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

Volume 1 Part 2: Graphics Core – MMIO, Media Registers, and Programming Environment (SandyBridge)

For the 2011 Intel Core Processor Family

May 2011

Revision 1.0

Doc Ref #: IHD-OS-V1 Pt2 – 05 11

Contents

1. Introduction

The HW supports three engines. The render command streamer interfaces to 3D/IE and display streams. The media command streamer interfaces to the fixed function media. The blitter command streamer interfaces to the blit commands.   Software interfaces of all 3 engines are very similar and should only differ on engine-specific functionality.

2. Virtual Memory

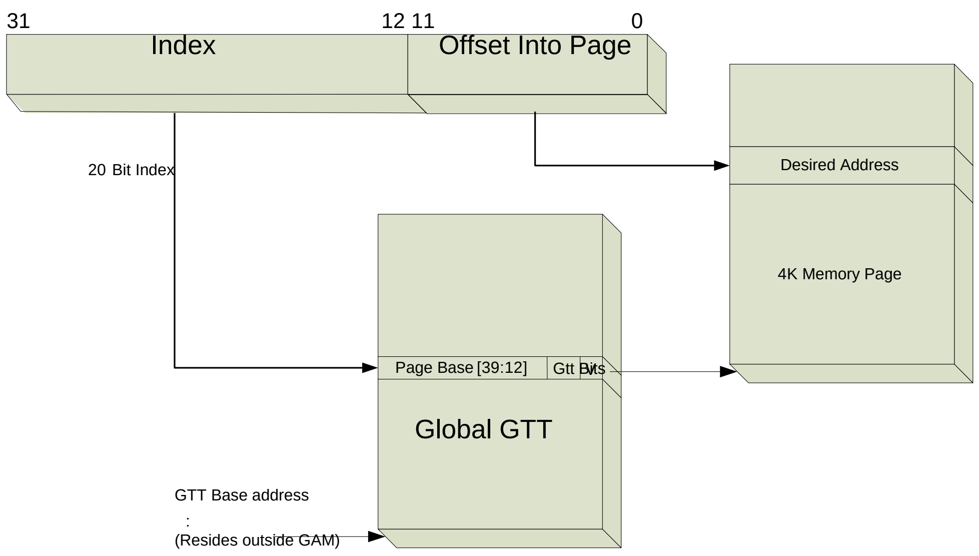
2.1 Virtual Memory Control

SandyBridge [DevSNB] products support different types of graphics translation tables for virtual memory. Global graphics translation table (GGTT) is a single common tranlation table used for all processes. There can be many Per-process graphics translation table (PPGTT). This requires an additional lookup for translation. The actual location is not accessable directly via software since their both located in graphics stolen memory (see graphics memory interface chapter for more detail).

Virtual Memory Structure	**[DevSNB]**
Global (GGTT)	GSM Only
Per-Process (PPGTT)	2-level, Page Directory in GSM, Page Tables anywhere

2.2 Global Virtual Memory

Global Virtual Memory is the default target memory if a PPGTT is not enabled. If a PPGTT is also present, the method to choose which is targeted by memory and rendering operations varies by product. See the sections on Per-Process Virtual Memory for more information. High priority graphics clients such as Display and Cursor always access global virtual memory.



2.2.1 Graphics Translation Table (GTT) Range (GTTADR)

Address Offset:	GTTADR in CPU Physical Space
Access:	Aligned DWord Read/Write

The GTTADR memory BAR defined in graphics device config space is an alias for the Global GTT.

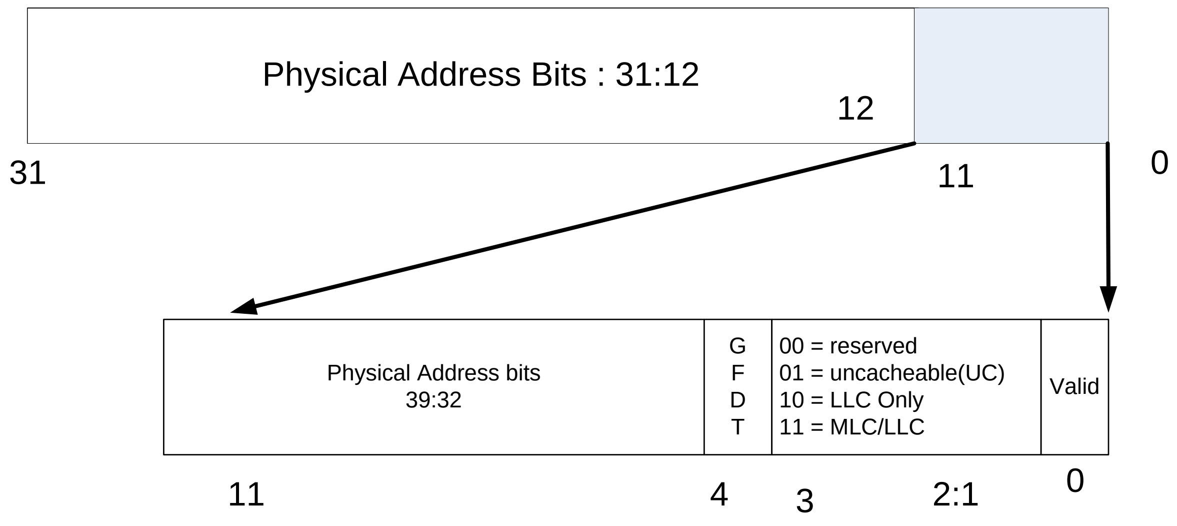
Programming Notes: It is recommended that the driver map all graphics memory pages in the GGTT to some physical page, if only a dummy page.

2.2.2 GTT Page Table Entries (PTEs)

2.2.2.1 GTT Page Table Entries (PTEs) SNB

Page Table Entry: 1 DWord per 4KB Graphics Memory page.

Graphics Page Table Entry Format



Bits	Description
31: 12	**Physical Page Address 31:12:** If the Valid bit is set, This field provides the page number of the physical memory page backing the corresponding Graphics Memory page.
11:4	Physical Start Address Extension: This field specified Bits 39:32 of the page table entry. This field must be zero for 32 bit addresses.

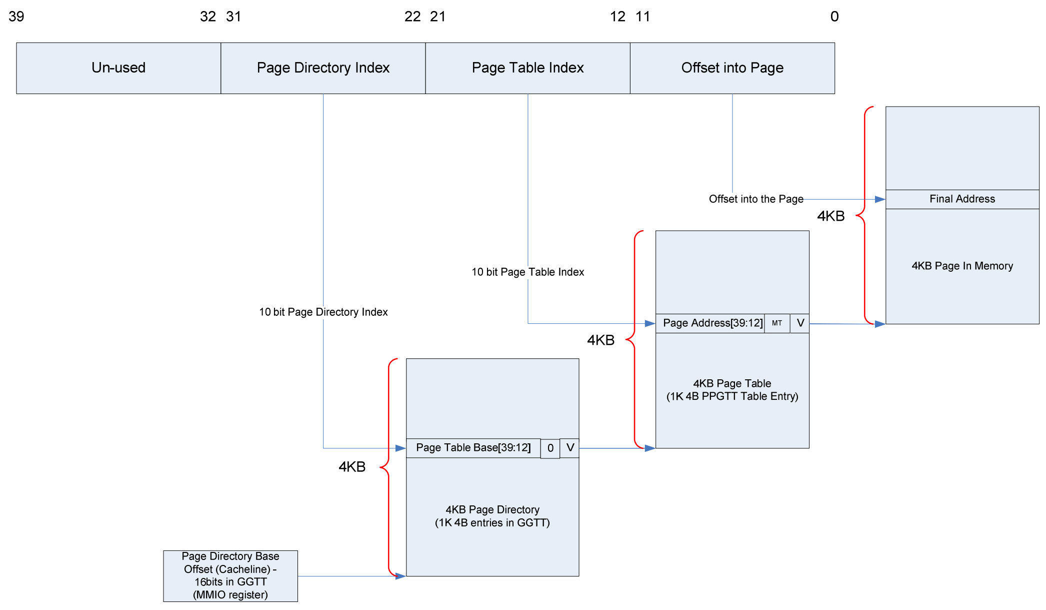
Bits	Description
3	**Graphics Data Type (GFDT)** This field contains the GFDT bit for this surface when writes occur. GFDT can also be set by various state commands and indirect objects. The effective GFDT is the logical OR of the GTT entry with this field. This field is ignored for reads. Format = U1 0: No GFDT (default) 1: GFDT
2	**LLC Cacheability Control (LLCCC)** This is the field used in GT interface block to determine what type of access need to be generated to uncore. For the cases where the LLCCC is set, cacheable transaction are generated to enable LLC usage for particular stream. 0: use cacheability controls from GTT entry 1: Data is cached in LLC
1	**L3 Cacheability Control (L3CC)** This field is used to control the L3 cacheability (allocation) of the stream. 0: not cacheable within L3 1: cacheable in L3 *Note: even if the surface is not cacheable in L3, it is still kept coherent with L3 content.*
0	**Valid PTE:** This field indicates whether the mapping of the corresponding Graphics Memory page is valid. 1: Valid 0: Invalid. An access (other than a CPU Read) through an invalid PTE will result in Page Table Error (Invalid PTE).

2.3 Two-Level Per-Process Virtual Memory

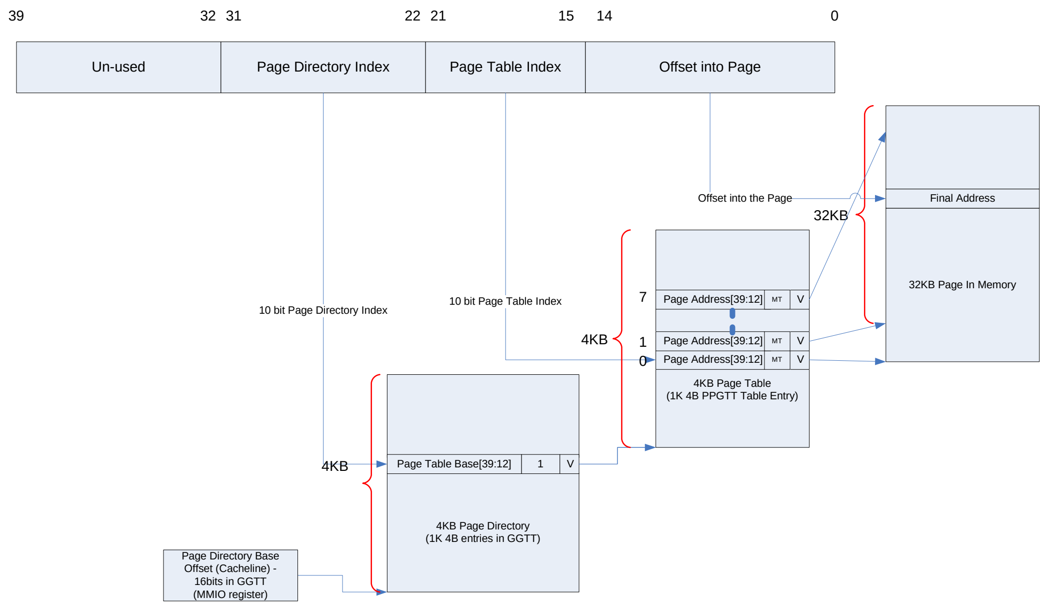
The DevSNB family supports a 2-level mapping scheme for PPGTT, consisting of a first-level page directory containing page table base addresses, and the page tables themselves on the 2nd level, consisting of page addresses. The motivation for the 2-level scheme is simple – it allows for the lookup table (the collection of page tables) to exist in discontiguous memory, making allocation of memory for these structures less problematic for the OS. The directory and each page table fit within a single 4K page of memory that can be located anywhere in physical memory space.

If a PPGTT is enabled, all rendering operations (including blit commands) target Per-process virtual memory. This means all commands *except* the Memory Interface Commands (MI_*). Certain Memory Interface Commands have a specifier to choose global virtual memory (mapped via the GGTT) instead of per-process memory. Global Virtual Memory can be thought of as “privileged” memory in this case. Commands that elect to access privileged memory must have sufficient access rights to do so. Commands executing directly from a ring buffer or from a “secure” batch buffer (see the MI_BATCH_BUFFER_START command in Memory Interface Commands) have these access rights; other commands do not and will not be permitted to access global virtual memory. See the Memory Interface Commands chapters for details on command access of privileged memory.

The PPGTT is disabled by resetting the **Per-Process GTT Enable** bit.



For 32KB Big Pages, the non-big page client the access has no different view, similar to 4KB pages. In case of a big page client, the following access path is used



<u>The starting address of a 32KB page needs to be natively aligned to a 32KB boundary in memory. Hardware uses certain bits (15 and up) to check for the TLB look-ups.</u>

2.3.1 PPGTT Directory Entries (PDEs)

Directory Entry: 1 DWord per 4KB page table (4MB Graphics Memory). Page directories must be located entirely within the GGTT (the table itself.) Directory entries should be updated only for non-active contexts. If a directory entry update is done for a running context, it is unknown when that update will take effect since the device caches directory entries. Directory entries can only be modified using GTTADDR (see *Memory Interface Commands for Rendering Engine*).

31	12	11	4	3	2	1	0
Physical Page Address 31:12		Physical Page Address 39:32		Reserved		Size ("0":4KB, "1":32KB)	Valid

Bits	Description
31:12	**Physical Page Address 31:12**: If the Valid bit is set, This field provides the page number of the physical memory page backing the corresponding Graphics Memory page.
11:8	Reserved: MBZ
7:4	**Physical Page Address Extension:** This field specifies bits 35:32 of the directory entry.
3:1	Reserved: MBZ
1	**Page Size**: Two page sizes are supported thru PDE. **0**: 4KB pages **1**: 32KB pages
0	**Valid PDE:** This field indicates whether this directory entry is valid. 1: Valid 0: Invalid. An access through an invalid PDE will result in a page fault.

2.3.2 PPGTT Table Entries (PTEs)

2.3.2.1 PPGTT Table Entries (PTEs)

Page Table Entry: 1 DWord per 4KB Graphics Memory page. Page Tables must be located in main memory (not snooped). They can be updated directly in memory if proper precautions are taken, or from the command stream by using the MI_UPDATE_GTT command (see *Memory Interface Commands for Rendering Engine*).

31 12	11:4	3	2 1	0
Physical Page Address 31:12	Physical Page Address 39:32	GFDT	Cacheability Control	Valid

Bits	Description
31:12	**Physical Page Address 31:12**: If the Valid bit is set, This field provides the page number of the physical memory page backing the corresponding Graphics Memory page.
11:4	Physical Start Address Extension: This field specified Bits 39:32 of the page table entry. This field must be zero for 32 bit addresses.
3	**Graphics Data Type (GFDT)** This field contains the GFDT bit for this surface when writes occur. GFDT can also be set by various state commands and indirect objects. The effective GFDT is the logical OR of the GTT entry with this field. This field is ignored for reads. Format = U1 0: No GFDT (default) 1: GFDT
2:1	**Cacheability Control** This field controls cacheability in the mid-level cache (MLC) and last-level cache (LLC). Format = U2 enumerated type 00: Reserved 01: data is not cached in LLC 10: data is cached in LLC 11: Reserved
0	**Valid PTE:** This field indicates whether the mapping of the corresponding Graphics Memory page is valid. 1: Valid 0: Invalid. An access (other than a CPU Read) through an invalid PTE will result in Page Table Error (Invalid PTE).

2.3.3 Context TLBs in DevSNB A-step

Due to a bug in DevSNB A-step, the context TLBs are not getting invalidated during a TLB invalidation sequence (i.e. context switching) creating a maintenance/coherency issues for the driver. The workaround of keeping the context pages same is already in place. Other option is to align context GTT entries to CL boundary.

This issue is fixed for B-step onwards.

2.3.4 PPGTT vs GGTT limitation

DevSNB hardware has the limitation to separate the TAGs between PPGTT vs GGTT mapped TLB entries. This requires Virtual Addresses (pages) mapped by PPGTT vs GGTT to be different, else the TLB can mix-match the mappings. Either they need to be different or use same physical mapping for Command Streamer, Vertex Buffers and Instruction Buffers. This limitation is applicable for all the Command Streamer, Vertex Fetch unit, State and Instruction and constant memory spaces.

For DevSNB, s/w will overlap the entire PPGTT and GGTT to work around the issue.

3. GFX MMIO – MCHBAR Aperture

Address Offset:	140000h – 17FFFFh
Default Value:	Same as MCHBAR
Access:	Aligned Word, Dword or Qword Read/Write

This range defined in the graphics MMIO range is an alias with which graphics driver can read and write registers defined in the MCHBAR MMIO space claimed thru Device #0. Attributes for registers defined within the MCHBAR space are preserved when the same registers are accessed via this space. Registers that the graphics driver requires access to are Rank Throttling, GMCH Throttling, Thermal Sensor etc.

The Alias functions works for MMIO access from the CPU only. A command stream load register immediate will drop the data and store register immediate will return all Zeros.

Graphics MMIO registers can be accessed thru MMIO BARs in function #0 and function #1 in Device #2. The aliasing mechanism is turned off if memory access to the corresponding function is turned off via software or in certain power states.

4. Graphics Memory Interface Functions

Note: In the interest of maintainability, functional descriptions of features being added, with the exception of GTT changes, are included in the MI Registers chapter and, to a lesser extent, the *Memory Interface Commands for Rendering* and *Memory Interface Commands for Video Decode* chapters.

4.1 Introduction

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

Information <u>not</u> presented in this chapter includes:

Microarchitectural and implementation-dependent features (e.g., internal buffering, caching and arbitration policies).

MI functions and paths specific to the operation of external (discrete) devices attached via external connections.

MI functions essentially unrelated to the operation of the internal graphics devices, .e.g., traditional "chipset functions" (refer to the device's C-Spec for this information).

4.2 Graphics Memory Clients

The MI function provides memory access functionality to a number of external and internal graphics memory *clients*, as described in Table 4-1.

Table 4-1. Graphics Memory Clients

MI Client	Access Modes
Host Processor	Read/Write of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses
External PEG Graphics Device	**Write-Only** of Graphics Operands located in Main Memory via the Graphics Aperture. (This client is not described in this chapter).
Peer PCI Device	**Write-Only** of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses (i.e., mapped by GTT*). Note that DMI access to Graphics registers is not supported.*
Snooped Read/Write (internal)	Internally-generated snooped reads/writes.
Command Stream (internal)	DMA Read of graphics commands and related graphics data.
Vertex Stream (internal)	DMA Read of indexed vertex data from Vertex Buffers by the 3D Vertex Fetch (VF) Fixed Function.

MI Client	Access Modes
Instruction/State Cache (internal)	Read of pipelined 3D rendering state used by the 3D/Media Functions and instructions executed by the EUs.
Render Cache (internal)	Read/Write of graphics data operated upon by the graphics rendering engines (Blt, 3D, MPEG, etc.) Read of render surface state.
Sampler Cache (internal)	Read of texture (and other sampled surface) data stored in graphics memory.
Display/Overlay Engines (internal)	Read of display, overlay, cursor and VGA data.

4.3 Graphics Memory Addressing Overview

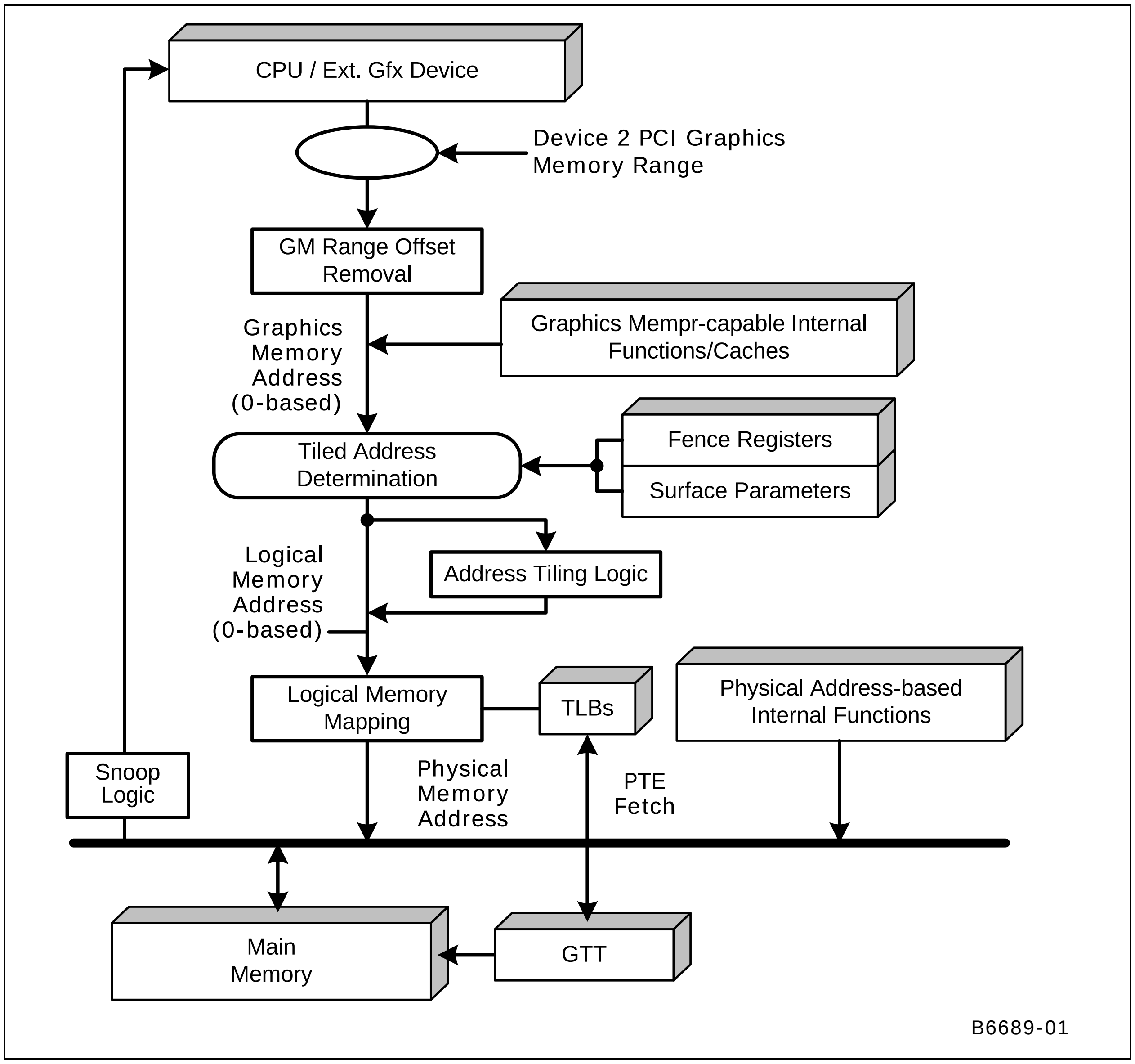
The Memory Interface function provides access to graphics memory (GM) clients. It accepts memory addresses of various types, performs a number of optional operations along *address paths*, and eventually performs reads and writes of graphics memory data using the resultant addresses. The remainder of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

4.3.1 Graphics Address Path

Figure 4-1 shows the internal graphics memory address path, connection points, and optional operations performed on addresses. Externally-supplied addresses are normalized to zero-based *Graphics Memory* (*GM) addresses* (GM_Address). If the GM address is determined to be a tiled address (based on inclusion in a fenced region or via explicit surface parameters), *address tiling* is performed. At this point the address is considered a *Logical Memory address*, and is translated into a *Physical Memory address* via the GTT and associated TLBs. The physical memory location is then accessed.

CPU accesses to graphics memory are not snooped on the front side bus post GTT translation. Hence pages that are mapped cacheable in the GTT will not be coherent with the CPU cache if accessed through graphics memory aperture. Also, such accesses may have side effects in the hardware.

Figure 4-1. Graphics Memory Paths



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

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

4.4 Graphics Memory Address Spaces

Table 4-2 lists the five supported Graphics Memory Address Spaces. Note that the Graphics Memory Range Removal function is automatically performed to transform system addresses to internal, zero-based Graphics Addresses.

Table 4-2. Graphics Memory Address Types

Address Type	Description	Range
Dev2_GM_Address	Address range allocated via the Device 2 (integrated graphics device) GMADR register. The processor and other peer (DMI) devices utilize this address space to read/write graphics data that resides in Main Memory. This address is internally converted to a GM_Address.	Some 64MB, 128MB, 256MB or 512MB address range normally above TOM
GM_Address	Zero-based logical Graphics Address, utilized by internal device functions to access GTT-mapped graphics operands. GM_Addresses are typically passed in commands and contained in state to specify operand location.	[0, 64MB-1], [0, 128MB-1], [0, 256MB-1] or [0, 512MB-1] [0, 1GB-1] on [DevCTG] Only
PGM_Address	Zero-based logical Per-Process Graphics Address, utilized by internal device functions to access render GTT (PPGTT) mapped graphics operands. Memory in this space is not accessible by the processor and other peer (DMI) devices unless aliased to a GM_Address.	[0, 64MB-1], [0,128MB-1], [0,256MB-1], [0,512MB-1] or [0, 1GB – 1]

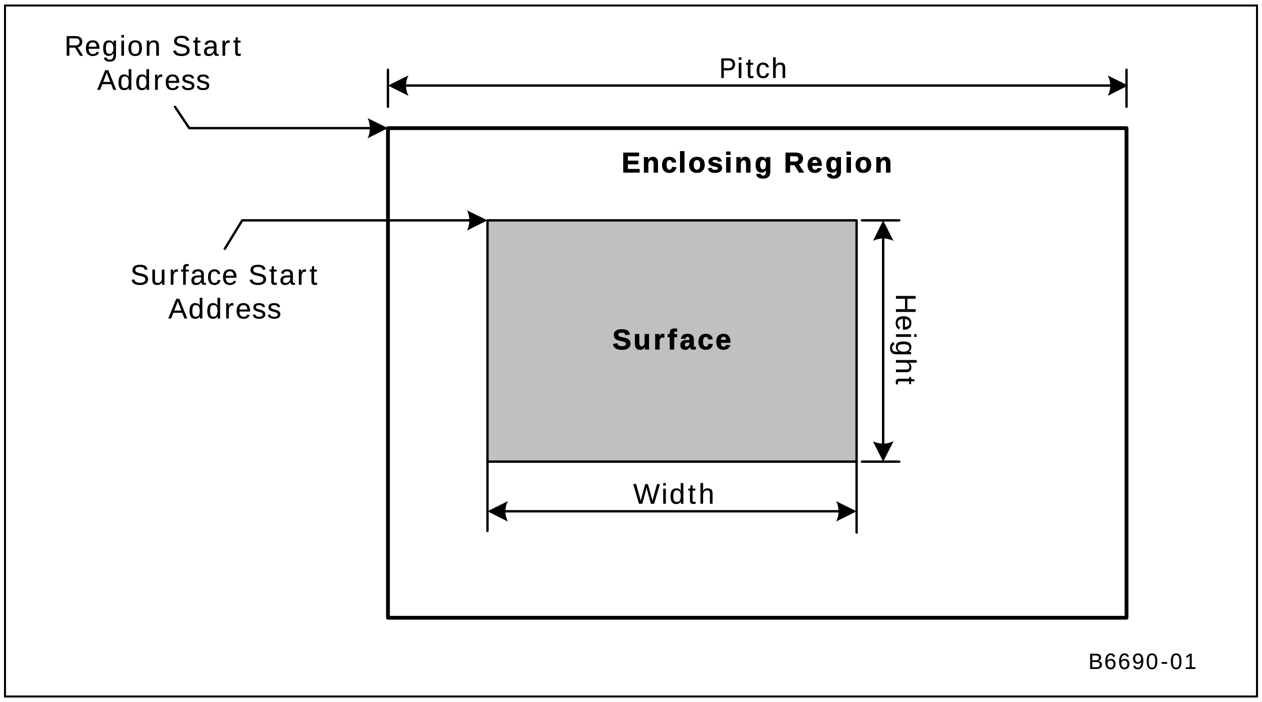
4.5 Address Tiling Function

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

4.5.1 Linear vs. Tiled Storage

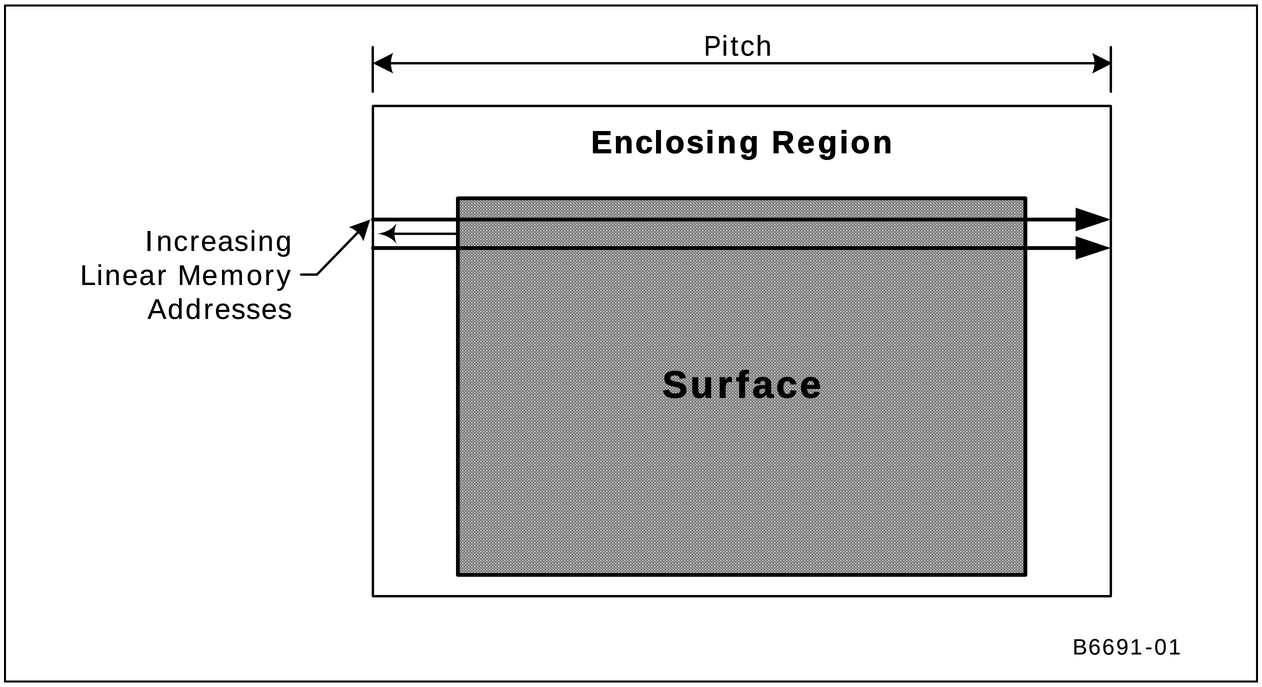
Regardless of the memory storage format, “rectangular” memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region must have widths less than or equal to the region pitch (indeed the enclosing region may coincide exactly with the surface). Figure 4-2 shows these parameters.

Figure 4-2. Rectangular Memory Operand Parameters



The simplest storage format is the *linear* format (see Figure 4-3), where each row of the operand is stored in sequentially increasing memory locations. If the surface width is less than the enclosing region's pitch, there will be additional memory storage between rows to accommodate the region's pitch. The pitch of the enclosing region determines the distance (in the memory address space) between vertically-adjacent operand elements (e.g., pixels, texels).

Figure 4-3. Linear Surface Layout

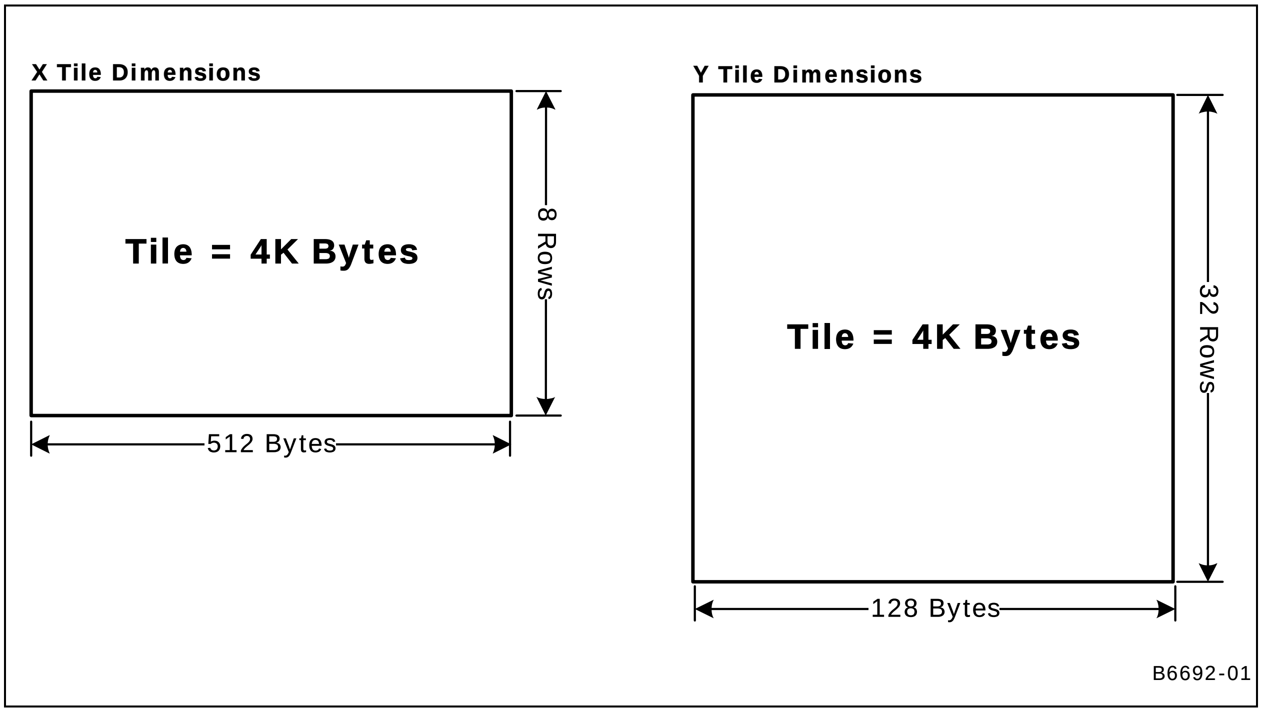


The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters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

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

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

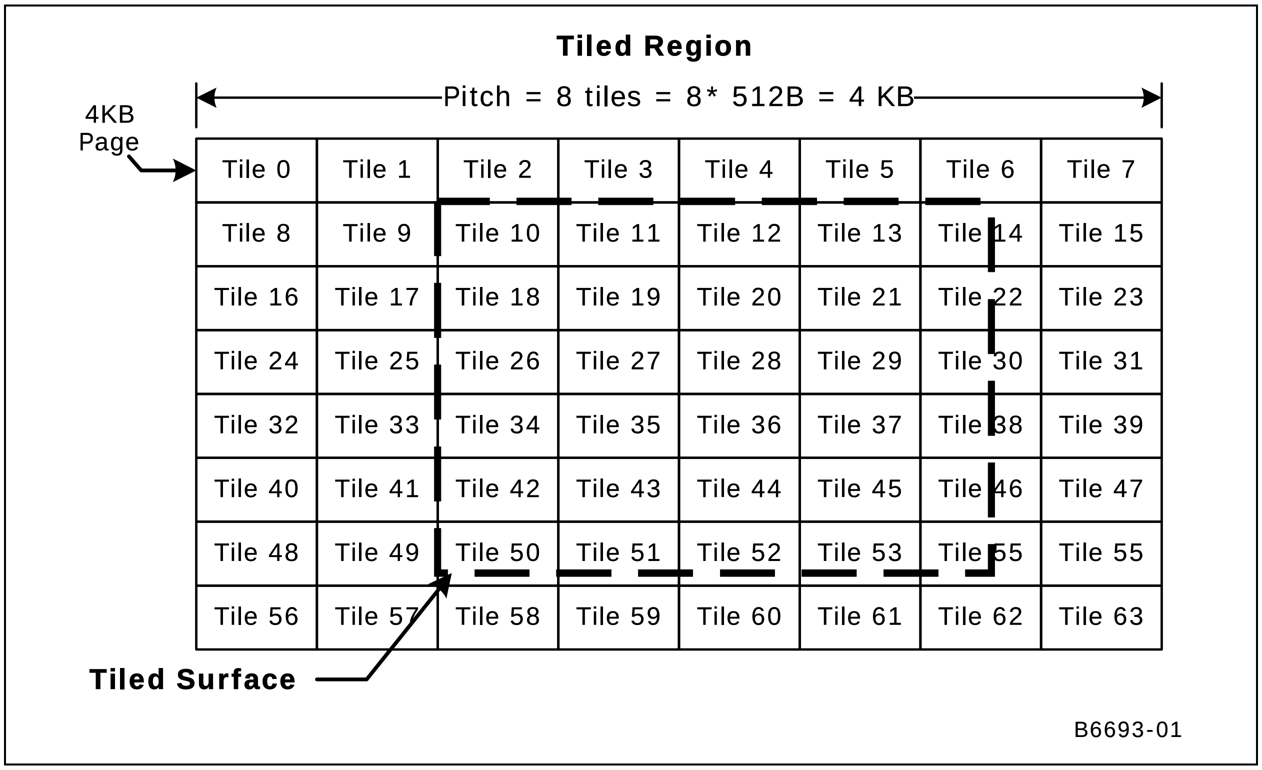
Figure 4-4. Memory Tile Dimensions



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

Figure 4-5 shows an example of a tiled surface located within a tiled region with a pitch of 8 tile widths (512 bytes * 8 = 4KB). Note that it is the enclosing region that is divided into tiles – the surface is not necessarily aligned or dimensioned to tile boundaries.

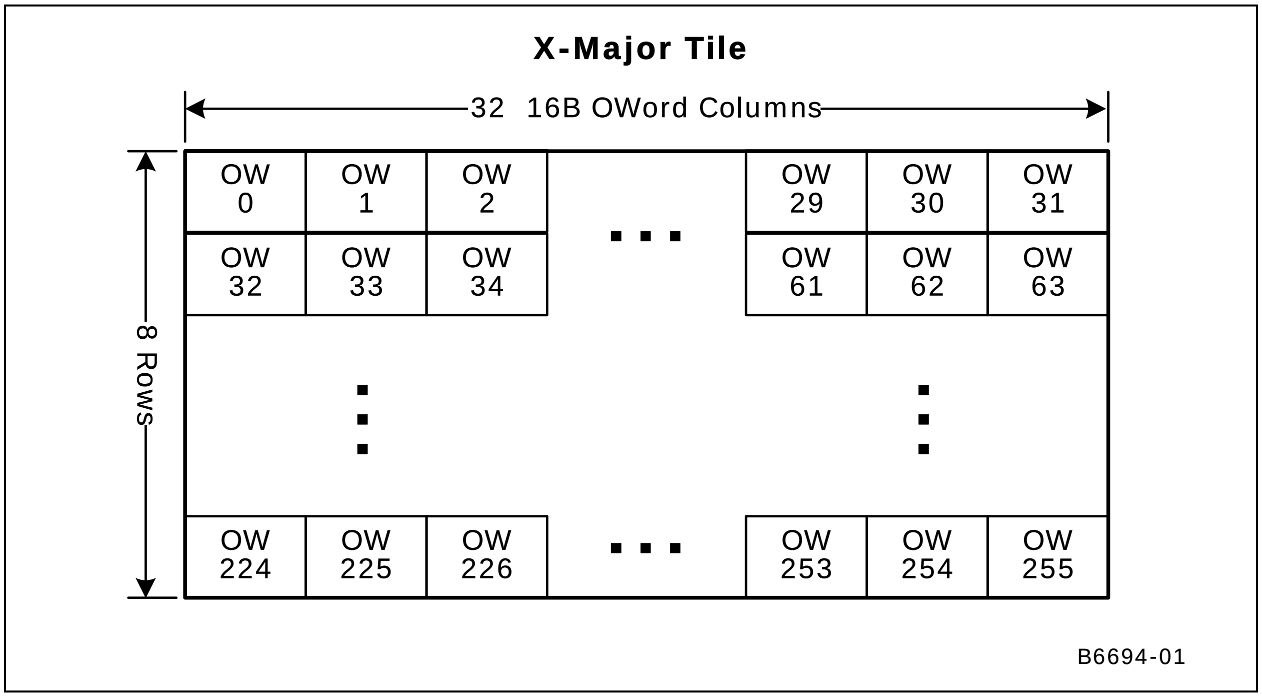
Figure 4-5. Tiled Surface Layout



4.5.2 Tile Formats

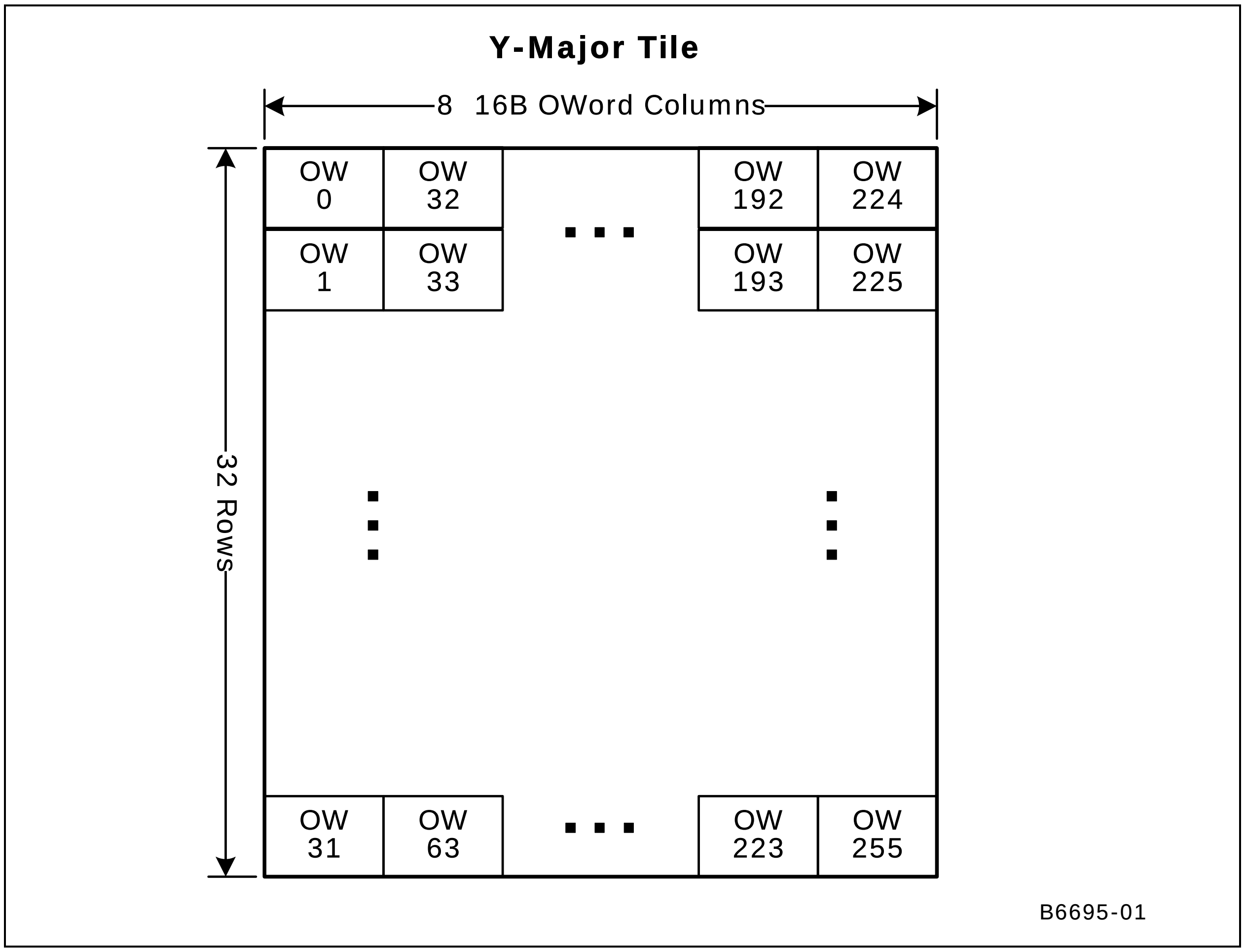
The device supports both *X-Major* (row-major) and *Y-Major* (column major) storage of tile data units, as shown in the following figures. A 4KB tile is subdivided into an 8-high by 32-wide array of 16-byte OWords for X-Major Tiles (X Tiles for short), and 32-high by 8-wide array of OWords for Y-Major Tiles (Y Tiles). The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. Note that the diagrams are not to scale – the first format defines the contents of an 8-high by 512-byte wide tile, and the 2nd a 32-high by 128-byte wide tile. The storage of tile data units in X-Major or Y-Major fashion is sometimes refer to as the *walk* of the tiling.

Table 4-3. X-Major Tile Layout



Note that an X-major tiled region with a tile pitch of 1 tile is actually stored in a linear fashion.

Figure 4-6. Y-Major Tile Layout



4.5.2.1 W-Major Tile Format

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

Figure 4-7. W-Major Tile Layout

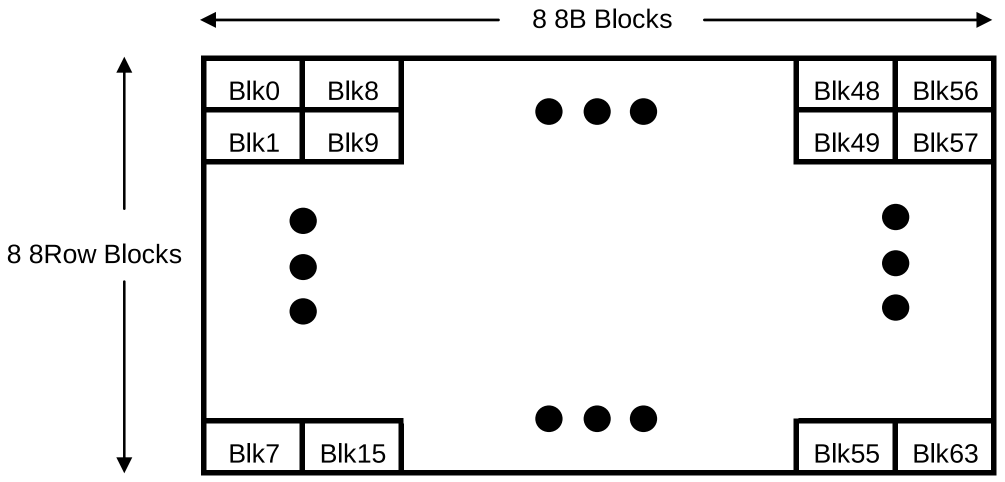
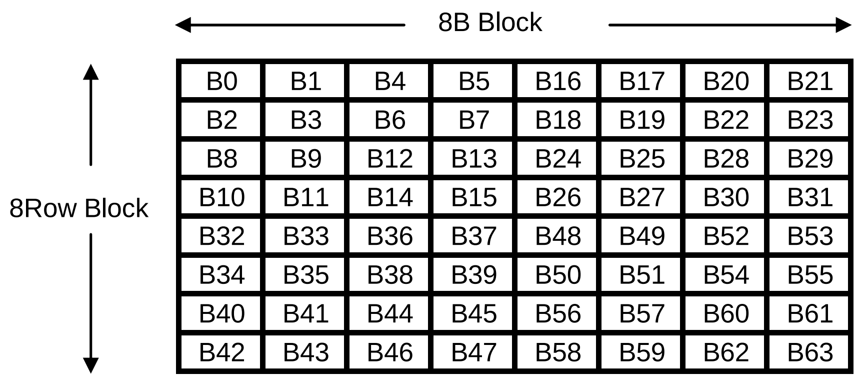


Figure 4-8. W-Major Block Layout



B0	B1	B4	B5	B16	B17	B20	B21
B2	B3	B6	B7	B18	B19	B22	B23
B8	B9	B12	B13	B24	B25	B28	B29
B10	B11	B14	B15	B26	B27	B30	B31
B32	B33	B36	B37	B48	B49	B52	B53
B34	B35	B38	B39	B50	B51	B54	B55
B40	B41	B44	B45	B56	B57	B60	B61
B42	B43	B46	B47	B58	B59	B62	B63

4.5.3 Tiling Algorithm

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

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

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


TileSize = TileH * TileW;
RowSize = Pitch * TileSize;


If (Fenced) {

      LinearAddress = LinearAddress – FenceBaseAddress

      LinearAddrInTileW = LinearAddress div TileW;

      Xoffset_inTile = LinearAddress mod TileW;

      Y = LinearAddrInTileW div Pitch;

      X = LinearAddrInTileW mod Pitch + Xoffset_inTile;

}


// Internal graphics clients that access tiled memory already have the
X, Y
// coordinates and can start here

YOff_Within_Tile = Y mod TileH;

XOff_Within_Tile = X mod TileW;


TileNumber_InY = Y div TileH;

TileNumber_InX = X div TileW;
```

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


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


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


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


TiledAddress = Tiled? (BaseAddress + TiledOffset): (BaseAddress +
            Y*LinearPitch + X);

}
```

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

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

4.5.4 Tiling Support

The rearrangement of the surface elements in memory must be accounted for in device functions operating upon tiled surfaces. (Note that not all device functions that access memory support tiled formats). This requires either the modification of an element's linear memory address or an alternate formula to convert an element's X,Y coordinates into a tiled memory address.

However, before tiled-address generation can take place, some mechanism must be used to determine whether the surface elements accessed fall in a linear or tiled region of memory, and if tiled, what the tile region pitch is, and whether the tiled region uses X-Major or Y-Major format. There are two mechanisms by which this detection takes place: (a) an implicit method by detecting that the pre-tiled (linear) address falls within a "fenced" tiled region, or (b) by an explicit specification of tiling parameters for surface operands (i.e., parameters included in surface-defining instructions).

The following table identifies the tiling-detection mechanisms that are supported by the various memory streams.

Access Path	Tiling-Detection Mechanisms Supported
Processor access through the Graphics Memory Aperture	Fenced Regions
3D Render (Color/Depth Buffer access)	Explicit Surface Parameters
Sampled Surfaces	Explicit Surface Parameters
Blt operands	Explicit Surface Parameters
Display and Overlay Surfaces	Explicit Surface Parameters

4.5.4.1 Tiled (Fenced) Regions

The only mechanism to support the access of surfaces in tiled format by the host or external graphics client is to place them within "fenced" tiled regions within Graphics Memory. A fenced region is a block of Graphics Memory specified using one of the sixteen FENCE device registers. (See *Memory Interface Registers* for details). Surfaces contained within a fenced region are considered tiled from an external access point of view. Note that fences cannot be used to untile surfaces in the PGM_Address space since external devices cannot access PGM_Address space. Even if these surfaces (or any surfaces accessed by an internal graphics client) fall within a region covered by an enabled fence register, that enable will be effectively masked during the internal graphics client access. Only the explicit surface parameters described in the next section can be used to tile surfaces being accessed by the internal graphics clients.

Each FENCE register (if its Fence Valid bit is set) defines a Graphics Memory region ranging from 4KB to the aperture size. The region is considered rectangular, with a pitch in tile widths from 1 tile width (128B or 512B) to 256 tile X widths (256 * 512B = 128KB) and 1024 tile Y widths (1024 * 128B = 128KB). Note that fenced regions must not overlap, or operation is UNDEFINED.

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

4.5.4.1.1 FENCE — Graphics Memory Fence Table Registers

FENCE — Graphics Memory Fence Table Registers

Register Type:	MMIO
Address Offset:	100000h
Project:	All
Default Value:	00000000h;
Access:	R/W
Size (in bits):	16x64
Trusted Type:	1

Address Offset: 00100000h – 001000007h: FENCE_0

:

:

00100078h – 0010007Fh: FENCE_15

The graphics device performs address translation from linear space to tiled space for a CPU access to graphics memory (See *Memory Interface Functions* chapter for information on these memory layouts) using the fence registers. Note that the fence registers are used **only for CPU accesses to gfx memory**. Graphics rendering/display pipelines use Per Surface Tiling (PST) parameters (found in SURFACE_STATE – see the *Sampling Engine* chapter) to access tiled gfx memory.

The intent of tiling is to locate graphics data that are close (in X and Y surface axes) in one physical memory page while still locating some amount of line oriented data sequentially in memory for display efficiency. All 3D rendering is done such that the QWords of any one span are all located in the same memory page, improving rendering performance. Applications view surfaces as linear, hence when the cpu access a surface that is tiled, the gfx hardware must perform linear to tiled address conversion and access the correct physical memory location(s) to get the data.

Tiled memory is supported for rendering and display surfaces located in graphics memory. A tiled memory surface is a surface that has a width and height that are subsets of the tiled region's pitch and height. The device maintains the constants required by the memory interface to perform the address translations. Each tiled region can have a different pitch and size. The CPU-memory interface needs the surface pitch and tile height to perform the address translation. It uses the GMAddr (PCI-BAR) offset address to compare with the fence start and end address, to determine if the rendering surface is tiled. The tiled address is generated based on the tile orientation determined from the matching fence register. Fence ranges are at least 4 KB aligned. Note that the fence registers are used only for CPU accesses to graphics memory.

A Tile represents 4 KB of memory. Tile height is 8 rows for X major tiles and 32 rows for Y major tiles. Tile Pitch is 512Bs for X major tiles and 128Bs for Y major tiles. The surface pitch is programmed in 128B units such that the pitch is an integer multiple of "tile pitch".

Engine restrictions on tile surface usage are detailed in Surface Placement Restrictions (Memory Interface Functions). Note that X major tiles can be used for Sampler, Color, Depth, motion compensation references and motion compensation destination, Display, Overlay, GDI Blt source and destination surfaces. Y major tiles can be used for Sampler, depth, color and motion compensation assuming they do not need to be displayed. GDI Blit operations, overlay and display cannot used Tiled Y orientations.

A "PST" graphics surface that will also be accessed via fence needs its base address to be tile row aligned.

FENCE — Graphics Memory Fence Table Registers

Hardware handles the flushing of any pending cycles when software changes the fence upper/lower bounds.

Fence Table Registers occupy the address range specified above. Each Fence Table Register has the following format.

FENCE registers are *not* reset by a graphics reset. They will maintain their values unless a full chipset reset is performed.

DWord	Bit	Description
0..15	63:44	**Fence Upper Bound** Project: All Address: GraphicsAddress[31:12] Bits 31:12 of the ending Graphics Address of the fence region. Fence regions must be aligned to a 4KB page. This address represents the last 4KB page of the fence region (Upper Bound is included in the fence region). Graphics Address is the offset within GMADR space.
	41:32	**Fence Pitch** Project: All Default Value: 0h DefaultVaueDesc Format: U10-1 Width in 128 bytes This field specifies the width (pitch) of the fence region in multiple of “tile width”. For Tile X this field must be programmed to a multiple of 512B (“003” is the minimum value) and for Tile Y this field must be programmed to a multiple of 128B (“000” is the minimum value). 000h = 128B 001h = 256B ... 3FFh = 128KB
	31:12	**Fence Lower Bound** Project: All Address: GraphicsAddress[31:12] Bits 31:12 of the starting Graphics Address of the fence region. Fence regions must be aligned to 4KB. This address represents the first 4KB page of the fence region (Lowe Bound is included in the fence region). Graphics Address is the offset within GMADR space.
	11:2	**Reserved** Project: All Format: MBZ

FENCE — Graphics Memory Fence Table Registers		
	1	**Tile Walk** Project: All Format: MI_TileWalk This field specifies the spatial ordering of QWords within tiles.
	0	**Fence Valid** Project: All Format: MI_ FenceValid This field specifies whether or not this fence register defines a fence region.

Tile Walk values:

Value	Name	Description	Project
0h	MI_TILE_XMAJOR	Consecutive SWords (32 Bytes) sequenced in the X direction	All
1h	MI_TILE_YMAJOR	Consecutive OWords (16 Bytes) sequenced in the Y direction	All

Fence Valid values:

Value	Name	Description	Project
0h	MI_FENCE_INVALID		All
1h	MI_FENCE_VALID		All

4.5.4.2 Tiled Surface Parameters

Internal device functions require explicit specification of surface tiling parameters via information passed in commands and state. This capability is provided to limit the reliance on the fixed number of fence regions.

The following table lists the surface tiling parameters that can be specified for 3D Render surfaces (Color Buffer, Depth Buffer, Textures, etc.) via SURFACE_STATE.

Surface Parameter	Description
Tiled Surface	If ENABLED, the surface is stored in a tiled format. If DISABLED, the surface is stored in a linear format.
Tile Walk	If Tiled Surface is ENABLED, this parameter specifies whether the tiled surface is stored in Y-Major or X-Major tile format.
Base Address	Additional restrictions apply to the base address of a Tiled Surface vs. that of a linear surface.
Pitch	Pitch of the surface. Note that, if the surface is tiled, this pitch must be a multiple of the tile width.

4.5.4.3 Tiled Surface Restrictions

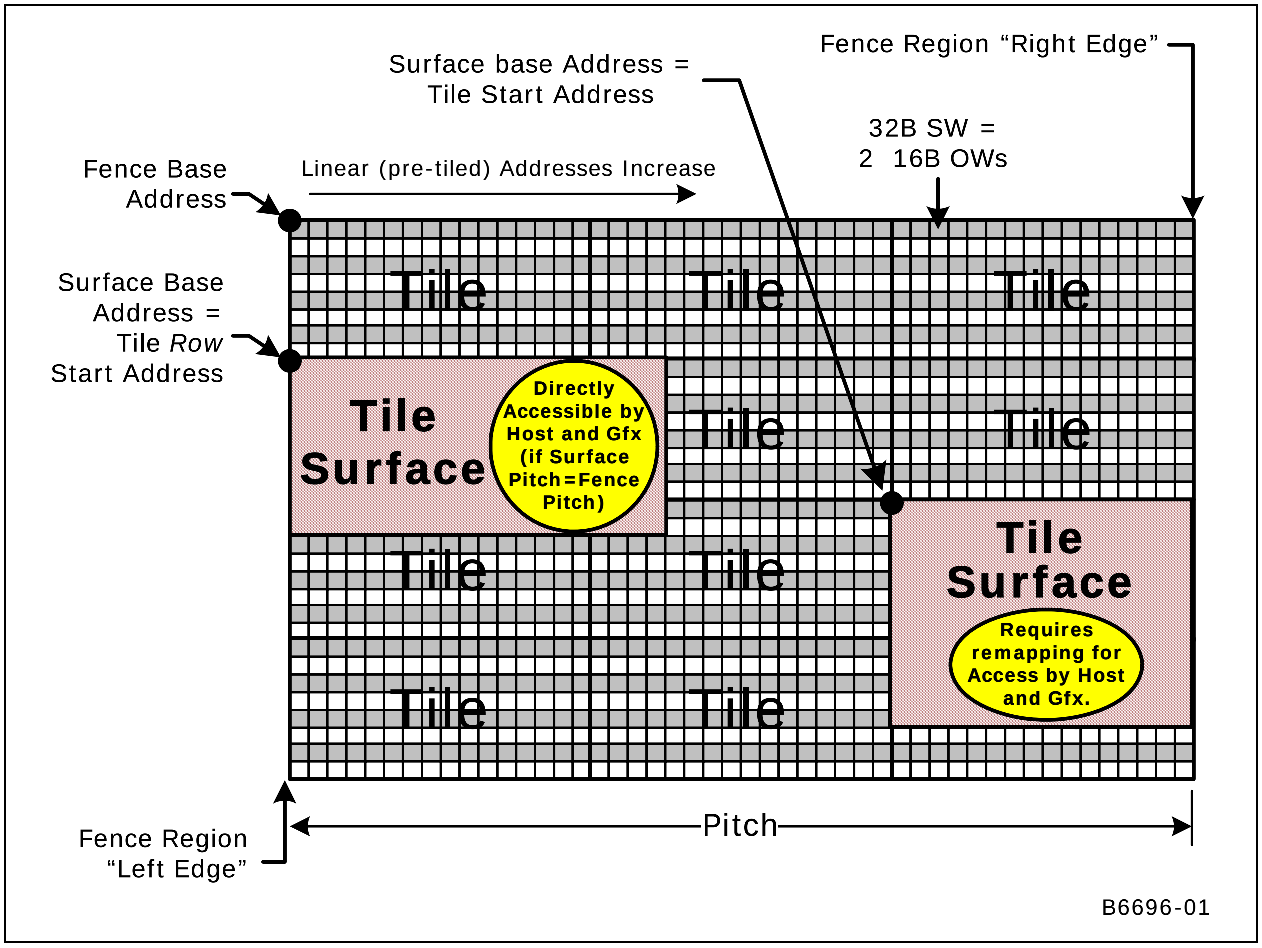
Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by internal device functions. An example of such a surface is a tiled texture that is initialized by the CPU and then sampled by the device.

The tiling algorithm for internal device functions is different from that of fence regions. Internal device functions always specify tiling in terms of a surface. The surface must have a base address, and this base address <u>is not subject to the tiling algorithm</u>. Only *offsets* from the base address (as calculated by X, Y addressing within the surface) are transformed through tiling.   The base address of the surface must therefore be 4KB-aligned. This forces the 4KB tiles of the tiling algorithm to exactly align with 4KB device pages once the tiling algorithm has been applied to the offset. The width of a surface must be less than or equal to the surface pitch. There are additional considerations for surfaces that are also accessed by the host (via a fence region).

Fence regions have no base address per se. Host linear addresses that fall in a fence region are translated in their entirety by the tiling algorithm. It is as if the surface being tiled by the fence region has a base address in graphics memory equal to the fence base address, and all accesses of the surfaces are (possibly quite large) offsets from the fence base address. Fence regions have a virtual "left edge" aligned with the fence base address, and a "right edge" that results from adding the fence pitch to the "left edge". Surfaces in the fence region must not straddle these boundaries.

Base addresses of surfaces that are to be accessed both by an internal graphics client and by the host have the tightest restrictions. In order for the surface to be accessed without GTT re-mapping, the surface base address (as set in SURFACE_STATE) must be a "Tile Row Start Address" (TRSA). The first address in each tile row of the fence region is a Tile Row Start Address. The first TRSA is the fence base address. Each TRSA can be generated by adding an integral multiple of the row size to the fence base address. The row size is simply the fence pitch in tiles multiplied by 4KB (the size of a tile.)

Figure 4-9. Tiled Surface Placement



The pitch in SURFACE_STATE must be set equal to the pitch of the fence that will be used by the host to access the surface if the same GTT mapping will be used for each access. If the pitches differ, a different GTT mapping must be used to eliminate the “extra” tiles (4KB memory pages) that exist in the excess rows at the right side of the larger pitch. Obviously no part of the surface that will be accessed can lie in pages that exist only in one mapping but not the other. The new GTT mapping can be done manually by SW between the time the host writes the surface and the device reads it, or it can be accomplished by arranging for the client to use a different GTT than the host (the PPGTT -- see Logical Memory Mapping below).

The width of the surface (as set in SURFACE_STATE) must be less than or equal to both the surface pitch and the fence pitch in any scenario where a surface will be accessed by both the host and an internal graphics client. Changing the GTT mapping will not help if this restriction is violated.

Surface Access	Base Address	Pitch	Width	Tile "Walk"
Host only	No restriction	Integral multiple of tile size <= 128KB	Must be <= Fence Pitch	No restriction
Client only	4KB-aligned	Integral multiple of tile size <= 256KB	Must be <= Surface Pitch	Restrictions imposed by the client (see Per-Stream Tile Format Support)
Host and Client, No GTT Remapping	Must be TRSA	Fence Pitch = Surface Pitch = integral multiple of tile size <= 256KB	Width <= Pitch	Surface Walk must meet client restriction, Fence Walk = Surface Walk
Host and Client, GTT Remapping	4KB-aligned for client (will be tile-aligned for host)	Both must be Integral multiple of tile size <=128KB, but not necessarily the same	Width <= Min(Surface Pitch, Fence Pitch)	Surface Walk must meet client restriction, Fence Walk = Surface Walk

4.5.5 Per-Stream Tile Format Support

MI Client	Tile Formats Supported
CPU Read/Write	All
Display/Overlay	Y-Major not supported. X-Major required for Async Flips
Blt	Linear and X-Major only No Y-Major support
3D Sampler	All Combinations of TileY, TileX and Linear are supported. TileY is the fastest, Linear is the slowest.
3D Color,Depth	

Rendering Mode Color-vs-Depth bpp	Buffer Tiling Supported
Classical Same Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY
Classical Mixed Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY

NOTE: 128BPE Format Color buffer (render target) MUST be either TileX or Linear.

4.6 Logical Memory Mapping

In order to provide a contiguous address space for graphics operands (surfaces, etc.) yet allow this address space to be mapped onto possibly discontiguous physical memory pages, the internal graphics device supports a Logical Memory Space (see Figure 4-12). A global *Graphics Translation Table* (GTT) is provided to map zero-based (and post-tiled) Logical Memory Addresses into a set of 4KB physical memory pages. (This mapping is also used for external PEG devices.)

There is another logical mapping function available local to each graphics process; this works identically to the global GTT with some additional restrictions. It is providing each graphics context with its own local translation table and protected memory space (see Rendering Context Management later in this chapter).

The GTT and PPGTT are arrays of 4-byte *Page Table Entries* (PTEs) physically located in Main Memory. The GTT and PPGTT are comprised of a number of locked (non-swappable) physically-contiguous 4KB memory pages, with a maximum size (each) of 128 4KB pages (128K DWords map 128K*4KB = 512MB max) for Global GTT, and up to 512 4KB pages for PPGTT, for total up to 2GB max. GTT and PPGTT base addresses must be 4KB-aligned.

Note that the PTEs within the global GTT must be written only through GTTADDR (see the Device #2 Config registers for a description of this range), the invalidation of hardware TLBs need to be done explicitly with a fence prior to enabling the new surface using updated PTEs. [Reserved content]

The PPGTT base address is also 4KB aligned, but it is programmed directly in physical memory space rather than through an alias mechanism like GTTADDR. Note that not all clients may use the PPGTT; only the global GTT is available for processor accesses as well as graphics accesses from display engines (including overlay and cursor). Any per-process access that occurs while the PPGTT is disabled will default to a translation via the global GTT.

4.6.1 Changes to GTT

The GTT is constrained to be located at the beginning of a special section of stolen memory called the GTT stolen memory (GSM). There is no longer an MMIO register containing the physical base address of the GTT as on prior devices. Instead of using the PGTBL_CTL register to specify the base address of the GTT, the GTT base is now defined to be at the bottom (offset 0) of GSM. See the Device Cspecs for information on the location of GSM.

Since the graphics device (including the driver) knows nothing about the location of GSM, it does not “know” where the GTT is located in memory. In fact, the CPU cannot directly access the GSM containing the GTT. GTT entries therefore cannot be updated directly in memory as was possible on prior devices. Instead, the driver must use GTTADR as it could before, or it must make use of a new command (MI_UPDATE_GTT) to perform command-buffer-synchronized updates to the GTT.

Similarly, the Per-process GTT (PPGTT) is also constrained to be located within the GSM. The PPGTT base is specified by an offset in the PGTBL_CTL2 register. The PPGTT is treated as an extension of the GTT; it must be updated using either GTTADDR or the MI_UPDATE_GTT command.

The use by graphics of “physical” addressing can no longer be supported under VT-d. Therefore all graphics commands and registers that could, on prior devices, contain physical addresses must now use graphics memory (virtual) addresses. Functions that still require physical addressing are prohibited when VT-d is active. An example is hardware binning which requires physically addressed batch buffers. The hardware binner can only be used when VT-d is not active. Frame Buffer Compression (FBC) also requires physical addressing. FBC is not supported on Bearlake-C since it is a mobile feature and Bearlake-C is a desktop chipset.

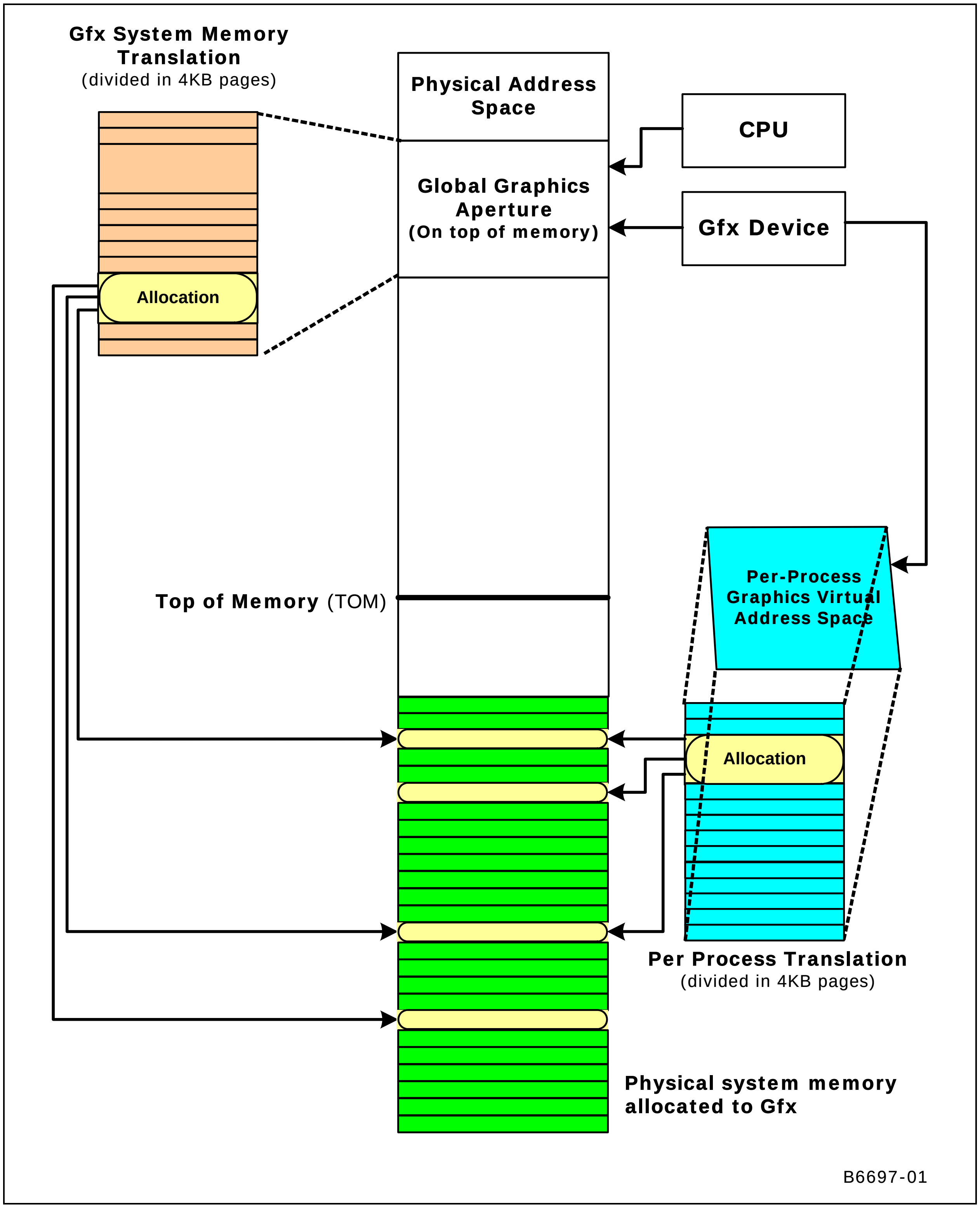
4.6.2 Logical Memory Space Mappings

Each valid PTE maps a 4KB page of Logical Memory to an independent 4KB page of:

- MM: Main Memory (unsnooped), or
- SM: System Memory (snooped, therefore coherent with the processor cache, must not be accessed through the Dev2_GM_Address range by the CPU)

PTEs marked as invalid have no backing physical memory, and therefore the corresponding Logical Memory Address pages must not be accessed in normal operation.

Figure 4-10. Global and Render GTT Mapping



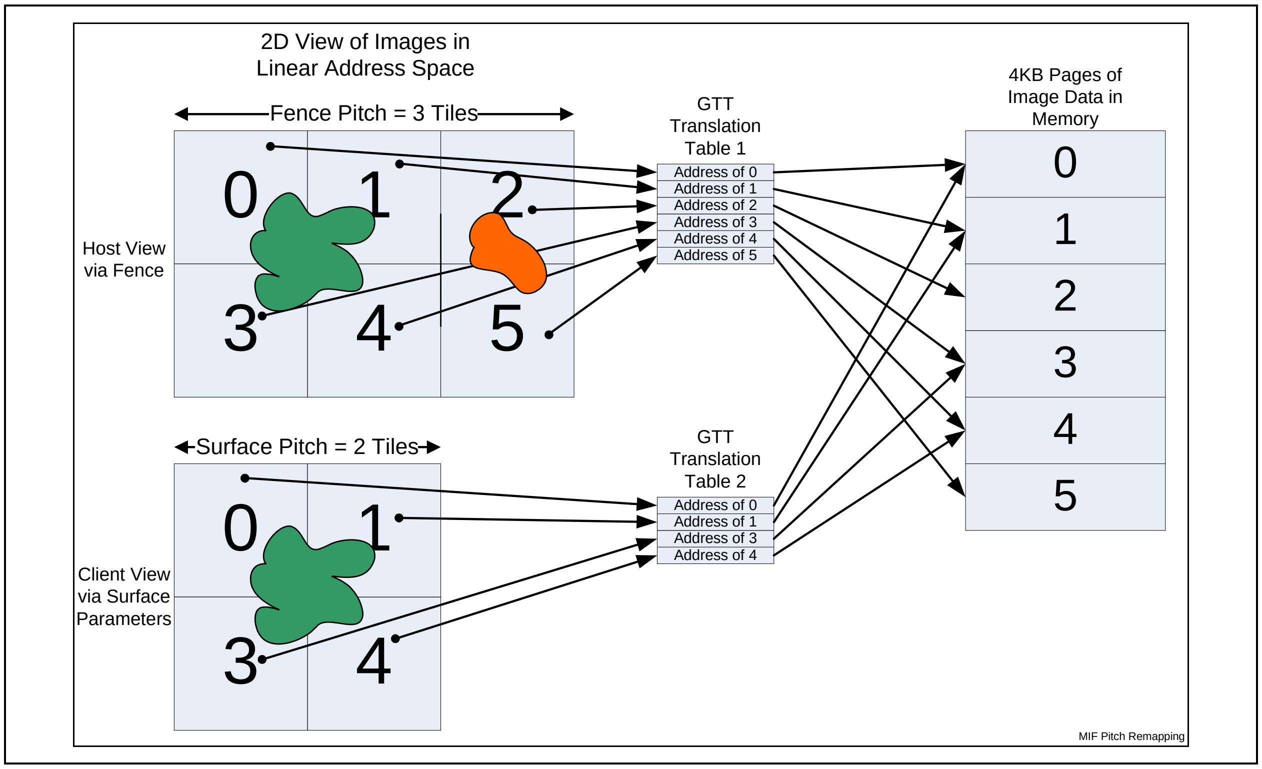
The following table lists the memory space mappings valid for each MI client:

MI Client	**Logical Memory Space Mappings Supported**	**xGTT Usage**
External Clients		
Host Processor	MM	GTT only
External PEG Device	None	n/a
Snooped Read/Write	None	n/a
Internal GPU Clients		
Render Command Ring Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<2>
Render Command Batch Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<5>
Indirect State Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<4>
CURBE Constant Data	MM	Same xGTT used to fetch the CONSTANT_BUFFER command.
Media Object Indirect Data	MM	Same xGTT used to fetch the MEDIA_OBJECT command.
Vertex Fetch Data	MM, SM	GTT/PGTT, selected by PGTBL_STR2<3>
Sampler Cache (RO)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<1>
DataPort Render Cache (R/W)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
Depth Buffer Cache (R/W)	MM	GTT/PGTT, selected by PGTBL_STR2<0>
Blit Engine	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
MI_STORE_DATA_IMM Destination (if virtual addressed)	MM, SM	Same xGTT used to fetch the command.
PIPE_CONTROL Write Destination	MM, SM	GTT/PGTT, selected by the command
Display/Overlay Engines (internal)	MM	GTT only

Usage Note: Since the CPU cannot directly access memory pages mapped through a Graphics Process' local GTT (PPGTT), these pages must also be mapped though the global GTT (at least temporarily) in order for the CPU to initialize graphics data for a Graphics Process.

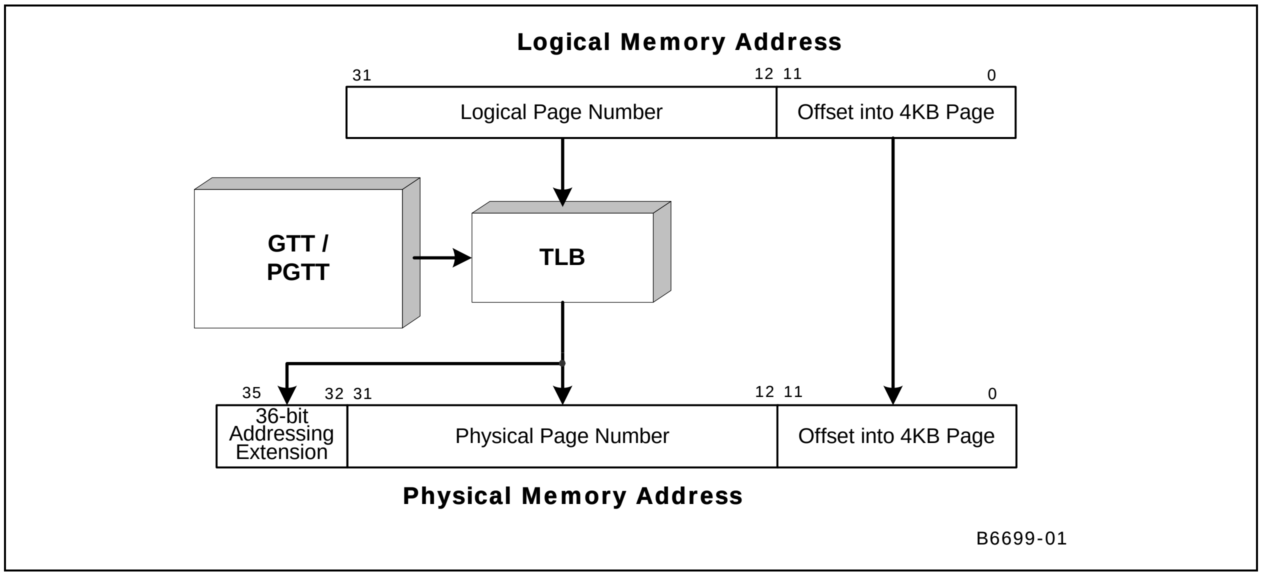
The PPGTT mechanism can be used by a client to access a surface with a pitch that is smaller than that of the fence region used by the host to initialize the surface, without having to physically move the data in memory.

Figure 4-11. GTT Re-mapping to Handle Differing Pitches



Refer to the "Graphics Translation Table (GTT) Range (GTTADR) & PTE Description" in *Memory Interface Registers* for details on PTE formats and programming information. Refer to the *Memory Data Formats* chapter for device-specific details/restrictions regarding the placement/storage of the various data objects used by the graphics device.

Figure 4-12. Logical-to-Physical Graphics Memory Mapping

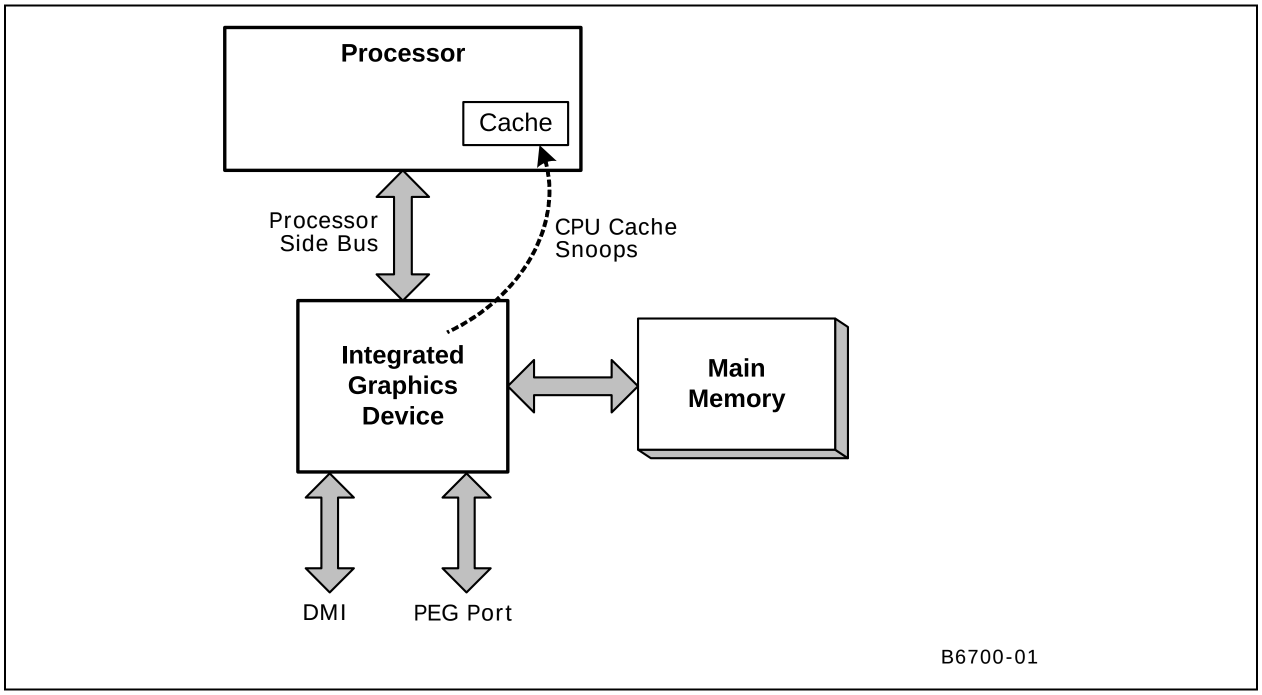


4.7 Physical Graphics Memory

The integrated graphics device satisfies all of its memory requirements using portions of main system memory. The integrated graphics device operates without any dedicated local memory, in a lower-cost configuration typically (though not necessarily officially) known as *Unified Graphics Memory* (UMA).

Figure 4-13 shows how the Main Memory is interfaced to the device.

Figure 4-13. Memory Interfaces



4.7.1 Physical Graphics Address Types

Table 4-4 lists the various physical address types supported by the integrated graphics device. Physical Graphics Addresses are either generated by Logical Memory mappings or are directly specified by graphics device functions. These physical addresses are not subject to tiling or GTT re-mappings.

Table 4-4. Physical Memory Address Types

Address Type	Description	Range
MM_Address	Main Memory Address. Offset into physical, unsnooped Main Memory.	[0,TopOfMemory-1]
SM_Address	System Memory Address. Accesses are snooped in processor cache, allowing shared graphics/ processor access to (locked) cacheable memory data.	[0,4GB]

4.7.2 Main Memory

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

4.7.2.1 Optimizing Main Memory Allocation

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

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

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

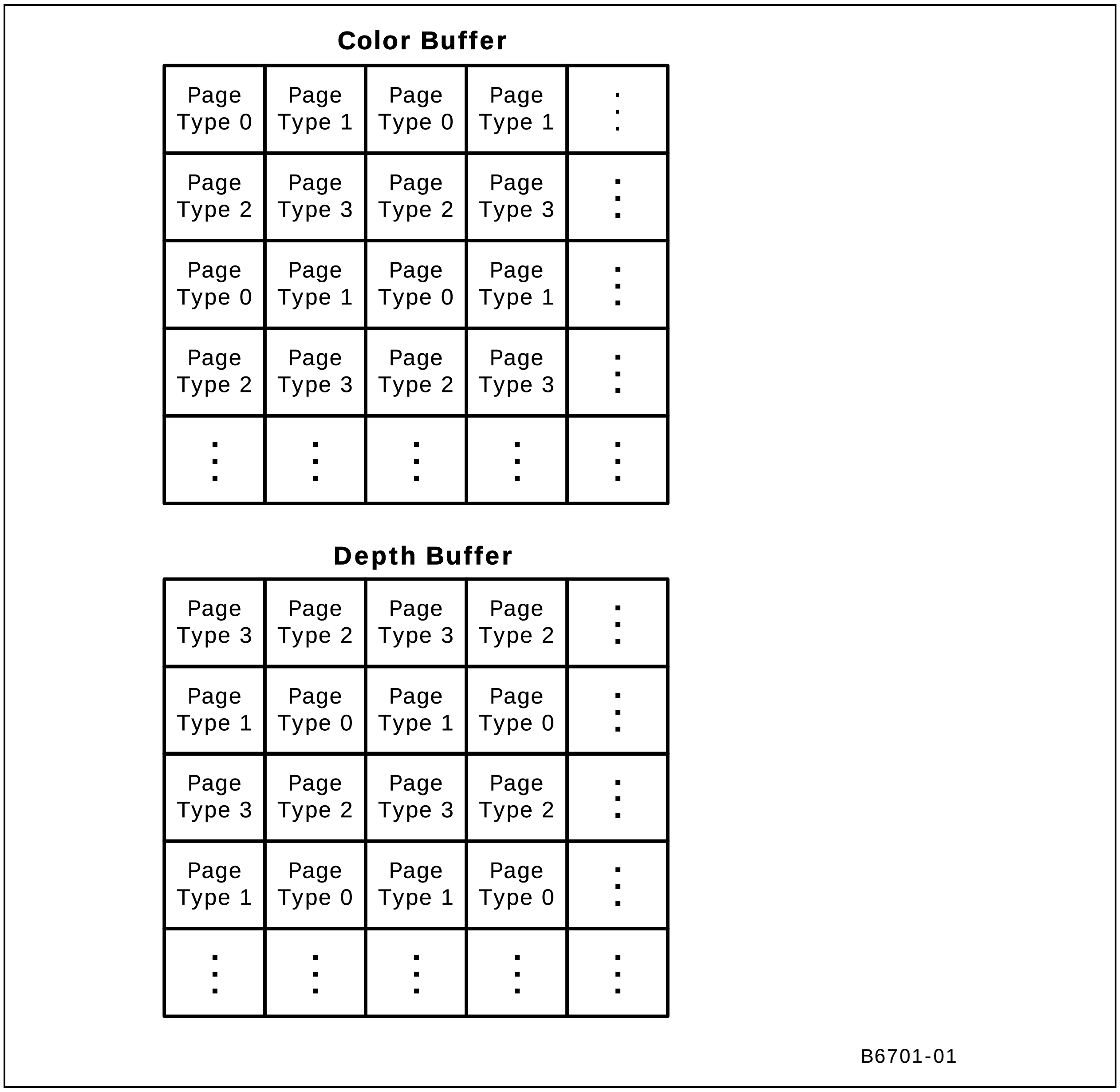
4.7.2.2 Application of the Theory (Page Coloring)

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

4.7.2.2.1 3D Color and Depth Buffers

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

Figure 4-14. Memory Pages Backing Color and Depth Buffers



Color Buffer

Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
⋮	⋮	⋮	⋮	⋮

Depth Buffer

Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
⋮	⋮	⋮	⋮	⋮

For higher performance, the Color and Depth Buffers could be allocated from <u>different</u> memory device rows.

4.7.2.2.2 Media/Video

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

5. Device Programming Environment

The graphics device contains an extensive set of registers and commands (also referred to as "commands" or "packets") for controlling 2D, 3D, video I/O, and other operations. This chapter describes the programming environment and software interface to these registers/commands. The registers and commands themselves are described elsewhere in this document.

5.1 Programming Model

The graphics device is programmed via the following three basic mechanisms:

POST-Time Programming of Configuration Registers

These registers are the graphics device registers residing in PCI space. A majority of these registers are programmed once during POST of the video device. Configuration registers are not covered in this section.

Direct (Physical I/O and/or Memory-Mapped I/O) Access of Graphics Registers

Various graphics functions can only be controlled via direct register access. In addition, direct register access is required to initiate the (asynchronous) execution of graphics command streams. This programming mechanism is "direct" and synchronous with software execution on the CPU.

Command Stream DMA (via the Command Ring Buffer and Batch Buffers)

This programming mechanism utilizes the indirect and asynchronous execution of graphics command streams to control certain graphics functions, e.g., all 2D, 3D drawing operations. Software writes commands into a command buffer (either a Ring Buffer or Batch Buffer) and informs the graphics device (using the Direct method above) that the commands are ready for execution. The graphics device's Command Parser (CP) will then, or at some point in the future, read the commands from the buffer via DMA and execute them.

5.2 Graphics Device Register Programming

The graphics device registers (except for the Configuration registers) are memory mapped. The base address of this 512 KB memory block is programmed in the MMADR Configuration register. For a detailed description of the register map and register categories, refer to the *Register Maps* chapter.

Programming Note:

- Software must only access GR06, MSR0, MSR1, and Paging registers (see *Register Maps*) via Physical I/O, never via Memory Mapped I/O.

5.3 Graphics Device Command Streams

This section describes how command streams can be used to initiate and control graphics device operations.

5.3.1 Command Use

Memory-resident commands are used to control drawing engines and other graphics device functional units:

- **Memory Interface (MI) Commands**. The MI commands can be used to control and synchronize the command stream as well as perform various auxiliary functions (e.g., perform display/overlay flips, etc.)
- **2D Commands (BLT).** These commands are used to perform various 2D (Blt) operations.
- **3D Commands.** 3D commands are used to program the 3D pipeline state and perform 3D rendering operations. There are also a number of 3D commands that can be used to accelerate 2D and video operations, e.g., “StretchBlt” operations, 2D line drawing, etc.
- **Video (MPEG, etc.) Decode Commands**. A set of commands are supported to perform video decode acceleration including Motion Compensation operations via the Sampling Engine of the 3D pipeline.

5.3.2 Command Transport Overview

Commands are not written directly to the graphics device – instead they are placed in memory by software and later read via DMA by the graphics device’s Command Parser (CP) within the Memory Interface function. The primary mechanism used to transport commands is through the use of a Ring Buffer.

An additional, indirect mechanism for command transport is through the use of Batch Buffers initiated from the Ring buffer.

The Command Parser uses a set of rules to determine the order in which commands are executed. Following sections in this chapter provide descriptions of the Ring Buffer, Batch Buffers, and Command Parser arbitration rules.

5.3.3 Command Parser

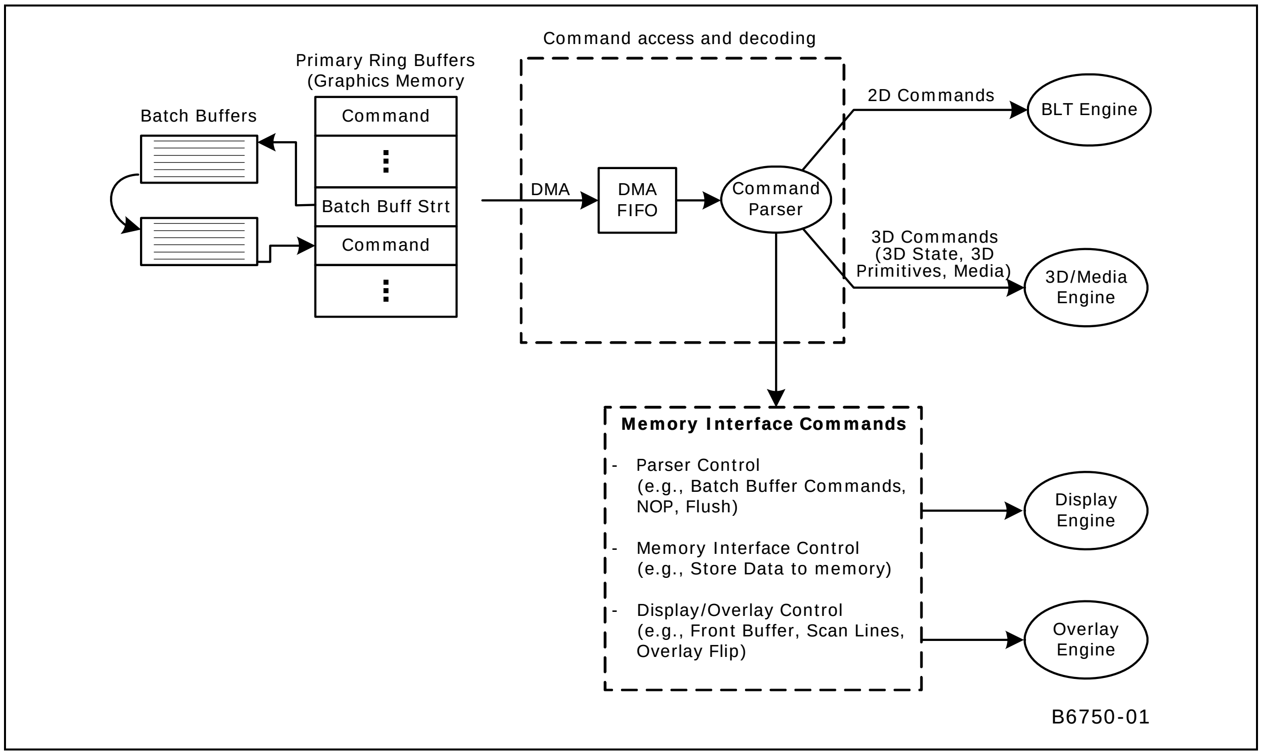
The graphics device’s Command Parser (CP) is responsible for:

- Detecting the presence of commands (within the Ring Buffer).
- Reading commands from the Ring Buffer and Batch Buffers via DMA. This includes support of the automatic head report function.
- Parsing the common "Command Type" (destination) field of commands.

- Execution of Memory Interface commands that control CP functionality, provide synchronization functions, and provide display and overlay flips as well as other miscellaneous control functions.
- Redirection of 2D, 3D and Media commands to the appropriate destination (as qualified by the INSTPM register) while enforcing drawing engine concurrency and coherency rules.
- Performing the “Sync Flush” mechanism
- Enforcing the Batch Buffer protection mechanism

Figure 5-1 is a high-level diagram of the graphics device command interface.

Figure 5-1. Graphics Controller Command Interface

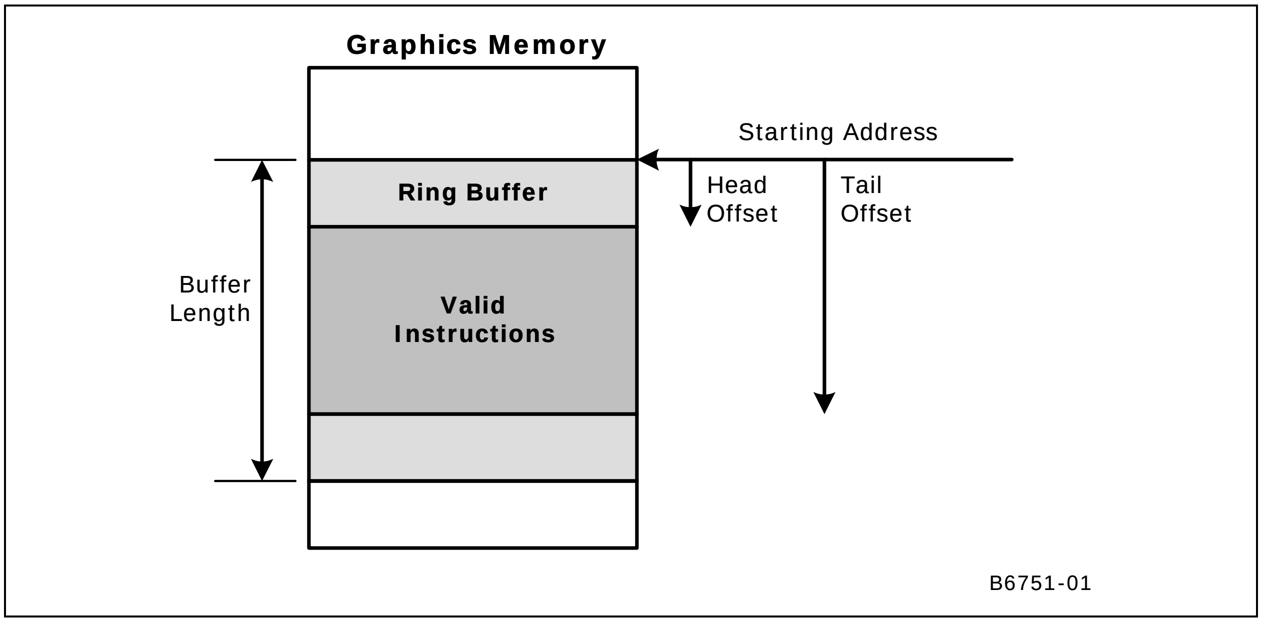


Command access and decoding
Primary Ring Buffers (Graphics Memory
Batch Buffers
Command
Batch Buff Strt
Command
DMA
DMA FIFO
Command Parser
2D Commands
BLT Engine
3D Commands (3D State, 3D Primitives, Media)
3D/Media Engine
Memory Interface Commands
- Parser Control (e.g., Batch Buffer Commands, NOP, Flush)
- Memory Interface Control (e.g., Store Data to memory)
- Display/Overlay Control (e.g., Front Buffer, Scan Lines, Overlay Flip)
Display Engine
Overlay Engine
B6750-01

5.3.4 The Ring Buffer

The ring buffesr are defined by a set of Ring Buffer registers and a memory area that is used to hold the actual commands. The Ring Buffer registers (described in full below) define the start and length of the memory area, and include two “offsets” (head and tail) into the memory area. Software uses the Tail Offset to inform the CP of the presence of valid commands that must be executed. The Head Offset is incremented by the CP as those commands are parsed and executed. The list of commands can wrap from the bottom of the buffer back to the top. Also included in the Ring Buffer registers are control fields that enable the ring and allow the head pointer to be reported to cacheable memory for more efficient flow control algorithms.

Figure 5-2. Ring Buffer



5.3.4.1 The Ring Buffer (RB)

Ring Buffer support:

- Batch Buffer initiation
- Indirect Data (operand access)

5.3.4.2 Ring Buffer Registers

A Ring Buffer is defined by a set of 4 Ring Buffer registers. Before a Ring Buffer can be used for command transport, software needs to program these registers. The fields contained within these registers are as follows:

- **Ring Buffer Valid:** This bit controls whether the Ring Buffer is included in the command arbitration process. Software must program all other Ring Buffer parameters before enabling a Ring Buffer. Although a Ring Buffer can be enabled in the non-empty state, it must not be disabled unless it is empty. Attempting to disable a Ring Buffer in the non-empty state is UNDEFINED. Enabling or disabling a Ring Buffer does not of itself change any other Ring Buffer register fields.
- **Start Address:** This field points to a contiguous, 4KB-aligned, linear (i.e., must not be tiled), mapped graphics memory region which provides the actual command buffer area.
- **Buffer Length:** The size of the buffer, in 4KB increments, up to 2MB.
- **Head Offset:** This is the DWord offset (from Start Address) of the next command that the CP will parse (i.e., it points one DWord past the last command parsed). The CP will update this field as commands are parsed – the CP typically continues parsing new commands before the previous

command operations complete. (Note that, if commands are pending execution, the CP will likely have prefetched commands past the Head Offset). As the graphics device does not "reset" the Head Offset when a Ring Buffer is enabled, software must program the Head Offset field before enabling the Ring Buffer. Software can enable a Ring Buffer with any legal values for Head/Tail (i.e., can enable the Ring Buffer in an non-empty state). It is anticipated, but not required, that software enable The Ring Buffer with Head and Tail Offsets of 0. Once the Head Offset reaches the QWord specified by the Tail Offset (i.e., the offsets are equal), the CP considers the Ring Buffer "empty".

- **Head Wrap Count:** This field is incremented by the CP every time the Head Offset wraps back to the start of the buffer. As it is included in the DWord written in the "report head" process, software can use this field to track CP progress as if the Ring Buffer had a "virtual" length of 2048 times the size of the actual physical buffer (up to 4GB).

- **Tail Offset:** This is the offset (from Start Address) of the next QWord of command data that software will request to be executed (i.e., it points one DWord past the last command DWord submitted for execution). The Tail Offset can only point to an command boundary – submitting partial commands is UNDEFINED. As the Tail Offset is a QWord offset, this requires software to submit commands in multiples of QWords (both DWords of the last QWord submitted must contain valid command data).   Software may therefore need to insert a “pad” command to meet this restriction. After writing commands into the Ring Buffer, software updates the Tail Offset field in order to submit the commands for execution (by setting it to the QWord offset past the last command). The commands submitted can wrap from the end of the buffer back to the top, in which case the Tail Offset written will be less than the previous value. As the “empty” condition is defined as “Head Offset == Tail Offset”, the largest amount of data that can be submitted at any one time is one QWord less than the Ring Buffer length.

- **IN USE Semaphore Bit:** This bit (included in the Tail Pointer register) is used to provide a HW semaphore that SW can use to manage access to the individual The Ring Buffer.   See the Ring Buffer Semaphore section below.

- **Automatic Report Head Enable:** Software can request to have the hardware Head Pointer register contents written ("reported") to snooped system memory on a periodic basis. Auto-reports can be programmed to occur whenever the Head Offset crosses either a 64KB or 128KB boundary. (Note therefore that a Ring Buffer must be at least 64KB in length for the auto-report mechanism to be useful). The complete Head Pointer register will be stored at a Ring Buffer-specific DWord offset into the "hardware status page" (defined by the HWSTAM register). The auto-report mechanism is desirable as software needs to use the Head Offset to determine the amount of free space in the Ring Buffer -- and having the Head Pointer periodically reported to system memory provides a fairly up-to-date Head Offset value automatically (i.e., without having to explicitly store a Head Pointer value via the MI_REPORT_HEAD command).

Table 5-1. Ring Buffer Characteristics

Characteristic	Description
Alignment	4 KB page aligned.
Max Size	2 MB
Length	Programmable in numbers of 4 KB pages.
Start Pointer	Programmable 4KB page-aligned address of the buffer
Head pointer	Hardware maintained DWord Offset into the ring buffer. Commands can wrap. Programmable to initially set up ring.
Tail pointer	Programmable QWord Offset into the ring buffer – indicating the *next* QWord where software can insert new commands.

5.3.4.3 Ring Buffer Placement

Ring Buffer memory buffers are defined via a Graphics Address and must physically reside in (uncached) Main Memory. There is no support for The Ring Buffer in cacheable system memory.

5.3.4.4 Ring Buffer Initialization

Before initializing a Ring Buffer, software must first allocate the desired number of 4KB pages for use as buffer space. Then the Ring Buffer registers associated with the Ring Buffer can be programmed. Once the Ring Buffer Valid bit is set, the Ring Buffer will be considered for command arbitration, and the Head and Tail Offsets will either indicate an empty Ring Buffer (i.e., Head Offset == Tail Offset), or will define some amount of command data to be executed.

5.3.4.5 Ring Buffer Use

Software can write new commands into the "free space" of the Ring Buffer, starting at the Tail Offset QWord and up to the QWord prior to the QWord indicated by the Head Offset. Note that this "free space" may wrap from the end of the Ring Buffer back to the start (hence the “ring” in the name).

While the “free space” wrap may allow commands to be wrapped around the end of the Ring Buffer, the wrap should only occur between commands. Padding (with NOP) may be required to follow this restriction.

Software is required to use some mechanism to track command parsing progress in order to determine the "free space" in the Ring Buffer. This can be accomplished in one of the following ways:

1. A direct read (poll) of the Head Pointer register. This gives the most accurate indication but is expensive due to the uncached read.

2. The automatic reporting of the Head Pointer register in the Hardware Status Page. This has low impact as no uncached reads or command overhead is involved. However, given the 64KB/128KB granularity of auto-reports, this mechanism only works well on fairly large The Ring Buffer.

3. The explicit reporting of the Head Pointer register via the MI_REPORT_HEAD command. This allows for flexible and more accurate reporting but comes at the cost of command bandwidth and execution time, in addition to the software overhead to determine how often to report the head.

4. Some other "implicit" means by which software can determine how far the CP has progressed in retiring commands from a Ring Buffer. This could include the use of "Store DWORD" commands to write sequencing data to system memory. This has similar characteristics to using the MI_REPORT_HEAD mechanism.

Once the commands have been written and, if necessary, padded out to a QWord, software can write the Tail Pointer register to submit the new commands for execution. The uncached write of the Tail Pointer register will ensure that any pending command writes are flushed from the processor.

If the Ring Buffer Head Pointer and the Tail Pointer are on the same cacheline, the Head Pointer must not be greater than the Tail Pointer.

5.3.4.6 Ring Buffer Semaphore

When the **Ring Buffer Mutex Enable** (RBME) bit if the INSTPM MI register is clear, all Tail Pointer IN USE bits are disabled (read as zero, writes ignored). When RBME is enabled, the IN USE bit acts as a Ring Buffer semaphore. If the Tail Pointer is read, and IN USE is clear, it is immediately set after the read. Subsequent Tail Pointer reads will return a set IN USE bit, until IN USE is cleared by a Tail Pointer write.

This allows SW to maintain exclusive ring access through the following protocol: A SW agent needing exclusive ring access must read the Tail Pointer before accessing the Ring Buffer: if the IN USE bit is clear, the agent gains access to the Ring Buffer; if the IN USE bit is set, the agent has to wait for access to the Ring Buffer (as some other agent has control). The mechanism to inform pending agents upon release of the IN USE semaphore is unspecified (i.e., left up to software).

5.3.5 Batch Buffers

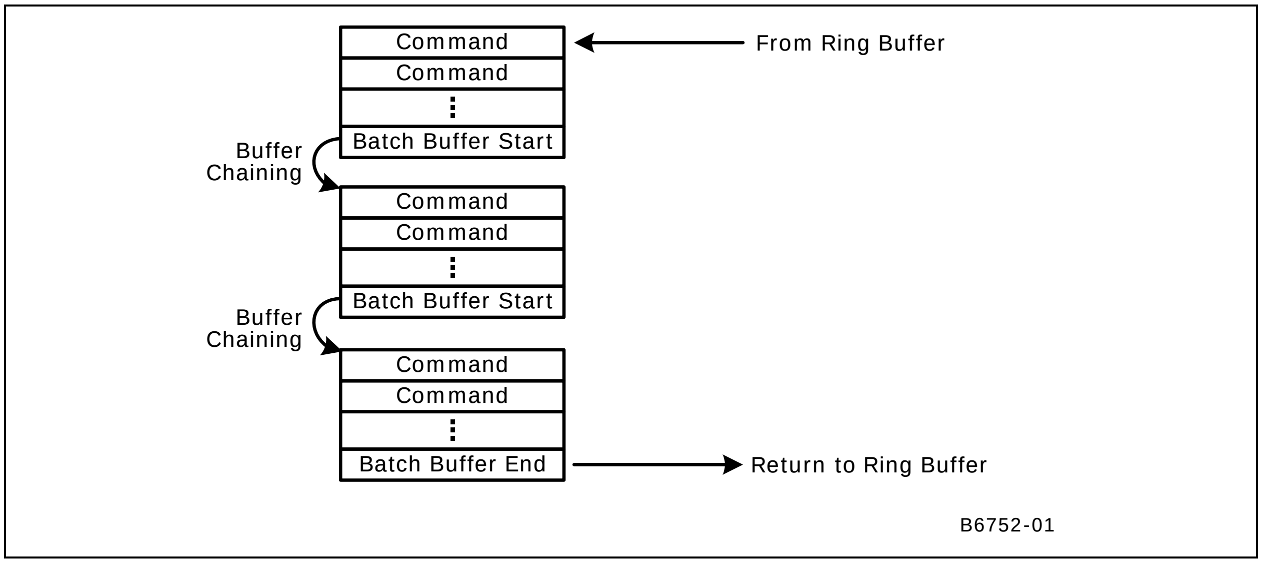
The graphics device provides for the execution of command sequences *external* to the Ring buffer. These sequences are called "Batch Buffers", and are initiated through the use of various Batch Buffer commands described below. When a Batch Buffer command is executed, a batch buffer sequence is initiated, where the graphics device fetches and executes the commands sequentially via DMA from the batch buffer memory.

5.3.5.1 Batch Buffer Chaining

What happens when the end of the Batch Buffer is reached depends on the final command in the buffer. Normally, when a Batch Buffer is initiated from a Ring Buffer, the completion of the Batch Buffer will cause control to pass back to the Ring Buffer at the command following the initiating Batch Buffer command.

However, the final command of a Batch Buffer can be another Batch Buffer-initiating command (MI_BATCH_BUFFER_START). In this case control will pass to the new Batch Buffer. This process, called *chaining*, can continue indefinitely, terminating with a Batch Buffer that does not chain to another Batch Buffer (ends with MI_BATCH_BUFFER_END) – at which point control will return to the Ring Buffer.

Figure 5-3. Batch Buffer Chaining



5.3.5.2 Ending Batch Buffers

The end of the Batch Buffer is determined as the buffer is being executed: either by (a) an MI_BATCH_BUFFER_END command, or (b) a "chaining" MI_BATCH_BUFFER_START command. There is no explicit limit on the size of a Batch Buffer that uses GTT-mapped memory. Batch buffers in physical space cannot exceed one physical page (4KB).

5.3.6 Indirect Data

In addition to Ring Buffer and Batch Buffers, the MI supports the access of *indirect* data for some specific command types. (Normal read/write access to surfaces isn't considered indirect access for this discussion).

5.3.6.1 Logical Contexts

Logical Contexts, indirectly referenced via the MI_SET_CONTEXT command, must reside in (unsnooped) Main Memory.

5.3.7 Command Arbitration

The command parser employs a set of rules to arbitrate among these command stream sources. This section describes these rules and discusses the reasoning behind the algorithm.

5.3.7.1 Arbitration Policies and Rationale

The Ring buffer (RB) is considered the primary mechanism by which drivers will pass commands to the graphics device.

The insertion of command sequences into the Ring Buffer must be a "synchronous" operation, i.e., software must guarantee mutually exclusive access to the Ring Buffer among contending sources (drivers). This ensures that one driver does not corrupt another driver's partially-completed command stream. There is currently no support for unsynchronized multi-threaded insertion of commands into ring buffer.

Another requirement for asynchronous command generation arises from competing (and asynchronous) drivers (e.g., "user-mode" driver libraries). In this case, the desire is to allow these entities to construct command sequences in an asynchronous fashion, via batch buffers. Synchronization is then only required to "dispatch" the batch buffers via insertion of Batch Buffer commands inserted into the Ring Buffer.

Software retains some control over this arbitration process. The MI_ARB_ON_OFF command disables all other sources of command arbitration until re-enabled by a subsequent MI_ARB_ON_OFF command from the same command stream. This can be used to define uninterruptible "critical sections" in an command stream (e.g., where some device operation needs to be protected from interruption). Disabling arbitration from a batch buffer without re-enabling before the batch is complete is UNDEFINED.

Batch Buffers can be (a) interruptible at command boundaries, (b) interruptible only at chain points, or (c) non-interruptible. See MI_BATCH_BUFFER_START in *Memory Interface Commands* for programming details.

5.3.7.2 Wait Commands

The MI_WAIT_EVENT command is provided to allow command streams to be held pending until an asynchronous event occurs or condition exists. An *event* is defined as occurring at a specific point in time (e.g., the leading edge of a signal, etc.) while a *condition* is defined as a finite period of time. A wait on an event will (for all intents and purposes) take some non-zero period of time before the subsequent command can be executed. A wait on a condition is effectively a noop if the condition exists when the MI_WAIT_EVENT command is executed.

A Wait in the Ring Buffer or batch buffer will cause the CP to treat the Ring Buffer as if it were empty until the specific event/condition occurs. This will temporarily stall the Ring Buffer.

While the Ring Buffer is waiting, the **RB Wait** bit of the corresponding RB*n*_CTL register will be set. Software can cancel the wait by clearing this bit (along with setting the **RB Wait Write Enable** bit). This will terminate the wait condition and the Ring Buffer will be re-enabled. This sequence can be included when software is required to flush all pending device operations and pending Ring Buffer waits cannot be tolerated.

5.3.7.3 Wait Events/Conditions

This section describes the wait events and conditions supported by the MI_WAIT_EVENT command. Only one event or condition can be specified in an MI_WAIT_EVENT, though different command streams can be simultaneously waiting on different events.

5.3.7.3.1 Display Pipe A,B Vertical Blank Event

The Vertical Blank event is defined as "shortly after" the *leading edge* of the next display VBLANK period of the corresponding display pipe. The delay from the leading edge is provided to allow for internal device operations to complete (including the update of display and overlay status bits, and the update of overlay registers).

5.3.7.3.2 Display Pipe A,B Horizontal Blank Event

The Horizontal Blank event is defined as "shortly after" the *leading edge* of the next display HBLANK period of the corresponding display pipe.

5.3.7.3.3 Display Plane A, B, Sprite A, Sprite B Flip Pending Condition

The Display Flip Pending condition is defined as the period starting with the execution of a "flip" (MI_DISPLAY_BUFFER_INFO) command and ending with the completion of that flip request. Note that the MI_DISPLAY_BUFFER_INFO command can specify whether the flip should be synchronized to vertical refresh or completed "as soon as possible" (likely some number of horizontal refresh cycles later).

5.3.7.3.4 Display Pipe A,B Scan Line Event

The Scan Line Event is defined as the start of a scan line specified in the Pipe A Display Scan Line Count Range Compare Register.

5.3.7.3.5 Semaphore Wait Condition

One of the 8 defined condition codes contained within the Execute Condition Code (EXCC) Register can be selected as the source of a wait condition. While the selected condition code bit is set, the initiating command stream will be removed from arbitration (i.e., paused). Arbitration of that command stream will resume once the condition code bit is clear. If the selected condition code is clear when the WAIT_ON_EVENT is executed, the command is effectively ignored.

5.3.7.4 Command Arbitration Points

The CP performs arbitration for command execution at the following points:

- Upon execution of an MI_ARB_CHECK command
- When the ring buffer becomes empty

5.3.7.5 Command Arbitration Rules

At an arbitration point, the CP will switch to the new head pointer contained in the UHPTR register if it is valid. Otherwise it will idle if empty, or continue execution in the current command flow if it arbitrated due to an MI_ARB_CHECK command.

5.3.7.6 Batch Buffer Protection

The CP employs a protection mechanism to help prevent random writes to system memory from occurring as a result of the execution of a batch buffer generated by a “non-secure” agent (e.g., client-mode library). Commands executed directly from a ring buffer, along with batch buffers initiated from a ring buffer and marked as “secure”, will not be subject to this protection mechanism as it is assumed they can only be generated by “secure” driver components.

This protection mechanism is enabled via a field in a Batch Buffer command that indicates whether the associated batch buffer is “secure” or “non-secure”. When the CP processes a non-secure batch buffer from the ring buffer it does not allow any MI_STORE_DATA_IMM commands that reference physical addresses, as that would allow the non-secure source to perform writes to any random DWord in the system. (Note that graphics engines will only write to graphics memory ranges, which by definition are virtual memory ranges mapped into physical memory pages by trusted driver components using the GTT/TGTT hardware). Placing an MI_STORE_DATA in a non-secure batch buffer will instead cause a Command Error. The CP will store the header of the command, the origin of the command, and an error code. In addition, such a Command Error can generate an interrupt or a hardware write to system memory (if these actions are enabled and unmasked in the IER and IMR registers respectively.) At this point the CP can be reactivated only by a **full reset**.

The security indication field of Batch Buffer instructions placed in batch buffers (i.e., “chaining” batch buffers) is ignored and the chained batch buffer will therefore inherit the security indication of the first Batch Buffer in the chain (i.e. the batch buffer that was initiated by an MI_BATCH_BUFFER_START command in the Ring Buffer).

5.3.8 Graphics Engine Synchronization

This table lists the cases where engine synchronization is required, and whether software needs to ensure synchronization with an explicit MI_FLUSH command or whether the device performs an implicit (automatic) flush instead. Note that a pipeline flush can be performed without flushing the render cache, but not vice versa.

Event	Implicit Flush or Requires Explicit Flush?
PIPELINE_SELECT	Requires explicit pipeline flush
Any Non-pipelined State Command	Device implicitly stalls the command until the pipeline has drained sufficiently to allow the state update to be performed without corrupting work-in-progress
MI_SET_CONTEXT	Device performs implicit flush
MI_DISPLAY_BUFFER_INFO (“display flip”)	Requires explicit render cache flush
MI_OVERLAY_FLIP	Requires explicit render cache flush
3D color destination buffer (render target) used as texture (i.e., “rendered texture”)	Requires explicit render cache flush
MEDIA_STATE_POINTERS	Requires explicit pipeline flush
MEDIA_OBJECT	Requires explicit pipeline flush
Media: Previous Destination Used as Source	Requires explicit render cache flush

Event	Implicit Flush or Requires Explicit Flush?
MI_SEMAPHORE_MBOX update to another pipe that requires output of current pipe to be in coherent memory. Example is blitter/render pipe synchronization	Requires explicit pipe flush

For GEN6, any time a pipeline flush is used it will need to follow by 8 separate storeDW commands. This is done as a work-around to a known issue where the pipeline flush does not push all cycles to the "GO" point. The storeDWs will not accomplish this process but will create enough stall time to ensure all cycles are at GO prior to execution of the next buffer or part of the buffer. This is required for producer/consumer model.

5.3.9 Graphics Memory Coherency

Table 5-2 lists the various types of graphics memory coherency provided by the device, specifically where the CPU writes to a 64B cacheline, and the device then accesses that same cacheline. Note that the coherency policy depends on the address type (GM or MM) involved in the accesses.

Table 5-2. Graphics Memory Coherency

CPU Access	Subsequent Device Access	Example Operand	Coherency
Write GM	Read GM	Texture loads	GM accesses from IA cores use WC memory type, and any tail-pointer update is US, hence flow is coherent when device accesses GM.
Write MM	Read MM	Batch Buffer	IA writes are always coherent with device accesses, hardware ensures the coherency.
Write GM	Write GM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).
Write MM	Write MM		"assumed to exclusive byte" No byte sharing w/o semaphores
Write GM	Read MM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).

5.3.10 Graphics Cache Coherency

There are several caches employed within the graphics device implementation. This section describes the impact of these caches on the programming model (i.e., if/when does software need to be concerned).

5.3.10.1 Rendering Cache

The rendering (frame buffer) cache is used by the blit and 3D rendering engines and caches portions of the frame buffer color and depth buffers. This cache is guaranteed to be flushed under the following conditions (note that the implementation may flush the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Render Flush Cache Inhibit** bit clear
- Execution of a PIPE_CONTROL instruction with the **Write Cache Flush Enable** bit set (Depth Stall must be clear).
- A SyncFlush handshake operation
- A change of rendering engines (e.g., going from 2D to 3D, 3D to 2D, etc.)
- Logical Context switch (via MI_SET_CONTEXT)

The render cache must be explicitly flushed using one of these mechanisms under certain conditions. See Graphics Engine Synchronization above.

5.3.10.2 Sampler Cache

The read-only sampler cache is used to cache texels and other data read by the Sampling Engine in the 3D pipeline. This cache can be enabled or disabled via the **Texture L2 Disable** bit of the Cache_Mode_0 register (see *Memory Interface Registers*). Note that, although there may be more than one level of sampler cache within the implementation, the sampler cache is exposed as a single entity at the programming interface.

The sampler cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Map Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the **Depth Stall Enable** bit clear.
- A SyncFlush handshake operation

The sampler cache must be invalidated prior to reallocation of physical texture memory (i.e., software must guarantee that stale texture data is invalidated before reusing physical texture memory for a new or modified texture).

5.3.10.3 Instruction/State Cache

The read-only ISC is used to cache pipelined state and EU instructions read in from memory. It also functions as a prefetch cache by reading in additional state information and instructions beyond those immediately requested in order to decrease latency and improve performance. As with the sampler cache, there may be more than one level of ISC within the implementation. The ISC is exposed as a single entity at the programming interface.

The instruction/state cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **State/Instruction Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the **Instruction/State Cache Flush Enable** bit set.
- A SyncFlush handshake operation

The instruction/state cache must be invalidated prior to reallocation of physical state/instruction memory (i.e., software must guarantee that stale state/instruction data is invalidated before reusing physical state/instruction memory for new or modified state or instructions).

5.3.10.4 Vertex Cache

The vertex cache consists of 2 sub-caches: one that caches vertex buffer data based on address, and another that caches (possibly shaded) vertex attribute data based on index (see the *Vertex Fetch* chapter for vertex index details). The latter cache is always invalidated between primitive topologies.

Both vertex caches are guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command
- Execution of a PIPE_CONTROL command
- A SyncFlush handshake operation
- Logical Context switch (via MI_SET_CONTEXT)

5.3.10.5 GTT TLBs

The following table summarizes when the various TLBs are invalidated.

TLB	Normal Invalidation Mechanism
Display	Refreshed on Vsync
Overlay	Refreshed on Vsync
Render/Blit	Internal Flush*
Host	Through a Page Table PTE write
Sampler Cache	Internal Flush*
Command Stream	Through a Page Table PTE write

* -- Includes MI_FLUSH, Engine switch, and Context switch.

On [DevCTG], all TLBs are invalidated on a VT-d translation enable (TE) status change.

5.3.11 Command Synchronization

This section describes the hardware mechanisms that can be used by software to provide synchronization with command stream parsing and execution.

The key point here is distinguishing between command *parsing* and *retirement* – in that, for most commands, there is some finite delay between the parsing of a command and the retirement (coherent completion) of the operation it specifies.

Interrogation of the Ring Buffer Head Pointer only gives an indication of the progress of command parsing. This information is required to discern the availability of command data within the Ring Buffer or Batch Buffers. If the Head Pointer indicates the command data has been parsed, those locations can be reused, otherwise the commands must be considered still pending parsing and left alone.

Given the CP rules for command execution, it is possible to use the indication of command parsing progress to infer the retirement status of parsed commands. The only indication of instruction retirement available from instruction parsing is that parsing of an MI instruction implies retirement of previous MI instructions with the following exceptions:

- The parsing of a Memory Interface (MI) command implies that all previously-parsed MI commands have completed, with the following exceptions:
    - Display and Overlay Flip commands: Only the submission of the flip request is guaranteed. The flip operation will occur some time later. Mechanisms to detect the actual completion of a flip operation are described below.
    - "Store-Data" type commands: Only the submission of the store operation is guaranteed. The write result will be complete (coherent) some time later (this is practically a finite period but there is no guaranteed latency).
    - Batch Buffer commands: There is no guarantee that the operations performed by the batch buffer have completed.

Other than the cases described above, additional measures must be taken to discern the progress of command retirement. These measures are described in the following subsections.

5.3.11.1 MI_FLUSH

The MI_FLUSH command pauses further command parsing until all drawing engines become idle and any internal rendering cache is flushed and invalidated. All previous rendering commands can therefore be considered retired.

This flush operation is considered complete once command parsing proceeds to the next command. Software can, for example, follow an MI_FLUSH command with an MI_STORE_DATA_IMM or MI_STORE_DATA_INDEX command – where the completion of the store operation implies that the flush operation has completed. (Note that if the last DWord in a ring buffer is an MI_FLUSH instruction, there is no way by simply looking at the Ring Buffer registers to determine whether the flush operation is complete or still pending.)

The successful completion of an MI_FLUSH command does not guarantee that *all* previous operations have completed. Operations that may still be pending include:

- Store Data type commands (MI_STORE_DATA_IMM, MI_STORE_DATA_INDEX, MI_REPORT_HEAD)
- Display or Overlay Flip operations

See section 5.3.10.2 for more information on when the sampler cache should be invalidated.

5.3.11.2 Sync Flush

Inserting MI_FLUSH commands, while effective at determining or forcing the retirement of previous rendering commands, may negatively impact performance if not absolutely required. For example, if the knowledge of rendering command retirement is not known a priori, it is likely undesirable to insert MI_FLUSH commands at intervals in the command stream. However, it may not be acceptable to insert an MI_FLUSH command (and wait for its completion) at the point that rendering command retirement is required – as there may be a large number of commands pending in ring/batch buffers at that point and flushing the entire device (including waiting for completion of pending commands that have not yet been parsed) may be prohibitive. There is a mechanism, however, where command stream synchronization can be performed on demand, without requiring earlier submitted commands and batch buffers to complete – it is called the "Sync Flush" mechanism.

Here's how it works:

- Software must (preferably at driver initialization time) unmask the Sync Status bit in the Hardware Status Mask Register (HWSTAM). This should be done unconditionally (at least whenever HW status writes are enabled), as any bandwidth increase due to Sync Status-initiated writes is negligible.
- At the point that synchronization is required, software must guarantee that command parsing has progressed past the point of interest in the command stream (i.e., past the last command whose retirement is required). Note that this step is required in any scheme.
- Software then reads the location where the Interrupt Status is reported in the Hardware Status Page (DWord offset 0) and saves that DWord in a temporary variable.
- Software then sets the Sync Enable bit of the Command Parser Mode Register (INSTPM) via an uncached write.
- The Command Parser will detect the Sync Enable bit set before it proceeds to the very next command (or immediately if the CP is idle). It will then perform an internal flush operation. This flush is identical to that performed by an MI_FLUSH command with all flush types enabled.
- Once this flush operation is complete, the CP will clear the Sync Enable bit of the INSTPM register and then *toggle* the Sync Status bit of the ISR register. This will initiate a write of the ISR register contents (with the toggled Sync Status) to DWord 0 of the Hardware Status page (as part the normal hardware status write mechanism).
- Software, following the write of the INSTPM register, should periodically poll the Hardware Status location. By comparing the current versus saved value of the Sync Status bit, software can then detect when the flush operation is complete. Note that the latency of this operation is typically small, as it will be initiated either immediately or at least before the next command is parsed (regardless of arbitration conditions).

5.4 Hardware Status

The graphics device supports a number of internal hardware status bits which can be used to detect and monitor hardware status conditions via polling or interrupts. This section will describe each hardware status bit. The following section describes the hardware status reporting (polling) mechanism. The mechanism to allow these status bits to generate interrupts is described in the Interrupts section. Note that the hardware status bits are actually reported in the Interrupt Status Register, so "hardware status" and “interrupt status” are used interchangeably here (though many hardware status bits won’t necessarily ever be used to generate interrupts).

The following subsections describe the various hardware (interrupt) status bits, as defined in the Interrupt Status Register.

5.4.1 Hardware-Detected Errors (Master Error bit)

This interrupt status bit is generated whenever an “unmasked” hardware-detected error status is detected. See the Errors section.

5.4.2 Thermal Sensor Event

This interrupt status bit is generated by “thermal events” detected by the Thermal Sensor logic. The bit corresponding to this event in the HWSTAM register must always be masked (i.e., set to ‘1’) so that thermal sensor events do not generate HW status DWord writes. See Hardware Status Writes.

5.4.3 Sync Status

This bit should only be used as described in Sync Flush, and should not be used to generate interrupts (i.e., the corresponding interrupt should not be enabled in the IER).

5.4.4 Display Plane A, B, (Sprite A, Sprite B) Flip Pending

These bits are used to report the status of “flip” operations on the corresponding Display Plane. Display Flip operations are requested via the MI_DISPLAY_BUFFER_INFO command. When that command is executed, the corresponding Display Flip Pending status in the ISR register will be set to ‘1’ indicating that a display flip has been requested but has not yet been performed. (Requesting a flip operation when one is already pending is UNDEFINED). This indicates that a flip is “pending”. At the appropriate time during the next vertical blank period (for that display stream), the flip operation will be performed (i.e., the display will switch to refreshing from the new display buffer). This causes the Display Flip Pending status to reset to ‘0’. When this occurs, and the Display Flip Pending status bit is unmasked by the Interrupt Mask Register (IMR), the Display Flip Pending status bit of the Interrupt Identity Register (IIR) is set. Note that this setting of an interrupt identity bit on the falling edge of the status bit is contrary to the general definition of interrupt status bits.

5.4.5 Display Pipe A,B VBLANK

These bits are set on the leading edge of the selected Display Pipe’s VBLANK signal.

5.4.6 User Interrupt

This bit is set in response to the execution of an MI_USER_INTERRUPT command. The Command Parser will continue parsing after processing that command. If a user interrupt is currently outstanding (set in the ISR) this packet has no effect.

Programming Note: User interrupts can be used to notify software of the progress of instruction parsing past the MI_USER_INTERRUPT instruction. Standard rules regarding instruction parsing versus instruction retirement documented in the section of this chapter still apply. In particular, user interrupts can be inserted into the command stream but effectively disabled for “normal operation” via the IMR and HWSTAM registers. Whenever software requires the notification afforded by the user interrupts, it can unmask this bit.

5.4.7 PIPE_CONTROL Notify Interrupt

This bit is set when a PIPE_CONTROL command with the **Notify Enable** bit set reaches the end of the pipeline and all required cache flushes have occurred.

5.4.8 Display Port Interrupt

This bit is set on a hot plug event. See the *Display Registers* chapter for details.

5.5 Hardware Status Writes

The graphics device supports the writing of the hardware status (ISR) bits into memory for optimized access from software. Software can select which (if any) status bits will trigger the write of the ISR contents to memory using the Hardware Status Mask (HWSTAM) register. Writing a ‘0’ to a defined bit position in the HWSTAM register will cause any change (0 → 1 or 1→ 0) in the corresponding ISR bit to trigger the write. The complete ISR contents will be written to DWord offset 0 of the hardware status page, located at the address programmed via the Hardware Status Page Address Register (HWS_PGA).

5.6 Interrupts

The graphics device supports the generation of an interrupt. This interrupt can be raised in response to one or more internal interrupt status conditions. Which interrupt status conditions are allowed to raise an interrupt is programmed via the Interrupt Mask Register (IMR) and Interrupt Enable Register (IER). The IMR is used to selectively “unmask” hardware status bits as to allow them to be reported in the Interrupt Identity Register (IIR). The IER holds a set of interrupt enable bits corresponding to each bit of the IIR – setting bits in the IER will allow interrupts to be generated by the corresponding bits in the IIR.

5.7 Errors

The graphics device supports the hardware detection of a number of *operational* errors. Operational errors occur out of the immediate control of driver software and must be anticipated and tolerated to the extent required by the relevant APIs. Software must therefore support the detection and proper handling of all relevant operational errors. [Reserved content]

5.7.1 Error Reporting

Regardless of the error classification, all errors funnel through the **Master Error** bit of the Interrupt Control Registers. This bit can be used to raise a device interrupt or trigger a hardware status write operation. (Needless to say it can also be polled directly, though this is clearly discouraged). Refer to Interrupt Control Registers in the *Memory Interface Registers* chapter for more information.

There are three registers dedicated to control, detect, and clear hardware error status conditions in a similar fashion to the Interrupt Control Registers. All three error registers share a common error status bit definition.

The Error Status Register (ESR) holds the actual error status bits (each of which may be the logical OR of "source" error bits in various functional registers). The Error Mask Register (EMR) is used to select which error status bit(s) are reported in the Error Identity Register (EIR). The EIR holds the "persistent" values of the unmasked error status bits, and is also used to clear error status conditions. Any bits set in the EIR will raise the Master Error interrupt status condition.

5.7.2 Page Table Errors

The following tables describe the various sources and types of Page Table Errors.

Table 5-3. Page Table Error Types

Error	Description	Streams
Invalid GTT PTE	In the process of mapping an address, the MI encountered a GTT PTE that was marked "Invalid". This would be the result of a programming error.	All (See Table 4-1)
Invalid TLB Miss	An unexpected TLB miss (detected at GTT request time) was encountered (e.g., during Display/Overlay/Sprite access).	Display, Overlay
Invalid PTE Data	Mapping to the physical page specified in the PTE is not permitted (e.g., a page in PAM, SMM or over the top of memory, etc.). This is the result of a programming error.	Host
Invalid Tiling	A tiling parameter was found inconsistent with the current operation. This includes the use of Y-Major tiling in the Render/Display/Overlay streams. This is the result of a programming error. This is detected during GTT request.	Blt, Display, Overlay

Note that Page Table Errors cannot be cleared. A device reset is required.

5.7.2.1 Page Fault Handling

A page fault happens when a GTT entry is read to be used, and it's valid bit is 0 (Bit 0 of the GTT DW)

When GAM encounters a page fault, it will record the faulty address into a register, and invalidate the cycle(s) that use that page translation. Invalidating a cycle means asserting the invalid bit in the GAM-CI interface. This bit commands the Super Queue to cancel an invalid write, and return all zeros for an invalid read.

Since a page fault is not recoverable for the client that generates it, the invalid cycles are lost (in the sense that intended writes will end up not happening, and reads will return garbage = all zeros)

In order to simplify GAM design, an invalid GTT entry will be loaded into the TLB, and will continue behave as a valid one, with the exception of the invalid bit generation.

This means that it is possible to have a TLB hit, which has an invalid GTT saved for the physical address, which will always genrate invalid cycles.

Register Recording:

For each engine (Graphics, Media, and Blitter), there is a register that will be used for recording the first page fault that occurs for that engine. Subsequent faults will be treated as described above, but will not be recorded, until the valid bit of that register is cleared by Software (Valid bit is set when the first fault for that engine happens). This will provide a lead to Software on when things started going wrong.

Graphics Engine Fault Register Definition:

Address Offset:	4094-4097h
Default Value:	XXXXXXXXxxx0h
Access:	RO; RW;
Size:	32 bits

GFX Arbiter Page Fault Register

Bit	Access	Default Value	RST/PWR	Description
31:12	RW	x	Core	**Virtual Address of Page Fault:** This is the original Address of the Page that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
11	RW	x	Core	**GTT Sel:** This bit indicates if the valid bit happened while using PPGTT or GGTT: 0 – PPGTT, 1 - GGTT This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
10:3	RW	x	Core	**SRCID of Page Fault:** This is the Source ID of the unit that requested the cycle that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
2:1	RO	x	Core	**RESERVED.**
0	RW	0	Core	**Valid Bit:** This bit indicates that the first fault for this engine has been recorded. It can only be cleared by SW, which will also clear the other fields.

Media Engine Fault Register Definition:

Address Offset:	4194-4197h
Default Value:	XXXXXXXXxxx0h
Access:	RO; RW;
Size:	32 bits

MEDIA Arbiter Page Fault Register

Bit	Access	Default Value	RST/PWR	Description
31:12	RW	x	Core	**Virtual Address of Page Fault:** This is the original Address of the Page that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
11	RW	x	Core	**GTT Sel:** This is the original Address of the Page that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
10:3	RW	x	Core	**SRCID of Page Fault:** This is the Source ID of the unit that requested the cycle that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
2:1	RW	x	Core	**RESERVED.**
0	RW	0	Core	**Valid Bit:** This bit indicates that the first fault for this engine has been recorded. It can only be cleared by SW, which will also clear the other fields.

Blitter Engine Fault Register Definition:

Address Offset:	4294-4297h
Default Value:	XXXXXXXxxx0h
Access:	RO; RW;
Size:	32 bits

BLT Arbiter Page Fault Register

Bit	Access	Default Value	RST/PWR	Description
31:12	RW	x	Core	**Virtual Address of Page Fault:** This is the original Address of the Page that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
11	RW	x	Core	**GTT Sel:** This is the original Address of the Page that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
10:3	RW	x	Core	**SRCID of Page Fault:** This is the Source ID of the unit that requested the cycle that generated the First Page fault for this engine. This value is locked and not updated on subsequent faults, until the valid bit of this register is cleared by SW
2:1	RW	x	Core	**RESERVED.**
0	RW	0	Core	**Valid Bit:** This bit indicates that the first fault for this engine has been recorded. It can only be cleared by SW, which will also clear the other fields.

5.7.3 Clearing Errors

For operational errors, software is responsible for taking the proper steps to recover from the error and then clearing the error indication. The actions required to recover from operational errors may be discussed in the various functional areas (not here). See the Hardware-detected Error Bit Definitions in *Memory Interface Registers* for more details. This subsection describes the actions required to clear the error indication.

In order to clear operational errors, software is responsible for clearing the error condition from the source, working back to the Master Error bit. Typically this will entail the following sequence.

- First the primary source of the error must be cleared. This requires clearing the functional register(s) containing the source error indication.

- Next, clear the particular error status bit by writing a '1' to the appropriate bit of the Error Identity Register (EIR). This will clear the error status bit in the Error Status Register (ESR). If multiple errors are present, all error status bits should be cleared simultaneously.
- Next, clear the Master Error interrupt status bit by writing a '1' to the Master Error bit of the Interrupt Identity Register (IIR).

Note: Page Table Errors cannot be cleared.

5.8 Rendering Context Management

The graphics device operation (rendering, etc.) is controlled via the settings of numerous hardware state variables. These state variables are divided into *global state* and *context state*.

There is only one copy of global state variables, and changing the settings of these variables requires explicit programming of the state variables. Examples of global state include:

- MI registers (HWSTAM, Ring Buffer, etc.) with the exception of those listed in the next paragraph (i.e, registers listed there *are* saved/restored)
- Configuration registers
- Display programming registers

On the other hand, context state is associated with a specific *context*, where switching to that context causes that context's state to be restored. While the associated context is active, the state variables and registers can be programmed via the command stream. Examples of context state include the PIPELINE_STATE_POINTERS command and most non-pipelined state. The following MI registers are considered part of context state and thus saved/restored with context:

- INSTPM
- CACHE_MODE_0
- CACHE_MODE_1
- MI_ARB_STATE
- 3D Pipeline Statistics Registers

The graphics device supports both a *hardware context* and *logical contexts*. The multiple logical context support provides robust rendering context support by swapping contexts to/from memory.

5.8.1 Multiple Logical Rendering Contexts

The graphics device supports multiple *logical rendering contexts* stored in Main Memory. Logical rendering contexts are referenced via a 2KB-aligned *Logical Context Address*.

The maximum size of a logical context entry (which is information required by the driver to allocate contexts) is currently 2K bytes. For forward compatibility, the maximum size of a logical context entry should be supplied to the drivers via a VBIOS mechanism as opposed to being hardcoded in the driver.

The actual size of a logical rendering context is the amount of data stored/restored during a context switch and is measured in 64B cache lines.

The format of the logical rendering context in memory is considered device-dependent; software must not attempt to modify the contents of a logical rendering context directly. This restriction is motivated by forward compatibility concerns because the location and definition of fields may change between implementations.

5.8.1.1 Current Context IDs

The ring buffer has an associated *Current Context ID* (CCID) register*.* The CCID includes a Logical Pipeline Context Address (LPCA).

The CCID for a ring buffer is set during the processing of the new MI_SET_CONTEXT command from that ring. The MI_SET_CONTEXT command provides a new CCID value (LPCA) to be loaded into the CCID register for the associated ring buffer. The MI_SET_CONTEXT command also contains a Restore Inhibit bit used to optionally inhibit the restoration (loading) of the new rendering context. This bit must be used during context initialization to avoid the loading of uninitialized (garbage) context data from memory. Failure to do so leads to UNDEFINED operation.

The initial values of the CCIDs are UNDEFINED. The first time a valid CCID is set from a ring buffer, the normal context save operation will be suppressed, as the previous CCID is invalid.

5.8.1.2 Intra-Ring Context Switch

Within a specific ring buffer, a new logical rendering context is specified via the MI_SET_CONTEXT command. Note that MI_SET_CONTEXT commands are permitted only within a ring buffer (not within a batch buffer).

As part of the execution of the MI_SET_CONTEXT command from within a ring buffer, the Logical Pipeline Context Address fields of the CCID register and MI_SET_CONTEXT command are compared. If they differ (or the CCID register is uninitialized), a rendering context switch operation will be performed, which includes:

1. If the CCID contents are valid, a context save operation will be performed. The contents of the HW context will be saved in memory starting at the Logical Pipeline Context Address specified in the CCID.

2. If the Restore Inhibit command field is not set, a context restore operation will be performed. Here the logical context values are read starting from the Logical Pipeline Context Address field of the command and used to set the internal HW context.

3. The relevant contents of the command will be loaded into the appropriate CCID register. (This occurs irrespective of the LPCA comparison result). At this point, the ring buffer has switched to using the new logical rendering context.

5.8.1.3 Logical Rendering Context Creation and Initialization

5.8.1.3.1 Rendering Context Creation Rules

1. Software only knows the **size** of the logical rendering context (2KB), for allocation purposes.

2. Given (1), software does **not** know the format of the context, and therefore is not allowed to write any portion of a logical rendering context. Software can, however, copy/move entire logical context blocks.

3. Given (2), software must never restore (load) a logical rendering context from memory that has not been previously stored by HW. I.e., software must never attempt to initialize a context itself and then cause it to be loaded. Breaking this rule causes UNDEFINED operation (as in the hang seen in BDG validation).

4. Initialization software must write **all** HW context variables with legal values before the first rendering context can be saved (this must be done before you can perform any rendering anyways). Given this, and the obvious rule that software must never program illegal state values, guarantees that the HW context will forever remain valid (and therefore be available to store into a logical rendering context). Note that software-visible context variables include 3D state, Blt register state, etc.

5.8.1.3.2 Context Initialization

Logical Rendering Contexts can be initialized (in memory) by software in the following way:

1. Issue an MI_SET_CONTEXT command w/ the **Restore Inhibit** bit set and the about-to-be-initialized logical pipeline context address. This will save the current rendering context and then change the LPCA to the new context (without loading it).

2. Use state commands to modify the context as desired.

3. Issue another MI_SET_CONTEXT command specifying some other LPCA (e.g., the previous one). This will cause the new context to be stored (initialized) in memory

5.8.1.4 Context Save

A context save will occur anytime all of the following apply:

- A rendering context switch occurs as a result of the execution of MI_SET_CONTEXT

- the CCID of the current context (CCID register of current ring) and the new CCID (the CCID register of the newly selected ring or the new CCID in the MI_SET_CONTEXT command) differ OR an MI_SET_CONTEXT with the “Force Restore” bit set initiated the context switch

- the current CCID is valid (has been previously set)

The current rendering context will be written out to memory starting at the LPCA in the format described by Logical Context Layout in *Memory Data Formats*. Note that this includes a limited number of Memory Interface Registers whose values are saved by embedding them in an MI_LOAD_REGISTER_IMM command that is written out to memory.

The Optional Extended Context will also be written if the Extended Save Enable bit is set in the current CCID register. Context saves DO NOT modify pipelined state stored in memory.

5.9 Reset State

This section describes the state of the programming interface following a hardware reset. Refer to the individual register definitions for details on reset (default) settings.

- The settings of the hardware context state variables are UNDEFINED. Software must program all state variables prior to their use in rendering.
- The ring buffer is disabled.
- All interrupts and error status bits are “masked” (disabled). All interrupts are disabled via IER. There will be no HW activity to cause any hardware/interrupt status bits to be set.
- The Hardware Status Page is located at 1FFFF000h (though HW status writes are effectively disabled)
- All FENCE registers are INVALID
- The GTT is disabled (accesses other than CPU reads, cursor and VGA reads will generate an error).
- All INSTDONE bits are set (“DONE”).
- The NOPID register is 0.
- All command groupings are enabled (via INSTPM)

Revision History

Revision Number	Description	Revision Date
1.0	First 2011 OpenSource edition	May 2011

§§