5:0 – DWord Count<br>5:0 – DWord Count<br>5:0 – DWord Count | |
| Reserved | 001 | Opcode – 11111 | 23:19<br>Sub Opcode 00h – | 18:16<br>Re-served | 15:0<br>DWord Count |

| Bits | | | | | | | |
|---|---|---|---|---|---|---|---|
| TYPE | 31:29 | 28:24 | | 23 | 22 | 21:0 | |
| | | | | 01h | | | |
| 2D | 010 | Opcode | | | | Command Dependent Data<br>4:0 – DWord Count | |
| TYPE | 31:29 | 28:27 | 26:24 | 23:16 | | 15:8 | 7:0 |
| Common | 011 | 00 | Opcode – 000 | Sub Opcode | | Data | DWord Count |
| Common (NP) | 011 | 00 | Opcode – 001 | Sub Opcode | | Data | DWord Count |
| Reserved | 011 | 00 | Opcode – 010 – 111 | | | | |
| Single Dword Command | 011 | 01 | Opcode – 000 – 001 | Sub Opcode | | | N/A |
| Reserved | 011 | 01 | Opcode – 010 – 111 | | | | |
| Media State | 011 | 10 | Opcode – 000 | Sub Opcode | | | Dword Count |
| Media Object | 011 | 10 | Opcode – 001 – 010 | Sub Opcode | | Dword Count | |
| Reserved | 011 | 10 | Opcode – 011 – 111 | | | | |
| 3DState | 011 | 11 | Opcode – 000 | Sub Opcode | | Data | DWord Count |
| 3DState (NP) | 011 | 11 | Opcode – 001 | Sub Opcode | | Data | DWord Count |
| PIPE_Control | 011 | 11 | Opcode – 010 | | | Data | DWord Count |
| 3DPrimitive | 011 | 11 | Opcode – 011 | | | Data | DWord Count |
| Reserved | 011 | 11 | Opcode – 100 – 111 | | | | |
| Reserved | 100 | XX | | | | | |
| Reserved | 101 | XX | | | | | |
| Reserved | 110 | XX | | | | | |
| Fulsim[2] | 111 | XX | | | | | |

Notes:

1. The qualifier “NP” indicates that the state variable is non-pipelined and the render pipe is flushed before such a state variable is updated. The other state variables are pipelined (default).
2. [31:29] == '111' is reserved for fulsim command decodings. It is invalid for HW to parse this command.

### VCCP Command Header Format

| Bits | | | | | | |
|---|---|---|---|---|---|---|
| TYPE | 31:29 | 28:24 | | 23 | 22 | 21:0 |
| Memory Interface (MI) | 000 | Opcode<br>00h – NOP<br>0Xh – Single DWord Commands<br>1Xh – Reserved<br>2Xh – Store Data Commands<br>3Xh – Ring/Batch Buffer Cmds | | | | Identification No./DWord Count<br>Command Dependent Data<br>5:0 – DWord Count<br>5:0 – DWord Count<br>5:0 – DWord Count |
| TYPE | 31:29 | 28:27 | 26:24 | 23:16 | | 15:0 |
| Reserved | 011 | 00 | XXX | XX | | |
| MFX Single DW | 011 | 01 | 000 | Opcode: 0h | | 0 |
| Reserved | 011 | 01 | 1XX | | | |
| Reserved | 011 | 10 | 0XX | | | |
| AVC State | 011 | 10 | 100 | Opcode: 0h – 4h | | DWord Count |

| Bits | | | | | | |
|---|---|---|---|---|---|---|
| TYPE | 31:29 | 28:24 | | 23 22 | | 21:0 |
| AVC Object | 011 | 10 | 100 | Opcode: 8h | | DWord Count |
| VC1 State | 011 | 10 | 101 | Opcode: 0h – 4h | | DWord Count |
| VC1 Object | 011 | 10 | 101 | Opcode: 8h | | DWord Count |
| Reserved | 011 | 10 | 11X | | | |
| Reserved | 011 | 11 | XXX | | | |
| TYPE | 31:29 | 28:27 | 26:24 | 23:21 | 20:16 | 15:0 |
| MFX Common | 011 | 10 | 000 | 000 | subopcode | DWord Count |
| Reserved | 011 | 10 | 000 | 001-111 | subopcode | DWord Count |
| AVC Common | 011 | 10 | 001 | 000 | subopcode | DWord Count |
| AVC Dec | 011 | 10 | 001 | 001 | subopcode | DWord Count |
| AVC Enc | 011 | 10 | 001 | 010 | subopcode | DWord Count |
| Reserved | 011 | 10 | 001 | 011-111 | subopcode | DWord Count |
| Reserved (for VC1 Common) | 011 | 10 | 010 | 000 | subopcode | DWord Count |
| VC1 Dec | 011 | 10 | 010 | 001 | subopcode | DWord Count |
| Reserved (for VC1 Enc) | 011 | 10 | 010 | 010 | subopcode | DWord Count |
| Reserved | 011 | 10 | 010 | 011-111 | subopcode | DWord Count |
| Reserved (MPEG2 Common) | 011 | 10 | 011 | 000 | subopcode | DWord Count |
| MPEG2 Dec | 011 | 10 | 011 | 001 | subopcode | DWord Count |
| Reserved (for MPEG2 Enc) | 011 | 10 | 011 | 010 | subopcode | DWord Count |
| Reserved | 011 | 10 | 011 | 011-111 | subopcode | DWord Count |
| Reserved | 011 | 10 | 100-111 | XXX | | |

## 5.2 Command Map

This section provides a map of the graphics command opcodes.

### 5.2.1 Memory Interface Command Map

All the following commands are defined in *Memory Interface Commands*.

**Memory Interface Commands for RCP**

| Opcode (28:23) | Command | Pipe | | | |
|---|---|---|---|---|---|
| | | Render | | Blitter | |
| 1-DWord | | | | | |
| 00h | MI_NOOP | All | All | All | All |
| 01h | | | | | |
| 02h | MI_USER_INTERRUPT | All | All | All | All |
| 03h | MI_WAIT_FOR_EVENT | All | All | All | All |
| 04h | | All | | | |
| 05h | MI_ARB_CHECK | All | All | All | All |
| 06h | | | | | |
| 07h | MI_REPORT_HEAD | All | All | All | All |

| Opcode (28:23) | Command | Pipe | | | |
|---|---|---|---|---|---|
| | | Render | | Blitter | |
| **1-DWord** | | | | | |
| 08h | MI_ARB_ON_OFF | | | | |
| 0Ah | MI_BATCH_BUFFER_END | All | All | All | All |
| 0Bh | MI_SUSPEND_FLUSH | | | | |
| 0Ch | MI_PREDICATE | | | | |
| 0Dh | MI_TOPOLOGY_FILTER | | | | |
| 0Eh | MI_SET_APPID | | | | |
| **2+ DWord** | | | | | |
| 10h | Reserved | | | | |
| 11h | Reserved | | | | |
| 12h | Reserved | | | | |
| 13h | Reserved | | | | |
| 14h | MI_DISPLAY_FLIP | All | | | |
| 15h | Reserved | | | | |
| 16h | MI_SEMAPHORE_MBOX | | | | |
| 17h | Reserved | | | | |
| 18h | MI_SET_CONTEXT | All | | | |
| 19h | | | | | |
| 1Ah | MI_MATH | | | | |
| 1Dh–1Fh | Reserved | | | | |
| **Store Data** | | | | | |
| 20h | MI_STORE_DATA_IMM | All | All | All | All |
| 21h | MI_STORE_DATA_INDEX | All | All | All | All |
| 22h | MI_LOAD_REGISTER_IMM | All | All | All | All |
| 23h | MI_UPDATE_GTT | | | | |
| 24h | MI_STORE_REGISTER_MEM | All | All | All | All |
| 25h | | | | | |
| 26h | MI_FLUSH_DW | | | | |
| 27h | MI_CLFLUSH | | | | |
| 28h | MI_REPORT_PERF_COUNT | | | | |
| 29h | MI_LOAD_REGISTER_MEM | | | | |
| 2Ah | | | | | |
| 2Bh | | | | | |
| 2Ch | | | | | |
| 2Dh | | | | | |

| Opcode (28:23) | Command | Pipe | | | |
|---|---|---|---|---|---|
| | | Render | | Blitter | |
| **1-DWord** | | | | | |
| 2Eh | | | | | |
| 2Fh | | | | | |
| **Ring/Batch Buffer** | | | | | |
| 30h | Reserved | | | | |
| 31h | MI_BATCH_BUFFER_START | All | All | All | All |
| 32h–35h | Reserved | | | | |
| 36h | MI_CONDITIONAL_BATCH_BUFFER_END | | | | |
| 37h–3Fh | Reserved | | | | |

## 5.2.2 2D Command Map

All the following commands are defined in *Blitter Instructions*.

### 2D Command Map

| Opcode (28:22) | Command |
|---|---|
| 00h | Reserved |
| 01h | XY_SETUP_BLT |
| 02h | Reserved |
| 03h | XY_SETUP_CLIP_BLT |
| 04h-10h | Reserved |
| 11h | XY_SETUP_MONO_PATTERN_SL_BLT |
| 12h-23h | Reserved |
| 24h | XY_PIXEL_BLT |
| 25h | XY_SCANLINES_BLT |
| 26h | XY_TEXT_BLT |
| 27h-30h | Reserved |
| 31h | XY_TEXT_IMMEDIATE_BLT |
| 32h-3Fh | Reserved |
| 40h | COLOR_BLT |
| 41h-42h | Reserved |
| 43h | SRC_COPY_BLT |
| 44h-4Fh | Reserved |
| 50h | XY_COLOR_BLT |
| 51h | XY_PAT_BLT |
| 52h | XY_MONO_PAT_BLT |
| 53h | XY_SRC_COPY_BLT |
| 54h | XY_MONO_SRC_COPY_BLT |
| 55h | XY_FULL_BLT |
| 56h | XY_FULL_MONO_SRC_BLT |
| 57h | XY_FULL_MONO_PATTERN_BLT |
| 58h | XY_FULL_MONO_PATTERN_MONO_SRC_BLT |
| 59h | XY_MONO_PAT_FIXED_BLT |
| 5Ah-70h | Reserved |
| 71h | XY_MONO_SRC_COPY_IMMEDIATE_BLT |

| Opcode (28:22) | Command |
|---|---|
| 72h | XY_PAT_BLT_IMMEDIATE |
| 73h | XY_SRC_COPY_CHROMA_BLT |
| 74h | XY_FULL_IMMEDIATE_PATTERN_BLT |
| 75h | XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT |
| 76h | XY_PAT_CHROMA_BLT |
| 77h | XY_PAT_CHROMA_BLT_IMMEDIATE |
| 78h-7Fh | Reserved |

## 5.2.3 3D/Media Command Map

For all commands listed in 6.2.3, 3D/Media Command Map, the Pipeline Type (bits 28:27) is 3h, indicating the 3D Pipeline.

### 3D/Media Command Map

| Opcode Bits 26:24 | Sub Opcode Bits 23:16 | Command | Definition Chapter |
|---|---|---|---|
| 0h | 03h | Reserved | |
| 0h | 04h | 3DSTATE_CLEAR_PARAMS | 3D Pipeline |
| 0h | 05h | 3DSTATE_DEPTH_BUFFER | 3D Pipeline |
| 0h | 06h | 3DSTATE_STENCIL_BUFFER | 3D Pipeline |
| 0h | 07h | 3DSTATE_HIER_DEPTH_BUFFER | 3D Pipeline |
| 0h | 08h | 3DSTATE_VERTEX_BUFFERS | Vertex Fetch |
| 0h | 09h | 3DSTATE_VERTEX_ELEMENTS | Vertex Fetch |
| 0h | 0Ah | 3DSTATE_INDEX_BUFFER | Vertex Fetch |
| 0h | 0Bh | 3DSTATE_VF_STATISTICS | Vertex Fetch |
| 0h | 0Ch | Reserved | |
| 0h | 0Dh | 3DSTATE_VIEWPORT_STATE_POINTERS | 3D Pipeline |
| 0h | 10h | 3DSTATE_VS | Vertex Shader |
| 0h | 11h | 3DSTATE_GS | Geometry Shader |
| 0h | 12h | 3DSTATE_CLIP | Clipper |
| 0h | 13h | 3DSTATE_SF | Strips & Fans |
| 0h | 14h | 3DSTATE_WM | Windower |
| 0h | 15h | 3DSTATE_CONSTANT_VS | Vertex Shader |
| 0h | 16h | 3DSTATE_CONSTANT_GS | Geometry Shader |
| 0h | 17h | 3DSTATE_CONSTANT_PS | Windower |
| 0h | 18h | 3DSTATE_SAMPLE_MASK | Windower |

| Opcode Bits 26:24 | Sub Opcode Bits 23:16 | Command | Definition Chapter |
|---|---|---|---|
| 0h | 19h | 3DSTATE_CONSTANT_HS | Hull Shader |
| 0h | 1Ah | 3DSTATE_CONSTANT_DS | Domain Shader |
| 0h | 1Bh | 3DSTATE_HS | Hull Shader |
| 0h | 1Ch | 3DSTATE_TE | Tesselator |
| 0h | 1Dh | 3DSTATE_DS | Domain Shader |
| 0h | 1Eh | 3DSTATE_STREAMOUT | HW Streamout |
| 0h | 1Fh | 3DSTATE_SBE | Setup |
| 0h | 20h | 3DSTATE_PS | Pixel Shader |
| 0h | 21h | Reserved | |
| 0h | 22h | 3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP | Strips & Fans |
| 0h | 23h | 3DSTATE_VIEWPORT_STATE_POINTERS_CC | Windower |
| 0h | 24h | 3DSTATE_BLEND_STATE_POINTERS | Pixel Shader |
| 0h | 25h | 3DSTATE_DEPTH_STENCIL_STATE_POINTERS | Pixel Shader |
| 0h | 26h | 3DSTATE_BINDING_TABLE_POINTERS_VS | Vertex Shader |
| 0h | 27h | 3DSTATE_BINDING_TABLE_POINTERS_HS | Hull Shader |
| 0h | 28h | 3DSTATE_BINDING_TABLE_POINTERS_DS | Domain Shader |
| 0h | 29h | 3DSTATE_BINDING_TABLE_POINTERS_GS | Geometry Shader |
| 0h | 2Ah | 3DSTATE_BINDING_TABLE_POINTERS_PS | Pixel Shader |
| 0h | 2Bh | 3DSTATE_SAMPLER_STATE_POINTERS_VS | Vertex Shader |
| 0h | 2Ch | 3DSTATE_SAMPLER_STATE_POINTERS_HS | Hull Shader |
| 0h | 2Dh | 3DSTATE_SAMPLER_STATE_POINTERS_DS | Domain Shader |
| 0h | 2Eh | 3DSTATE_SAMPLER_STATE_POINTERS_GS | Geometry Shader |
| 0h | 2Fh | Reserved | |
| 0h | 30h | 3DSTATE_URB_VS | Vertex Shader |
| 0h | 31h | 3DSTATE_URB_HS | Hull Shader |
| 0h | 32h | 3DSTATE_URB_DS | Domain Shader |

| Opcode Bits 26:24 | Sub Opcode Bits 23:16 | Command | Definition Chapter |
|---|---|---|---|
| 0h | 33h | 3DSTATE_URB_GS | Geometry Shader |
| 0h | 48h-4Bh | Reserved | |
| 0h | 4Ch | 3DSTATE_WM_CHROMA_KEY | Windower |
| 0h | 4Dh | 3DSTATE_PS_BLEND | Windower |
| 0h | 4Eh | 3DSTATE_WM_DEPTH_STENCIL | Windower |
| 0h | 4Fh | 3DSTATE_PS_EXTRA | Windower |
| 0h | 50h | 3DSTATE_RASTER | Strips & Fans |
| 0h | 51h | 3DSTATE_SBE_SWIZ | Strips & Fans |
| 0h | 52h | 3DSTATE_WM_HZ_OP | Windower |
| 0h | 53h | 3DSTATE_INT (internally generated state) | 3D Pipeline |
| 0h | 54h | Reserved | |
| 0h | 56h-FFh | Reserved | |
| 1h | 00h | 3DSTATE_DRAWING_RECTANGLE | Strips & Fans |
| 1h | 02h | 3DSTATE_SAMPLER_PALETTE_LOAD0 | Sampling Engine |
| 1h | 03h | Reserved | |
| 1h | 04h | 3DSTATE_CHROMA_KEY | Sampling Engine |
| 1h | 05h | Reserved | |
| 1h | 06h | 3DSTATE_POLY_STIPPLE_OFFSET | Windower |
| 1h | 07h | 3DSTATE_POLY_STIPPLE_PATTERN | Windower |
| 1h | 08h | 3DSTATE_LINE_STIPPLE | Windower |
| 1h | 0Ah | 3DSTATE_AA_LINE_PARAMS | Windower |
| 1h | 0Bh | 3DSTATE_GS_SVB_INDEX | Geometry Shader |
| 1h | 0Ch | 3DSTATE_SAMPLER_PALETTE_LOAD1 | Sampling Engine |
| 1h | 0Dh | 3DSTATE_MULTISAMPLE | Windower |
| 1h | 0Eh | 3DSTATE_STENCIL_BUFFER | Windower |
| 1h | 0Fh | 3DSTATE_HIER_DEPTH_BUFFER | Windower |
| 1h | 10h | 3DSTATE_CLEAR_PARAMS | Windower |
| 1h | 11h | 3DSTATE_MONOFILTER_SIZE | Sampling Engine |
| 1h | 12h | 3DSTATE_PUSH_CONSTANT_ALLOC_VS | Vertex Shader |
| 1h | 13h | 3DSTATE_PUSH_CONSTANT_ALLOC_HS | Hull Shader |
| 1h | 14h | 3DSTATE_PUSH_CONSTANT_ALLOC_DS | Domain Shader |
| 1h | 15h | 3DSTATE_PUSH_CONSTANT_ALLOC_GS | Geometry Shader |
| 1h | 16h | 3DSTATE_PUSH_CONSTANT_ALLOC_PS | Pixel Shader |
| 1h | 17h | 3DSTATE_SO_DECL_LIST | HW Streamout |
| 1h | 18h | 3DSTATE_SO_BUFFER | HW Streamout |

| Opcode Bits 26:24 | Sub Opcode Bits 23:16 | Command | Definition Chapter |
|---|---|---|---|
| 1h | 1Ch | 3DSTATE_SAMPLE_PATTERN | Windower |
| 1h | 1Dh | 3DSTATE_URB_CLEAR | 3D Pipeline |
| 1h | 1Eh-FFh | Reserved | |
| 2h | 00h | PIPE_CONTROL | 3D Pipeline |
| 2h | 01h-FFh | Reserved | |
| 3h | 00h | 3DPRIMITIVE | Vertex Fetch |
| 3h | 01h-FFh | Reserved | |
| 4h-7h | 00h-FFh | Reserved | |

| Pipeline Type (28:27) | Opcode | Sub Opcode | Command | Definition Chapter |
|---|---|---|---|---|
| **Common (pipelined)** | **Bits 26:24** | **Bits 23:16** | | |
| 0h | 0h | 03h | STATE_PREFETCH | Graphics Processing Engine |
| 0h | 0h | 04h-FFh | Reserved | |
| **Common (non-pipelined)** | **Bits 26:24** | **Bits 23:16** | | |
| 0h | 1h | 00h | Reserved | n/a |
| 0h | 1h | 01h | STATE_BASE_ADDRESS | Graphics Processing Engine |
| 0h | 1h | 02h | STATE_SIP | Graphics Processing Engine |
| 0h | 1h | 03h | SWTESS BASE ADDRESS | 3D Pipeline |
| 0h | 1h | 04h | GPGPU CSR BASE ADDRESS | Graphics Processing Engine |
| 0h | 1h | 04h–FFh | Reserved | n/a |
| **Reserved** | **Bits 26:24** | **Bits 23:16** | | |
| 0h | 2h–7h | XX | Reserved | n/a |

## 5.2.4 Video Codec Command Map

### 5.2.4.1 MFX Common Command Map

MFX state commands support direct state model and indirect state model. Recommended usage of indirect state model is provided here (as a software usage guideline).

| Pipeline Type (28:27) | Opcode (26:24) | SubopA (23:21) | SubopB (20:16) | Command | Chapter | Recommended Indirect State Pointer Map | Interruptable ? |
|---|---|---|---|---|---|---|---|
| MFX Common (State) | | | | | | | |

| Pipeline Type (28:27) | Opcode (26:24) | SubopA (23:21) | SubopB (20:16) | Command | Chapter | Recommended Indirect State Pointer Map | Interruptable ? |
|---|---|---|---|---|---|---|---|
| 2h | 0h | 0h | 0h | MFX_PIPE_MODE_SELECT | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 1h | MFX_SURFACE_STATE | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 2h | MFX_PIPE_BUF_ADDR_STATE | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 3h | MFX_IND_OBJ_BASE_ADDR_STATE | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 4h | MFX_BSP_BUF_BASE_ADDR_STATE | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 5h | | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 6h | MFX_ STATE_POINTER | MFX | IMAGE | n/a |
| 2h | 0h | 0h | 7-8h | Reserved | n/a | n/a | n/a |
| MFX Common (Object) | | | | | | | |
| 2h | 0h | 1h | 9h | MFD_ IT_OBJECT | MFX | n/a | Yes |
| 2h | 0h | 0h | 4-1Fh | Reserved | n/a | n/a | n/a |
| AVC Common (State) | | | | | | | |
| 2h | 1h | 0h | 0h | MFX_AVC_IMG_STATE | MFX | IMAGE | n/a |
| 2h | 1h | 0h | 1h | MFX_AVC_QM_STATE | MFX | IMAGE | n/a |
| 2h | 1h | 0h | 2h | MFX_AVC_DIRECTMODE_STATE | MFX | SLICE | n/a |
| 2h | 1h | 0h | 3h | MFX_AVC_SLICE_STATE | MFX | SLICE | n/a |
| 2h | 1h | 0h | 4h | MFX_AVC_REF_IDX_STATE | MFX | SLICE | n/a |
| 2h | 1h | 0h | 5h | MFX_AVC_WEIGHTOFFSET_STATE | MFX | SLICE | n/a |
| 2h | 1h | 0h | 6-1Fh | Reserved | n/a | n/a | n/a |
| AVC Dec | | | | | | | |
| 2h | 1h | 1h | 0-7h | Reserved | n/a | n/a | n/a |
| 2h | 1h | 1h | 8h | MFD_AVC_BSD_OBJECT | MFX | n/a | No |
| 2h | 1h | 1h | 9-1Fh | Reserved | n/a | n/a | n/a |
| AVC Enc | | | | | | | |
| 2h | 1h | 2h | 0-1h | Reserved | n/a | n/a | n/a |
| 2h | 1h | 2h | 2h | MFC_AVC_FQM_STATE | MFX | IMAGE | n/a |
| 2h | 1h | 2h | 3-7h | Reserved | n/a | n/a | n/a |
| 2h | 1h | 2h | 8h | MFC_AVC_PAK_INSERT_OBJECT | MFX | n/a | n/a |
| 2h | 1h | 2h | 9h | MFC_AVC_PAK_OBJECT | MFX | n/a | Yes |
| 2h | 1h | 2h | A-1Fh | Reserved | n/a | n/a | n/a |
| 2h | 1h | 2h | 0-1Fh | Reserved | n/a | n/a | n/a |
| VC1 Common | | | | | | | |
| 2h | 2h | 0h | 0h | MFX_VC1_PIC_STATE | MFX | IMAGE | n/a |
| 2h | 2h | 0h | 1h | MFX_VC1_PRED_PIPE_S | MFX | IMAGE | n/a |

| Pipeline Type (28:27) | Opcode (26:24) | SubopA (23:21) | SubopB (20:16) | Command | Chapter | Recommended Indirect State Pointer Map | Interruptable ? |
|---|---|---|---|---|---|---|---|
| | | | | TATE | | | |
| 2h | 2h | 0h | 2h | MFX_VC1_DIRECTMODE_STATE | MFX | SLICE | n/a |
| 2h | 2h | 0h | 2-1Fh | Reserved | n/a | n/a | n/a |
| VC1 Dec | | | | | | | |
| 2h | 2h | 1h | 0-7h | Reserved | n/a | n/a | n/a |
| 2h | 2h | 1h | 8h | MFD_VC1_BSD_OBJECT | MFX | n/a | Yes |
| 2h | 2h | 1h | 9-1Fh | Reserved | n/a | n/a | n/a |
| VC1 Enc | | | | | | | |
| 2h | 2h | 2h | 0-1Fh | Reserved | n/a | n/a | n/a |
| MPEG2Common | | | | | | | |
| 2h | 3h | 0h | 0h | MFX_MPEG2_PIC_STATE | MFX | IMAGE | n/a |
| 2h | 3h | 0h | 1h | MFX_MPEG2_QM_STATE | MFX | IMAGE | n/a |
| 2h | 3h | 0h | 2-1Fh | Reserved | n/a | n/a | n/a |
| MPEG2 Dec | | | | | | | |
| 2h | 3h | 1h | 1-7h | Reserved | n/a | n/a | n/a |
| 2h | 3h | 1h | 8h | MFD_MPEG2_BSD_OBJECT | MFX | n/a | Yes |
| 2h | 3h | 1h | 9-1Fh | Reserved | n/a | n/a | n/a |
| MPEG2 Enc | | | | | | | |
| 2h | 3h | 2h | 0-1Fh | Reserved | n/a | n/a | n/a |
| The Rest | | | | | | | |
| 2h | 4-5h, 7h | x | x | Reserved | n/a | n/a | n/a |

# 6. Memory Data Formats

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

## 6.1 Memory Object Overview

Any memory data accessed by the device is considered part of a *memory object* of some memory object type.

### 6.1.1 Memory Object Types

The following table lists the various memory objects types and an indication of their role in the system.

| Memory Object Type | Role |
|---|---|
| Graphics Translation Table (GTT) | Contains PTEs used to translate “graphics addresses” into physical memory addresses. |
| Hardware Status Page | Cached page of sysmem used to provide fast driver synchronization. |
| Logical Context Buffer | Memory areas used to store (save/restore) images of hardware rendering contexts. Logical contexts are referenced via a pointer to the corresponding Logical Context Buffer. |
| Ring Buffers | Buffers used to transfer (DMA) instruction data to the device. Primary means of controlling rendering operations. |
| Batch Buffers | Buffers of instructions invoked indirectly from Ring Buffers. |
| State Descriptors | Contains state information in a prescribed layout format to be read by hardware. Many different state descriptor formats are supported. |
| Vertex Buffers | Buffers of 3D vertex data indirectly referenced through “indexed” 3D primitive instructions. |
| VGA Buffer<br>(Must be mapped UC on PCI) | Graphics memory buffer used to drive the display output while in legacy VGA mode. |
| Display Surface | Memory buffer used to display images on display devices. |
| Overlay Surface | Memory buffer used to display overlaid images on display devices. |
| Overlay Register, Filter Coefficients Buffer | Memory area used to provide double-buffer for Overlay register and filter coefficient loading. |
| Cursor Surface | Hardware cursor pattern in memory. |
| 2D Render Source | Surface used as primary input to 2D rendering operations. |
| 2D Render R-M-W Destination | 2D rendering output surface that is read in order to be combined in the rendering function. Destination surfaces that accessed via this Read-Modify-Write mode have somewhat different restrictions than Write-Only Destination surfaces. |
| 2D Render Write-Only Destination | 2D rendering output surface that is written but not read by the 2D rendering function. Destination surfaces that accessed via a Write-Only mode have somewhat different restrictions than Read-Modify-Write Destination surfaces. |
| 2D Monochrome | 1 bpp surfaces used as inputs to 2D rendering after being converted to foreground/background |

| Memory Object Type | Role |
|---|---|
| Source | colors. |
| 2D Color Pattern | 8x8 pixel array used to supply the “pattern” input to 2D rendering functions. |
| DIB | “Device Independent Bitmap” surface containing “logical” pixel values that are converted (via LUTs) to physical colors. |
| 3D Color Buffer | Surface receiving color output of 3D rendering operations. May also be accessed via R-M-W (aka blending). Also referred to as a Render Target. |
| 3D Depth Buffer | Surface used to hold per-pixel depth and stencil values used in 3D rendering operations. Accessed via RMW. |
| 3D Texture Map | Color surface (or collection of surfaces) which provide texture data in 3D rendering operations. |
| “Non-3D” Texture | Surface read by Texture Samplers, though not in normal 3D rendering operations (e.g., in video color conversion functions). |
| Motion Comp Surfaces | These are the Motion Comp reference pictures. |
| Motion Comp Correction Data Buffer | This is Motion Comp intra-coded or inter-coded correction data. |

# 6.2 Channel Formats

## 6.2.1 Unsigned Normalized (UNORM)

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

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

## 6.2.2 Gamma Conversion (SRGB)

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

## 6.2.3 Signed Normalized (SNORM)

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

## 6.2.4 Unsigned Integer (UINT/USCALED)

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

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

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

## 6.2.5 Signed Integer (SINT/SSCALED)

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

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

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

## 6.2.6 Floating Point (FLOAT)

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

### 6.2.6.1 32-bit Floating Point

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

The value of this data type is derived as:

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

### 6.2.6.2 64-bit Floating Point

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

The value of this data type is derived as:

- if e == b'11..11' and f != 0, then v is NaN regardless of s
- if e == b'11..11' and f == 0, then v = $(-1)^s$*infinity (signed infinity)

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

### 6.2.6.3 16-bit Floating Point

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

The value of this data type is derived as:

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

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

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

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

### 6.2.6.4 11-bit Floating Point

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

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

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

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

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

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

### 6.2.6.5 10-bit Floating Point

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

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

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

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

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

### 6.2.6.6 Shared Exponent

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

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

| Bit | Description |
|---|---|
| 31:27 | **Exponent (e)** Biased Exponent |
| 26:18 | **Blue Fraction** |

| Bit | Description |
|---|---|
| 17:9 | **Green Fraction** |
| 8:0 | **Red Fraction** |

## 6.2.7 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.

## 6.2.8 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.

## 6.2.9 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.

## 6.2.10 Luminance Formats

All surface formats contaning “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.

## 6.2.11 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**<br>On texture reads, this (unsigned) 1-bit value is replicated to all color channels.<br>Format: U1 |
| ... | **...** |
| T7 | **Texel 7**<br>On texture reads, this (unsigned) 1-bit value is replicated to all color channels.<br>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.

## 6.2.12 Palette Formats

### 6.2.12.1 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**<br>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.<br>Format: U4 |
| 3:0 | **Palette Index**<br>A 4-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx)<br>Format: U4 |

### 6.2.12.2 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**<br>A 4-bit color index which is used to lookup a 24-bit RGB value in the texture palette.<br>Format: U4 |
| 3:0 | **Alpha**<br>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.<br>Format: U4 |

### 6.2.12.3 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 |
|---|---|
| 7:4 | **Alpha**<br>Alpha value which will be divided by 255 to yield a [0.0,1.0] Alpha value.<br>Format: U8 |
| 3:0 | **Palette Index**<br>An 8-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx)<br>Format: U8 |

### 6.2.12.4 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**<br>An 8-bit color index which is used to lookup a 24-bit RGB value in the texture palette.<br>Format: U8 |
| 7:0 | **Alpha**<br>Alpha value which will be divided by 255 to yield a [0.0,1.0] alpha value.<br>Format: U8 |

### 6.2.12.5 P8_UNORM

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

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

### 6.2.12.6 P2_UNORM

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

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

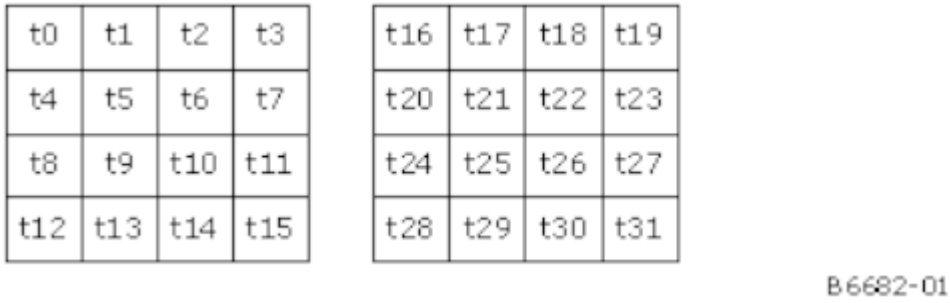
## 6.3 Compressed Surface Formats

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

### 6.3.1 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**




### 6.3.1.1 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. |

### 6.3.1.2 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 8$^{th}$ 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.

#### 6.3.1.2.1 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 |

| Bit | Description |
|---|---|
| 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 |

#### 6.3.1.2.2 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.

### 6.3.1.3 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).

#### 6.3.1.3.1 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 |
| 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 |

#### 6.3.1.3.2 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] |

| Expanded Color Bit | Expanded Channel Bit | Encoded Block Source Bit |
|---|---|---|
| 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] |
| 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 |

### 6.3.1.4 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).

#### 6.3.1.4.1 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 |
| 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 |

| Bit | Description |
|---|---|
| 63:62 | Texel 31 Select |
| ... | ... |
| 33:32 | Texel 16 Select |
| 31:30 | Texel 15 Select |
| ... | ... |
| 1:0 | Texel 0 Select |

#### 6.3.1.4.2 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] |
| 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] |

| Expanded Color Bit | Expanded Channel Bit | Encoded Block Source Bit |
|---|---|---|
| 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] |

| Expanded Color Bit | Expanded Channel Bit | Encoded Block Source Bit |
|---|---|---|
| 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] |
| 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.

### 6.3.1.5 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.

#### 6.3.1.5.1 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 |

| Bit | Description |
|---|---|
| 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 |

#### 6.3.1.5.2 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] |
| 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 |

## 6.3.2 DXT 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)

**Notes:**

- 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

### 6.3.2.2 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.

### 6.3.2.3 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. 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) |

### 6.3.3 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 |

| Bit | Description |
|---|---|
| 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
}
```

### 6.3.4 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 |

| Bit | Description |
|---|---|
| 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 |
| 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.

## 6.3.5 BC6H

These formats (BC6H_UF16 and BC6H_SF16) compress 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.

### 6.3.5.1 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] |
| 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] |

| Bit | Description |
|---|---|
| 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] |
| 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] |

| Bit | Description |
|---|---|
| 60 | bz[1] |
| 64:61 | by[3:0] |
| 69:65 | ry[4:0] |
| 70 | bz[2] |
| 75:71 | rz[4:0] |
| 76 | bz[3] |
| 81:77 | partition |
| 127:82 | indices |

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

Green: 11-bit endpoint, 5-bit deltas

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

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

Blue: 11-bit endpoint, 5-bit deltas

| Bit | Description |
|---|---|
| 4:0 | mode = 01010 |
| 14:5 | rw[9:0] |
| 24:15 | gw[9:0] |
| 34:25 | bw[9:0] |
| 38:35 | rx[3:0] |
| 39 | rw[10] |
| 40 | by[4] |
| 44:41 | gy[3:0] |
| 48:45 | gx[3:0] |
| 49 | gw[10] |
| 50 | bz[0] |

| Bit | Description |
|---|---|
| 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 |
| 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] |

| Bit | Description |
|---|---|
| 40:35 | rx[5:0] |
| 44:41 | gy[3:0] |
| 49:45 | gx[4:0] |
| 50 | bz[0] |
| 54:51 | gz[3:0] |
| 59:55 | bx[4:0] |
| 60 | gz[1] |
| 64:61 | by[3:0] |
| 70:65 | ry[5:0] |
| 76:71 | rz[5:0] |
| 81:77 | partition |
| 127:82 | indices |

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

Green: 8-bit endpoint, 6-bit deltas

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

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

Blue: 8-bit endpoint, 6-bit deltas

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

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

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

| Bit | Description |
|---|---|
| 4:0 | mode = 11110 |
| 10:5 | rw[5:0] |
| 11 | gz[4] |
| 12 | bz[0] |
| 13 | bz[1] |
| 14 | by[4] |
| 20:15 | gw[5:0] |
| 21 | gy[5] |
| 22 | by[5] |
| 23 | bz[2] |
| 24 | gy[4] |
| 30:25 | bw[5:0] |
| 31 | gz[5] |
| 32 | bz[3] |
| 33 | bz[5] |
| 34 | bz[4] |
| 40:35 | rx[5:0] |
| 44:41 | gy[3:0] |
| 50:45 | gx[5:0] |
| 54:51 | gz[3:0] |
| 60:55 | bx[5:0] |
| 64:61 | by[3:0] |
| 70:65 | ry[5:0] |
| 76:71 | rz[5:0] |
| 81:77 | partition |
| 127:82 | indices |

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

| Bit | Description |
|---|---|
| 4:0 | mode = 00011 |

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

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

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

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

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

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

| Bit | Description |
|---|---|
| 4:0 | mode = 01111 |
| 14:5 | rw[9:0] |
| 24:15 | gw[9:0] |
| 34:25 | bw[9:0] |
| 38:35 | rx[3:0] |
| 39 | rw[15] |
| 40 | rw[14] |
| 41 | rw[13] |
| 42 | rw[12] |
| 43 | rw[11] |

| Bit | Description |
|---|---|
| 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 |

| Bit | Description |
|---|---|
| 112:110 | texel[2][2] index |
| 115:113 | texel[2][3] index |
| 118:116 | texel[3][0] index |
| 121:119 | texel[3][1] index |
| 124:122 | texel[3][2] index |
| 127:125 | texel[3][3] index |

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

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

**ONE mode *indices* field**

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

### 6.3.5.2 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).

#### 6.3.5.3 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 |

Note that 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.

### 6.3.5.4 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.

#### 6.3.5.4.1 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.

#### 6.3.5.4.2 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.

## 6.3.6 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.

### 6.3.6.1 Field Definition

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

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

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

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

| Bit | Description |
|---|---|
| 1:0 | mode = 10 |
| 7:2 | partition |
| 13:8 | R0 |
| 19:14 | R1 |
| 25:20 | R2 |
| 31:26 | R3 |
| 37:32 | G0 |
| 43:38 | G1 |
| 49:44 | G2 |
| 55:50 | G3 |
| 61:56 | B0 |
| 67:62 | B1 |

| Bit | Description |
|---|---|
| 73:68 | B2 |
| 79:74 | B3 |
| 80 | P0 |
| 81 | P1 |
| 127:82 | indices |

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

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

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

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

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

| Bit | Description |
|---|---|
| 4:0 | mode = 10000 |
| 6:5 | rotation |
| 7 | index selector |
| 12:8 | R0 |
| 17:13 | R1 |
| 22:18 | G0 |
| 27:23 | G1 |
| 32:28 | B0 |
| 37:33 | B1 |
| 43:38 | A0 |
| 49:44 | A1 |
| 80:50 | 2-bit indices |
| 127:81 | 3-bit indices |

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

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

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

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

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

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

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

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

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

### 6.3.6.2 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.

### 6.3.6.3 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **weight** | 0 | 4 | 9 | 13 | 17 | 21 | 26 | 30 | 34 | 38 | 43 | 47 | 51 | 55 |

Note that 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.

### 6.3.6.4 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.

#### 6.3.6.4.1 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.

#### 6.3.6.4.2 TWO Mode

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

| | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | [0] | 0 | 1 | 1 | [0] | 0 | 0 | 1 | [0] | 1 | 1 | 1 | [0] | 0 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| | 0 | 0 | 1 | [1] | 0 | 0 | 0 | [1] | 0 | 1 | 1 | [1] | 0 | 1 | 1 | [1] |
| 04 | [0] | 0 | 0 | 0 | [0] | 0 | 1 | 1 | [0] | 0 | 0 | 1 | [0] | 0 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| | 0 | 0 | 1 | [1] | 1 | 1 | 1 | [1] | 1 | 1 | 1 | [1] | 0 | 1 | 1 | [1] |
| 08 | [0] | 0 | 0 | 0 | [0] | 0 | 1 | 1 | [0] | 0 | 0 | 0 | [0] | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 1 | [1] | 1 | 1 | 1 | [1] | 1 | 1 | 1 | [1] | 0 | 1 | 1 | [1] |
| 0C | [0] | 0 | 0 | 1 | [0] | 0 | 0 | 0 | [0] | 0 | 0 | 0 | [0] | 0 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | [1] | 1 | 1 | 1 | [1] | 1 | 1 | 1 | [1] | 1 | 1 | 1 | [1] |
| 10 | [0] | 0 | 0 | 0 | [0] | 1 | [1] | 1 | [0] | 0 | 0 | 0 | [0] | 1 | [1] | 1 |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | [1] | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 1 | 1 | 1 | [1] | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 14 | [0] | 0 | [1] | 1 | [0] | 0 | 0 | 0 | [0] | 0 | 0 | 0 | [0] | 1 | 1 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 0 | 0 | 0 | 0 | [1] | 1 | 0 | 0 | [1] | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | [1] |
| 18 | [0] | 0 | [1] | 1 | [0] | 0 | 0 | 0 | [0] | 1 | [1] | 0 | [0] | 0 | [1] | 1 |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 0 | 0 | 0 | 1 | [1] | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1C | [0] | 0 | 0 | 1 | [0] | 0 | 0 | 0 | [0] | 1 | [1] | 1 | [0] | 0 | [1] | 1 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| | [1] | 1 | 1 | 0 | [1] | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 20 | [0] | 1 | 0 | 1 | [0] | 0 | 0 | 0 | [0] | 1 | 0 | 1 | [0] | 0 | 1 | 1 |
| | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | [1] | 0 | 0 | 0 | 1 | 1 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | [1] | 1 | 0 | 0 |
| | 0 | 1 | 0 | [1] | 1 | 1 | 1 | [1] | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 24 | [0] | 0 | [1] | 1 | [0] | 1 | 0 | 1 | [0] | 1 | 1 | 0 | [0] | 1 | 0 | 1 |
| | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

| | 0 | | | | 1 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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] |

#### 6.3.6.4.3 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 | | | | 2 | | | | 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

## 6.4 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.

### 6.4.1 Packed Memory Organization

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

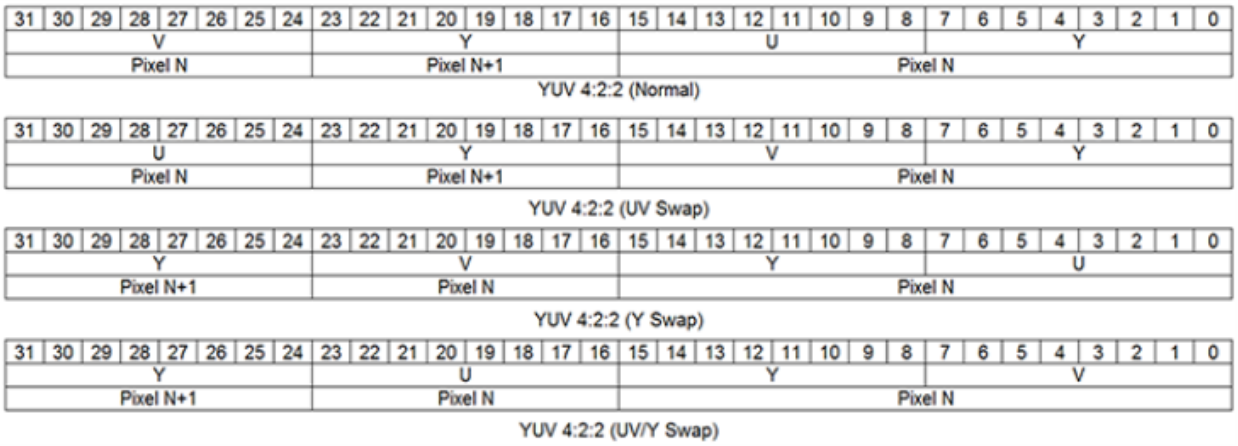
The following four YUV 4:2:2 surface formats are supported, listed with alternate names:

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

The channels are mapped as follows:

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

**Memory layout of packed YUV 4:2:2 formats**




### 6.4.2 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.

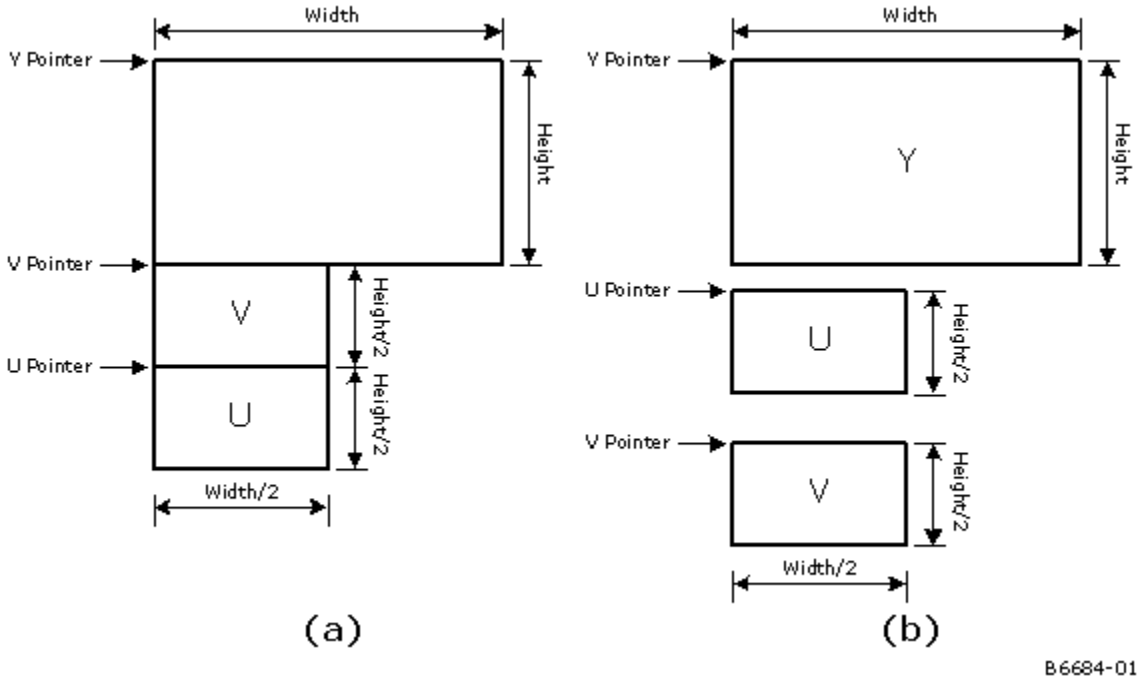
***Note:*** There is no direct support for use of planar video surfaces as textures. The sampling engine can be used to operate on each of the 8bpp buffers separately (via a single-channel 8-bit format such as I8_UNORM). The U and V buffers can be written concurrently by using multiple render targets 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.

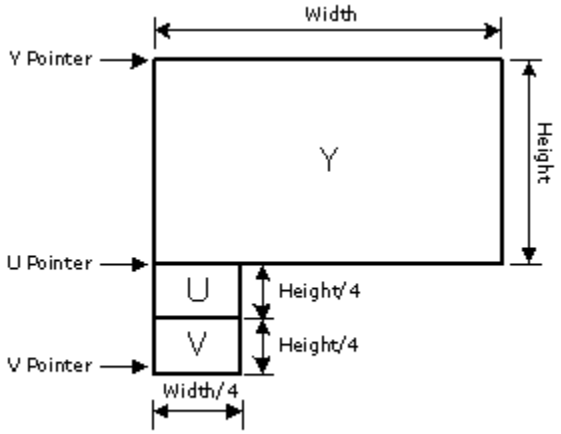
2. An alternative memory structure that the addresses of the three planes are independent but satisfy certain alignment restrictions.

### 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.

### YUV 4:1:0 Format Memory Organization




B6685-01

## 6.5 Additional Video Formats

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| DXGI_FORMAT_AYUV | DXGI_FORMAT_R8G8B8A8_UNORM (V->R8, U->G8, Y->B8, A->A8) | Packed | 1 | R8G8B8A8_UNORM | NA | NA | Sampler, PB |
| DXGI_FORMAT_AYUV | DXGI_FORMAT_R8G8B8A8_UINT (V->R8, U->G8, Y->B8, A->A8) | Packed | 1 | R8G8B8A8_UINT | NA | NA | Sampler, HDC, PB |
| DXGI_FORMAT_YUY2 | DXGI_FORMAT_R8G8B8A8_UNORM (Y0->R8, U0->G8, Y1->B8, V0->A8) | Packed | 1 | R8G8B8A8_UNORM | NA | NA | Sampler, |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| DXGI_FORMAT_YUY2 | DXGI_FORMAT_R8G8B8A8_UINT (Y0->R8, U0->G8, Y1->B8, V0->A8) | Packed | 1 | R8G8B8A8_UINT | NA | NA | Sampler, HDC |
| DXGI_FORMAT_YUY2 | DXGI_FORMAT_R8G8_B8G8_UNORM | Packed | 1 | R8G8B8A8_UNORM<br>In this case the width of the view will appear to | NA | NA | Sampler |

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| | | | | be twice the R8G8B8A8 view, with hardware reconstruction of RGBA done automatically on read (and before filtering). | | | |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R8G8B8A8_UINT | NA | NA | Sampler, HDC |
| DXGI_FORMAT_NV12 | Y = DXGI_FORMAT_R8_UNORM<br>U/V = DXGI_FORMAT_R8G8_UNORM (U->R8, V->G8) | Planar | 2 | R8_UNORM | R8G8_UNORM<br>chomra pixel dimensions 1/2 in both x and y from the Luma view | NA | Sampler, PB |
| NA | NA | Planar | 3 | R16_UNIT | R16_UNIT | R16_UNIT | Sampler |
| NA | NA | Planar | 3 | R16_UNIT | R16_UNIT | R16_UNIT | Sampler |
| NA | NA | Planar | 3 | R16_UNIT | R16_UNIT | R16_UNIT | Sampler |
| NA | NA | Planar | 3 | R8_UNIT | R8_UNIT | R8_UNIT | Sampler |
| NA | NA | Planar | 2 | R16_UNIT | R16G16_UINT<br>chomra pixel dimensions halved in both x and y from the Luma view | NA | Sampler, HDC |
| NA | NA | Planar | 2 | R16_UNIT | R16G16_UINT<br>chomra pixel dimensions halved in | NA | Sampler, HDC |

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| | | | | | both x and y from the Luma view | | |
| NA | NA | Planar | 2 | R16_UNIT | R16G16_UINT<br>chomra pixel dimensions halved in both x and y from the Luma view | NA | Sampler, HDC |
| DXGI_FORMAT_NV12 | Y = DXGI_FORMAT_R8_UINT<br>U/V = DXGI_FORMAT_R8G8_UINT (U->R8, V->G8) | Planar | 2 | R8_UNIT | R8G8_UINT<br>chomra pixel dimensions halved in both x and y from the Luma view | NA | Sampler, HDC, PB |
| DXGI_FORMAT_NV11 | Y = DXGI_FORMAT_R8_UNORM<br>U/V = DXGI_FORMAT_R8G8_UNORM (U->R8, V->G8) | Planar | 2 | R8_UNORM | R8G8_UNORM<br>chomra pixel dimensions 1/4 in both x and y from the Luma view | NA | Sampler, PB |
| DXGI_FORMAT_NV11 | Y = DXGI_FORMAT_R8_UINT<br>U/V = DXGI_FORMAT_R8G8_UINT (U->R8, V->G8) | Planar | 2 | R8_UNIT | R8G8_UINT<br>chomra pixel dimensions 1/4 in both x and y from the Luma view | NA | Sampler, HDC, PB |
| DXGI_FORMAT_P016 | Y = DXGI_FORMAT_R16_UNORM<br>U/V = DXGI_FORMAT_R16G | Planar | 2 | R16_UNORM | R16G16_UNORM<br>chomra pixel dimensions | NA | Sampler, HDC, PB |

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| | 16_UNORM<br>(U->R16, V->G16) | | | | 1/2 in both x and y from the Luma view | | |
| DXGI_FORMAT_P016 | Y = DXGI_FORMAT_R16_UINT<br>U/V = DXGI_FORMAT_R16G16_UINT<br>(U->R16, V->G16) | Planar | 2 | R16_UNIT | R16G16_UINT<br>chomra pixel dimensions 1/2 in both x and y from the Luma view | NA | Sampler, PB |
| DXGI_FORMAT_P010 | Y = DXGI_FORMAT_R16_UNORM<br>U/V = DXGI_FORMAT_R16G16_UNORM<br>(U->R16, V->G16) | Planar | 2 | R16_UNORM | R16G16_UNORM<br>chroma pixel dimensions 1/2 in both x and y from the Luma view | NA | Sampler, HDC, PB |
| DXGI_FORMAT_P010 | Y = DXGI_FORMAT_R16_UINT<br>U/V = DXGI_FORMAT_R16G16_UINT<br>(U->R16, V->G16) | Planar | 2 | R16_UNIT | R16G16_UINT<br>chomra pixel dimensions 1/2 in both x and y from the Luma view | NA | Sampler, HDC, PB |
| DXGI_FORMAT_Y216 | DXGI_FORMAT_R16G16B16A16_UNORM<br>(Y0->R16, U->G16, Y1->B16, V->A16). | Packed | 1 | R16G16B16A16_UNORM | NA | NA | Sampler, |
| DXGI_FORMAT_Y216 | DXGI_FORMAT_R16G16B16A16_UINT<br>(Y0->R16, U->G16, Y1->B16, V->A16). | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| DXGI_FORMAT_Y210 | DXGI_FORMAT_R16G16B16A16_UNORM<br>(Y0->R16, U->G16, Y1->B16, V->A16). | Packed | 1 | R16G16B16A16_UNORM | NA | NA | Sampler, |

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| DXGI_FORMAT_Y210 | DXGI_FORMAT_R16G16B16A16_UINT (Y0->R16, U->G16, Y1->B16, V->A16). | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, HDC |
| DXGI_FORMAT_Y416 | DXGI_FORMAT_R16G16B16A16_UNORM (U->R16, Y->G16, V->B16, A->A16) | Packed | 1 | R16G16B16A16_UNORM | NA | NA | Sampler, HDC |
| DXGI_FORMAT_Y416 | DXGI_FORMAT_R16G16B16A16_UINT (U->R16, Y->G16, V->B16, A->A16) | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, |
| DXGI_FORMAT_Y410 | DXGI_FORMAT_R16G16B16A16_UNORM (U->R16, Y->G16, V->B16, A->A16) | Packed | 1 | R16G16B16A16_UNORM | NA | NA | Sampler, HDC |
| DXGI_FORMAT_Y410 | DXGI_FORMAT_R16G16B16A16_UINT (U->R16, Y->G16, V->B16, A->A16) | Packed | 1 | R16G16B16A16_UINT | NA | NA | Sampler, |
| NA | NA | Packed | 1 | R16G16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16G16_UINT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R8G8_UINT | NA | NA | Sampler, HDC |
| NA | NA | Planar | 2 | R16_UNIT | R16_UNIT | NA | Sampler, HDC |
| NA | NA | Planar | 2 | R16_UNIT | R16_UNIT | NA | Sampler, HDC |
| NA | NA | Planar | 2 | R16_UNIT | R16_UNIT | NA | Sampler, HDC |
| NA | NA | Planar | 2 | R8_UNIT | R8_UNIT | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R8_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Pack | 1 | R16_UNIT | NA | NA | Sampler, |

| Dx YUV format name | DxRGB format name | Packed / Planar | # Surface States | Surface Format #1 | Surface Format #2 | Surface Format #3 | Support By |
|---|---|---|---|---|---|---|---|
| | | ed | | | | | HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R8_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R16_UNIT | NA | NA | Sampler, HDC |
| NA | NA | Packed | 1 | R8_UNIT | NA | NA | Sampler, HDC |

## 6.6 Raw Format

A new surface format is added that is only supported with the untyped surface read/write and atomic operation data port messages. This new format is called simply RAW. It means that the surface has no inherent format. Surfaces of type RAW are addressed with byte-based offsets that must be DWord aligned (multiple of 4). Data is returned in DWord quantities. The RAW surface format can be applied only to surface types of BUFFER and STRBUF.

## 6.7 Surface Memory Organizations

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

## 6.8 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 graphics translation tables must reside in (unsnooped) system memory.

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 GTTADR. 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.

## 6.9 Hardware Status Page

The hardware status page is a naturally-aligned 4KB page residing in snooped system memory. This page exists primarily to allow the device to report status via PCI master writes – thereby allowing the driver to read/poll WB memory instead of UC reads of device registers or UC memory.

The address of this page is programmed via the HWS_PGA MI register. The definition of that register (in *Memory Interface Registers*) includes a description of the layout of the Hardware Status Page.

## 6.10 Instruction Ring Buffers

Instruction ring buffers are the memory areas used to pass instructions to the device. Refer to the Programming Interface chapter for a description of how these buffers are used to transport instructions.

The RINGBUF register sets (defined in Memory Interface Registers) are used to specify the ring buffer memory areas. The ring buffer must start on a 4KB boundary and be allocated in linear memory. The length of any one ring buffer is limited to 2MB.

Note that “indirect” 3D primitive instructions (those that access vertex buffers) must reside in the same memory space as the vertex buffers.

## 6.11 Instruction Batch Buffers

Instruction batch buffers are contiguous streams of instructions referenced via an MI_BATCH_BUFFER_START and related instructions (see Memory Interface Instructions, Programming Interface). They are used to transport instructions external to ring buffers.

Note that batch buffers should not be mapped to snooped SM (PCI) addresses. The device will treat these as MainMemory (MM) address, and therefore not snoop the CPU cache.

The batch buffer must be QWord aligned and a multiple of QWords in length. The ending address is the address of the last valid QWord in the buffer. The length of any single batch buffer is “virtually unlimited” (i.e., could theoretically be 4GB in length).

## 6.12 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.

## 6.13 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.

## 6.14 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.

## 6.15 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, 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.

## 6.16 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 7.17, 3D Depth Buffer Surfaces, 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 |

## 6.17 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 |
|---|---|---|
| S8_UINT/R8_ UNIT | 8 | 8-bit stencil value in a byte |

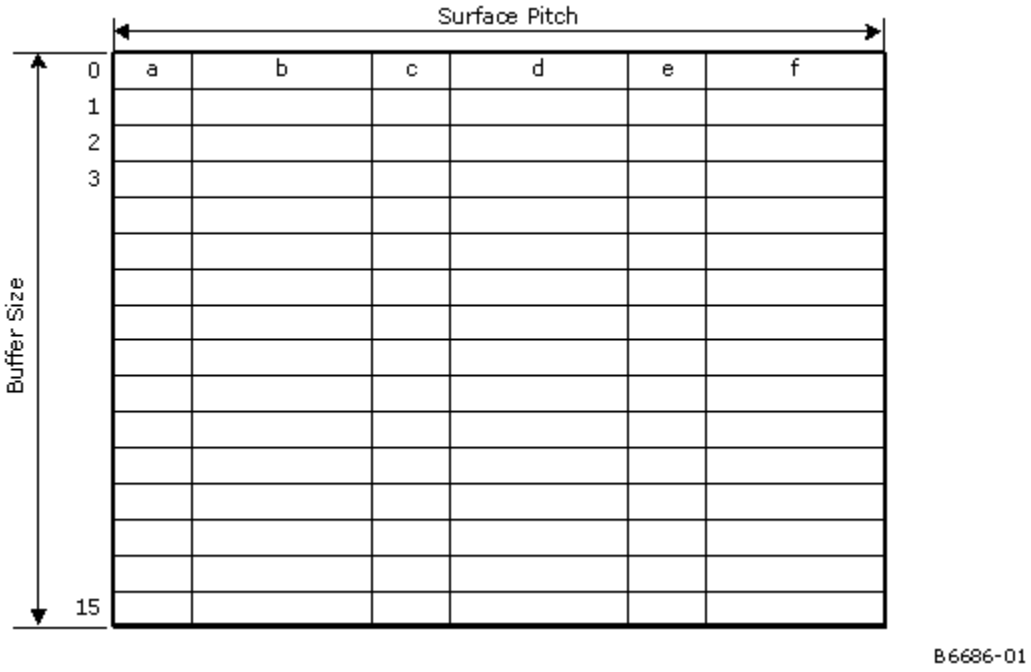
## 6.18 Surface Layout

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.

### 6.18.1 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.



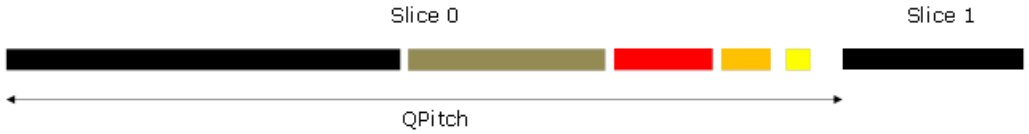

## 6.18.2 Structured Buffers

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**.

## 6.18.3 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.




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

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

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

Next, the aligned width parameter for each LOD “L” is computed. The “i” parameter is the horizontal alignment parameter set by a state field or defined as a constant, depending on the surface. The

alignment parameter may change at one point in the mip chain based on Mip Tail Start LOD. The equation uses the I value that applies to the LOD being computed.

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

...

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

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

## 6.18.4 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.)

### 6.18.4.1 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:

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

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

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 |

### 6.18.4.2 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 in rectangular memory space. 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:

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

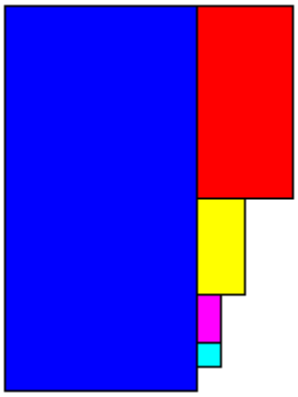
$$h_L = j * ceil\left(\frac{H_L}{j}\right)$$

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:

$LOD_0 = (0,0)$

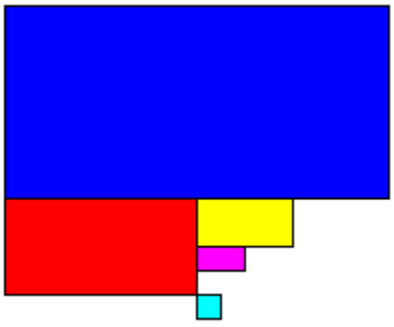
$LOD_1 = (0,h_0)$

$LOD_2 = (w_1,h_0)$

$LOD_3 = (w_1,h_0 + h_2)$

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

...



### 6.18.4.3 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.

### 6.18.4.4 Alignment Unit Size

This section documents the alignment parameters *i* and *j* that are used depending on the surface.

| surface defined by | surface format | alignment unit width "*i*" | alignment unit height "*j*" |
|---|---|---|---|
| 3DSTATE_DEPTH_BUFFER | D16_UNORM | 8 | 4 |
| | not D16_UNORM | 4 | 4 |
| 3DSTATE_STENCIL_BUFFER | N/A | 8 | 8 |
| SURFACE_STATE | BC*, ETC*, EAC* | 4 | 4 |
| | FXT1 | 8 | 4 |
| | all others | set by **Surface Horizontal Alignment** | set by **Surface Vertical Alignment** |

### 6.18.4.5 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).

### 6.18.4.6 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.

### 6.18.4.7 Surface Arrays

#### 6.18.4.7.1 For all surface other than separate stencil buffer

Both 1D and 2D surfaces can be specified as an array. The only difference in the surface state is the presence of a depth value greater than one, indicating multiple array “slices”.

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 MIPLAYOUT_BELOW 2D surface. *MIPLAYOUT_BELOW is the only format supported by 1D non-arrays and both 2D and 1D arrays, the programming of the MIP Map Layout Mode state variable is ignored when using a TextureArray.*

The following equation is used for surface formats other than compressed textures:

$$QPitch = (h_0 + h_1 + 11j)$$

The input variables in this equation are defined in sections above.

The equation for compressed textures (BC* and FXT1 surface formats) follows:

$$QPitch = \frac{(h_0 + h_1 + 11j)}{4}$$

#### 6.18.4.7.2 For all surfaces

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.

The **Surface Array Spacing** field in SURFACE_STATE has two possible values, which affect the QPitch formula.

If **Surface Array Spacing** is set to ARYSPC_FULL (note that the *depth buffer* and stencil buffer have an implied value of ARYSPC_FULL):

$$QPitch = (h_0 + 1_h + 12j)$$

$$QPitch = \frac{(h_0 + h_1 + 12j)}{4}$$

$$QPitch = \left( h_0 + h_1 + \frac{12j}{2} \right)$$

Note that $h_0$ and $h_1$ have been halved as described earlier.

If **Surface Array Spacing** is set to ARYSPC_LOD0:

QPitch = $h_0$

$$QPitch = \frac{h_0}{4}$$

#### 6.18.4.8 Multisampled Surfaces

Starting with, 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.

##### 6.18.4.8.1 Interleaved Multisampled Surfaces

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

4x MSAA: 2x width and 2x height of non-multisampled surface

8x MSAA: 4x width and 2x height of non-multisampled surface

### 6.18.5 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.

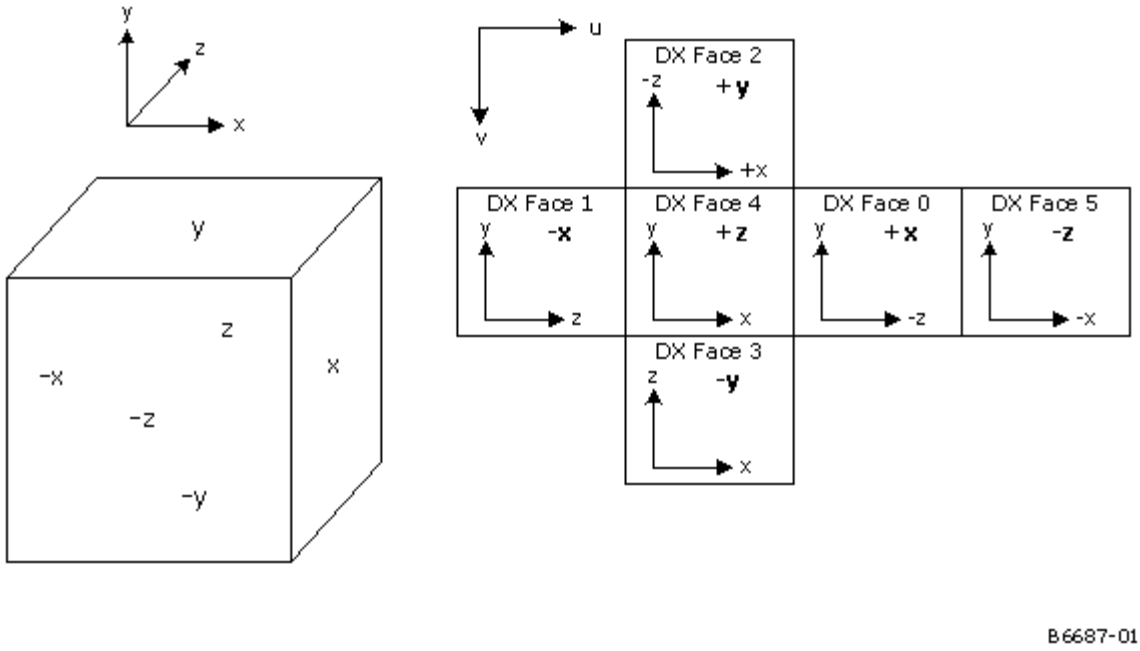
Each texture map level is represented as a group of six, square *cube face* texture surfaces. The faces are identified by their relationship to the 3D texture coordinate system. The subsections below describe the cube maps as described at the API as well as the memory layout dictated by the hardware.

### 6.18.5.1 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**




### 6.18.5.2 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 |

### 6.18.5.3 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**

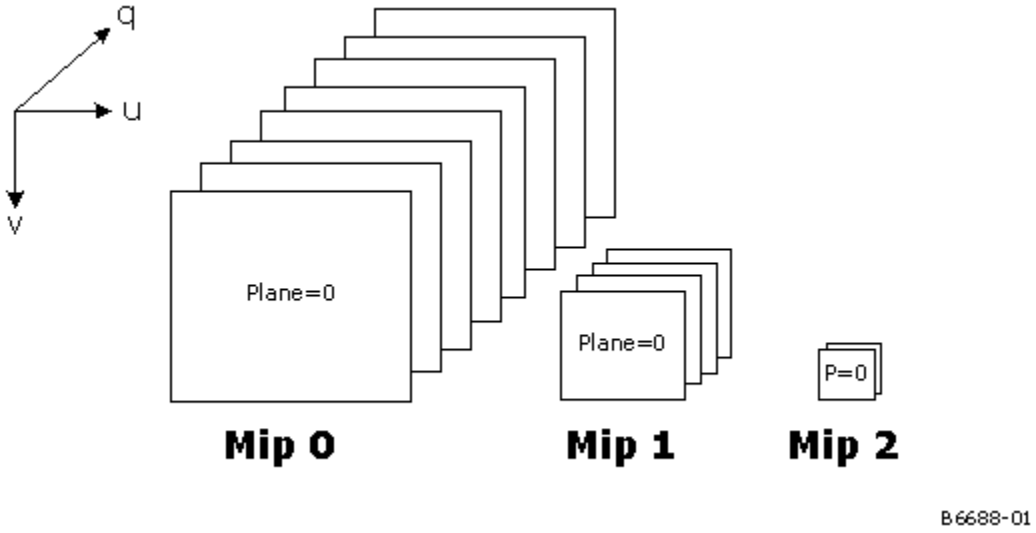
### 6.18.5.4 Cube Arrays

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.

## 6.18.6 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**




Note that 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:

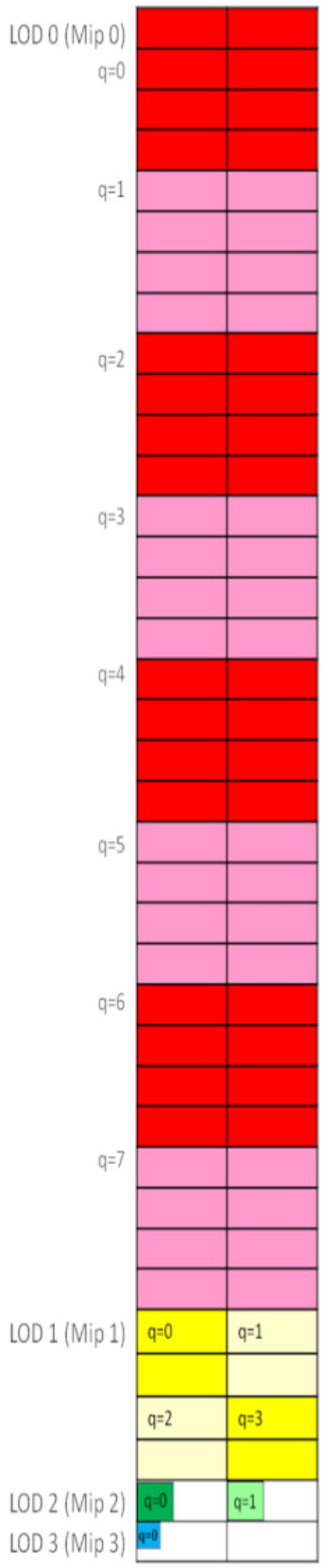
$W_L = ((\text{width} >> L) > 0 ? \text{width} >> L : 1)$

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

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

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

Cache-line aligned width and height are as follows, with i and j being a function of the map format as shown in the table entitled 7.19.4.4, 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

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

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

Note that 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.

#### 6.18.6.1 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.

## 6.19 Surface Padding Requirements

### 6.19.1 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 Section *Alignment Unit Size* for the i and j parameters for the surface format in use. The surface must then be extended to the next multiple of the alignment unit size in each dimension, and all texels contained in this extended surface must have valid GTT entries.

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

For buffers, which have no inherent “height,” padding requirements are different. A buffer must be padded to the next multiple of 256 array elements, with an additional 16 bytes added beyond that to account for the L1 cache line.

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 compressed textures (BC* and FXT1 surface formats), padding at the bottom of the surface is to an even compressed row, which is equal to a multiple of 8 uncompressed texel rows. Thus, for padding purposes, these surfaces behave as if j = 8 only for surface padding purposes. The value of 4 for j still applies for mip level alignment and QPitch calculation.

For YUV, 96 bpt, and 48 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.

### 6.19.2 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.

If the surface contains an odd number of rows of data, a final row below the surface must be allocated. If the surface will be accessed in field mode (**Vertical Stride** = 1), enough additional rows below the surface must be allocated to make the extended surface height (including the padding) a multiple of 4.

## 6.20 BSD Logical Context Data (MFX)

### 6.20.1 Register/State Context

| | | Valid Only When PPGTT Enabled | | | | | |
|---|---|---|---|---|---|---|---|
| **DW Range** | **DW Count** | **State Field** | **Restore Inhibited** | **PPGTTEnableds** | **PPGTT Disabled** | **Power Context** | **Set Before Submitting Context?** |
| 00h | 1 | Context Control | R | S/R | X | S/R | Yes |
| 01h | 1 | Ring Head Pointer Register | R | S/R | X | S/R | Yes |
| 02h | 1 | Ring Tail Pointer Register | R | R | X | S/R | Yes |
| 03h | 1 | Batch Buffer Current Head Register | NR | S/R | X | S/R | No |
| 04h | 1 | Batch Buffer State Register | NR | S/R | X | S/R | No |
| 05h | 1 | PPGTT Directory Cache Valid | R | R | X | S/R | Yes |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | Valid Only When PPGTT Enabled | | | | | |
| | | Register<br>(Software always populates via host) | | | | | |
| 06h | 1 | Reserved | X | X | X | S/R | X |
| 07h | 1 | PD Base Virtual Address Register | R | R | X | S/R | Yes |
| 08h | 1 | MFX_STATE_POINTER 0 | NR | S/R | X | S/R | Yes |
| 09h | 1 | MFX_STATE_POINTER 1 | NR | S/R | X | S/R | Yes |
| 0Ah | 1 | MFX_STATE_POINTER 2 | NR | S/R | X | S/R | Yes |
| 0Bh | 1 | MFX_STATE_POINTER 3 | NR | S/R | X | S/R | Yes |
| 0Ch | 1 | VCS_CNTR— Media Watchdog Counter Control | NR | S/R | X | S/R | No |
| 0Dh | 1 | VCS_THRSH— Media Watchdog Counter Threshold | NR | S/R | X | S/R | No |
| 0Eh | 1 | Current Context ID Register | NR | S/R | X | S/R | No |
| 0Fh | 1 | Reserved | X | X | X | S/R | X |

## 6.20.2 The Per-Process Hardware Status Page

The following table defines the layout of the Per-process Hardware Status Page:

| DWord Offset | Description |
|---|---|
| (3FFh – 020h) | These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions. |
| 1F:5 | **Reserved.** |
| 4 | **Ring Head Pointer Storage:** The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an "automatic report" (see RINGBUF registers). |
| 3:0 | **Reserved.** |

This page is designed to be read by SW in order to glean additional details about a context beyond what it can get from the context status.

Accesses to this page will automatically be treated as cacheable and snooped. It is therefore illegal to locate this page in any region where snooping is illegal (such as in stolen memory).

## 6.21 Copy Engine Logical Context Data

### 6.21.1 Register/State Context

| | | Valid Only When PPGTT Enabled | | | | | |
|---|---|---|---|---|---|---|---|
| **DW Range** | **DW Count** | **State Field** | **Render Restore Inhibited** | **PPGTTEnabled** | **PPGTT Disabled** | **Power Context** | **Set Before Submitting Context?** |
| 00h | 1 | Reserved | NR | X | X | X | X |
| 01h | 1 | Ring Head Pointer Register | R | S/R | X | S/R | Yes |
| 02h | 1 | Ring Tail Pointer Register | R | R | X | S/R | Yes |
| 03h | 1 | Reserved | NR | X | X | X | X |
| 04h | 1 | Reserved | NR | X | X | X | X |
| 05h | 1 | PPGTT Directory Cache Valid Register<br>(Software always populates via host) | R | R | X | X | Yes |
| 06h | 1 | BCS_SWCTRL Register | NR | S/R | X | S/R | Yes |
| 07h | 1 | PD Base Virtual Address Register | R | R | X | X | Yes |
| 08h | 1 | Reserved | NR | X | X | X | X |
| 09h | 1 | Reserved | NR | X | X | X | X |
| 0Ah | 1 | Reserved | NR | X | X | X | X |
| 0Bh | 1 | Reserved | NR | X | X | X | X |
| 0Ch | 1 | Reserved | NR | X | X | X | X |
| 0Dh | 1 | Reserved | NR | X | X | X | X |
| 0Eh | 1 | Reserved | NR | X | X | X | X |
| 0Fh | 1 | Reserved | NR | X | X | X | X |

### 6.21.2 The Per-Process Hardware Status Page

The following table defines the layout of the Per-process Hardware Status Page:

| DWord Offset | Description |
|---|---|
| (3FFh – 020h) | These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions. |
| 1F:5 | **Reserved.** |
| 4 | **Ring Head Pointer Storage:** The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an "automatic report" (see RINGBUF registers). |
| 3:0 | **Reserved.** |

This page is designed to be read by SW in order to glean additional details about a context beyond what it can get from the context status.

Accesses to this page will automatically be treated as cacheable and snooped. It is therefore illegal to locate this page in any region where snooping is illegal (such as in stolen memory).

## 6.22 Render Logical Context Data

Logical Contexts are memory images used to store copies of the device's rendering and ring context.

Logical Contexts are aligned to 256-byte boundaries.

Logical contexts are referenced by their memory address. The format and contents of rendering contexts are considered ***device-dependent*** and software must not access the memory contents directly. The definition of the logical rendering and power context memory formats is included here primarily for internal documentation purposes.

### 6.22.1 Overall Context Layout

#### 6.22.1.1 Register/State Context

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| NOOP | | CS | 1 |
| Load_Register_Immediate header | 0x1100_105D | CS | 1 |
| RING_BUFFER_START | 0x2038 | CS | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| RING_BUFFER_CONTROL | 0x203C | CS | 2 |
| RVSYNC | 0x2040 | CS | 2 |
| RBSYNC | 0x2044 | CS | 2 |
| RC_PWRCTX_MAXCNT | 0x2054 | CS | 2 |
| CTX_WA_PTR | 0x2058 | CS | 2 |
| NOPID | 0x2094 | CS | 2 |
| HWSTAM | 0x2098 | CS | 2 |
| FF_THREAD_MODE | 0x20A0 | CS | 2 |
| IMR | 0x20A8 | CS | 2 |
| EIR | 0x20B0 | CS | 2 |
| EMR | 0x20B4 | CS | 2 |
| CMD_CCTL_0 | 0x20C4 | CS | 2 |
| GAFS_Mode | 0x212C | CS | 2 |
| UHPTR | 0x2134 | CS | 2 |
| BB_PREEMPT_ADDR | 0x2148 | CS | 2 |
| RING_BUFFER_HEAD_PREEMPT_REG | 0x214C | CS | 2 |
| CXT_SIZE | 0x21A8 | CS | 2 |
| CXT_OFFSET | 0x21AC | CS | 2 |
| CXT_PIPESTATEBASE | 0x21B0 | CS | 2 |
| PREEMPT_DLY | 0x2214 | CS | 2 |
| GFX_MODE | 0x229C | CS | 2 |
| MTCH_CID_RST | 0x222C | CS | 2 |
| SYNC_FLIP_STATUS | 0x22D0 | CS | 2 |

| POWER CONTEXT | | | |
|---|---|---|---|
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| SYNC_FLIP_STATUS_1 | 0x22D4 | CS | 2 |
| NOOP | | CS | 12 |
| NOOP | | GPM | 1 |
| Load_Register_Immediate header | 0x1100_10d5 | GPM | 1 |
| GPM Data(Inc GAM) | | GPM | 214 |
| NOOP | | GPM | 1 |
| Load_Register_Immediate header | 0x1100_104b | GPM | 1 |
| MBCunit | | GPM | 76 |
| NOOP | | GPM | 1 |
| Load_Register_Immediate header | 0x1100_1013 | GPM | 1 |
| GCPunit | | GPM | 20 |
| NOOP | | GPM | 1 |
| Load_Register_Immediate header | 0x1100_103f | GPM | 1 |
| GDTunit | | GPM | 64 |
| NOOP | | GPM | 1 |
| Load_Register_Immediate header | 0x1100_1035 | GPM | 1 |
| GAMunit | | GPM | 52 |
| NOOP | | GPM | 28 |
| NOOP | | CS | 1 |
| Load_Register_Immediate header | 0x1100_1017 | CS | 1 |
| Context Control | 0x2244 | CS | 2 |
| Ring Head Pointer Register | 0x2034 | CS | 2 |
| Ring Tail Pointer Register | 0x2030 | CS | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| Batch Buffer Current Head Register | 0x2140 | CS | 2 |
| Batch Buffer State Register | 0x2110 | CS | 2 |
| PPGTT Directory Cache Valid Register | 0x2220 | CS | 2 |
| PP_DIR_BASE | 0x2228 | CS | 2 |
| Read Offset in Piipelined State Page (8 CL aligned) | 0x224C | CS | 2 |
| Committed Vertex Number | 0x21C4 | CS | 2 |
| Committed Instance ID | 0x21C8 | CS | 2 |
| Committed Primitive ID | 0x21CC | CS | 2 |
| CCID Register | 0x2180 | CS | 2 |
| NOOP | | CS | 6 |
| NOOP | | CS | 1 |
| Load_Register_Immediate header | 0x1100_105F | CS | 1 |
| EXCC | 0x2028 | CS | 2 |
| MI_MODE | 0x209C | CS | 2 |
| INSTPM | 0x20C0 | CS | 2 |
| PR_CTR_CTL | 0x2178 | CS | 2 |
| PR_CTR_THRSH | 0x217C | CS | 2 |
| IA_VERTICES_COUNT | 0x2310 | CS | 4 |
| IA_PRIMITIVES_COUNT | 0x2318 | CS | 4 |
| VS_INVOCATION_COUNT | 0x2320 | CS | 4 |
| HS_INVOCATION_COUNT | 0x2300 | CS | 4 |
| DS_INVOCATION_COUNT | 0x2308 | CS | 4 |
| GS_INVOCATION_COUNT | 0x2328 | CS | 4 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| GS_PRIMITIVES_COUNT | 0x2330 | CS | 4 |
| CL_INVOCATION_COUNT | 0x2338 | CS | 4 |
| CL_PRIMITIVES_COUNT | 0x2340 | CS | 4 |
| PS_INVOCATION_COUNT | 0x2348 | CS | 4 |
| PS_DEPTH_COUNT | 0x2350 | CS | 4 |
| VFSKPD | 0x2470 | CS | 2 |
| TIMESTAMP Register (LSB) | 0x2358 | CS | 2 |
| GPUGPU_DISPATCHDIMX | 0x2500 | CS | 2 |
| GPUGPU_DISPATCHDIMY | 0x2504 | CS | 2 |
| GPUGPU_DISPATCHDIMZ | 0x2508 | CS | 2 |
| MI_PREDICATE_SRC0 | 0x2400 | CS | 2 |
| MI_PREDICATE_SRC0 | 0x2404 | CS | 2 |
| MI_PREDICATE_SRC1 | 0x2408 | CS | 2 |
| MI_PREDICATE_SRC1 | 0x240C | CS | 2 |
| MI_PREDICATE_DATA | 0x2410 | CS | 2 |
| MI_PREDICATE_DATA | 0x2414 | CS | 2 |
| MI_PRED_RESULT | 0x2418 | CS | 2 |
| 3DPRIM_END_OFFSET | 0x2420 | CS | 2 |
| 3DPRIM_START_VERTEX | 0x2430 | CS | 2 |
| 3DPRIM_VERTEX_COUNT | 0x2434 | CS | 2 |
| 3DPRIM_INSTANCE_COUNT | 0x2438 | CS | 2 |
| 3DPRIM_START_INSTANCE | 0x243C | CS | 2 |
| 3DPRIM_BASE_VERTEX | 0x2440 | CS | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| GPGPU_THREADS_DISPATCHED | 0x2290 | CS | 4 |
| MI_TOPOLOGY_FILTER | | CS | 1 |
| MI_URB_CLEAR | | CS | 2 |
| MI_SET_APPID | | CS | 1 |
| PIPELINE_SELECT | | CS | 1 |
| STATE_BASE_ADDRESS | | CS | 10 |
| 3DSTATE_PUSH_CONSTANT_ALLOC_VS | | CS | 2 |
| 3DSTATE_PUSH_CONSTANT_ALLOC_HS | | CS | 2 |
| 3DSTATE_PUSH_CONSTANT_ALLOC_DS | | CS | 2 |
| 3DSTATE_PUSH_CONSTANT_ALLOC_GS | | CS | 2 |
| 3DSTATE_PUSH_CONSTANT_ALLOC_PS | | CS | 2 |
| NOOP | | CS | 5 |
| NOOP | | SARB | 1 |
| Load_Register_Immediate header | 0x1100_105B | SARB | 1 |
| SARB Error Status | 0xB004 | SARB | 2 |
| L3CD Error Status register 1 | 0xB008 | SARB | 2 |
| L3CD Error Status register 2 | 0xB00C | SARB | 2 |
| L3 SQC registers 1 | 0xB010 | SARB | 2 |
| L3 SQC registers 2 | 0xB014 | SARB | 2 |
| L3 SQC registers 3 | 0xB018 | SARB | 2 |
| L3 Control Register1 | 0xB01C | SARB | 2 |
| L3 Control Register2 | 0xB020 | SARB | 2 |
| L3 Control Register3 | 0xB024 | SARB | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| L3 SLM Register | 0xB028 | SARB | 2 |
| Arbiter Control Register | 0xB02C | SARB | 2 |
| L3 bank0 reg0 log error | 0xB070 | SARB | 2 |
| L3 bank0 reg1 log error | 0xB074 | SARB | 2 |
| L3 bank0 reg2 log error | 0xB078 | SARB | 2 |
| L3 bank0 reg3 log error | 0xB07C | SARB | 2 |
| L3 bank0 reg4 log error | 0xB080 | SARB | 2 |
| L3 bank0 reg5 log error | 0xB084 | SARB | 2 |
| L3 bank0 reg6 log error | 0xB088 | SARB | 2 |
| L3 bank0 reg7 log error | 0xB08C | SARB | 2 |
| L3 bank1 reg0 log error | 0xB090 | SARB | 2 |
| L3 bank1 reg1 log error | 0xB094 | SARB | 2 |
| L3 bank1 reg2 log error | 0xB098 | SARB | 2 |
| L3 bank1 reg3 log error | 0xB09C | SARB | 2 |
| L3 bank1 reg4 log error | 0xB0A0 | SARB | 2 |
| L3 bank1 reg5 log error | 0xB0A4 | SARB | 2 |
| L3 bank1 reg6 log error | 0xB0A8 | SARB | 2 |
| L3 bank1 reg7 log error | 0xB0AC | SARB | 2 |
| L3 bank2 reg0 log error | 0xB0B0 | SARB | 2 |
| L3 bank2 reg1 log error | 0xB0B4 | SARB | 2 |
| L3 bank2 reg2 log error | 0xB0B8 | SARB | 2 |
| L3 bank2 reg3 log error | 0xB0BC | SARB | 2 |
| L3 bank2 reg4 log error | 0xB0C0 | SARB | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| L3 bank2 reg5 log error | 0xB0C4 | SARB | 2 |
| L3 bank2 reg6 log error | 0xB0C8 | SARB | 2 |
| L3 bank2 reg7 log error | 0xB0CC | SARB | 2 |
| L3 bank3 reg0 log error | 0xB0D0 | SARB | 2 |
| L3 bank3 reg1 log error | 0xB0D4 | SARB | 2 |
| L3 bank3 reg2 log error | 0xB0D8 | SARB | 2 |
| L3 bank3 reg3 log error | 0xB0DC | SARB | 2 |
| L3 bank3 reg4 log error | 0xB0E0 | SARB | 2 |
| L3 bank3 reg5 log error | 0xB0E4 | SARB | 2 |
| L3 bank3 reg6 log error | 0xB0E8 | SARB | 2 |
| L3 bank3 reg7 log error | 0xB0EC | SARB | 2 |
| L3 SQC register 4 | 0xB034 | SARB | 2 |
| 3DSTATE_VS | | SVG | 6 |
| 3DSTATE_BINDING_TABLE_POINTERS_VS | | SVG | 2 |
| 3DSTATE_SAMPLER_STATE_POINTERS_VS | | SVG | 2 |
| 3DSTATE_CONSTANT_VS | | SVG | 7 |
| 3DSTATE_URB_VS | | SVG | 2 |
| 3DSTATE_HS | | SVG | 7 |
| 3DSTATE_BINDING_TABLE_POINTERS_HS | | SVG | 2 |
| 3DSTATE_SAMPLER_STATE_POINTERS_HS | | SVG | 2 |
| 3DSTATE_CONSTANT_HS | | SVG | 7 |
| 3DSTATE_URB_HS | | SVG | 2 |
| 3DSTATE_TE | | SVG | 4 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| 3DSTATE_DS | | SVG | 6 |
| 3DSTATE_BINDING_TABLE_POINTERS_DS | | SVG | 2 |
| 3DSTATE_SAMPLER_STATE_POINTERS_DS | | SVG | 2 |
| 3DSTATE_CONSTANT_DS | | SVG | 7 |
| 3DSTATE_URB_DS | | SVG | 2 |
| 3DSTATE_GS | | SVG | 7 |
| 3DSTATE_BINDING_TABLE_POINTERS_GS | | SVG | 2 |
| 3DSTATE_SAMPLER_STATE_POINTERS_GS | | SVG | 2 |
| 3DSTATE_CONSTANT_GS | | SVG | 7 |
| 3DSTATE_URB_GS | | SVG | 2 |
| 3DSTATE_STREAMOUT | | SVG | 3 |
| 3DSTATE_CLIP | | SVG | 4 |
| 3DSTATE_VIEWPORT_STATE_POINTERS_CL_SF | | SVG | 2 |
| 3DSTATE_SF | | SVG | 7 |
| 3DSTATE_SCISSOR_STATE_POINTERS | | SVG | 2 |
| 3DSTATE_MULTISAMPLE | | SVG | 4 |
| 3DSTATE_DRAWING_RECTANGLE | | SVG | 4 |
| SWTESS_BASE_ADDRESS | | SVG | 2 |
| NOOP | | SVG | 2 |
| 3DSTATE_WM | | SVL | 3 |
| 3DSTATE_VIEWPORT_STATE_POINTERS_CC | | SVL | 2 |
| 3DSTATE_CC_STATE_POINTERS | | SVL | 2 |
| 3DSTATE_DEPTHSTENCIL_STATE_POINTERS | | SVL | 2 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| 3DSTATE_SAMPLE_MASK | | SVL | 2 |
| 3DSTATE_SBE | | SVL | 14 |
| 3DSTATE_CONSTANT_PS | | SVL | 7 |
| 3DSTATE_PS | | SVL | 8 |
| 3DSTATE_BINDING_TABLE_POINTERS_PS | | SVL | 2 |
| 3DSTATE_SAMPLER_STATE_POINTERS_PS | | SVL | 2 |
| 3DSTATE_BLEND_STATE_POINTERS | | SVL | 2 |
| Load_Register_Immediate header | 0x1100_100B | SVL | 1 |
| Cache_Mode_0 | 0x7000 | SVL | 2 |
| Cache_Mode_1 | 0x7004 | SVL | 2 |
| GT_MODE | 0x7008 | SVL | 2 |
| FBC_RT_BASE_ADDR_REGISTER | 0x7020 | SVL | 2 |
| STATE_SIP | | SVL | 2 |
| 3DSTATE_DEPTH_BUFFER | | SVL | 7 |
| 3DSTATE_STENCIL_BUFFER | | SVL | 3 |
| 3DSTATE_HIER_DEPTH_BUFFER | | SVL | 3 |
| 3DSTATE_CLEAR_PARAMS | | SVL | 3 |
| NOOP | | SVL | 3 |
| NOOP | | TDL0 | 1 |
| Load_Register_Immediate header | 0x1100_1011 | TDL0 | 1 |
| TD_CTL2 | 0xE404 | TDL0 | 2 |
| TD_VF_VS_EMSK | 0xE408 | TDL0 | 2 |
| TD_GS_EMSK | 0xE40C | TDL0 | 2 |

| POWER CONTEXT | | | |
|---|---|---|---|
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| TD_WIZ_EMSK | 0xE410 | TDL0 | 2 |
| TD_TS_EMSK | 0xE428 | TDL0 | 2 |
| TD_HS_EMSK | 0xE4B0 | TDL0 | 2 |
| TD_DS_EMSK | 0xE4B4 | TDL0 | 2 |
| NOOP | | TDL0 | 12 |
| NOOP | | TDL1 | 1 |
| Load_Register_Immediate header | 0x1100_1011 | TDL1 | 1 |
| TD_CTL2 | 0xF404 | TDL1 | 2 |
| TD_VF_VS_EMSK | 0xF408 | TDL1 | 2 |
| TD_GS_EMSK | 0xF40C | TDL1 | 2 |
| TD_WIZ_EMSK | 0xF410 | TDL1 | 2 |
| TD_TS_EMSK | 0xF428 | TDL1 | 2 |
| TD_HS_EMSK | 0xF4B0 | TDL1 | 2 |
| TD_DS_EMSK | 0xF4B4 | TDL1 | 2 |
| NOOP | | TDL1 | 12 |
| NOOP | | WM | 1 |
| Load_Register_Immediate header | 0x1100_1003 | WM | 1 |
| SuperSpan Count | 0x5520 | WM | 2 |
| 3DSTATE_POLY_STIPPLE_PATTERN | | WM | 33 |
| 3DSTATE_AA_LINE_PARAMS | | WM | 3 |
| 3DSTATE_POLY_STIPPLE_OFFSET | | WM | 2 |
| 3DSTATE_LINE_STIPPLE | | WM | 3 |
| NOOP | | WM | 1 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| NOOP | | SC0 | 1 |
| Load_Register_Immediate header | 0x1100_1003 | SC0 | 1 |
| NOOP | | SC0 | 10 |
| NOOP | | SC1 | 1 |
| Load_Register_Immediate header | 0x1100_1003 | SC1 | 1 |
| NOOP | | SC1 | 10 |
| 3DSTATE_MONOFILTER_SIZE | | SC0/1 | 2 |
| 3DSTATE_CHROMA_KEY | | SC0/1 | 16 |
| NOOP | | SC0/1 | 6 |
| MEDIA_VFE_STATE | | VFE | 8 |
| MEDIA_CURBE_LOAD | | VFE | 4 |
| MEDIA_INTERFACE_DESCRIPTOR_LOAD | | VFE | 4 |
| MEDIA_OBJECT_PRT/GPGPU_WALKER | | VFE | 16 |
| MEDIA_STATE_FLUSH | | VFE | 2 |
| NOOP | | VFE | 6 |
| 3DSTATE_SAMPLER_PALETTE_LOAD0 | | DM0/1 | 257 |
| 3DSTATE_SAMPLER_PALETTE_LOAD1 | | DM0/1 | 257 |
| NOOP | | DM0/1 | 14 |
| NOOP | | SOL | 1 |
| Load_Register_Immediate header | 0x1100_1027 | SOL | 1 |
| SO_NUM_PRIMS_WRITTEN0 | 0x5200 | SOL | 4 |
| SO_NUM_PRIMS_WRITTEN1 | 0x5208 | SOL | 4 |
| SO_NUM_PRIMS_WRITTEN2 | 0x5210 | SOL | 4 |

| | | | |
|---|---|---|---|
| POWER CONTEXT | | | |
| | | | |
| MAIN CONTEXT | | | |
| EXTENDED CONTEXT | | | |
| | | | |
| **Description** | | | **# of DW** |
| SO_NUM_PRIMS_WRITTEN3 | 0x5218 | SOL | 4 |
| SO_PRIM_STORAGE_NEEDED0 | 0x5240 | SOL | 4 |
| SO_PRIM_STORAGE_NEEDED1 | 0x5248 | SOL | 4 |
| SO_PRIM_STORAGE_NEEDED2 | 0x5250 | SOL | 4 |
| SO_PRIM_STORAGE_NEEDED3 | 0x5258 | SOL | 4 |
| SO_WRITE_OFFSET0 | 0x5280 | SOL | 2 |
| SO_WRITE_OFFSET1 | 0x5284 | SOL | 2 |
| SO_WRITE_OFFSET2 | 0x5288 | SOL | 2 |
| SO_WRITE_OFFSET3 | 0x528C | SOL | 2 |
| 3DSTATE_SO_BUFFER | | SOL | 16 |
| NOOP | | SOL | 3 |
| 3DSTATE_SO_DECL_LIST | | SOL | 259 |
| 3DSTATE_INDEX_BUFFER | | VF | 3 |
| 3DSTATE_VERTEX_BUFFERS | | VF | 133 |
| 3DSTATE_VERTEX_ELEMENTS | | VF | 69 |
| 3DSTATE_VF_STATISTICS | | VF | 1 |
| NOOP | | VF | 2 |

## 6.22.2 Pipelined State Page

This page is used a scratch area for the pipeline to store pipelined state that is not referenced indirectly. Under no circumstances should SW read from or write to this page.

## 6.22.3 Ring Buffer

This page is used a scratch area for the pipeline to store ring buffer commands that need to be reissued. Under no circumstances should SW read from or write to this page.

## 6.22.4 The Per-Process Hardware Status Page

The following table defines the layout of the Per-process Hardware Status Page:

| DWord Offset | Description |
|---|---|
| (3FFh – 020h) | These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions. |
| 1F:1C | **Reserved.** |
| 1B | **Context Save Finished Timestamp** |
| 1A | **Context Restore Complete Timestamp** |
| 19 | **Pre-empt Request Received Timestamp** |
| 18 | **Last Switch Timestamp** |
| 17:12 | **Reserved.** |
| F:5 | **Reserved.** |
| 4 | **Ring Head Pointer Storage:** The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an “automatic report” (see RINGBUF registers). |
| 3:0 | **Reserved.** |

# Revision History

| Revision Number | Description | Revision Date |
|---|---|---|
| 1.0 | First 2012 OpenSource edition | May 2012 |

§§