3.5.4 CONSTANT_BUFFER

CONSTANT_BUFFER			
Project:	All	**LenSNBh Bias:**	2

The CONSTANT_BUFFER packet is used to define the memory address of data that will be read by the CS unit and stored into the current CURBE entry.

Programming Notes:

- Issuing a CONSTANT_BUFFER packet with **Valid** set when the CS unit does not have any CURBE entries allocated in the URB results in UNDEFINED behavior.
- 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.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h GFXPIPE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 02h CONSTANT_BUFFER Format: OpCode
	15:9	**Reserved** Project: All Format: MBZ
	8	**Valid** Project: All Format: Enable If TRUE, a Constant Buffer will be defined and possibly used in the pipeline (depending on FF unit state programming). The **Buffer Starting Address** and **Buffer LenSNBh** fields are valid. If FALSE, the Constant Buffer becomes undefined and unused. The **Buffer Starting Address** and **Buffer LenSNBh** fields are ignored. The FF unit state descriptors must not specify the use of CURBE data, or behavior is UNDEFINED.
	7:0	**DWord LenSNBh** Default Value: 0h Excludes DWord (0,1) Format: =n Total LenSNBh - 2 Project: All

CONSTANT_BUFFER		
1	31:6	**Buffer Starting Address** Project: All Format: GeneralStateOffset[31:6] or GraphicsAddress[31:6] (see below) If **Valid** is set and INSTPM<**CONSTANT_BUFFER Address Offset Disable**> is clear (enabled), this field defines the location of the memory-resident constant data via a 64Byte-granular offset from the **General State Base Address**. If **Valid** is set and INSTPM<**CONSTANT_BUFFER Address Offset Disable**> is set (disabled), this field defines the location of the memory-resident constant data via a 64Byte-granular Graphics Address (not offset). **Programming Notes** Constant Buffers can only be allocated in linear (not tiled) graphics memory Constant Buffers can only be mapped to Main Memory (UC)
	5:0	**Buffer LenSNBh** Project: All Format: U6 Count-1 in 512-bit units If **Valid** is set, this field specifies the lenSNBh of the constant data to be loaded from memory into the CURBE in 512-bit units (minus one). The lenSNBh must be less than or equal to the **URB Entry Allocation Size** specified via the CS_URB_STATE command.

3.5.5 MEMORY_OBJECT_CONTROL_STATE

This 4-bit field is used in various state commands and indirect state objects to define MLC/LLC cacheability, graphics data type, and encryption attributes for memory objects.

Bit De	scription
3	**Encrypted Data** This field controls whether data is decrypted while being read. This field is ignored for writes. Format = Enable
2	**Graphics Data Type (GFDT)** This field contains the GFDT bit for this surface when writes occur. GFDT can also be set by the SNBT. The effective GFDT is the logical OR of this field with the GFDT from the SNBT entry. This field is ignored for reads. 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. Format = U1
1:0	**Cacheability Control** This field controls cacheability in the mid-level cache (MLC) and last-level cache (LLC). . Format = U2 enumerated type 00: use cacheability control bits from SNBT entry 01: data is not cached in LLC or MLC 10: data is cached in LLC but not MLC 11: data is cached in both LLC and MLC

3.6 Memory Access Indirection

The GPE supports the indirection of certain graphics (SNBT-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.

Table 3-2. Base Address Utilization

Base Address Used	Memory Accesses
General State Base Address	CS unit reads from **CURBE Constant Buffers** via CONSTANT_BUFFER when INSTPM< **CONSTANT_BUFFER Address Offset Disable**> is clear (enabled).
	3D Pipeline FF state read by the 3D FF units, as referenced by state pointers passed via 3DSTATE_PIPELINE_POINTERS.
	Media pipeline FF state, as referenced by state pointers passed via MEDIA_PIPELINE_POINTERS
	DataPort memory accesses resulting from **'stateless' DataPort Read/Write requests**. See *DataPort* for a definition of the 'stateless' form of requests.
General State Base Address	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
General State Base Address [Pre-DevILK] Instruction Base Address [DevILK] 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 unit reads from **CURBE Constant Buffers** via CONSTANT_BUFFER when INSTPM< **CONSTANT_BUFFER Address Offset Disable**> is set (disabled).
	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 *Data Port* for a definition of the 'stateless' form of requests.
	Memory reads resulting from **STATE_PREFETCH** commands
	Any **physical memory access** by the device
	SNBT-mapped accesses not included above (i.e., default)

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 SNBT)
GraphicsAddress[n:m]	Corresponding bits of an absolute, virtual graphics memory byte address (mapped by a SNBT)
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 SNBT)
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 SNBT)
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 SNBT)
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 SNBT)

3.6.1 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 Table 3-2. Base Address Utilization 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.

3.6.1.1 [Pre-DevILK]

STATE_BASE_ADDRESS			
Project:	[Pre-DevILK]	**LenSNBh 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 3-2. Base Address Utilization 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.
- MI_FLUSH command with ISC invalidate bit set should always be programmed prior to STATE_BASE_ADDRESS command.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h GFXPIPE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h STATE_BASE_ADDRESS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord LenSNBh** Default Value: 4h Excludes DWord (0,1) Format: =n Total LenSNBh - 2 Project: All

STATE_BASE_ADDRESS

1	31:12	**General State Base Address** Project: All Format: GraphicsAddress[31:12] Specifies the 4K-byte aligned base address for general state accesses. See Table 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ
	0	**General State Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set.

Value Name	Description	Project
0h	Disable	Ignore the updated address	All
1h	Enable	Modify the address	All

2	31:12	**Surface State Base Address** Project: All Format: GraphicsAddress[31:12] Specifies the 4K-byte aligned base address for binding table and surface state accesses. See Table 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ
	0	**Surface State Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set.

Value Name	Description	Project
0h	Disable	Ignore the updated address	All
1h	Enable	Modify the address	All

3	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 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ

	0	**Indirect Object Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set. **Value Name** / **Description** / **Project** 0h Disable — Ignore the updated address — All 1h Enable — Modify the address — All
4	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 3-2). 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
5	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

STATE_BASE_ADDRESS		
	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

3.6.1.2 [DevILK]

STATE_BASE_ADDRESS			
Project:	[DevILK]	**LenSNBh 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 3-2. Base Address Utilization 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.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h GFXPIPE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h STATE_BASE_ADDRESS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord LenSNBh** Default Value: 6h Excludes DWord (0,1) Format: =n Total LenSNBh - 2 Project: All

STATE_BASE_ADDRESS

1	31:12	**General State Base Address** Project: All Format: GraphicsAddress[31:12] Specifies the 4K-byte aligned base address for general state accesses. See Table 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ
	0	**General State Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set.

Value Name	Description	Project
0h	Disable	Ignore the updated address	All
1h	Enable	Modify the address	All

2	31:12	**Surface State Base Address** Project: All Format: GraphicsAddress[31:12] Specifies the 4K-byte aligned base address for binding table and surface state accesses. See Table 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ
	0	**Surface State Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set.

Value Name	Description	Project
0h	Disable	Ignore the updated address	All
1h	Enable	Modify the address	All

3	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 3-2 for details on where this base address is used.
	11:1	**Reserved** Project: All Format: MBZ

	0	**Indirect Object Base Address Modify Enable** Project: All Format: Enable The address in this dword is updated only when this bit is set. **Value Name** / **Description** / **Project** 0h Disable — Ignore the updated address — All 1h Enable — Modify the address — All
4	31:12	**Instruction Base Address** Project: All Format: GraphicsAddress[31:12] Specifies the 4K-byte aligned base address for all EU instruction accesses.
	11:1	**Reserved** Project: All Format: MBZ
	0	**Instruction Base Address Modify Enable** Project: All Format: Enable The address in this dword is 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	**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 3-2). 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

6	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

7	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

3.7 State Invalidation ([DevCTG+])

The STATE_POINTER_INVALIDATE command is provided as an optional mechanism to invalidate 3D/Media state pointers and pointers to constant data. This is sometimes desirable to prevent prefetching of state between the time the pointed-to state is no longer needed, and the time the commands above are re-issued to point to new state.

3.7.1 STATE_ POINTER_INVALIDATE ([DevCTG+])

STATE_POINTER_INVALIDATE			
Project:	[DevCTG], [DevILK]	**LenSNBh Bias:**	1

The STATE_POINTER_INVALIDATE command marks the state pointers of the selected type(s) as invalid. The corresponding state pointer command must be issued again prior to attempting any rendering operations that depend on the state whose pointers have been marked as invalid.

The pointers initialized by the following commands are (potentially) invalidated by this command:

- 3DSTATE_PIPELINE_POINTERS
- 3DSTATE_CC_POINTERS
- CONSTANT_BUFFER
- MEDIA_STATE_POINTERS

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 1h GFXPIPE_SINGLE_DW Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h GFXPIPE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 02h STATE_POINTER_INVALIDATE Format: OpCode
	15:3	**Reserved** Project: All Format: MBZ
	2	**Pipelined State Pointers Invalidate** Project: All Format: Invalidate Enable The pointers initialized with the last 3DSTATE_PIPELINED_POINTERS are marked as invalid if this bit is set. Said pointers are unaffected if this bit is clear.
	1	**Constant Buffer Invalidate** Project: All Format: Invalidate Enable The pointer initialized with the last CONSTANT_BUFFER is marked as invalid. Said pointer is unaffected if this bit is clear.

STATE_POINTER_INVALIDATE		
	0	**Media State Pointers Invalidate** Project: All Format: Invalidate Enable The pointers initialized with the last MEDIA_STATE_POINTERS are marked as invalid. Said pointers are unaffected if this bit is clear.

3.8 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).

3.8.1 STATE_ PREFETCH

STATE_PREFETCH			
Project:	All	**LenSNBh 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** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h GFXPIPE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 03h STATE_PREFETCH Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord LenSNBh** Default Value: 0h Excludes DWord (0,1) Format: =n Total LenSNBh - 2 Project: All
1	31:6	**Prefetch Pointer** Project: All Format: GraphicsAddress[31:6] 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** Project: All Format: MBZ

STATE_PREFETCH		
	2:0	**Prefetch Count** Project: All Format: U3 count of cache lines (minus one) Range [0,7] indicating a count of [1,8] Indicates the number of contiguous 64-byte cache lines that will be prefetched.

3.9 System Thread Configuration

3.9.1 STATE_ SIP

STATE_SIP			
Project:	All	**LenSNBh Bias:**	2
The STATE_SIP command specifies the starting instruction location of the System Routine that is shared by all threads in execution.			

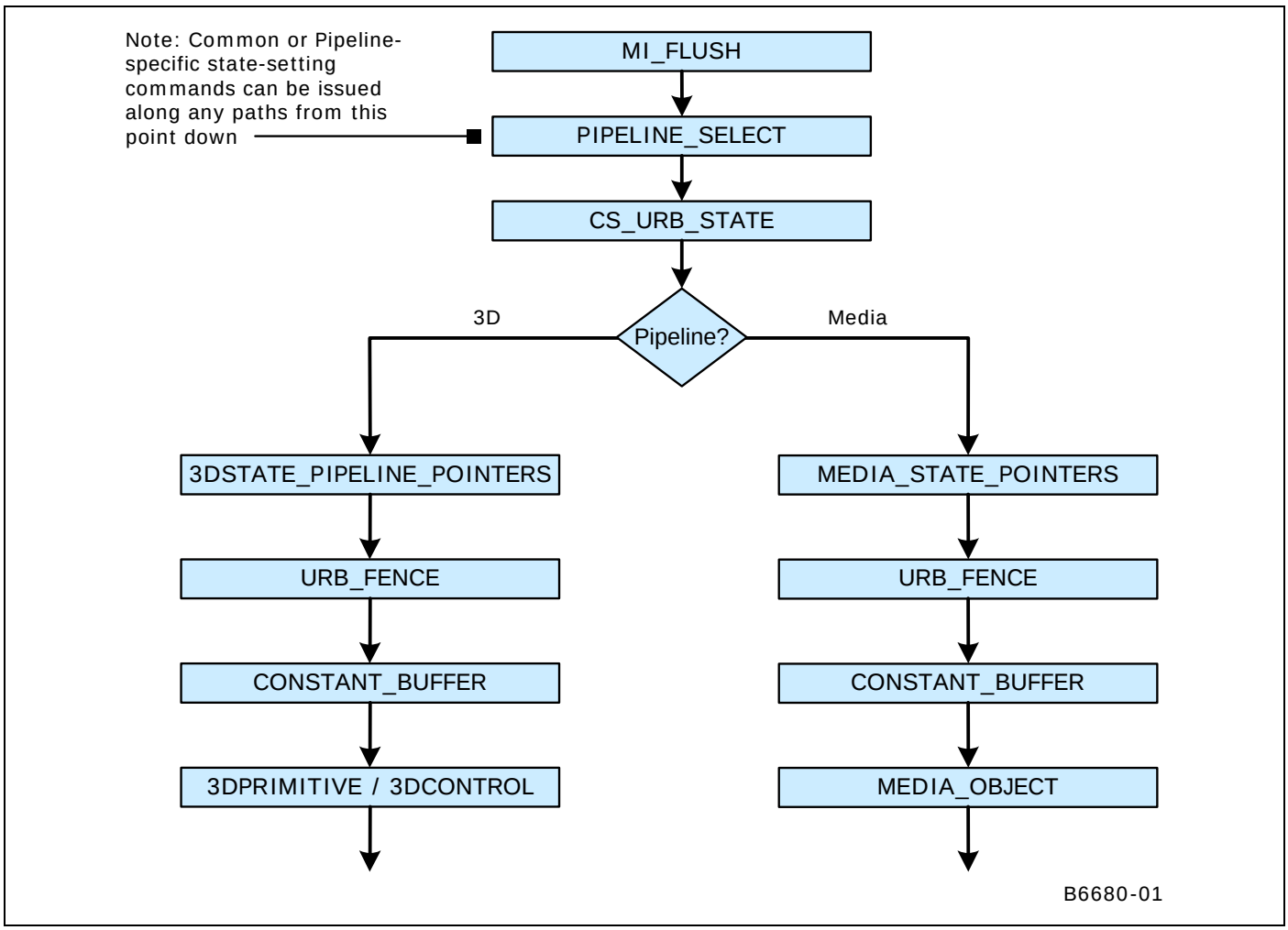
DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h GFXPIPE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 02h STATE_SIP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

Errata	Description	Project
BWT007	Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)	**[DevBW-A]**

3.10 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.



3.10.1 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 Section 3.3 for details on the PIPELINE_SELECT command.

3.10.2 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.

3.10.3 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 below).

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.

3.10.4 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).

3.10.5 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.

3.10.6 Media Pipeline-Specific State-Setting Commands

The following commands are 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.

3.10.7 URB_FENCE (URB Fencing & Entry Allocation)

URB_FENCE command is used to initiate URB entry deallocation/allocation processes within pipeline FF units. The URB_FENCE command is first processed by the CS FF unit, and is then directed down the currently selected pipeline to the FF units comprising that pipeline.

As the FF units receive the URB_FENCE command, a URB entry deallocation/allocation process with be initiated if (a) the FF unit is currently enabled (note that some cannot be disabled) and (b) the **ModifyEnable** bit associated with that FF unit's **Fence** value is set. If these conditions are met, the deallocation of the FF unit's currently-allocated URB entries (if any) commences. (Implementation Note: For better performance, this deallocation proceeds in parallel with allocation of new handles).

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.

The allocation of new handles (if any) for the FF unit then commences. The parameters used to perform this allocation come from (a) the URB_FENCE **Fence** values, and (b) the relevant URB entry state associated with the FF unit: specifically, the **Number of URB Entrie**s and the **URB Entry Allocation Size**. For the CS unit, this state is programmed via CS_URB_STATE, while the other FF units receive this state indirectly via PIPELINED_STATE_POINTERS or MEDIA_STATE_POINTERS commands.

Although a FF unit's allocation process relies on it's URB **Fence** as well as the relevant FF unit pipelined state, only the URB_FENCE command initiates URB entry deallocation/allocation. This imposes the following restriction: If a change is made to (a) the **Number of URB Entries** or **URB Entry Allocation Size** state for a given FF unit or (b) the **Maximum Number of Threads** state for the GS or CLIP FF units, a URB_FENCE command specifying a valid URB Fence state for that FF unit must be subsequently issued – at some point prior to the next CONSTANT_BUFFER, 3DPRIMITIVE (if using the 3D pipeline) or MEDIA_OBJECT (if using the Media pipeline). It is invalid to change **Number of URB Entries** or **URB Entry Allocation Size** state for an enabled FF units without also issuing a subsequent URB_FENCE command specifying a valid **Fence** valid for that FF unit.

It is valid to change a FF unit's Fence value without specifying a change to its **Number of URB Entries** or **URB Entry Allocation Size** state, though the values must be self-consistent.

3.10.8 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.

3.10.9 3DPRIMITIVE

Before issuing a 3DPRIMITIVE command, all state (with the exception of MEDIA_STATE_POINTERS) 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 3DPRIMITIVE.

3.10.10MEDIA_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.

3.11 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 Video Engine.

Its logic functions include

- MMIO register programming interface.
- DMA action for fetching of run lists and ring data from memory.
- Management of the Head pointer for the Ring Buffer.
- Decode of ring data and sending it to the appropriate destination
- Handling of user interrupts and ring context switch interrupt.
- Flushing the Video 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 Video 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.

4. Graphics Command Formats

4.1 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 Genx 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 (includes Trusted Ops)
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

On [DevBW] and [DevCL], no Video Codec Command Parser is present.

Graphics commands vary in lenSNBh, though are always multiples of DWords. The lenSNBh 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 lenSNBh of the command

Note that command *sequences* require QWord alignment and padding to QWord lenSNBh 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.

4.1.1 Memor y 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.

4.1.2 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 lenSNBh, though there are a few commands that include a variable amount of "inline" data at the end of the command.

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

4.1.3 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.

4.1.4 Video Codec Commands

4.1.4.1 AVC Commands [DevCTG/DevILK]

The AVC commands are used to program the AVC Bit-Stream Serial Decoder attached to the Video Codec Command Parser. See the *AVC BSD* chapter for a description of these commands.

4.1.4.2 VC1 Commands [DevCTG/DevILK]

The VC1 commands are used to program the VC1 Bit-Stream Serial Decoder attached to the Video Codec Command Parser. See the *VC1 BSD* chapter for a description of these commands.

4.1.5 Command Header

Table 4-1. RCP Command Header Format

Bits							
TYPE 31:29		**28:24**		**23**	**22**	**21:0**	
Memory Interface (MI)	000	Opcode 00h – NOP 0Xh – Single DWord Commands 1Xh – Two+ DWord Commands 2Xh – Store Data Commands 3Xh – Ring/Batch Buffer Cmds			Identification No./DWord Count Command Dependent Data 5:0 – DWord Count 5:0 – DWord Count 5:0 – DWord Count		
Reserved	001	Opcode – 11111		23:19 Sub Opcode 00h – 01h	18:16 Re-served	15:0 DWord Count	
2D	010	Opcode				Command Dependent Data 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	1XX	XX					

NOTES:

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).

4.2 Command Map

This section provides a map of the graphics command opcodes.

4.2.1 Memory Interface Command Map

All the following commands are defined in *Memory Interface Commands*.Table 4-2. Memory Interface Commands for RCP

Opcode (28:23)	Command	Pipe		
		Render	Video [DevCTG+]	Blitter
1-DWord				
00h	MI_NOOP	All	All	All
01h	Reserved			
02h	MI_USER_INTERRUPT	All	All	All
03h	MI_WAIT_FOR_EVENT	All	All	All
04h	MI_FLUSH	All	All	
05h	MI_ARB_CHECK	All	All	All
06h	Reserved			
07h	MI_REPORT_HEAD	All	All	All
08h	MI_ARB_ON_OFF	[DevCTG+]		
09h	Reserved			
0Ah	MI_BATCH_BUFFER_END	All	All	All
0Bh	MI_SUSPEND_FLUSH	[DevILK]		
0Fh	Reserved			
2+ DWord				
10h	Reserved			
11h	MI_OVERLAY_FLIP Reserved [DevCTG+]	[pre-DevCTG]		
12h	MI_LOAD_SCAN_LINES_INCL Reserved	All		
13h	MI_LOAD_SCAN_LINES_EXCL Reserved	All		
14h	MI_DISPLAY_BUFFER_INFO [DevBW], [DevCL] MI_DISPLAY_FLIP [DevCTG+]	All		
15h	Reserved			
16h	MI_SEMAPHORE_MBOX [DevBW], [DevCL] Reserved	[DevCTG+]	All	All
17h	Reserved			

Opcode (28:23)	Command	Pipe		
		Render	Video [DevCTG+]	Blitter
18h	MI_SET_CONTEXT	All		
1Ah–1Fh	Reserved			
Store Data				
20h	MI_STORE_DATA_IMM	All	All	All
21h	MI_STORE_DATA_INDEX	All	All	All
22h	MI_LOAD_REGISTER_IMM	All	All	All
23h	MI_UPDATE_SNBT	[DevCTG+]		
24h	MI_STORE_REGISTER_MEM	All	All	All
25h	MI_PROBE	[DevCTG] [DevILK]		
26h	MI_FLUSH_DW [DevILK] This is the opcode for MI_REPORT_PERF_COUNT. It only applied to Render pipe		All	All
28h	MI_REPORT_PERF_COUNT	[DevILK]		
2Ah–2Fh	Reserved			
Ring/Batch Buffer				
30h	Reserved			
31h	MI_BATCH_BUFFER_START	All	All	All
32h–35h	Reserved			
37h–3Fh	Reserved			

4.2.2 2D Command Map

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

Opcode (28:22)	Command	Comments
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	
23h–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	
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	

4.2.3 3D/Media Command Map

Pipeline Type (28:27)	Opcode Sub	Opcode	Command De	finition Chapter
Common (pipelined)	**Bits 26:24**	**Bits 23:16**		
0h	0h	00h	URB_FENCE	Graphics Processing Engine
0h	0h	01h	CS_URB_STATE [Pre-DevSNB]	Graphics Processing Engine
0h	0h	02h	CONSTANT_BUFFER [Pre-DevSNB]	Graphics Processing Engine
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	04h–FFh	Reserved	n/a
Reserved	**Bits 26:24**	**Bits 23:16**		
0h	2h–7h	XX	Reserved	n/a

Pipeline Type (28:27)	Opcode Sub	Opcode	Command De	finition Chapter
Single DW	**Opcode (26:24)**	**Bits 23:16**		
1h	0h	00h-01h	Reserved	n/a
1h	0h	02h	STATE_POINTER_INVALIDATE [DevCTG+]	Graphics Processing Engine
1h	0h	03h-0Ah	Reserved	n/a
1h	0h	0Bh	3DSTATE_VF_STATISTICS	Vertex Fetch
1h	0h	0Ch-FFh	Reserved	n/a
1h	1h	00h-03h	Reserved	n/a
1h	1h	04h	PIPELINE_SELECT	Graphics Processing Engine
1h	1h	05h-FFh	Reserved	n/a
1h	2h-7h	XX	Reserved	n/a

Media	Opcode (26:24)	Bits 23:16		
2h	0h	00h	MEDIA_STATE_POINTERS	Media
2h	0h	05h-FFh	Reserved	n/a
2h	1h	00h	MEDIA_OBJECT	Media
2h	1h	01h	MEDIA_OBJECT_EX	Media
2h	1h	02h	MEDIA_OBJECT_PRT	Media
2h	1h	04h-FFh	Reserved	n/a
2h	2h–7h	XX	Reserved	n/a

Pipeline Type (28:27)	Opcode	Sub Opcode	Command De	finition Chapter
3D State (Pipelined)	**Bits 26:24**	**Bits 23:16**		
3h	0h	00h	3DSTATE_PIPELINED_POINTERS	3D Pipeline
3h	0h	03h	Reserved	n/a
3h	0h	05h	Reserved	3D Pipeline
3h	0h	08h	3DSTATE_VERTEX_BUFFERS	Vertex Fetch
3h	0h	09h	3DSTATE_VERTEX_ELEMENTS	Vertex Fetch
3h	0h	0Ah	3DSTATE_INDEX_BUFFER	Vertex Fetch
3h	0h	0Bh	3DSTATE_VF_STATISTICS	Vertex Fetch
3h	0h	0Ch	Reserved	n/a
3h	0h	11h	3DSTATE_GS [DevSNB+]	Geometry Shader
3h	0h	12h	3DSTATE_CLIP [DevSNB+]	Clipper
3h	0h	13h	3DSTATE_SF [DevSNB+]	Strips & Fans
3h	0h	14h	3DSTATE_WM [DevSNB+]	Windower
3D State (Non-Pipelined)	**Bits 26:24**	**Bits 23:16**		
3h	1h	00h	3DSTATE_DRAWING_RECTANGLE	Strips & Fans
3h	1h	01h	3DSTATE_CONSTANT_COLOR	Color Calculator
3h	1h	02h	3DSTATE_SAMPLER_PALETTE_LOAD0	Sampling Engine
3h	1h	03h	Reserved	
3h	1h	04h	3DSTATE_CHROMA_KEY	Sampling Engine

Pipeline Type (28:27)	Opcode	Sub Opcode	Command De	finition Chapter
3h	1h	05h	3DSTATE_DEPTH_BUFFER	Windower
3h	1h	06h	3DSTATE_POLY_STIPPLE_OFFSET	Windower
3h	1h	07h	3DSTATE_POLY_STIPPLE_PATTERN	Windower
3h	1h	08h	3DSTATE_LINE_STIPPLE	Windower
3h	1h	09h	3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP	Windower
3h	1h	0Ah	[DevCTG]: 3DSTATE_AA_LINE_PARAMS [DevCTG+]	Windower
3h	1h	0Bh	3DSTATE_GS_SVB_INDEX [DevCTG+]	Geometry Shader
3h	1h	0Ch	3DSTATE_SAMPLER_PALETTE_LOAD1 [DevCTG-B+]	Sampling Engine
3h	1h	0Eh	3DSTATE_STENCIL_BUFFER [DevILK] Reserved [ILK,	Windower
3h	1h	0Fh	3DSTATE_HIER_DEPTH_BUFFER [ILK, Reserved [ILK,]	Windower
3h	1h	10h	3DSTATE_CLEAR_PARAMS [ILK,	Windower
3h	1h	11h	3DSTATE_MONOFILTER_SIZE [ILK]	Sampling Engine
3h	1h	17h	3DSTATE_SO_DECL_LIST	HW Streamout
3h	1h	18h	3DSTATE_SO_BUFFER	HW Streamout
3h	1h	19h–FFh	Reserved	n/a
3D (Control)	**Bits 26:24**	**Bits 23:16**		
3h	2h	00h	PIPE_CONTROL	3D Pipeline
3h	2h	01h–FFh	Reserved	n/a
3D (Primitive)	**Bits 26:24**	**Bits 23:16**		
3h	3h	00h	3DPRIMITIVE	Vertex Fetch
3h	3h	01h–FFh	Reserved	n/a
3h	4h–7h	00h–FFh	Reserved	n/a

4.2.4 Video Codec Command Map

4.2.4.1 AVC BSD Command Map [DevCTG/DevILK]

This map is N/A to [DevBW], [DevCL]

Table 4-3. AVC Commands for the VCCP

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command De	finition Chapter
AVC State				
2h	4h	0h	AVC_BSD_IMG_STATE	AVC BSD
2h	4h	1h	AVC_BSD_QM_STATE	AVC BSD
2h	4h	2h	AVC_BSD_SLICE_STATE	AVC BSD
2h	4h	3h	AVC_BSD_BUF_BASE_STATE	AVC BSD
2h	4h	4h	BSD_IND_OBJ_BASE_ADDR	AVC BSD
2h	4h	5h-7h	Reserved	n/a
AVC Object				
2h	4h	8h	AVC_BSD_OBJECT	AVC BSD
2h	4h	9h-FFh	Reserved	n/a

4.2.4.2 VC1 BSD Command Map [DevCTG/DevILK]

This map is N/A to [DevBW], [DevCL].

Pipeline Type (28:27)	Opcode (26:24)	Sub Opcode (23:16)	Command De	finition Chapter
VC1 State				
2h	5h	0h	VC1_BSD_PIC_STATE	VC1 BSD
2h	5h	1h	Reserved	n/a
2h	5h	2h	Reserved	n/a
2h	5h	3h	VC1_BSD_BUF_BASE_STATE	VC1 BSD
2h	5h	4h	Reserved	n/a
2h	5h	5h-7h	Reserved	n/a

VC1 Object				
2h	5h	8h	VC1_BSD_OBJECT	VC1 BSD
2h	5h	9h-FFh	Reserved	n/a

5. Register Address Maps

5.1 Graphics Register Address Map

This chapter provides address maps of the graphics controllers I/O and memory-mapped registers. Individual register bit field descriptions are provided in the following chapters. PCI configuration address maps and register bit descriptions are provided in the following chapter.

5.1.1 Memory and I/O Space Registers

This section provides a high-level register map (register groupings per function). The memory and I/O maps for the graphics device registers are shown in the following table, except PCI Configuration registers that are described in the following chapter.

NOTE: The VGA and Extended VGA registers can be accessed via standard VGA I/O locations as well as via memory-mapped locations.

NOTE: All graphics MMIO registers can also be accessed via CPU I/O. See IOBASE, MMIO_INDEX and MMIO_DATA I/O registers in the *MontaraGM Cspec*.

The memory space address listed for each register is an offset from the base memory address programmed into the MMADR register (PCI configuration offset 14h).

Table 5-1. Graphics Controller Register Memory and I/O Map

Start Offset	End Offset	Description
00000h	00FFFh	**VGA and Extended VGA Control Registers.** These registers are located in both I/O space and memory space. The VGA and Extended VGA registers contain the following register sets: General Control/Status, Sequencer (SRxx), Graphics Controller (GRxx), Attribute Controller (Arxx), VGA Color Palette, and CRT Controller (CRxx) registers. Detailed bit descriptions are provided in the *VGA and Extended VGA Register* Chapter. The registers within a set are accessed using an indirect addressing mechanism as described at the beginning of each section. Note that some of the register description sections have additional operational information at the beginning of the section
01000h	01FFFh	***Reserved***

Start Offset	End Offset	Description
02000h	02FFFh	**Instruction, Memory, and Interrupt Control Registers:** **Instruction Control Registers** Ring Buffer registers and page table control registers are located in this address range. Various instruction status, error, and operating registers are located in this group of registers. **Graphics Memory Fence Registers.** The Graphics Memory Fence registers are used for memory tiling capabilities. **Interrupt Control/Status Registers.** This register set provides interrupt control/status for various GC functions. **Display Interface Control Register.** This register controls the FIFO watermark and provides burst lenSNBh control. **Logical Context Registers** **Pipeline Statistic Counters**
03000h	031FFh	**FENCE & Per Process SNBT Control registers**
03200h	03FFFh	**Frame Buffer Compression Registers**
04000h	043FFh	**Instruction Control Registers for Secondary (BSD) Command Streamer.** **On [DevBW] and [DevCL] this range is *Reserved.***
04400h	04FFFh	**Video Decode Fixed Function Control Registers.** **On [DevBW] and [DevCL] this range is *Reserved.***
05000h	05FFFh	**I/O Control Registers**
06000h	06FFFh	**Clock Control Registers**. This memory address space is the location of the GC clock control and power management registers
09000h	09FFFh	***Reserved***
0A000h	0AFFFh	**Display Palette Registers**
0B000h	0FFFFh	***Reserved***
10000h	13FFFh	**MMIO MCHBAR.** Alias through which the graphics driver can access registers in the MCHBAR accessed through device 0.
14000h	2FFFFh	***Reserved***
30000h	3FFFFh	**Overlay Registers.** These registers provide control of the overlay engine. The overlay registers are double-buffered with one register buffer located in graphics memory and the other on the device. On-chip registers are not directly writeable. To update the on-chip registers software writes to the register buffer area in graphics memory and instructs the device to update the on-chip registers.
40000h	5FFFFh	***Reserved***
60000h	6FFFFh	**Display Engine Pipeline Registers**
70000h	72FFFh	**Display and Cursor Registers**
73000h	73FFFh	**Performance Counters**
74000h	7FFFFh	***Reserved***

5.1.2 PCI Configuration Space

See the releveant EDS/C-Specs for details on accessing PCI configuration space, PCI address map tables, and register descriptions.

5.1.3 Graphics Register Memory Address Map

All graphics device registers are directly accessible via memory-mapped I/O and indirectly accessible via the MMIO_INDEX and MMIO_DATA I/O registers. In addition, the VGA and Extended VGA registers are I/O mapped.

5.2 VGA and Extended VGA Register Map

For I/O locations, the value in the address column represents the register I/O address. For memory mapped locations, this address is an offset from the base address programmed in the MMADR register.

5.2.1 VGA and Extended VGA I/O and Memory Register Map

Table 5-2. I/O and Memory Register Map

Address	Register Name (Read)	Register Name (Write)
2D Registers		
3B0h–3B3h	Reserved	Reserved
3B4h	VGA CRTC Index (CRX) (monochrome)	VGA CRTC Index (CRX) (monochrome)
3B5h	VGA CRTC Data (monochrome)	VGA CRTC Data (monochrome)
3B6h–3B9h	Reserved	Reserved
3Bah	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3BBh–3BFh	Reserved	Reserved
3C0h	VGA Attribute Controller Index (ARX)	VGA Attribute Controller Index (ARX)/ VGA Attribute Controller Data (alternating writes select ARX or write ARxx Data)
3C1h	VGA Attribute Controller Data (read ARxx data)	Reserved
3C2h	VGA Feature Read Register (ST00)	VGA Miscellaneous Output Register (MSR)
3C3h	Reserved	Reserved
3C4h	VGA Sequencer Index (SRX)	VGA Sequencer Index (SRX)
3C5h	VGA Sequencer Data (SRxx)	VGA Sequencer Data (SRxx)
3C6h	VGA Color Palette Mask (DACMASK)	VGA Color Palette Mask (DACMASK)
3C7h	VGA Color Palette State (DACSTATE)	VGA Color Palette Read Mode Index (DACRX)

Address	Register Name (Read)	Register Name (Write)
3C8h	VGA Color Palette Write Mode Index (DACWX)	VGA Color Palette Write Mode Index (DACWX)
3C9h	VGA Color Palette Data (DACDATA)	VGA Color Palette Data (DACDATA)
3CAh	VGA Feature Control Register (FCR)	Reserved
3CBh	Reserved	Reserved
3CCh	VGA Miscellaneous Output Register (MSR)	Reserved
3CDh	Reserved	Reserved
3CEh	VGA Graphics Controller Index (GRX)	VGA Graphics Controller Index (GRX)
3CFh	VGA Graphics Controller Data (GRxx)	VGA Graphics Controller Data (GRxx)
3D0h–3D1h	Reserved	Reserved
2D Registers		
3D4h	VGA CRTC Index (CRX)	VGA CRTC Index (CRX)
3D5h	VGA CRTC Data (CRxx)	VGA CRTC Data (CRxx)
System Configuration Registers		
3D6h	GFX/2D Configurations Extensions Index (XRX)	GFX/2D Configurations Extensions Index (XRX)
3D7h	GFX/2D Configurations Extensions Data (XRxx)	GFX/2D Configurations Extensions Data (XRxx)
2D Registers		
3D8h–3D9h	Reserved	Reserved
3DAh	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3DBh–3DFh	Reserved	Reserved

5.3 Indirect VGA and Extended VGA Register Indices

The registers listed in this section are indirectly accessed by programming an index value into the appropriate SRX, GRX, ARX, or CRX register. The index and data register address locations are listed in the previous section. Additional details concerning the indirect access mechanism are provided in the *VGA and Extended VGA Register Description* Chapter (see SRxx, GRxx, ARxx or CRxx sections).

Table 5-3. 2D Sequence Registers (3C4h / 3C5h)

Index Sy m		Description
00h	SR00	Sequencer Reset
01h	SR01	Clocking Mode
02h	SR02	Plane / Map Mask
03h	SR03	Character Font
04h	SR04	Memory Mode
07h	SR07	Horizontal Character Counter Reset

Table 5-4. 2D Graphics Controller Registers (3CEh / 3CFh)

Index	Sym	Register Name
00h	GR00	Set / Reset
01h	GR01	Enable Set / Reset
02h	GR02	Color Compare
03h	GR03	Data Rotate
04h	GR04	Read Plane Select
05h	GR05	Graphics Mode
06h	GR06	Miscellaneous
07h	GR07	Color Don't Care
08h	GR08	Bit Mask
10h	GR10	Address Mapping
11h	GR11	Page Selector
18h	GR18	Software Flags

Table 5-5. 2D Attribute Controller Registers (3C0h / 3C1h)

Index	Sym	Register Name
00h	AR00	Palette Register 0
01h	AR01	Palette Register 1
02h	AR02	Palette Register 2
03h	AR03	Palette Register 3
04h	AR04	Palette Register 4
05h	AR05	Palette Register 5
06h	AR06	Palette Register 6
07h	AR07	Palette Register 7
08h	AR08	Palette Register 8
09h	AR09	Palette Register 9
0Ah	AR0A	Palette Register A
0Bh	AR0B	Palette Register B
0Ch	AR0C	Palette Register C
0Dh	AR0D	Palette Register D
0Eh	AR0E	Palette Register E
0Fh	AR0F	Palette Register F
10h	AR10	Mode Control
11h	AR11	Color
12h	AR12	Memory Plane Enable
13h	AR13	Horizontal Pixel Panning
14h	AR14	Color Select

Table 5-6. 2D CRT Controller Registers (3B4h / 3D4h / 3B5h / 3D5h)

Index	Sym	Register Name
00h	CR00	Horizontal Total
01h	CR01	Horizontal Display Enable End
02h	CR02	Horizontal Blanking Start
03h	CR03	Horizontal Blanking End
04h	CR04	Horizontal Sync Start
05h	CR05	Horizontal Sync End
06h	CR06	Vertical Total
07h	CR07	Overflow
08h	CR08	Preset Row Scan
09h	CR09	Maximum Scan Line
0Ah	CR0A	Text Cursor Start
0Bh	CR0B	Text Cursor End
0Ch	CR0C	Start Address High
0Dh	CR0D	Start Address Low
0Eh	CR0E	Text Cursor Location High
0Fh	CR0F	Text Cursor Location Low
10h	CR10	Vertical Sync Start
11h	CR11	Vertical Sync End
12h	CR12	Vertical Display Enable End
13h	CR13	Offset
14h	CR14	Underline Location
15h	CR15	Vertical Blanking Start
16h	CR16	Vertical Blanking End
17h	CR17	CRT Mode
18h	CR18	Line Compare
22h	CR22	Memory Read Latch Data
24h	CR24	Test Register for Toggle State of Attribute Control Register

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 Memor y 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 (SNBT)	Contains PTEs used to translate “graphics addresses” into physical memory addresses.
Hardware Status Page	Cached page of sysmem used to provide fast driver synchronization.
Logical Context Buffer	Memory areas used to store (save/restore) images of hardware rendering contexts. Logical contexts are referenced via a pointer to the corresponding Logical Context Buffer.
Ring Buffers	Buffers used to transfer (DMA) instruction data to the device. Primary means of controlling rendering operations.
Batch Buffers	Buffers of instructions invoked indirectly from Ring Buffers.
State Descriptors	Contains state information in a prescribed layout format to be read by hardware. Many different state descriptor formats are supported.
Vertex Buffers	Buffers of 3D vertex data indirectly referenced through “indexed” 3D primitive instructions.
VGA Buffer (Must be mapped UC on PCI)	Graphics memory buffer used to drive the display output while in legacy VGA mode.
Display Surface	Memory buffer used to display images on display devices.
Overlay Surface	Memory buffer used to display overlaid images on display devices.
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.

Memory Object Type	Role
2D Render R-M-W Destination	2D rendering output surface that is read in order to be combined in the rendering function. Destination surfaces that accessed via this Read-Modify-Write mode have somewhat different restrictions than Write-Only Destination surfaces.
2D Render Write-Only Destination	2D rendering output surface that is written but not read by the 2D rendering function. Destination surfaces that accessed via a Write-Only mode have somewhat different restrictions than Read-Modify-Write Destination surfaces.
2D Monochrome Source	1 bpp surfaces used as inputs to 2D rendering after being converted to foreground/background colors.
2D Color Pattern	8x8 pixel array used to supply the “pattern” input to 2D rendering functions.
DIB	“Device Independent Bitmap” surface containing “logical” pixel values that are converted (via LUTs) to physical colors.
3D Color Buffer	Surface receiving color output of 3D rendering operations. May also be accessed via R-M-W (aka blending). Also referred to as a Render Target.
3D Depth Buffer	Surface used to hold per-pixel depth and stencil values used in 3D rendering operations. Accessed via RMW.
3D Texture Map	Color surface (or collection of surfaces) which provide texture data in 3D rendering operations.
“Non-3D” Texture	Surface read by Texture Samplers, though not in normal 3D rendering operations (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 (though GENX deviates slightly from those behaviors).

6.2.6.1 32-bit Floating Point

Bit De	scription
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 De	scription
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 De	scription
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

Bit De	scription
10:6	**Exponent (e)** Biased Exponent
5:0	**Fraction (f)** Does not include “hidden one”

The value of this data type is derived as:

- if e == 31 and f != 0 then v = 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)

6.2.6.5 10-bit Floating Point

Bit De	scription
9:5	**Exponent (e)** Biased Exponent
4:0	**Fraction (f)** Does not include “hidden one”

The value of this data type is derived as:

- if e == 31 and f != 0 then v = 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)

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

6.3 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.

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.3.4 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

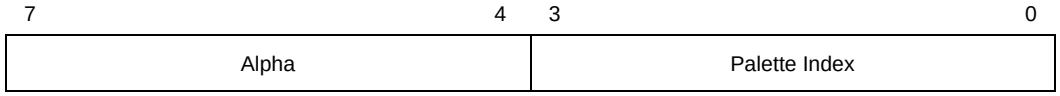
Bit De	scription
T0	**Texel 0** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1
...	**...**
T7	**Texel 7** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1

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

6.3.5 Palette Formats

6.3.5.1 P4A4_UNORM

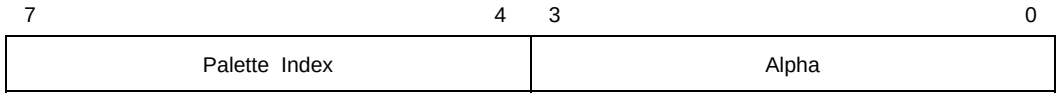
This surface format contains a 4-bit Alpha value (in the high nibble) and a 4-bit Palette Index value (in the low nibble).



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

6.3.5.2 A4P4_UNORM

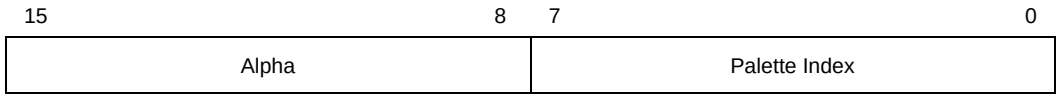
This surface format contains a 4-bit Alpha value (in the low nibble) and a 4-bit Color Index value (in the high nibble).



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

6.3.5.3 P8A8_UNORM

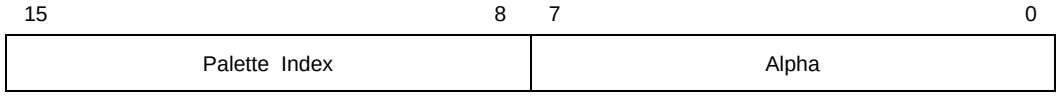
This surface format contains an 8-bit Alpha value (in the high byte) and an 8-bit Palette Index value (in the low byte).



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

6.3.5.4 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).



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

6.3.5.5 P8_UNORM

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

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

6.3.5.6 P2_UNORM

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

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

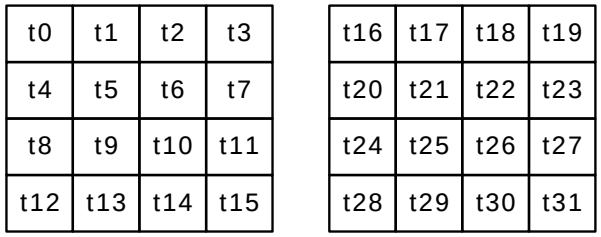
6.4 Compressed Surface Formats

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

6.4.1 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:

Figure 6-1. FXT1 Encoded Blocks



6.4.1.1 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:

Table 6-1. 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.

6.4.1.2 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 8th entry is transparent black). The encoded block contains a 3-bit index value per texel that is used to lookup a color from the table.

6.4.1.2.1 CC_HI Block Encoding

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

Table 6-2. FXT CC_HI Block Encoding

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

6.4.1.2.2 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:

Table 6-3. 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:

Table 6-4. 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.

6.4.1.3 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).

6.4.1.3.1 CC_CHROMA Block Encoding

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

Table 6-5. FXT CC_CHROMA Block Encoding

Bit De	scription
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

6.4.1.3.2 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:

Table 6-6. FXT CC_CHROMA Decoded Colors

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

Table 6-7. FXT CC_CHROMA Interpolated Color Table

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

6.4.1.4 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).

6.4.1.4.1 CC_MIXED Block Encoding

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

Table 6-8. FXT CC_MIXED Block Encoding

Bit De	scription
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

Bit De	scription
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

6.4.1.4.2 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:

Table 6-9. 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:

Table 6-10. 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]
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:

Table 6-11. 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

Table 6-12. 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:

Table 6-13. 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.

Table 6-14. FXT CC_MIXED Decoded Colors (Alpha[0] = 1)

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

Table 6-15. 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

Table 6-16. 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.

6.4.1.5 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.

6.4.1.5.1 CC_ALPHA Block Encoding

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

Table 6-17. FXT CC_ALPHA Block Encoding

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

6.4.1.5.2 CC_ALP HA 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:

Table 6-18. 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.

Table 6-19. 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:

Table 6-20. 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

Table 6-21. 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

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

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

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

6.4.3 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 De	scription
7:0	red_0
15:8	red_1
18:16	texel[0][0] red bit code
21:19	texel[0][1] red bit code
24:22	texel[0][2] red bit code
27:25	texel[0][3] red bit code
30:28	texel[1][0] red bit code
33:31	texel[1][1] red bit code
36:34	texel[1][2] red bit code
39:37	texel[1][3] red bit code
42:40	texel[2][0] red bit code
45:43	texel[2][1] red bit code
48:46	texel[2][2] red bit code
51:49	texel[2][3] red bit code
54:52	texel[3][0] red bit code
57:55	texel[3][1] red bit code
60:58	texel[3][2] red bit code
63:61	texel[3][3] red bit code
71:64	green_0
79:72	green_1
82:80	texel[0][0] green bit code
85:83	texel[0][1] green bit code
88:86	texel[0][2] green bit code
91:89	texel[0][3] green bit code
94:92	texel[1][0] green bit code
97:95	texel[1][1] green bit code
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

Bit De	scription
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.

6.5 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.

6.5.1 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

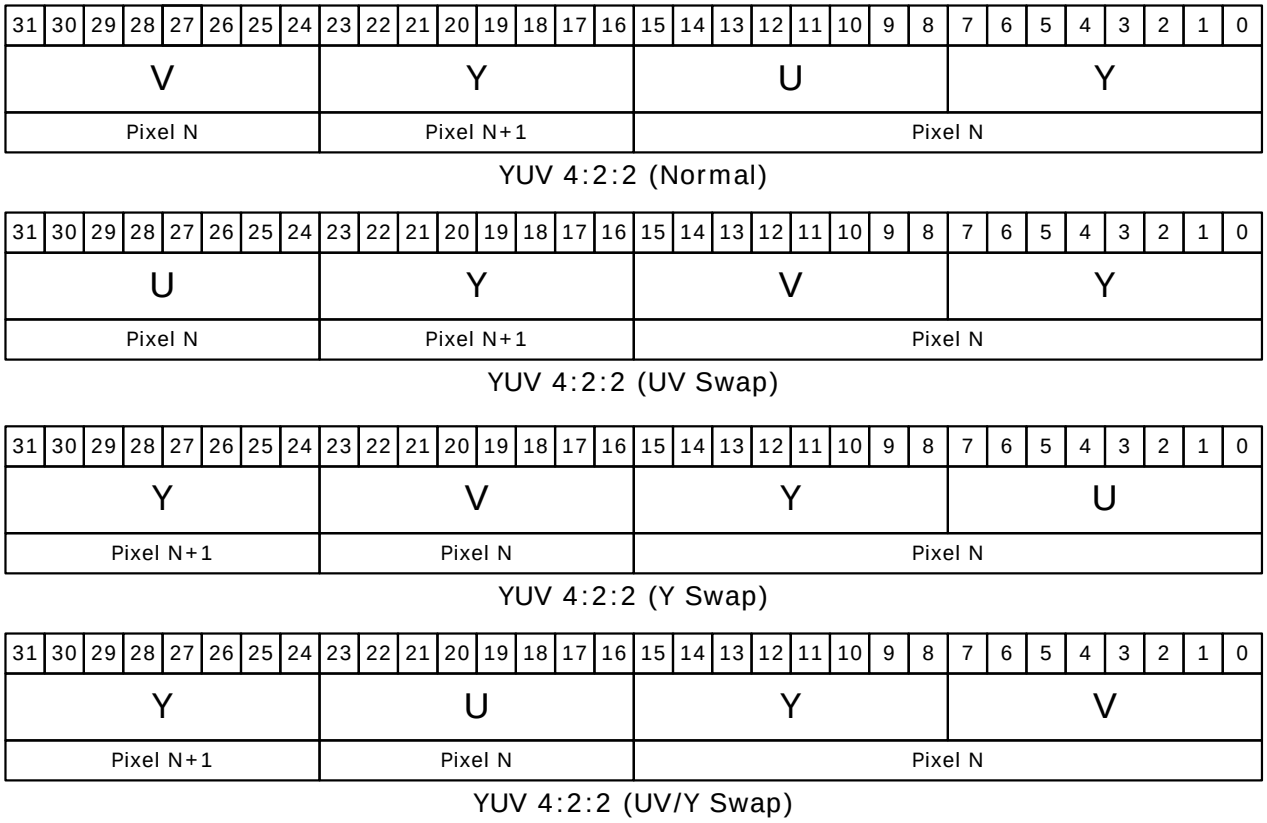


Figure 6-2. Memory layout of packed YUV 4:2:2 formats

6.5.2 Planar Memory Organization

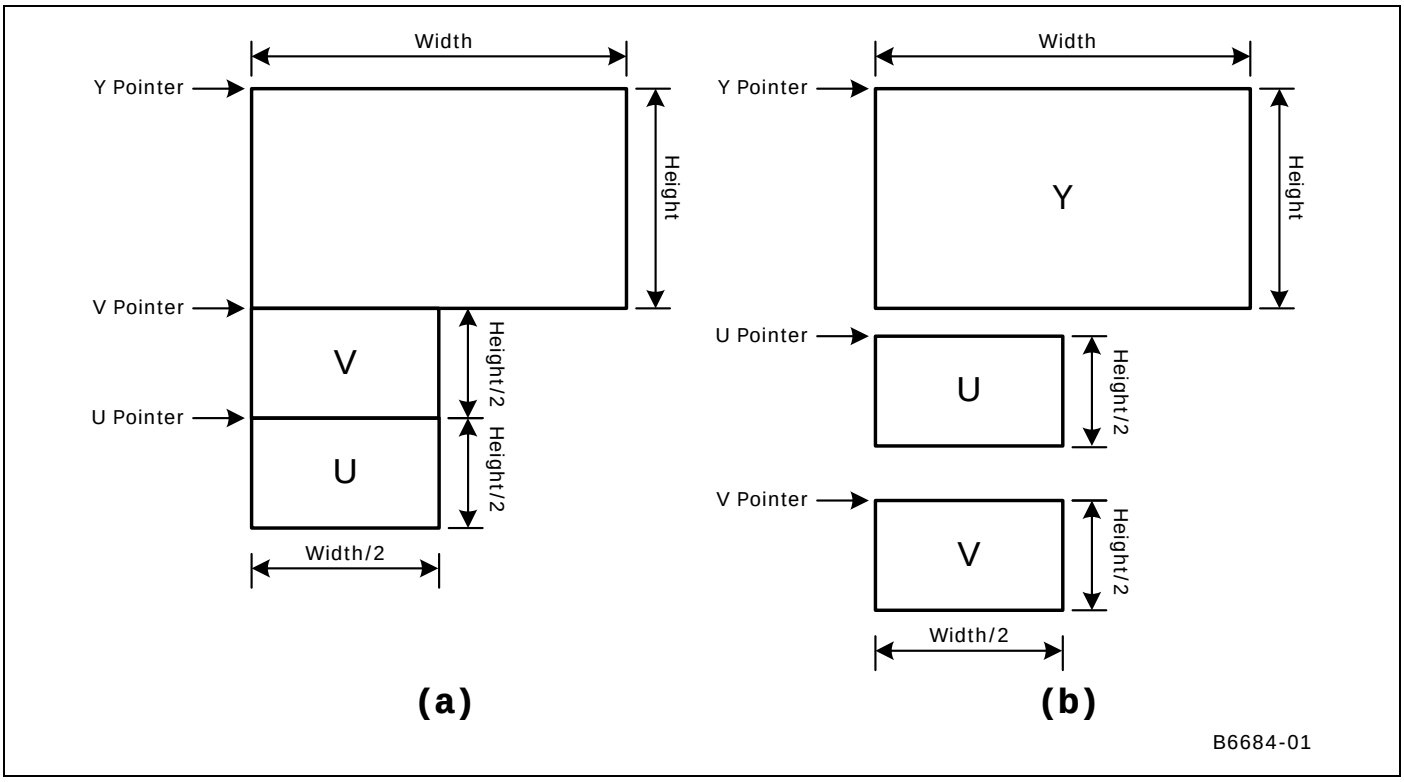
Planar formats use what could be thought of as separate buffers for the three color components. Because there is a separate stride for the Y and U/V data buffers, several memory footprints can be supported.

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:

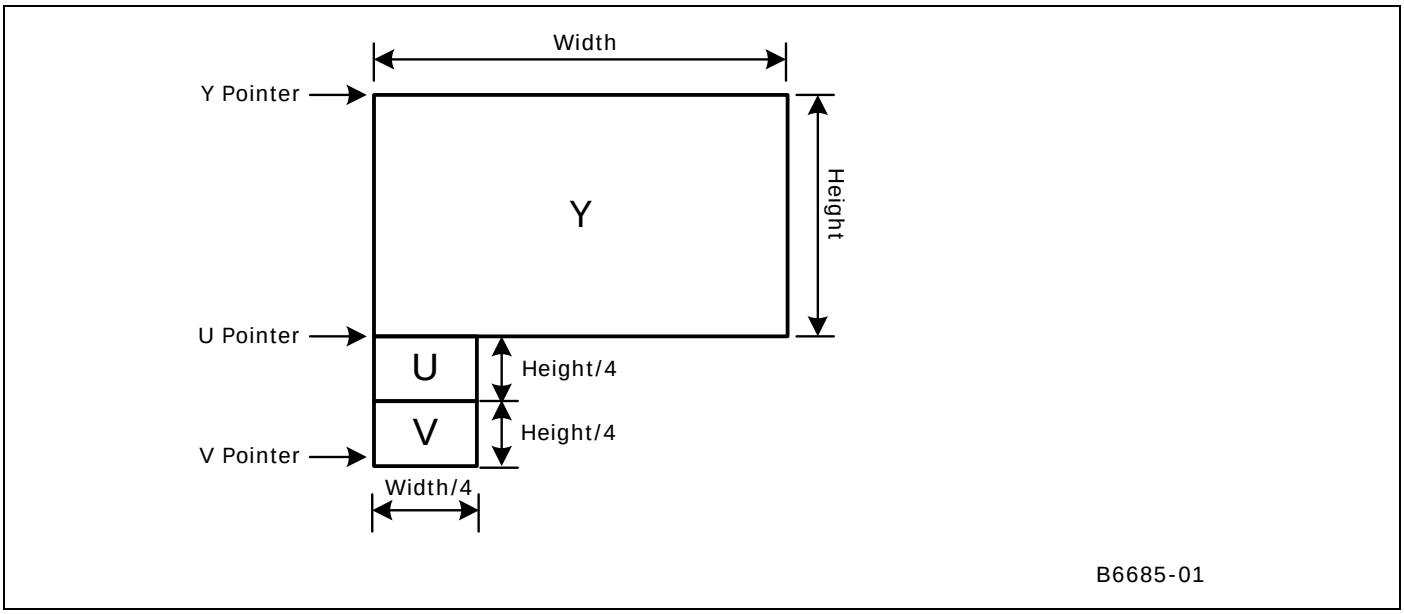
1. The memory organization of the common YV12 data, where all three planes are contiguous and the strides of U and V components are half of that of the Y component.
2. An alternative memory structure that the addresses of the three planes are independent but satisfy certain alignment restrictions.

Figure 6-3. 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. The stride of the U and V planes is a quarter of that of the Y plane.

Figure 6-4. YUV 4:1:0 Format Memory Organization



6.6 Surface Memory Organizations

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

6.7 Graphics Translation Tables

The Graphics Translation Tables SNBT (Graphics Translation Table, sometimes known as the global SNBT) and PPSNBT (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 SNBT and the PPSNBT are programmed via the PSNBBL_CTL and PSNBBL_CTL2 MI registers, respectively. The translation table base addresses must be 4KB aligned. The SNBT size can be either 128KB, 256KB or 512KB (mapping to 128MB, 256MB, and 512MB aperture sizes respectively) and is physically contiguous. The global SNBT should only be programmed via the range defined by SNBTADR. The PPSNBT is programmed directly in memory. The per-process SNBT (PPSNBT) size is controlled by the PSNBBL_CTL2 register. The PPSNBT can, in addition to the above sizes, also be 64KB in size (corresponding to a 64MB aperture). Refer to the SNBT Range chapter for a bit definition of the PTE entries.

6.8 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.

6.9 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 lenSNBh 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.

6.10 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 lenSNBh. The ending address is the address of the last valid QWord in the buffer. The lenSNBh of any single batch buffer is “virtually unlimited” (i.e., could theoretically be 4GB in lenSNBh).

6.11 Display, Overlay, Cursor Surfaces

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

6.12 2D Render Surfaces

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

Note that the device provides no coherency between 2D render surfaces and the texture cache – i.e., the texture cache must be explicitly invalidated prior to the use of 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 1bpp surfaces are used as source operands to certain 2D Blt operations, where the Blt engine expands the 1bpp source into the required color depth.

The device uses the texture cache to store monochrome sources. There is no mechanism to maintain coherency between 2D render surfaces and (texture)-cached monochrome sources, software is required to 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 are used to hold per-pixel color values for use in the 3D pipeline. Note that the 3D pipeline always requires a Color buffer to be defined.

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

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 and a linear Depth buffers are used together:

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

6.16 3D Depth Buffer Surfaces

Depth buffer surfaces are used to hold per-pixel depth values and per-pixel stencil values for use in the 3D pipeline. Note that the 3D pipeline does not require a Depth buffer to be allocated, though a Depth buffer is required to perform (non-trivial) Depth Test and Stencil Test operations.

The following table summarizes the possible formats of the Depth buffer. Refer to Depth Buffer Formats section in this chapter for details on the pixel formats. Refer to the *Windower* and *DataPort* chapters for details on the usage of the Depth Buffer.

Table 6-22. Depth Buffer Formats

DepthBufferFormat / DepthComponent	bpp	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

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

6.17 3D Separate Stencil Buffer Surfaces [ILK+]

Separate Stencil buffer surfaces are used to 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.

The following table summarizes the possible formats of the Stencil buffer. Refer to Stencil Buffer Formats section in this chapter for details on the pixel formats. Refer to the *Windower* chapters for details on the usage of the Stencil Buffer.

Table 6-23. Depth Buffer Formats

DepthBufferFormat / DepthComponent	bpp	Description
S8_UINT	8	8-bit stencil value in a byte

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

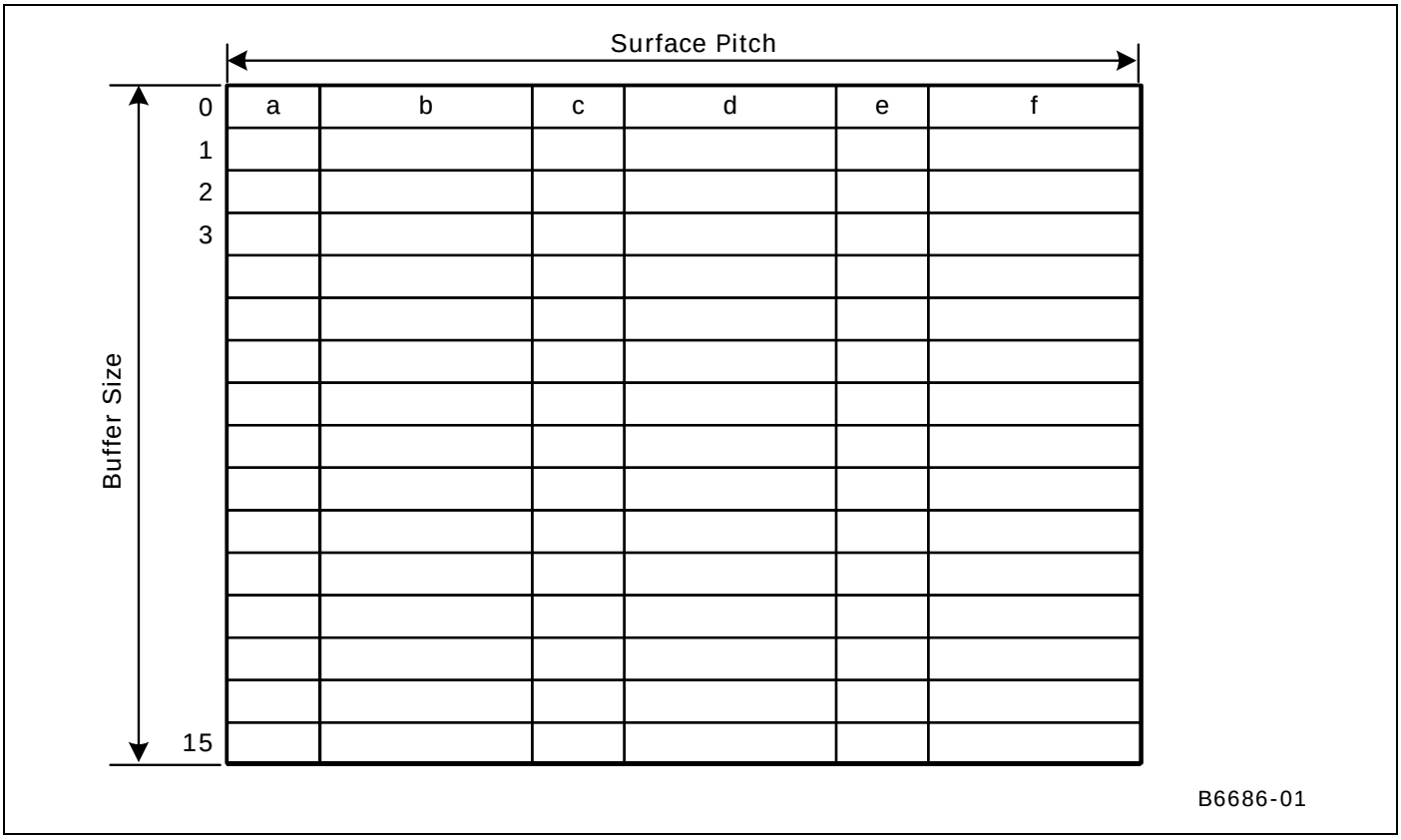
6.18 Surface Layout

This section describes the formats of surfaces and data within the surfaces.

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 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.

6.18.3 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: i.e., 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).

6.18.3.1 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:

$$W_L = \left((width >> L) > 0 ? width >> L : 1\right)$$
$$H_L = \left((height >> L) > 0 ? height >> L : 1\right)$$

6.18.3.2 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:

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

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

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:

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

6.18.3.3 Minimum Pitch

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.

6.18.3.4 Alignment Unit Size

The following table indicates the *i* and *j* values that should be used for each map format. Note that the compressed formats are padded to a full compression cell.

Table 6-24. Alignment Units for Texture Maps

surface format	alignment unit width "*i*"	alignment unit height "*j*"
YUV 4:2:2 formats	4	* see below
BC1-5	4	4
FXT1	8	4
all other formats	4	* see below

* For these formats, the vertical alignment factor "j" is determined as follows:

- For **[All:**
  - j = 4 for any separate stencil buffer surface (**[DevILK]** only)
  - j = 2 for all other surfaces

6.18.3.5 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 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 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).

6.18.3.6 Compressed Mipmap Layout

Mipmaps of textures using compressed (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.

6.18.3.7 Surface Arrays

6.18.3.7.1 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:

$$QPitch = \left(h_0 + h_1 + 11j\right) * Pitch$$

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

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

$$QPitch = \frac{\left(h_0 + h_1 + 11j\right)}{4} * Pitch$$

6.18.3.7.2 For separate stencil buffer [DevILK]

The separate stencil buffer does not support mip mapping, thus the storage for LODs other than LOD 0 is not needed. The following *QPitch* equation applies only to the separate stencil buffer:

$$QPitch = h_0 * Pitch$$

6.18.3.7.3 8.19.4.8.1 MCS Surface

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

6.18.3.8 4x MCS

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

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

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

6.18.3.9 8x MCS

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

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

Other than this, the 8x algorithm is the same as the 4x algorithm. The MCS value indicating clear state is 0x00ffffff.

6.18.3.9.1 MSS Surface

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

6.18.4 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.

6.18.4.1 Hardware Cube Map Layout

6.18.4.1.1 [Pre-Dev ILK]

The cube face textures are stored in the same way as 3D surfaces are stored (see section 0 for details). For cube surfaces, however, the depth is equal to the number of faces (always 6) and is not reduced for each MIP. The equation for D_L is replaced with the following for cube surfaces:

$$D_L = 6$$

The "q" coordinate is replaced with the face identifier as follows:

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

6.18.4.1.2 [Dev ILK+]

The cube face textures are stored in the same way as 2D array surfaces are stored (see section 6.18.3 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

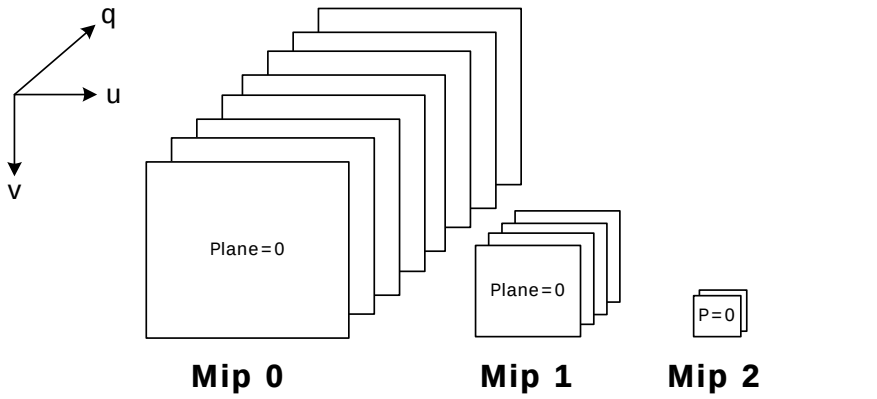
6.18.4.2 Restrictions

- 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 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.

Figure 6-5. 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:

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

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

$$D_L = \left((depth >> L) > 0 \,?\, depth >> L : 1\right)$$

Cache-line aligned width and height are as follows, with i and j being a function of the map format as shown in Table 6-24.

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

Note that it is not necessary to cache-line align in the "depth" dimension (i.e. lower case "*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.

6.18.5.1 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 SNBT, a SNBT error will result when the cache line is accessed. In order to avoid these SNBT 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 SNBT entries to avoid errors. To determine the necessary padding on the bottom and right side of the surface, refer to the table in Section 6.18.3.4 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 SNBT 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 SNBT, a SNBT error will result when the cache request is processed. In order to avoid these SNBT 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.