Permedia4®

Programmer's Guide - Volume I

DRAFT ONLY

PROPRIETARY AND CONFIDENTIAL INFORMATION

3Dlabs®

Permedia4®

Programmer's Guide - Volume I

PROPRIETARY AND CONFIDENTIAL INFORMATION

Issue 2

Proprietary Notice

The material in this document is the intellectual property of **3D***labs*. It is provided solely for information. You may not reproduce this document in whole or in part by any means. While every care has been taken in the preparation of this document, **3D***labs* accepts no liability for any consequences of its use. Our products are under continual improvement and we reserve the right to change their specification without notice. **3D***labs* may not produce printed versions of each issue of this document. The latest version will be available from the **3D***labs* web site.

3D*labs* products and technology are protected by a number of worldwide patents. Unlicensed use of any information contained herein may infringe one or more of these patents and may violate the appropriate patent laws and conventions.

3D*labs* is the worldwide trading name of **3D***labs* Inc. Ltd.

3D*labs*, Permedia4 and PERMEDIA are registered trademarks of **3D***labs* Inc. Ltd.

Microsoft, Windows and Direct3D are either registered trademarks or trademarks of Microsoft Corp. in the United States and/or other countries. OpenGL is a registered trademark of Silicon Graphics, Inc. All other trademarks are acknowledged and recognized.

© Copyright **3D***labs* Inc. Ltd. 1999. All rights reserved worldwide.

Email: info@3dlabs.com
Web: http://www.3dlabs.com

3D*labs* Ltd.
Meadlake Place
Thorpe Lea Road, Egham
Surrey, TW20 8HE
United Kingdom
Tel: +44 (0) 1784 470555
Fax: +44 (0) 1784 470699

3D*labs* K.K
Shiroyama JT Mori Bldg 16F
40301 Toranomon
Minato-ku, Tokyo, 105, Japan
Tel: +81-3-5403-4653
Fax: +91-3-5403-4646

3D*labs* Inc.
480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: (408) 530-4700
Fax: (408) 530-4701

Change History

Document	Issue	Date	Change
160.3.0	1	1 October 99	First DRAFT Issue.
160.3.0	2	16 June 2001	Improvements/corrections to LB and FB; removed incorrect footnote ref to PEREN004 (DMA Continue); clarified opaque span color masking; corrected PixelSize use; deleted FCP reference in GID position text, changed PCI map access from 2&4 to 1&2.

Contents

1 Introduction

The Permedia4 family of high performance PCI/AGP graphics processors combine workstation class 3D graphics acceleration and state of the art 2D performance in a single chip. All 3D rendering operations are accelerated by Permedia4, including Gouraud shading, depth buffering, antialiasing, alpha blending and texture mapping.

Implemented around a scaleable memory architecture, Permedia4 reduces the cost and complexity of delivering high performance 3D graphics within a windowing environment - making it ideal for a wide range of graphics products from PC boards to high end workstation accelerators.

This document has been written as the reference for programmers and system designers who wish to develop software to drive the Permedia4. There are separate manuals for related members of the Permedia, GLINT Gamma and GLINT Delta families. Familiarity with the OpenGL Specification will be useful when reading this document.

1.1 How to use this manual

The *Permedia4 Programmers' Guide* (Volumes I and II) should be read together with the *Permedia4 Reference Guide*, which contains all of the register descriptions. The Programmers Guide is in two volumes. The present volume contains:

- Chapter 2 - an overview of Permedia4, its capabilities and architecture, and highlights key differences between the Permedia4 and GLINT MX or Permedia2/3.
- Chapter 3 - details of the programming model for the chip, including the DMA interface, host bypass route to unified framebuffer, vertex loading and context changing.
- Chapter 4 - describes the data structures that Permedia4 supports in the framebuffer and the localbuffer, including virtual texture management
- Chapter 5 - describes the Video System including timing, RAMDAC and overlays.

In Volume II of the Programmers Guide we examine Permedia4 graphics rendering. The chapters begin with an overview and discussion of 3d and 2d graphics pipelining, followed by a walkthrough of the major functional groups (Scissor, Fog, Alpha Test, etc.) and a closing discussion of initialization and performance issues. In addition, volume II contains two appendices and a glossary:

- Appendix A gives the format used in the pseudocode examples throughout the document.
- Appendix B gives example code for rendering a triangle accurately.
- Following the body of the manual, a glossary of technical terms defines many of the 2D/3D graphics terms used throughout.

1.2 Further Reading

- *Permedia4 Reference Guide*, 3Dlabs
- *Permedia4 Architecture Overview*, 3Dlabs
- *OpenGL Programming Guide*, Jackie Neider et al, Reading MA: Addison-Wesley
- *OpenGL Reference Manual*, Jackie Neider et al, Reading MA: Addison-Wesley
- *The OpenGL Graphics System: A Specification* (Version 1.1), Mark Segal and Kurt Akeley, SGI (see below)
- *PCI Local Bus Specification Rev2.1*, 1Jun95, PCI Special Interest Group, PO Box 14070, Hillsboro, Oregon 97214 (503-797-4207)
- *Multiprocessor Methods For Computer Graphics Rendering*, Scott Whitman, ISBN 0-86720-229-7
- Microsoft WIN32 Software Development Kit 3.1, Microsoft
- *Windows NT 3.1 Graphics Programming*, Emeryville CA, Ziff-Davis Press
- *The X Window System*, Sebastopol CA, O'Reilly & Associates Inc.
- *The X Window System Server*, Elias Israel and Erik Fortune, Digital Press
- *Computer Graphics: Principles and Practice*, James D. Foley et al, Reading MA: Addison-Wesley

2

Architecture Overview

2.1 Functional Overview

Permedia4 is a 3D graphics processor consisting of a geometry set-up processor and rastering engine together with external interfaces. It fully implements the functionality of "The OpenGL Machine" from edge walk and span interpolation through fragment level processing including:

- Point, Line, Triangle and Bitmap primitives
- Flat and Gouraud shading
- Texture and Fog
- Antialiasing
- Scissor and Stipple
- Alpha test, Stencil test, Depth (Z) buffer test
- Alpha Blending
- Dithering
- Logical Operations
- Writemasks, spans and images

The 300 MFLOPS geometry set-up unit accepts vertex, color, depth, fog and texture parameters in IEEE single precision floating point format. It is both OpenGL and Direct3D compliant and supports the Direct3D TLVERTEX data type and the DirectX 6 Flexible Vertex Format (FVF) extensions.

Systems using Permedia4 can easily be configured to address a wide range of price, performance and functionality points by simply tuning the external memory design. Permedia4 supports 8, 16 and 32-bit RGBA and 8-bit color index framebuffers.

The unified memory can be up to 32Mbytes in size. Permedia4 also supports DMA mode virtual texture and demand page texturing using a 4K local L2 texture cache to incrementally access up to 256MB of addressable host texture.

2.2 Block Diagram

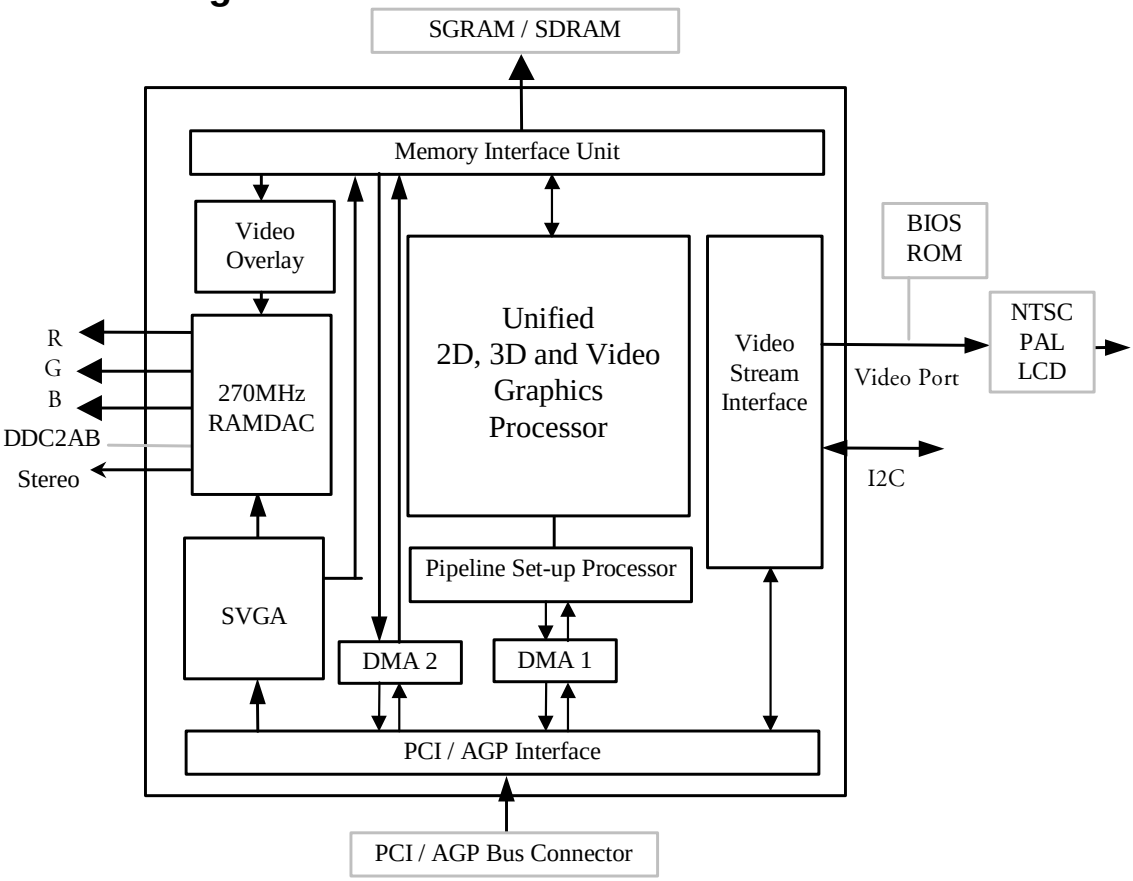


Figure 2-1 High level blocks in the Permedia4 architecture

2.3 Host Interfaces

The Permedia4 architecture consists of a Graphics Core augmented by I/O and memory interfaces as shown in Figure 2-1. There are three external interfaces: the Host Bus Interface (PCI/AGP Local Bus), the SGRAM/SDRAM unified memory Interface, and a Video Stream interface.

There are two ways to program the graphics processor, either by writing to memory mapped registers, or by sending command packets to the input FIFO (which may be done by HostIn DMA or programmed IO). Both methods produce tag data that is sent to the pipeline.

A write to a register creates a tag/data pair "message" which has a tag formed from the lower bits of the address. This tag and the data associated with it are sent to the input FIFO. If the data has been written directly to the input FIFO address range (or fetched by DMA) a special tag marking it as part of a packet is sent with the data to the input FIFO. The decode process converts packets into tag/data streams, so that after this process both methods of progaming are indistinguishable.

Conceptually Permedia4 can be viewed as either a register file or a message passing system. As a register file, it appears as a flat block of memory-mapped registers. When a driver is initialized it maps the register file into address space. Each register has an associated address tag containing the offset from the file base in multiples of 8 bytes

(since all registers reside on a 64-bit boundary). The most straightforward way to load a data value into a register is to write the data to its mapped address.

As a message passing system, Permedia4 consists of processing units connected in a long pipeline with message-passing communication between adjacent units. There is a small amount of buffering between units, the size being appropriate to the local communications requirements and unit throughput. Although the units operate asynchronously they are in practice synchronised by the common clocks.

Units are in fact functional groupings - virtual units - rather than strictly physical entities. For example, although Frame and Local Buffer "units" exist they operate on contiguous, unified memory. The difference is that local buffer functions involve GID, stencil and depth facilities while frame buffer functionality primarily relates to the 8/16/32 bpp color formats.

Each unit can be disabled, and the host can mimic any unit in the chain by inserting messages which that unit would normally generate, effectively bypassing the unit. The host can also directly access the framebuffer, and context dump/restore is also supported.

When the control registers have been primed with the data needed to render the primitive, the render process is triggered by writing to a Command register such as **Render2D** or **ContinueNewLine**.

2.3.1 Task Switching

Where multiple applications wish to make simultaneous access to Permedia4, it is the responsibility of the software driving the chip to handle loading and to load the correct state. Permedia4 has been designed to support a number of different software architectures. Facilities available include:

- Synchronous operation - a new task can load its context without waiting for current rendering to complete
- All loadable states can be read back
- Sync command to flush all rendering - can be polled or return an interrupt

2.4 Processor evolution and programming changes

Although part of the same family as the Permedia 1 and 2, Permedia4 functionality changes affect almost all areas of operation, including performance and feature benefits and programming differences from earlier chipsets such as Permedia3. Performance and feature changes include:

- Increased clock speeds
- Low latency command DMA
- 2M drawn texture-mapped polygons/sec
- 250M perspective correct, bilinear filtered, dual texture texels/sec
- 125M perspective correct, per pixel MIP-mapped trilinear filtered, texture mapped, depth buffered, fogged and blended pixels/sec
- 8M backface-culled polygons/sec
- Enhanced Texture format (Delta) unit
- Support for HDTV screen resolutions, e.g. 1920x1080
- Optimized for 32-bit displays at resolutions up to 2048x1536
- 266 MHz AGP 4X DMA and Execute Mode support
- 3.3V and 1.5V AGP signal support
- 128 bit internal bus

- LCD flat panel and TV display support
- MPEG2 compatible video playback acceleration with motion compensation, hardware scaling and filtering

2.4.1 Programming changes from earlier products

Relative to the GLINT MX or Permedia2, for example, there are a large number of new, changed or deleted registers. There are also a significant number of minor corrections in implementation. These result in significant programming changes, particularly in

- buffer addressing,
- register width,
- context management,
- texture formatting,
- bitmask spans and span alignment,
- logical operations,
- unit enabling,
- non-linear Z,
- pixel size definition and subpixel correction,
- scale by Q, texture wrap and LOD calculations
- OGL blend modes with multiple texturing
- DMA interrupts

...as well as minor changes in register and bitfield nomenclature and functionality.

2.4.2 Permedia3/Permedia4 Programming Differences

Relative to Permedia3, changes to Permedia4 primarily affect hardware, improving reliability and performance by removing technical obstacles, clearing errata and adding bandwidth enhancements, particularly support for PCI 2.2 and the AGP4X bus standard. Programming changes are noteworthy but few:

- Introduction of a new DeltaFormat unit with early backface cull and texture functionality
- Additional OGL Alphablend interpolations
- A number of bitwise register changes

For a list of errata changes refer to the *Permedia3 Errata* document and the *Permedia4 Reference Guide* volume I, section 1.2 - P3/P4 Differences.

3

Programming Model

This chapter describes the programming model for Permedia4. It describes the interface conceptually rather than detailing specific registers and their exact usage. In-depth descriptions of how to program Permedia4 for specific drawing operations may be found in later chapters. Register specifications may be found in the *Permedi3 Reference Guide*.

3.1 Permedia4 as a Register file

Of the two I/O models (Message passing or Register file) of Permedia4 the simplest approach is to view it as a flat block of memory-mapped registers. This register file appears as part of Region 0 of the PCI address map[1]. When a Permedia4 host software driver is initialized it maps the register file into its address space.

Each register has an associated address tag giving its offset from the base of the register file. Since all registers reside on a 64-bit boundary, the tag offset is measured in multiples of 8 bytes. The obvious way to load a value into a register is to write the data to its mapped address.

In reality, the chip interface comprises a 32-entry deep FIFO, and each write to a register causes the written *value* and the register's *address tag* to be written as a new entry in the FIFO. Programming Permedia4 to draw a primitive then consists of writing initial values to the appropriate registers followed by a write to a command register. The last write triggers the start of rendering.

Permedia4 has more than 850 registers. These are individually defined in the *Permedia4 Reference Guide* (RG). All registers are 32 bits wide and should be 32-bit addressed. Many registers are paired to allow 64bit data handling or split into bit fields for mode setting.

Note: Bit 0 is the least significant bit.

In future chip revisions the register file may be extended and currently unused bits in certain registers may be assigned new meanings. Developers should ensure that only defined registers are written to and that undefined bits in registers are written as zeros. The exception to this rule is in some floating point registers where good practice is to use sign extended values.

Fields marked "not used" in register definitions are not usually being considered for internal use or development. Fields marked "reserved" are either used internally or reserved for future development.

3.1.1 Register Types

Permedia4 has three main types of register:

- Control Registers
- Command Registers

[1] See the *Permedia4 Reference Guide* (chapter 2) for more information about the PCI map.

- Internal Registers

3.1.1.1 Control Registers

Most registers are Control Registers. These are updated only by the host - Permedia4 effectively uses them as read-only registers. Examples of control registers are the Scissor Clip unit min and max registers (**ScissorMaxXY, ScissorMinXY**). Once initialized by the host, the chip only reads these registers to determine the scissor clip extents.

3.1.1.2 Command Registers

Command Registers are those which, when written to, start a rendering task (although some command registers such as **ResetPickResult** or **Sync** perform other tasks).

There are two types of command registers: begin-draw and continue-draw:

- Begin-draw commands start rendering with the values specified by the control registers.
- Continue draw commands cause drawing to continue with internal register values as they were when the previous drawing operation completed. Using continue-draw commands can significantly reduce the amount of data that has to be loaded when drawing multiple connected objects such as polylines.

Render and **ContinueNewLine** are typical command registers.

Note: For convenience in this document we often refer to "sending a Render command" rather than saying "the Render Command register is written to, which initiates drawing".

3.1.1.3 Internal Registers

Internal Registers are not writeable by host software. They are used internally by the chip to keep track of changing values. Some control registers have corresponding internal registers. When a begin-draw command is sent the internal registers are updated with the values in the corresponding control registers before rendering starts. If a continue-draw command is sent then this update does not happen and drawing continues with the current values in the internal registers.

For example, during line drawing the **StartXDom** and **StartY** control registers specify the (x, y) coordinates of the first point in the line. When a begin-draw command is sent these values are copied into internal registers. As the line drawing progresses these internal registers are updated to contain the (x, y) coordinates of the pixel being drawn. When drawing has completed the internal registers contain the (x, y) coordinates of the next point that would have been drawn. If a continue-draw command is now given these final (x, y) internal values are not modified and further drawing uses these values. If a begin-draw command had been used the internal registers would have been re-loaded from the StartXDom and StartY registers.

Internal registers can usually be ignored except when context switching. It is helpful to appreciate that they exist in order to understand the continue-draw commands.

3.1.2 Efficiency Issues and Register Types

Software developers wishing to write device drivers for Permedia4 should become familiar with the different types of registers. Some control registers such as the **StartX** and **StartY** registers have to be updated for almost every primitive whereas other control registers

such as the **ScissorMaxXY** or the **LogicalOpMode** can be updated much less frequently. Pre-loading of the appropriate control registers can reduce the amount of data that has to be loaded into the chip for a given primitive thus improving efficiency. In addition, as described above, the final values in internal registers can sometimes be used for subsequent drawing operations.

The cross-reference listing in the *Permedia4 Reference Guide* (Chapter 6) identifies the graphics registers by type.

Due to the structure of the internal HyperPipeline, when several graphics control registers are being loaded it is slightly more efficient to load them in pipeline order. For instance registers in the rasterizer should be loaded before registers in the GID/Stencil/Depth unit.

3.2 Permedia I/O Interface

There are two ways to program the graphics processor: either by writing to memory mapped registers, or by sending command packets to the input FIFO (which may be done by HostIn DMA or programmed IO). Both methods convert to tag/data "messages" that are sent to the pipeline, as described earlier[2].

3.2.1 FIFO control

The actual register is not updated until Permedia4 processes an entry. When the chip is busy performing a time consuming operation (e.g. drawing a large polygon), and not draining the FIFO very quickly, it is possible for the FIFO to become full. If a write to a register is performed when the FIFO is full no entry is put into the register.

3.2.1.1 PCI Disconnect

Setting bit 0 of the **FIFODiscon** register to 1 enables FIFO disconnection. It does not require host polling, but forces host write retries until the data is accepted. This may affect other time-critical peripherals on the PCI bus, e.g. sound cards, and sometimes drops interrupts. Despite the speed advantage it is therefore advisable to avoid this approach when large primitives are likely to slow FIFO clearing.

3.2.1.2 Polling InFIFOSpace

The input FIFO is 32 entries deep and each entry consists of a tag/data pair. The **InFIFOSpace** register can be read to determine how many entries are free. The value returned by this register will never be greater than 32[3].

An example of loading registers using the FIFO is given below. The pseudocode fills a series of rectangles. Details of the conventions used in the pseudocode examples may be found in Appendix B.

Assume that the data to draw a single rectangle consists of 8 words (including the **Render** command).

Note: Some data values are in 16.16 fixed point format.

```
for (i = 0; i < nrects; ++i) {
    while (*InFIFOSpace < 8);
```

[2] Chapter 2, section 3.

[3] When the InFIFOspace register is read, the value must be clamped to a maximum of 120 before it is used - refer to *Permedia4 Errata and Alerts*, PEREN0011.

```
        // wait for room
    StartXDom(rect->x1 << 16);
    StartXSub(rect->x2 << 16);
    dXDom(0x0);
    dXSub(0x0);
    Count(rect->y2 - rect->y1);
    YStart(rect->y1 << 16);
    dY(1 << 16);
    Render(PERMEDIA4_TRAPEZOID_PRIMITIVE);
        }
```

Checking the status of the FIFO before each write is inefficient so it is checked before loading the data for each complete rectangle. Since the FIFO is 32 entries deep, a further optimization is to wait for all 32 entries to be free after every second rectangle. Further optimization is possible by moving **dXDom**, **dXSub** and **dY** outside the loop (as they are constant for each rectangle) and doing the FIFO wait after every third rectangle.

The **InFIFOSpace** FIFO control register contains a count of the number of entries currently free in the FIFO. The chip increments this register for each entry it removes from the FIFO and decrements it every time the host puts an entry into the FIFO.

The graphics core registers cannot be read through the core FIFO interface. Command buffers generated to be sent to the FIFO interface can be read directly using the DMA controller.

3.2.1.3 Direct writes to the FIFO

In addition to writing to the register file you can send data directly to the core via the input FIFO. Any address in the PCI region 0 address map range can be read (normally a program will choose the first address and use this as the address for the output FIFO). All 32-bit addresses in this region perform the same function – the range of addresses is provided for data transfer schemes which force the use of incrementing addresses.

Writing to a location in this address range provides raw access to the input FIFO. Thus the same address can be used for both input and output FIFOs. Reading gives access to the output FIFO; writing gives access to the input FIFO.

When writing to a memory-mapped register the register file has a unique address for each register. This allows Permedia4 to construct the appropriate tag, associate it with the data and inserted the tag/data pair into the input FIFO.

When writing to the raw FIFO address an address tag description must first be written *followed by* the associated data. In fact, the format of the tag descriptions and their data is identical to that described below for DMA buffers[4]. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

Note: *When writing to the raw FIFO address the FIFO full condition must still be checked by reading the **InFIFOSpace** register. However, writing tag*

[4] Instead of using the Permedia4 DMA it is possible to transfer data to Permedia4 by constructing a DMA-style buffer of data and then copying each item in this buffer to the raw input FIFO address. Based on the tag descriptions and data written Permedia4 constructs tag/data pairs to enter as real FIFO entries.

descriptions does not put tag entries into the FIFO – it simply establishes a set of tags to be paired with the subsequent data.

Because direct writes to the FIFO do not place tag values in the FIFO itself, the FIFO space can be used for data only - free space need be ensured only for actual data items that are written (not the tag values). For example, where e.g. each tag is followed by a single data item, it would be possible to do 32 writes to an empty buffer before checking for free space.

See the *Permedia4 Reference Guide* for more details of the Graphics Processor FIFO Interface address range.

3.2.2 The DMA Interface

Loading registers directly via the FIFO can be an inefficient way to download data. The FIFO can accommodate only a small number of entries, so it has to be interrogated often to find out how much space is left. Also, if an API function requires a large amount of data then the function cannot return until almost all the data has been consumed. This may take some time depending on the types of primitives being drawn.

To avoid these problems Permedia4 provides an on-chip DMA controller which can be used to load data from arbitrary sized (< 64K 32-bit words) host buffers into the FIFO. The DMA may be either rectangular or sequential. The address of the latest DMA data read from memory (for the input DMA only) can be readback; this allows the progress of the DMA to be monitored and the DMA buffer reused as soon as it is free. The readback is resynchronizing so the pipeline does not have to be sync'd first.

3.2.2.1 DMA Operation

The normal mode of operation is to build a series of small DMA buffers in system memory and then send commands to the input FIFO to load them. These commands may be for complete DMA operations or continue operations to extend the current DMA. Continues are more efficient and should be used where possible.

Two DMA buffers can be maintained at the same time. If a new address is sent, it can be read from the FIFO before the current DMA completes. At chip reset the *BusMasterEnable* bit in the **CFGCommand** register must be set to allow DMA to operate (see the *Permedia4 Reference Guide* for further details). Then, for the simplest form of DMA, the host software has to prepare a host buffer containing register address tag descriptions and data values.

The host then writes the base address of this buffer to the **DMAAddress** register and the count of the number of words to transfer to the **DMACount** register. Writing to the **DMACount** register starts the DMA transfer and the host can now perform other work.

In general, if the complete set of rendering commands required by a given call to a driver function can be loaded into a single DMA buffer, then the driver function can return immediately. Meanwhile, in parallel, Permedia4 is reading data from the host buffer and loading it into its FIFO. In addition, some algorithms require that data be loaded multiple times (e.g. drawing the same object across multiple clipping rectangles). Since the DMA only reads the buffer data it can be downloaded many times simply by restarting the DMA. This can be very beneficial if composing the buffer data is a time consuming task.

An additional mechanism, the **HostInID** register, can be used to mark any point in the command stream so that the use of index and vertex buffers can be monitored. This is a register that is loaded with an ID field; like the DMA address register, this can be read at any time.

To prevent overflows the DMA controller automatically waits until there is room in the FIFO before doing any transfers. The only restriction on the use of DMA control registers is that before attempting to reload the **DMACount** register the host software must wait until previous DMA has completed. The **DMAAddress** register can, however, be reloaded while the previous DMA is in progress since the address is latched internally at the start of the DMA transfer.

3.2.2.2 Typical use

Many display driver functions can be implemented using the following skeleton structure:

```
do any pre-work
DMAAddress(address of dma_buffer);
while (*DMACount != 0);  // wait for DMA to complete
        // note use of backoff algorithm here
copy render data into DMA buffer
DMACount(number of words in DMA buffer)
return
```

A further optimization is to use a double buffered mechanism with two DMA buffers. This allows the second buffer to be filled before waiting for the previous DMA to complete thus further improving the parallelism between host and Permedia4 processing.

```
do any pre-work
get free DMA buffer and mark as in use
put render data into this new buffer
DMAAddress(address of new buffer)
while (*DMACount != 0)
    ;    // wait for DMA to complete
         // using a back off algorithm
DMACount(number of words in new buffer)
mark the old buffer as free
return
```

Double Buffering is discussed in greater detail in Chapter 4, section 10.

In general the DMA buffer format consists of a 32-bit address tag description word followed by one or more data words. The DMA buffer consists of one or more sets of these formats. The following paragraphs describe the different types of tag description words that can be used.

3.2.2.3 DMA Tag Description Format

When DMA is performed each 32-bit tag description in the DMA buffer conforms to the following format:

A packet is made up of a header followed by some number of data items. The format of the header is:

Bits	Field	Description
0-3	Offset	Index into 16 tags in each group
4-13	Group	16 tags usually grouped by association
14-15	Type	Type of packet, 0 = hold tag 1 = increment tag 2 = indexed tag 3 = reserved
16-31	Count or mask	Meaning changes with type of packet

Figure 3.1 DMA Tag Description Format

There are 3 different tag addressing modes for DMA: hold, increment and indexed. The different DMA modes are provided to reduce the amount of data which needs to be transferred, hence making better use of the available DMA bandwidth.

Hold: Tag formed from group and offset, kept constant for the next count + 1 data items.

Increment: Tag formed from group and offset, incremented by one for each of the next count + 1 data items.

Index: Tag formed from group (offset ignored) and mask showing which of the 16 tags in the group are valid. Tags formed in incrementing order and paired with the data.

These are described in *DMA Tags* below.

3.2.2.4 DMA and Security

To avoid accidental writes during DMAs to registers which can hang the entire graphics pipeline, Permedia4 supports a security mode. When the **Security** register *secure* bit is set the following tags are filtered out of DMA command buffers:

FilterMode

VTGAddress

VTGData

Security

DMARectangleWrite

DMAOutputCount

DMAFeedback

ContextDump

ContextRestore

ContextData

3.2.2.5 DMA Tags - Hold Format

In this format the 32-bit tag description contains a tag value and a count specifying the number of data words following in the buffer. The DMA controller writes each of the data words to the same address tag. This is useful e.g. for image download where pixel data is continuously written to the FrameBuffer. The bottom 11 bits specify the register to which the data should be written; the high-order 16 bits specify the number of data words (minus 1) which follow in the buffer and which should be written to the address tag

Note: The 2-bit mode field for this format is zero so a given tag value can simply be loaded into the low order 16 bits.

A special case of this format is where the top 16 bits are zero indicating that a single data value follows the tag (*i.e.* the 32-bit tag description is simply the address tag value itself). This allows simple DMA buffers to be constructed which consist of tag/data pairs. For example to render a horizontal span 10 pixels long starting from (2,5) the DMA buffer could look like this:

StartXDom
2 << 16
StartY
5 << 16
StartXSub
12 << 16
Count
1
Render
(trapezoid render command)

3.2.2.6 DMA Tags - Increment Format

address-tag with Count=n-1, Mode=1
value 1
...
value n

This format is similar to the Hold format except that as each data value is loaded the address tag is incremented (the value in the DMA buffer is not changed; Permedia4 updates an internal copy). Thus, this mode allows contiguous Permedia4 registers to be loaded by specifying a single 32-bit tag value followed by a data word for each register.

The low-order 11 bits specify the address tag of the first register to be loaded. The 2 bit mode field is set to 1 and the high-order 16 bits are set to the count (minus 1) of the number of registers to update. To enable use of this format, the Permedia4 register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 32 **AreaStipplePattern** registers can be loaded as follows:

AreaStipplePattern0, Count=31, Mode=1
row 0 bits
row 1 bits
...
row 31 bits

3.2.2.7 DMA Tags - Indexed Format

Permedia4 address tags are 11 bit values. For the purposes of the Indexed DMA Format they are organized into major groups and within each group there are up to 16 tags. The low-order 4 bits of a tag give its offset within the group. The high-order bits give the major group number. The *Permedia4 Reference Guide* (chapter 6) lists the individual registers with their Major Group and Offset.

This format allows up to 16 registers within a group to be loaded while still only specifying a single address tag description word.

address tag with Mask, Mode=2
value 1
...
value n

If the Mode of the address tag description word is set to Indexed mode then the high-order bits are used as a mask to indicate which registers within the group are to be used. The bottom 4 bits of the address tag description word are unused.

The group is specified by bits 4 to 8. Each bit in the mask is used to represent a unique tag within the group. If a bit is set then the corresponding register will be loaded. The number of bits set in the mask determines the number of data words that should be following the tag description word in the DMA buffer. The data is stored in order of increasing corresponding address tag. For example,

0x003280F0
value 1
value 2
value 3

The Mode bits are set to 2 so this is indexed mode. The Mask field (0x0032) has 3 bits set so there are three data words following the tag description word. Bits 1, 4 and 5 are set so the tag offsets are 1, 4 and 5. The major group is given by the bits 4-8 which are 0x0F (in indexed mode bits 0-3 are ignored). Thus the actual registers to update have address tags 0x0F1, 0x0F4 and 0x0F5 corresponding to registers dRdx, dGdx and dRdyDom. These are updated with value 1, value 2 and value 3 respectively.

3.2.2.8 DMA Example

The following pseudo-code example shows how to draw a series of rectangles using the DMA controller. This example uses a single DMA buffer and the simplest case which is Hold Mode for the tag description words in the buffer.

```
UINT32  *pbuf;
DMAAddress(physical address of dma_buffer)
while (*DMACount != 0)
    ;     // wait for DMA to complete
```

```
pbuf = dma_buffer;

*pbuf++ = Permedia4TagdXDom;
*pbuf++ = 0;
*pbuf++ = Permedia4TagdXSub;
*pbuf++ = 0;
*pbuf++ = Permedia4TagdY;
*pbuf++ = 1 << 16;
for (i = 0; i < nrects; ++i) {
    *pbuf++ = Permedia4TagStartXDom;
    *pbuf++ = rect->x1 << 16; // Start dominant edge
    *pbuf++ = Permedia4TagStartXSub
    *pbuf++ = rect->x2 << 16; // Start of subordinate
    *pbuf++ = Permedia4TagCount;
    *pbuf++ = rect->y2 - rect->y1;
    *pbuf++ = Permedia4TagYStart;
    *pbuf++ = rect->y1 << 16;
    *pbuf++ = Permedia4TagRender;
    *pbuf++ = Permedia4_TRAPEZOID_PRIMITIVE;
}
// initiate DMA
DMACount((int)(pbuf - dma_buffer))
```

The example assumes that a host buffer has been previously allocated and is pointed at by "dma_buffer".

3.2.2.9 DMA Buffer Addresses

Host software must generate the correct DMA buffer address for the DMA controller. Normally this means that the address passed to Permedia4 must be the physical address of the DMA buffer in host memory. The buffer must also reside at contiguous physical addresses as accessed by Permedia4. On a system which uses virtual memory for the address space of a task, some method of allocating and mapping contiguous physical memory within this space must be used.

Note: *This does not apply to Virtual Texturing using the TextureDownloadControl register with SlaveTexture enabled for DMA texel data. Texture pages are relocatable and do not need to be contiguous in local memory.*

If the virtual memory buffer maps to non-contiguous physical memory then the buffer must be divided into sets of contiguous physical memory pages and each of these sets transferred separately. In such a situation the whole DMA buffer cannot be transferred in one go; the host software must wait for each set to be transferred. Often the best way to handle these fragmented transfers is via an interrupt handler.

3.2.2.10 DMA Interrupts

Permedia4 provides interrupt support as an alternative means of determining when a DMA transfer is complete. If enabled, the interrupt is generated whenever the **DMACount** register changes from having a non-zero to having a zero value. Since the **DMACount** register is decremented every time a data item is transferred from the DMA buffer this happens when the last data item is transferred from the DMA buffer:

```
case CommandInterruptTag:
    if (Data != 0)
    {
            // Assert interrupt at end of current output DMA.
            if (CheckFifoEmpty(PciWriteControlFifo))
            {
                    HICommandInterrupt = True;
                    Code = True;
            }
            else
                    Code = False;
    }
    else
    {
            HICommandInterrupt = True;
            Code = True;
    }
    break;
```

To enable the DMA interrupt, the *OutputDMA* bit must be set in the **CommandDMA** register. The interrupt handler should also check the *ControlDMA* flag bit in the **IntFlags** register to determine that a DMA interrupt has actually occurred. To clear the interrupt, reset the flag by writing a 1 to the **IntFlags** *ControlDMA* flag bit.

A typical use of DMA interrupts might be as follows:

```
prepare DMA buffer
DMACount(n);            // start a DMA transfer
prepare next DMA buffer
while (*DMACount != 0) {
    mask interrupts
    set DMA Interrupt Enable bit in IntEnable register
    sleep on interrupt handler wake up
    unmask interrupts
}
```

```
DMACount(n)          // start the next DMA sequence
```

The interrupt handler could then be:

```
if (*IntFlags & DMA Flag bit) {
    reset DMA Flag bit in IntFlags
    send wake up to main task
}
```

Interrupts are complicated and depend on the facilities provided by the host operating system. The above pseudocode only hints at the system details.

This scheme frees the host processor for other work while DMA is being completed. Since the overhead of handling an interrupt is often quite high for the host processor, the scheme should be tuned to allow a period of polling before sleeping on the interrupt.

3.2.2.11 Using Run Length Encoding

Image data frequently contains runs of the same pixel data. Run Length Encoding (RLE) is a convenient image compression method which counts adjacent identical pixel values instead of replicating them. This does not speed up the image transfer from the core's viewpoint (it still needs to read the data) but it reduces the amount of data carried over the PCI bus and, potentially, the host effort in processing/copying the image data.

When run length encoding is enabled then any data (but not tags) which matches the 32 bit run length value is added to the run length count instead of being written to the FIFO. The accumulated run length is written to the FIFO when:

- The new 32 bit word is different from the run being encoded.
- A new scanline is started.
- The end of the primitive occurs.

The amount of data produced during the run length encoding is not known when the DMA controller is set up so an alternative mechanism is used to tell the DMA controller the upload data has finished. This is done by using the **EndOfFeedback** command. When detected in the DMA controller the DMA can be terminated as all the data has been received.

Tags are not included in the FIFO and an extra bit (on the FIFO width) is used to tell the DMA controller to finish. This bit is only set when an **EndOfFeedback** tag is received *during RLE processing*. This allows run length data to be uploaded at twice the rate available with an external DMA controller.

Note: There is the potential for the software to hang if the Permedia4 Output DMA controller is in feedback mode and the RunLengthEncodeData *bit is not set.*

If the RLE bit is not set during Feedback mode DMA, the **EndOfFeedback** tag is ignored by the DMA controller, hence the software will not be informed the upload has finished. The graphics core, etc. will continue to function as normal but the Output DMA controller will loop forever discarding data once the buffer count has expired. If the software is running with a time-out the Output DMA controller can be recovered by setting the *RunLengthEncodedData* bit in the **FilterMode** register and sending an **EndOfFeedback**.

Note: This situation only arises during Feedback Mode DMA with RLE, since in all other cases the amount of data to load is known.

Software must make allowance for the fact than run length encoding is not guaranteed to result in a smaller representation of the image and in the worst case could double the size.

Run length encoding is done 32 bits at a time irrespective of the pixel depth and is masked by the RLEMask before the comparison is done. This allows bits in the data to be excluded from the test, e.g. because they are unused and have 'random' values. The masked data is returned.

3.2.3 Vertex Loading for Primitives and Data Re-ordering

The I/O units support various strategies to make data loading more efficient including early culling, vertex loading and texture formatting, described below.

3.2.3.1 Primitives

Further I/O pre-processing may be done if the data represents a primitive. Vertex data for primitives can be loaded into the Permedia4 graphics memory space in a number of ways. It can be:

- written directly to the appropriate vertex store
- loaded indirectly from an index into a vertex array, or
- loaded as part of a special vertex sequence corresponding to one of the following primitive types:

Name	Description
TriangleList	Individual triangles with no shared vertices.
TriangleFan	Triangles with a common center vertex, and another shared vertex between each adjacent triangle.
TriangleStrip	Triangles with two shared vertices between adjacent triangles.
LineList	Individual lines with no shared vertices.
LineStrip	Lines joined head to tail.
PointList	Individual points

Each of these is available in an indexed form (e.g. **IndexedTriangleList**) and a vertex form (e.g **VertexTriangleList**). The indexed form specifies vertices through an index into a vertex array; the vertex form specifies a vertex array directly, so the vertices must be in the correct order.

The register tag marks the start of a primitive and the data field is a count of the number of vertices to be processed. If the number is zero then the data is assumed to be inline and the primitive is terminated by another primitive command.

IndexedVertex or **IndexedDoubleVertex** can be used as inline indices into a vertex array by first sending the **IndexedTriangleList** primitive (for example) with data equal to zero. **IndexedVertex** supplies a single 32 bit index into the array, **IndexedDoubleVertex** supplies two 16 bit indices into the vertex array.

The vertex data can be loaded without specifying a primitive type using **Vertex0**, **Vertex1**, and **Vertex2.** These registers specify the vertex store to load, and the data field holds the index into the array. The **VertexData** register is used for inline vertex data.

The format of the vertex is specified by four controls:

- The vertex size controls the amount of memory each vertex takes (i.e. the Stride).

- The tag list describes the order of the data within the vertex, so the tag loaded into TagList0 defines the data type of the first entry in the vertex.
- The **VertexFormat** mask controls which data elements are present, so if bit 0 is clear but bit 1 set, the first data element read is associated with TagList1 instead of TagList0.
- The **VertexValid** mask specifies which items of data within the current vertex should be read.

Typically we would use the tag list to define the order in which data is delivered. The format mask and vertex size set modes (so if z is enabled the z bit in the format mask is set and the vertex size increased by 1).

The **VertexValid** mask is generally used for multi-pass algorithms, such as emulation of multiple textures. The vertex structure holds several sets of texture coordinates for the same x,y,z coordinates, and on each pass a different set is enabled by defining it as valid. Because invalid data is discarded the format only needs to recognize a single field (although the vertex size must be big enough to include all data).

A state register supports context switching within a primitive. This allows a primitive such as a triangle strip to be interrupted part way through and then restarted cleanly. If data is being read by DMA it completes before a sync can get through the pipeline: this applies to single vertices read in by inline indexes and to multiple vertices read in via index arrays.

To restart an interrupted primitive the **HostInState** register should be restored along with the rest of the pipeline and the primitive continued from the point at which it was broken.

A simple 2D line primitive is enabled by setting the *Line2D* bit in the **VertexControl** register. This works in conjunction with **LineList** or **LineStrip** and the **LineCoord01** and **LineCoord10** registers. Only XY data (packed in one word) can be sent in this mode. The contents of the tag list are ignored.

3.2.3.2 Data Re-ordering

In addition to vertex processing, I/O processing supports data re-ordering for a write combined FIFO. Up to 64 bytes (16x32 bit words) can be sent out of order, stored, and issued in the correct order. Write combined acesses are identified by the address space they are sent to, and this is passed down the input FIFO by a flag. Typically:

```
// Special case input fifo including write combined flag.
struct InputFifo
{
    INT32       Data[32][4];
    INT11       Tag[32][4];
    INT1        WriteCombined[32][4];
    INT1        Enables[32][4];
};
```

A counter cycles through the the registers trying to write them out in ascending order. While the next register required is flagged "not valid" the process locks.

Note: This mechanism assumes that the maximum amount of write buffering in the CPU is 64 bytes and that the if multiple buffers are supported they are flushed according to address.

3.2.4 Backface Cull and Texture Setup Functions

In addition to vertex loading and re-ordering, it is possible to cull triangles facing away from the viewer to avoid unnecessary rendering overheads[5]. This is controlled by the *backfacecull* bit in the **DeltaMode** register and by the *TextureEnable* and *RejectNegativeFace* bits in the **DrawTriangle** command.

Note: Backface cull was supported in Permdia3, but in Permedia4 it is implemented ahead of the delta unit to avoid loading vertex data for a triangle that will subsequently be culled.

A typical backface cull algorithm would be:

```
ScreenArea = dXac * dYbc – dXbc * dYac;
    if (ScreenArea is negative or zero && RejectNegativeFace)
        abort processing and discard draw command;
    else if (ScreenArea is positive or zero && !RejectNegativeFace)
        abort processing and discard draw command;
    else
        continue processing;
    where:
    dXac = a.x – c.x;    // etc for each parameter.
```

3.2.4.1 Texture Setup

When texture is enabled in the **DeltaMode** register and the relevant **Draw** command, Permedia4 can also perform some texture optimization functions:

- ForceQ - used in unusual cases where the application does not want perspective correction of textures and either does not supply equal Q values or supplies no Q at all. Permedia4 either over-rides or makes up the numbers as appropriate.
- EqualQ - used to avoid rounding problems where the chip encounters calculations of the form 1/(1/x) and x isn't a power of 2.
- Share S,T,Q - used to provide S, T or Q values for the second texture when multitexturing if the application only supplies data for texture 0 but indicates that both textures should use the same set of texture coordinates.

These functions are enabled in the DeltaFormatControl register - for details see the *Permedia4 Reference Guide* volume II.

3.2.4.2 Texture Wrap

Generally, textures are interpolated linearly from start value to end value under Direct3D rules. (For a general description of Texture Interpolation and Level of Detail (LOD) principles please refer to the *Permedia4 Programmer's Guide* volume II, section 5.2) For

[5] A D3D compliance requirement

cylindrical objects, texture interpolation neds to take the shortest distance from start value to end value.

The implementation looks at the difference in texture co-ordinates along the one edge and if it is greater than ½ then 1.0 is subtracted from the start value. If the difference is less than –½ then 1.0 is added to the start value. The effect of these operations is to move the start value to the 'other side' of the end value, so the direction of interpolation is flipped. The operation is repeated for n-1 sides, where n is the number of vertices in the primitive.

Note: *Tthis relies on texture units downstream handling co-ordinates outside the range 0 to 1 and repeating the texture accordingly. This is done for each set of texture co-ordinates individually.*

The algorithm for triangles is:

```
if ((a.s – c.s) > 0.5)
    a.s -= 1.0;
else if ((a.s – c.s) < -0.5)
    a.s += 1.0;

if ((b.s – c.s) > 0.5)
    b.s -= 1.0;
else if ((b.s – c.s) < -0.5)
    b.s += 1.0;
```

The algorithm for lines is:

```
if ((a.s – b.s) > 0.5)
    a.s -= 1.0;
else if ((a.s – b.s) < -0.5)
    a.s += 1.0;
```

Texture co-ordinates for points are not modified.

3.2.4.3 Per-poly Mipmapping

Polygon Mipmapping generates a Level of Detail (LOD) value for the x,y area of an entire triangle. This requires determining a ratio between the triangle area and the projected area in the texture map taken from the S,T coordinates. The ratio determines the LOD value which is sent to the downstream texture units.

The calculation is as shown below.

Note: *The triangle x,y area has already been calculated if Backface Cull is enabled.*

```
ScreenArea = dXac * dYbc – dXbc * dYac;
TextureArea = (dSac * dTbc – dSbc * dTac) * TextureLODScale;
LOD = (exponent of texture area – exponent of screen area) / 2;
```

```
if (LOD < 0)
    LOD = 0;
if (LOD > 0xF)
    LOD = 0xF;
// LOD is 4.8 format, fraction always set to zero.
LOD <<= 8;
where:
dXac = a.x – c.x;    // etc for each parameter.
repeat for S1 and T1 using TextureLODScale1.
```

The LOD is clamped to lie between 0 and 15 inclusive.

Note *Subtracting the exponents is an easy way of getting the ratio of the two values. The divide by 2 is applied because logs of areas are being compared so it is equivalent of a square root.*

TextureLODScale is used to control the appearance of the texture; making it larger makes the texture over-filtered, making it smaller makes the texture under-filtered. This operation is done to both sets of texture co-ordinates (if enabled) and there is a separate **TextureLODScale1** for S1 and T1. For a general discussion of LOD interpolation refer to Volume II, section 5.2.

When the LOD has been calculated it is sent to the texture coord unit in either the LOD (for S and T) or LOD1 (for S1 and T1) registers as appropriate.

3.2.4.4 Texture Shift

Texture Shift is a means of recovering significance bits from the integer of a texture to the fraction part where the integer has become very large relative to the range (e.g. 70.0 to 70.5) but is actually redundant because the texture repeats.

The shift function finds the vertex with the largest Q and subtracts the integer parts of those S,T texture coordinates from the other vertices' coordinates. The integer part is rounded to an even value to avoid interference with mirrored textures (refer to Volume II, section 5.2.2.1, Texture Coordinate Wrapping Modes. Texture Shift is enabled by the **DeltaFormatControl** register *TextureShift* bit.

3.2.4.5 Scale by Q

This function multipleis the S,T values for each vertex of a triangle by the vertex's Q value. This anticipates the Divide by Q operation performed later during perspective correction.

3.3 Output FIFO

An output FIFO provides data readback from Permedia4. Each entry in this FIFO is 32 bits wide and can hold tag or data values. This differs from the input FIFO whose entries are always tag/data pairs (effectively 41 bits – 9 bits for the tag and 32 bits for the data).

The **FilterMode** register controls the type of data written to the output FIFO.. This register allows filtering of output data in various categories including the following:

- Depth: output in this category results from an image upload of the Depth buffer.
- Stencil: output in this category results from an image upload of the Stencil buffer.

- Color: output in this category results from an image upload of the framebuffer.
- Synchronization: synchronization data is sent in response to a **Sync** command.

The **FilterMode** register uses 2 bits for each category. Setting the least significant bit (0x1) enables output of the register ***tag*** for that category. Setting the most significant bit (0x2) enables output of the ***data*** for that category. If both tag and data output are enabled the tag is written first to the FIFO followed by the data. The **FilterMode** register is described in more detail in Chapter 5 - Graphics Programming.

For example, to perform an image upload from the framebuffer the **FilterMode** register should have output enabled for the *FBColorData* bit. Then the rectangular area to be uploaded should be described to the rasterizer. Each pixel read from the framebuffer is then placed into the output FIFO. If the output FIFO becomes full Permedia4 blocks internally until space becomes available.

It is the programmer's responsibility to read all data from the output FIFO. For example, it is important to know how many pixels should result from an image upload and to read exactly that many from the FIFO.

To read data from the output FIFO the **OutFIFOWords** register should first be read to establish the number of entries in the FIFO (reading from the FIFO when it is empty returns undefined data). Then read the same number of 32-bit data items from the FIFO. Repeat the procedure until all the expected data or tag items have been read. The address of the output FIFO is described below.

Note: *All expected data must be read back. The graphics core will block if the FIFO becomes full. Programmers must be careful to avoid the deadlock condition that will result if the host is waiting for space to become free in the input FIFO while Permedia4 is waiting for the host to read data from the output FIFO.*

3.4 Other Interrupts

Permedia4 also provides interrupt facilities for the following:

- Sync: If a Sync command is sent and the Sync interrupt has been enabled then once all rendering has been completed, a data value is entered into the Host Out FIFO, and a Sync interrupt is generated when this value reaches the output end of the FIFO. Synchronization is described further in the next section.
- External: this provides the capability for external hardware on a Permedia4 board (such as an external video timing generator) to generate interrupts to the host processor.
- Error: if enabled the error interrupt will occur when Permedia4 detects certain error conditions, such as an attempt to write to a full FIFO.
- Vertical Retrace: if enabled a vertical retrace interrupt is generated at the start of the video blank period.

Each of these are enabled and cleared in a similar way to the DMA interrupt. See **IntEnable** in the *Permedia4 Reference Guide* for more details.

3.5 Synchronization

There are three main cases where the host must synchronize with the graphics core:

1. before reading back from registers

2. before directly accessing the framebuffer or the localbuffer via the bypass mechanism
3. framebuffer management tasks such as double buffering (though this may be better handled using the **SuspendUntilFrameBlank** command).[6]

Synchronizing with Permedia4 implies waiting for any pending DMA to complete ***and*** waiting for the chip to complete any processing currently being performed. The following pseudo-code shows the general scheme:

```
Permedia4Data data;
// wait for DMA to complete
while (*DMACount != 0) {
      poll or wait for interrupt
}
while (*InFIFOSpace < 2) {
      ;      // wait for free space in the FIFO
}
// enable sync output and send the Sync command
data.Word = 0;
data.FilterMode Synchronization = 0x1;
FilterMode(data.Word);
Sync(0x0);

/* wait for the sync output data */
do {
      while (*OutFIFOWords == 0)
            ;                  // poll waiting for data in output FIFO
} while (*OutputFIFO != Sync_tag);
```

Initially we wait for DMA to complete as normal. We then also have to wait for space to become free in the FIFO (since the DMA controller actually loads the FIFO). We need space for 2 registers: one to enable generation of an output sync value, and the **Sync** command itself. The enable flag can be set at initialization time. The output value will be generated only when a **Sync** command has actually been sent, and Permedia4 has then completed all processing.

Rather than polling it is possible to use a **Sync** interrupt as mentioned in the previous section. As well as enabling the interrupt and setting the filter mode, the data sent in the **Sync** command must have the most significant bit set in order to generate the interrupt. The interrupt is generated when the tag or data reaches the output end of the Host Out FIFO. Use of the Sync interrupt has to be considered carefully as Permedia4 will generally empty the FIFO more quickly than it takes to set up and handle the interrupt.

[6] And ref *Permedia4 Errata and Alerts*, PEREN004 - Check FIFO space before sending HostIn DMA commands.

3.6 Host Framebuffer Bypass

Normally, the host will access the framebuffer indirectly via commands sent to the FIFO interface. However, Permedia4 maps the complete framebuffer as a linear 32-bit addressable memory region. Access to the framebuffer via this memory mapped route is independent of the Permedia4 FIFO.

Drivers may choose to use direct access to the framebuffer for algorithms which are not supported by Permedia4. The framebuffer bypass supports big-endian, little-endian and GIB-endian formats. These are described in a later section.

A driver making use of the framebuffer bypass mechanism should synchronize framebuffer accesses made through the FIFO with those made directly through the memory map. If data is written to the FIFO followed by a direct access to the framebuffer, the FIFO may not have had a chance to complete before the bypass access happens.

Once mapped in, the framebuffer can be read or written with 8, 16 or 32-bit accesses. Permedia4 does not use bank switching since it is a PCI device and the PCI bus provides a 32 bit address space[7].

The framebuffer is accessible via Regions 1 and 2 of the PCI address map.!

3.6.1 Framebuffer Dimensions and Depth

Address calculation is controlled by the **FBWriteMode** register and the address is a function of X, Y, **FBWriteBufferAddr**, **FBWriteBufferOffset**, **FBWriteBufferWidth** and **PixelSize** parameters. The X and Y parameters are passed in the active fragment or span registers. The Address, Offset and Width are specified independently for each of the write buffers defined in **LBWriteFormat**.

The address calculation is further modified by the mode bits: *Origin*, *StripePitch*, *StripeHeight* and *Layout*.

Once the framebuffer parameters have been defined, determining the visible screen width and height becomes a clipping issue. The visible screen width and height are set up in the **ScreenSize** register and enabled by setting the *ScreenScissorEnable* bit in the **ScissorMode** register.

The framebuffer depth (8, 16 or 32-bit) is controlled by the **PixelSize** register. This register provides a 2 bit field to control which of the three pixel depths is being used. The pixel depth can be changed at any time without the need for any synchronization but should normally be set at initialization.[8]

3.7 Host Localbuffer Bypass

As with the framebuffer, the localbuffer can be mapped in and accessed directly. The host should synchronize with Permedia4 before making any direct access to the localbuffer.

In Permedia4, Localbuffer access is defined as reads from up to four mappable destinations, or writes to a single mappable destination. Addresses for reads and writes

---

[7] On address limited buses such as ISA, devices limit the amount of address space that they occupy by using bank switching hardware. This typically provides a 64K byte window through which part of the framebuffer is visible. The region 0 Video Stream hardware registers define which part of the framebuffer is visible through this window - refer to the *Permedia4 Reference Guide*, chapter 4.

[8] On earlier systems without span operation it was useful to change the pixel depth temporarily to optimize some 2D rendering operations. This is no longer necessary but code written to use the technique will continue to work provided pixel size is set using the **PixelSize** register.

are managed locally, i.e. the base address, offset value and width are not shared with the read address generation**.**

Localbuffer management facilities are described in section 4.2 - Localbuffer. The localbuffer is accessible via Regions 1 and 3 of the PCI address map. Localbuffer bypass supports big-endian and little-endian formats, described in *Byte Swapping*, below.

3.8 Register Read back and Context Dump/Restore

In some operating environments multiple tasks want access to the Permedia4 chip. Sometimes a server task or driver wants to arbitrate access to Permedia4 on behalf of multiple applications. In other situations a host process may wish to extract specific information on the fly, usually for diagnostic purposes. In both these cases register readback is required, but in the former a simple global context switch is involved, while in the other we want selective readback.

3.8.1 Context Dump/Restore

To perform a context switch the host must first synchronize with Permedia4. This means waiting for any outstanding DMAs to complete, sending a **Sync** command and waiting for the sync output data to appear in the output FIFO.

In Permedia4, Context switching is supported by a new dedicated register group: **ContextData**, **ContextDump**, and **ContextRestore.**

The **ContextDump** command forces the Permedia4 to dump the selected context. Context switching can be done on any command boundary but not during internal processing or texture/image downloads.

Note: *Care must be taken when context switching a task which is making use of continue-draw commands. Continue-draw commands rely on the internal registers maintaining previous state. Any rendering work for a new task destroys the previous state. To prevent this, continue-draw commands should be performed via DMA where possible since the context switch code has to wait for outstanding DMA to complete. Alternatively, continue-draw commands can be performed in a non-preemptable code segment.*

The context is dumped over the full length of the graphics pipeline by the **ContextDump** command and restored by the **ContextRestore** command.

The data sent with this command (the context mask) dictates what subset of the full context is to be dumped. The context for each unit is defined by the Context Mask. It appears in the Host Output FIFO tagged as **ContextData** where the host of the output DMA controller can read it. The amount of data sent depends on the context mask sent with the command. The last tag and data sent to the FIFO is the **ContextDump** tag and mask, but this is not included in the word counts above.

Usually, the intention is to restore a context exactly as it was saved earlier. For paired context dump and restore operations the same mask is used for both dump and restore commands.

For further information see the **ContextRestore, EndofFeedback, FilterMode** and **ContextData** register descriptions in the *Permedia4 Reference Guide*.

3.8.2 Register Readback

To read a Permedia4 register the host reads the same address which would be used for a write, i.e. the base address of the register file plus the offset value for the register. See the warning in *Context Dump*, above, about state preservation with **Continue** commands.

Normally, reading back individual registers should be avoided. The need to synchronize with the chip can adversely affect performance. It is usually more efficient to keep a software copy of the register which is updated when the actual register is updated.

3.9 Byte Swapping

Internally Permedia4 operates in little-endian mode. However, it is designed to work with both big- and little-endian host processors. Since the PCI Bus specification defines that byte ordering is preserved regardless of the size of the transfer operation, Permedia4 provides facilities to handle byte swapping. The choice of mapping typically depends on the endian ordering of the host processor.

The Control Space in PCI address region 0, is 128K bytes in size and consists of two 64K spaces. The first 64K provides little endian access to the control space registers; the second 64K provides big endian access to the same registers (or, alternatively, can be configured as a Write-Combine buffer).

The Configuration Space may be set using reset pin VSBData(3) [9] to be either little endian or big endian. The same effect can be achieved by resetting the *WCEnable* bit in the **ChipConfig** register.

Additional support is provided within Permedia4 to byte swap images and bitmasks as they are transferred to and from the host, in particular:

- ControlDMAControl
- ByApertureOneMode / ByApertureTwoMode
- ByDMAReadMode
- ByDMAWriteMode
- DMARectangleRead
- DMARectangleWrite
- RasterizerMode
- TextureReadMode

Byte swapping may be on a 32-bit word, 16-bit word or byte offset depending on the register and use. See the *Permedia4 Reference Guide*, chapter 4, for more details of these control registers.

3.10 Red and Blue Color Ordering

For a specific graphics board the RAMDAC and/or API will usually force an interpretation for true color pixel values. For example, 32-bit pixels will be interpreted as either ARGB (alpha at byte 3, red at byte 2, green at byte 1 and blue at byte 0) or ABGR (blue at byte 2 and red at byte 0). The byte position for red and blue may be important for software which has been written to expect one byte order or the other, in particular when handling image data stored in a file.

Permedia4 allows color ordering to be specified in a number of registers along the pipeline. The **AlphaBlendColorMode** register contains a 1-bit field called *ColorOrder*. If

[9] ...where 0 = Upper half of region Zero is byte-swapped, or 1= Upper half of region Zero flagged internally as write-combined.

this bit is set to zero then the byte ordering is BGR; if the bit is set to one then the ordering is RGB. As well as setting this bit for AlphaBlend work it must also normally be set in the Color Formatting unit, though in some cases it may be useful to set them differently. The **DitherMode** register contains a *ColorOrder* bit with the same interpretation. The order applies to all of the RGB pixel formats, regardless of the pixel depth.

Image and bitmask data can also be optionally byte/word swapped as part of the download process by setting the appropriate bit in the **RasterizerMode** register. Similarly the LUTMode register allows extensive color order and format conversion. Finally image data can be optionally byte/word swapped by setting the appropriate bit in the **FilterMode** register of the Host Out unit. These operations are controlled independently of DMA byte swapping operations.

4

Buffer and Cache Management

4.1 Introduction

This chapter provides an overview of local and frame buffer implementation and describes virtual texture mapping and double buffering.

4.2 Localbuffer (LB)

The LB Read Unit is connected to the Memory Controller interface to the memory subsystem via a FIFO. The Permedia4 data bus width in the LB Memory is 128 bits, as it is for the Framebuffer.

The information passed includes:

- The target memory (FB, LB or PCI)
- The required control/write action
- The write address
- Any Byte Enables (see *LBDestReadEnables* in the *Permedia4 Reference Guide*)
- Data to be written

The localbuffer holds per-pixel information corresponding to each displayed pixel:

- Graphic ID (GID)
- Depth ("Z")
- Stencil

...which may be handled as pixel read/writes or as spans of 4 consecutive words.

The values are referred to as fields because they contain pixel parameter data in a structured format. Conceptually, they can be thought of as buffers or planes. The possible formats for each of these and their use are described below.

4.2.1 Localbuffer Management

With unified Permedia4 memory the Localbuffer can be considered a functional package rather than a predefined area of memory or distinct device type. Specifically, localbuffer functions include:

- Managing the read from a destination memory buffer,
- Managing the read from a source memory buffer,
- Managing the updates to the buffer,
- Calculating the write address of the fragment in the memory,
- Combining multiple fragments in the same memory word,
- Calculating the write addresses of the spans in the memory,
- Aligning span clears and issuing multiple normal writes,
- Calculating the read address(es) of the fragment in the memory,
- Caching memory data to reduce actual memory accesses,
- Prefetching data when possible,

- Calculating the read addresses of the spans in the memory,
- GID testing single pixels (from steps) or multiple pixels (spans)
- Interfacing to the Memory Controller,
- Formatting memory data into a standard Z, stencil and GID format.

An optimization in Stencil and Depth functionality avoids unnecessary writes when the **LBWrite** contents are the same as those currently in memory.

4.2.2 Layout

The **LBWriteMode** *Layout* field selects how data is to be laid out in memory. The options are:

- Linear. Here the rows are stored contiguously, one after another in memory.
- Patched64. In this layout the pixel data is arranged into 64x16 pixel "patches" or tiles if the buffer depth is 32 bpp, up to 128x16 pixels for 16 bpp buffer configurations. This is the preferred layout for the depth, etc. buffer to give equal access times for X and Y paths. Note the memory page size is 1Kx32 bits so patching keeps page loads to a minimum. This also matches a patch format used for Framebuffer functions.

Localbuffer memory itself can be from 16 bits (assuming a depth plane is always needed) to 48 bits wide in steps of 4 bits. Each location in one field/plane is contiguous with the corresponding location in the adjacent field/plane; for example, GID_0, $Depth_0$, $Stencil_0$, GID_1, $Depth_1$, $Stencil_1$ etc. The allowed lengths and positions of the fields supported in the localbuffer are shown in the **LBReadFormat** register description below:

Name	Type	Offset	Format
LBReadFormat	Localbuffer	0x8888	Bitfield
	Control register		

Bits	Name	Read	Write	Reset	Description
0…1	DepthWidth	✔	✔	x	This field specifies the width of the depth field. The depth field always starts at bit position 0. The width options are: 0 = 16 bits 1 = 24 bits 2 = 31 bits 3 = 15 bits When the depth width is 15 the GID and Stencil fields are ignored and a one bit GID and Stencil are taken from bit 15. Only one of the GID or Stencil operation are enabled to select the desired field type.
2…5	StencilWidth	✔	✔	x	This field specifies the width of the stencil field. The legal range of values are 0…8. The stencil field always starts at bit position given in the next field.
6…10	StencilPosition	✔	✔	x	This field holds position of the least significant bit of the stencil field. The legal range of values are 0…23, representing bit positions 16…39 respectively.
11…14	FCPWidth	0	0	x	Reserved
15…19	FCPPosition	0	0	x	Reserved
20…22	GIDWidth	✔	✔	x	This field specifies the width of the Graphics ID field. The legal range of values are 0…4. The GID field always starts at bit position given in the next field.

23…27	GIDPosition	✔	✔	x	This field holds position of the least significant bit of the Graphics ID field. The legal range of values are 0…23, representing bit positions 16…39 respectively.

The order of the planes/fields is as shown above with the depth field at the least significant end and GID field at the most significant end. The GID is at the most significant end so that various combinations of the Stencil and depth field widths can be used on a per window basis without the position of the GID fields moving. If the GID field is in different positions in different windows then ownership tests become impossible to do.

4.2.3 Pixel Formats

See volume II, section 2.1.7 - Pixel Sizes.

4.2.4 Pixels and Spans

Operations are either single pixel or pixel spans. For a more general discussion of upload, download and copy for fragments, bitmasks and spans see Volume II, section 1.1.8 - Bitmaps, Spans and Images.

The LB supports two forms of write request:

- Single pixel. This is the normal mode for 3D operation but is only used for exotic 2D operations. The calculated address is always a pixel address and this is shifted to take into account the width of a pixel (8, 16 or 32 bits) in calculating the memory address and byte enables. The pixel data is always right justified within the 32 data bus on input to this unit so is shifted into the correct byte lanes for the memory. Successive writes to the same memory word (128 bits) are combined so only a single memory write access is done. The memory write is done when the new fragment is for a new memory address, or the **Render**, **SuspendReads** or a mode change command intervenes.
- Pixel spans. Span operations can be used to write multiple pixels simultaneously in the local buffer. A span is a group of 64 consecutive pixels starting at the span start address and at rising addresses. The data written is constant for the span and held in the **LBClearDataU** and **LBClearDataL** registers. The data is replicated to a four pixel memory word. In Packed16 mode (8 pixels per word) software must replicate the 16 bits of constant data into the 32bit **LBClearDataL** register. Byte masking is possible using the **LBWriteMode** byte enables, provided the field to clear is an aligned multiple of bytes. If not (e.g. a 3-bit stencil field) then selective clearing requires a read-modify-write loop.

4.2.5 Clearing the Localbuffer using FBWrite

Spans are useful for clearing down the local buffer but do not use any block fill capabilities of the memory. However Block fill for SGRAM (or its emulation for SDRAM) is available using Framebuffer Write functionality or emulated in the LB Memory interface. Write mask data is ignored by the LB and requires FB functions. If the local buffer is implemented in SGRAM then the FB write mask has a resolution to the bit level so a field with any width or position can be masked to prevent it being cleared during a write only operation (using the FBWrite registers).

Span operations can also be used to write multiple pixels simultaneously in the local buffer as described in section 4.2.4 - Pixels and Spans - above. Span operation is enabled using the *FastFillEnable* bit in the **Render** register. For a more general discussion of upload,

download and copy for fragments, bitmasks and spans see Volume II, section 1.1.8 - Bitmaps, Spans and Images.

4.2.6 GID field

The GID field may optionally be used for pixel ownership tests to allow per pixel window clipping. Each window using this facility is assigned one of the GID values, and the visible pixels in the window have their GID field set to this value. If the test is enabled the current GID (set to correspond with the current window) is compared with the GID in the localbuffer for each relevant fragment. If they are equal this pixel belongs to the window so the localbuffer and framebuffer at this coordinate may be updated. If not, the pixel is left unchanged.

If the *GIDWidth* is set to its maximum (4) only 16 GID values are available which limits the number of open windows which can be tested. Some other methods of achieving the same result are:

- clip the primitive to the window's boundary (or rectangular tiles which make up the window's area) and render only the visible parts of the primitive
- use the scissor test to define the rectangular tiles which make up the window's visible area and render the primitive once per tile (This may be limited to only those tiles which the primitive intersects).

It is possible to apply GID testing to LB spans which involves testing the pixels the span represents. This allows all the span operations to be modified by GIDs but the test can only be done four pixels at a time so span performance is constrained (but is still four times faster than doing clears, etc., one pixel at a time).

More details on the GID field and related registers may be found in the *Permedia4 Reference Guide.*

Note: GID planes are distinct from and serve a different purpose to Windows ID planes which are described later.

4.2.7 Stencil Field

The stencil field holds the stencil value associated with a pixel and can be 0 to 8 bits wide in increments of 1 bit.

The width of the stencil buffer is also stored in the **StencilMode** register and is needed for clamping and masking during the update methods. The stencil compare mask should be set up to exclude any absent bits from the stencil compare operation.

4.2.8 FrameCount Field

The Frame Count Field which controls the Fast Clear Planes mechanism is not supported in Permedia4. The same result can be achieved using the **MinRegion** and **MaxRegion** registers:

1. Record the extent of all updates to the localbuffer and framebuffer using the **MinRegion** and **MaxRegion** registers
2. Use the **MinHitRegion** and **MaxHitRegion** commands to send the bounding box of the smallest area to clear to the FIFO.

For some applications this will be a significantly smaller area than clearing the whole window or screen, therefore faster.

4.2.9 Texture Map Storage

With the introduction of unified framebuffer memory texture storage becomes a separate topic rather than an aspect of localbuffer data handling.

Permedia4 implements a virtual demand-page texture management system using 4k pages in conjunction with other more conventional approaches such as AGP execute mode. Details are given in section 4.3 - Texture Mapping.

4.2.10 Source and Destination Reads

Permedia4 supports both source and destination reads. The destination read parameters are normally set up identically to those used in the **LBWrite** registers so the same pixel data is read for a given XY coordinate as will later be written. If the destination reads and writes are set up differently then the destination read can almost be viewed as a source read as far as copies are concerned.

The reason for the apparent duplication is that distinct source reads are still needed when (for example) source stencil planes are to replace the destination stencil planes while leaving the depth field unchanged[10].

Source data is sent downstream by the **LBSourceData** register when any destination reads are enabled for a fragment. When no destination reads are enabled the source data is carried internally.

4.2.10.1 Address Calculation

The source address calculation is controlled by the **LBSourceReadMode** register and the address is a function of X, Y, **LBSourceReadBufferAddr**, **LBSourceReadBufferOffset**, width and Packed16 parameters.

The destination address calculation is controlled by the **LBDestReadMode** register and the address is a function of X, Y, **LBDestReadBufferAddr**, **LBDestReadBufferOffset**, width and Packed16 parameters.

The Localbuffer calculates pixel addresses to convert x,y current step coordinates into linear byte addresses using different approaches for linear and patched memory. Because conversion for patched memory is non-trivial we do not recommend bypass accesses to patched (non-linear) memory.

Coordinates can be defined window-relative or screen-relative. This only matters when the coordinates are converted to an actual physical address. In general the windowing system will use absolute coordinates and the graphics system will use relative coordinates (to be independent of where the window really is).

In the calculations below, X and Y are the coordinates from the step or span message. The *offset*, *width* and *address* values are taken from the appropriate buffers details:

1. X, Y are set from the step message.
2. The first step is to add in the buffer offset to the coordinates.

   X = X + offset.x

   Y = Y + offset.y

[10] The size and/or alignement may prevent byte masking from being used in a SDRAM local buffer. For an SGRAM local buffer bit masking via the framebuffer operations can always be used. Also for a P4 with 160 bits of local buffer the framebuffer units cannot process the extra 32 bits.

This offset can be used to convert the window relative coordinates to screen relative. This is only necessary if the local buffer is patched. If the buffer is linear, the base address of the window (as a byte address) can be held in **LBSourceReadAddr** or **LBDestReadAddr**.

1. The Y coordinate is compressed using the *StripePitch* and *StripeHeight* values in **LBDestReadMode**. The following diagram shows how the Y bit is interpreted:
2. Y = Y with (StripePitch - StripeHeight) bits chopped out as in the diagram.
3. For linear layout the pixel offset is:

   bottom left origin: -Y * width + X

   top left origin: Y * width + X

The buffer width is in the range 0…4095. For Patch64 the 2D XY coordinate space is mapped to a 1D address range.

The Origin may be either top left or bottom left as shown in the following diagram:

GUI systems (such as Microsoft Windows, Microsoft Windows NT and The X Window System) usually have the origin of the coordinate system at the top left corner of the screen but this is not true for all graphics systems. For instance OpenGL uses the bottom left corner as its origin. The *Origin* bit in the **LBDestReadMode** register selects the top left (0) or bottom left (1) as the origin.

4.2.11 LB Writes

The Localbuffer address calculation is controlled by the **LBWriteMode** register abd the address is a function of the current X, Y and the **LBWriteBufferAddr**, offset, *width* and *Packed16* parameters:

LBWriteMode
LBWriteModeAnd
LBWriteModeOr

Name	Type	Offset	Format
LBWriteMode	Localbuffer	0x88C0	Bitfield
LBWriteModeAnd	Localbuffer	0xAC80	Bitfield
LBWriteModeOr	Localbuffer	0xAC88	Bitfield

Control register

Bits	Name	Read [11]	Write	Reset	Description
0	WriteEnable	✔	✔	x	This bit, when set, causes fragments or spans to written to the destination buffer. Note each byte must also be enabled in the ByteEnables field.
1…2	Reserved	0	0	x	
3…5	StripePitch	✔	✔	x	This field specifies the number of scanlines between the first scanline in a stripe and the first scanline in the next stripe. The options are: 0 = 1 4 = 16 1 = 2 5 = 32 2 = 4 6 = 64 3 = 8 7 = 128 This field will normally be set to zero for P4.
6…8	StripeHeight	✔	✔	x	This field specifies the number of scanlines in a stripe. The options are: 0 = 1 3 = 8 1 = 2 4 = 16 2 = 4 This field will normally be set to zero for P4.
9	Layout	✔	✔	x	This field selects the layout of the pixel data in memory for the destination buffer. The options are: 0 = Linear 1 = Patch64
10	Origin	✔	✔	x	This field selects where the window origin is for the destination buffer. The options are: 0 = Top Left. 1 = Bottom Left

[11] Logic Op register readback is via the main register only

11	Packed16	✔	✔	x	When this bit is set the pixel size is 16 bits so a single memory word can hold 8 depth values.
12…23	Width	✔	✔	x	This field holds the width of the destination buffer. Its range is 0…4095.
24…28	ByteEnables	✔	✔	x	This field holds the byte enables for each byte in the pixel. A byte enable bit must be set for the corresponding byte to be written. Ideally the depth, stencil, etc. fields are byte aligned and integral bytes in length so these can be used to disable modifying a field, otherwise read-modify-write operations will need to be done.
29…31	Operation	✔	✔	x	This field defines where the data is to be taken from to do the write and what is to happen to it afterwards. This is only of interest during an upload or download operation. The options are: 0 = No operation 1 = Download depth 2 = Download stencil 3 = Upload depth 4 = Upload stencil

The origin is either top left or bottom left, as for Reads but in the **LBWriteMode** register.

4.2.11.1 Stripes

LBWrite supports Stripes, which allow more efficient texture cache usage than individual scanlines where multiprocessor implementations are to be supported in future. Currently the *StripePitch* and *StripeHeight* fields are reserved.

4.2.11.2 Write Combining

Since the memory data width is significantly wider than the pixel widths, it is possible to combine pixels from the same data word in a wrtie combine buffer (WCB). The WCB is written to memory when the pixel address changes.

Write combine operation is exactly as for Framebuffer write combining - see section 4.4.6 below.

4.2.11.3 Byte Enables

Mode changes and similar software state changes can have complicated effects on Localbuffer read/write enabling. The application software usually monitors the state change and enables buffer reads/writes as required but this is not always possible.

Permedia4 introduces a new register, **LBDestReadEnables**, whichs allow the user's software to assign paired bits to control major mode monitoring and read enables. There are 8 pairs of bits, E0 to E3 and R0 to R3. In a typical case, E0 could be assigned to Depth Enable, R0 set whenever a depth mode requires a read. E1 could be allocated to Stencil enable, and R1 set whenever a stencil read is required.

The **LBDestReadEnables** register also supports logical operators (AND, OR). These can be very useful for, e.g., Windows OGL applications where the bits can be mapped onto the various state changes which affect whether a read is necessary. Some of the state variables are maintained internally by drivers, others in the OGL ICD at the user system level. Since the two cannot communicate directly, the AND/OR operators allow changes to individual bits in **LBDestReadEnables** without modifying bits controlled by another driver or application. This in turn avoids the need for slow ExtEscape calls whenever a state change takes place.

4.2.11.4 Data Download

LBWriteMode normally downloads data using the *SyncOnHostData* mechanism in the **Render** command, which places downloaded data into the color field of current fragments. However the other LB fields can also be loaded by setting the **LBWriteMode** *Operation* field to Download Depth or Download Stencil:

- If Download Depth is specified, the user-supplied depth data is clamped to fit the range of depth values supported by the LB format. The other fields are updated from the LB or related registers.
- If Download Stencil is specified, the user-supplied stencil field is masked to fit into the range of stencil values supported in the LB format. GID and Depth are updated as above.

In both cases, the ByteEnables in **LBWriteMode** may be usable to control the buffer update if the value is byte-aligned.

For a more general discussion of upload, download and copy for fragments, bitmasks and spans see Volume II, section 1.1.8 - Bitmaps, Spans and Images.

4.2.11.5 Data Upload

The **LBWriteMode** Operation field can also be used to upload Depth or Stencil into the fragment's color field before formatting is carried out. The GID data is loaded concurrently.

Note: If non-linear Depth is used then the Z value remains as it was encoded.

Since the data is loaded into the color field, it can be overwritten by subsequent operations which affect the fragment's color. E.g. alpha blend. Similarly the **FilterMode** *LBDepth* and *Stencil* fields will be ignored - use the *FBColor* field in **FilterMode** instead.

For a more general discussion of upload, download and copy for fragments, bitmasks and spans see Volume II, section 1.1.8 - Bitmaps, Spans and Images.

4.3 Framebuffer (FB)

The framebuffer (FB) is a region of memory where the information produced during rasterization is written prior to being displayed. This information is not restricted to color but can include window control data for LUT management and double buffering[12].

Conceptualy, the FB is more correctly a grouped set of register facilities than a physical pipeline component or memory device type. With unified memory the FB functionality is

[12] Although the term 'double buffering' is used here everything is just as applicable to single or double buffered stereoscopic displays.

very similar to the Localbuffer, however FB supports additional memory layouts (Patch2, Patched32_2) for texture management etc.

4.3.1 Framebuffer Management

Framebuffer functionality includes:

- Managing updates to and reads from up to 4 destination memory buffers,
- Managing the read from a single source memory buffer,
- Calculating read and write address(es) of fragments in memory,
- Combining multiple fragments in the same memory word,
- Calculating the write addresses of the spans in the memory,
- Aligning span data and issuing multiple normal writes,
- Implementing transparent or opaque fills,
- Interfacing to the Memory Controller
- Cacheing the memory data to reduce actual memory accesses,
- Prefetching data where ever possible,
- Calculating the read addresses of the spans in the memory,
- Aligning span data and issuing aligned span data,

4.3.2 Framebuffer Layout

The Layout field selects how the color data is to be laid out in memory. The options are:

- **Linear**. Here the rows are stored one after another in memory. This is typically used for small texture maps (less than 32 x 32 x 32bpp which fit into one page) and is preferably *not* used for color buffers. In this case the page miss per scanline will reduce the small triangle and line performance.
- **Patched64**. In this layout the pixel data is arranged into 64x16 patches for 32 bpp, 128x16 for 16 bpp and 256x16 for 8 bpp. This is the preferred layout for the color buffer to give equal access times for X and Y paths. Note the page size is 1Kx32 bits. This also matches the patching of the Z, etc. data.
- **Patched32_2**. The texture data is arranged into 32x32 patches, but also patched to a finer level so that one read always returns a 2x2 block of texel data (for 32 bit texels) or a 2x4 block for 16 bit texels.
- **Patch2**. The texture data is arranged into 2x2 patches. This is used for texture maps where the total number of texels is less than 1K so it all fits into a page.

4.3.3 Block Writes

Block writes write consecutive pixels (spans) in the framebuffer simultaneously. This is useful when filling large areas but does have some restrictions:

- No depth, stencil or GID testing can be done
- All the pixels must be written with the same value so no color interpolation, blending, dithering or logical ops can be done

Block writes are not restricted to rectangular areas and can be used for any trapezoid. Hardware writemasking is available during block writes.

The **FBBlockColor** and **FBBlockColorBack** registers need to be loaded with the value to write to each pixel before block fills can be used.

Sending a **Render** command with the *PrimitiveType* field set to "trapezoid" and the *FastFillEnable* field set enables block filling of the area. Note that during a block fill of a trapezoid any inappropriate state is ignored so even if color interpolation, depth testing and logical ops, for example, are enabled they have no effect.

4.3.4 Pixels and Spans

During reading, spans are analysed into the number of 64-bit word reads involved, which can be up to 16 addresses and repeated for each possible buffer (one source buffer and four destination buffers). Each buffer is cached and data may be prefetched if only a single read is involved. Prefetch (which may be enabled/disabled in hardware) is enabled by the *SubPixelCorrectionEnable* bit in the **Render** commands and triggered at the start of a scanline.

Span data can also be written to scratch pad memory instead of being forwarded. When both source and destination reads are enabled the source read is stored in scratch memory.

FB Span operations fall into one of three categories described below. For a more general discussion of upload, download and copy for fragments, bitmasks and spans see Volume II, section 1.1.8 - Bitmaps, Spans and Images.

4.3.4.1 Constant color transparent stipple

The pixel mask is used and no color mask is expected. The Write Combine Buffer (see 4.3.7) is used to merge the new span mask with the current accumulated mask (if the masks overlap). When the span mask is full or needs to be flushed out of the WCB the block write capability of the SGRAM memories or normal writes for SDRAM are used (this is determined by the memory controller).

Any alignment of the span mask to the memory pixel structure is done in this unit.

The color to use is loaded by the **FBBlockColor** message or the **FBBlockColor[0…3]** registers beforehand. The 128 bits can take different colors (useful for pattern filling). Transparent spans can be replicated or use the multi-buffer support to allow up to 4 writes.

4.3.4.2 Constant color opaque stipple

The foreground mask is the AND of the pixel mask and color mask. The background mask is the AND of the pixel mask and the NOT color mask. Two WCBs are used to merge the new span masks with the current accumulated mask - one for the foreground mask and one for the background mask.

SGRAM block writes or normal SDRAM writes are used to flush the WCB when the span mask is full or needs to be flushed (this is determined by the memory controller). The span is tagged to use the foreground color or the background color and the memory controller will multiplex these into the single color register in the SGRAM

Any alignment of the masks to the framebuffer pixel structure is done in this unit.

The foreground color to use will have been previously loaded by the **FBBlockColor** message or **FBBlockColor[0…3]** where the 128 bits can take different colors (useful for pattern filling). The background color to use will have been previously loaded by the **FBBlockColorBack** message or **FBBlockColorBack[0…3]** where the 128 bits can take different colors.

Opaque spans can be replicated or use multi-buffer support to allow up to 2 writes, however only single buffer writes are handled efficiently in Permedia4 - the address, offset, layout, etc. parameters must be duplicated for the pair of buffers used in this mode.

4.3.4.3 Variable color

The pixel mask is used and no color mask is expected. This is converted into normal 128 bit writes and the span data is aligned as necessary to the memory pixel structure. The WCB is used as for normal pixel writes.

Variable color spans can be replicated or use multi-buffer support to allow up to 4 writes.

The amount of data associated with a span mask (if the modes require any data at all) depends on the pixel size:

32 bpp	One word of data is sent for each pair of bits which are non zero. The number of words is found by first reducing the span mask to 16 bits by oring pairs of bits (0, 1), (2, 3), etc. together and doing a population count on the reduced mask.
16 bpp	One word of data is sent for each nibble of bits which are non zero. The number of words is found by first reducing the span mask to 8 bits by oring nibbles of bits (0, 1, 2, 3), (4, 5, 6, 7), etc. together and doing a population count on the reduced mask.
8 bpp	One word of data is sent for each byte of bits which are non zero. The number of words is found by first reducing the span mask to 4 bits by oring bytes of bits together and doing a population count on the reduced mask.

4.3.4.4 Clearing FB Memory

Framebuffer writes are handled generally as for Localbuffer writes with the addition of the Patch2 and Patch32_2 formats intended primarily for textures. Hardware block fills are only available using Framebuffer functionality and bitmasking is only supported in FB with SGRAM devices.

4.3.4.5 Hardware Writemasks.

These allow writemasking in the framebuffer without incurring a performance penalty. If hardware writemasks are not available, Permedia4 must be programmed to read the memory, merge the value with the new value using the writemask, and write it back.

To use hardware writemasking, the required writemask is written to the **FBHardwareWriteMask** register. The **FBSoftwareWriteMask** register is set to all 1's, and the number of framebuffer reads are disabled using the *Enable* bits in the **FBDestReadMode** and **FBSourceReadMode** registers.

4.3.4.6 Software Writemasks.

To use software writemasking, the required writemask is written to the **FBSoftwareWriteMask** register and framebuffer reads disabled as above (Hardware Writemasks). This is achieved by setting the ReadDestination enable in the FBReadMode register.

4.4 Double Buffering

Double buffering is a technique used to achieve visually smooth animation by rendering a scene to an offscreen buffer before quickly displaying it.

Which techniques are available depend on the board design. This section discusses how Permedia4 can be used to support specific types of double buffering:

- BitBLT
- Page Flipping
- Mixed 2D/3D Buffering

Colorplane (pixel) double buffering is not supported by the Permedia4 RAMDAC.

Triple buffering is supported indirectly, using the vblank interrupt to indicate when the **ScreenBase** register can be safely updated.

4.4.1 BitBlt Double Buffering

BLT double buffering in its simplest form requires a complete duplicate buffer of non-displayed memory to be maintained. To swap buffers a BLT is performed onto the displayable area. The features are:

- takes significant time to swap buffers
- the offscreen buffer requires as much RAM as is displayed
- any number of windows can be independently double buffered
- pixel depth is limited only by the available RAM.

4.4.2 Page Flipping

To perform page flip double buffering the available memory must be partitioned into buffer 0 and buffer 1, each of which contains enough memory to display a full screen of pixel information. The partitioning requires calculating the offset into the framestore at which each buffer starts. This offset is used to program the relevant LB and FB registers. For a given resolution and pixel depth there must be enough configured memory on the display adapter for this to be possible.

There are two factors to consider for page flip double buffering:

- The video output hardware must be set up to display the pixels from the correct buffer.
- The relevant registers (**FBDestReadBufferAddr**, **FBWriteBufferAddr** etc.) must be programmed to render into the correct buffer.

4.4.2.1 Mixed 2D/3D Buffering

Where a single 3D application is running on a 2D desktop it is possible to use page flipping instead of BLT buffering and achieve a significant performance improvement . This also requires enabling the 2D desktop to double write to front and back buffers simultaneously. Although two writes are still required, once the appropriate enables are set up the 2D desktop algorithms can be unaware that they are in a dual write mode. Usually driver software will enable that write mode dynamically when page flipping is enabled.

The constraint for page flipping is that only one 3D window may be visible. If multiple 3D windows are visible then page flipping cannot be used (because the flip happens on a complete framebuffer basis, not per window).

Multi buffer reads are enabled in the **FBDestReadMode** register using the *Enable0* - *Enable3* bits, and qualified by the *FBSourceReadEnable* bit in the **Render** command. When the **FBDestReadMode** *ReadEnable* bit is set then writes are done to the same buffers which were read. If this bit is clear then no writes to any buffer are made.

Note *Setting the* Replicate *bit in* ***FBWriteMode*** *selects the FBWriteEnable[0…3] bits and corresponding FBWrite buffers instead of the* ***FBDestReadMode*** *enables. Do not set the* Replicate *bit unless you wish to use the* ***FBWrite*** *buffer selections.*

4.4.3 Video Output

Video output is controlled using bits 0-20 of the **ScreenBase** register in the video unit (offset 0x3000). This is changed each vblank by writing to it directly via the bypass or by writing to the *ScreenBase* field of the **SuspendUntilFrameblank** message through the graphics core.

For double buffering the **SuspendUntilFrameblank** command is not sent until back buffer rendering has completed. This prevents further rendering until the next Vblank to avoid corrupting the front (displayed) buffer.

For further information on Video display parameters see chapter 5, Video System.

4.4.4 Texture Map Management

Framebuffer functionality is specifically geared to texture mapping, hence the texture layouts (Patch2 and Patch32_2). Physical, logical and virtual texture management is discussed in *Texture Mapping,* section 4.9 below.

4.4.5 Source and Destination Address Calculation

Addresses are calculated in the same general manner as for Localbuffer addresses, although two additional formats (Patch2 and Patch32_2) are also supported. As with Localbuffer, bypass accesses are not recommended when using patched (non-linear) data layouts.

The address calculation is controlled by the **FBWriteMode** register and the address is a function of X, Y, **FBWriteBufferAddr**, **FBWriteBufferOffset**, **FBWriteBufferWidth** and **PixelSize** parameters. The *Address*, *Offset* and *Width* parameters are specified independently for each of the four possible write buffers.

The address calculation is further modified by the **FBWriteMode** mode bits: *Origin*, *StripePitch*, *StripeHeight*, and *Layout*. For register details see the *Permedia4 Reference Guide*.

4.4.6 Origin and Stripe Data

Coordinate origin and Stripe information is defined exactly as for the Localbuffer (section 4.2.11.1 above)

4.4.7 Write Combining

The data width of the memory is 128 bits. Access to the local buffer using FB functionality is also treated as if it were 128 bits wide. This is much larger than the pixel width, so pixels which occupy the same memory word are combined where possible to make best use of the memory bandwidth.

Write combining is done in the write combine buffer (WCB) and any new pixel data is written into this buffer at the appropriate byte offset when the pixel's memory address matches that of the data already in the WCB. If the addresses do not match and the WCB holds valid data then the WCB is written to memory before the new pixel is loaded in.

For single pixel writes (i.e. rendering via individual fragments) the write combed data will always fit exactly into the WCB. Span data (which is 64 bits wide) may overflow the WCB which cause it to be updated, written and then updated again.

The WCB is flushed to the framebuffer (if it contains any valid data) in the following circumstances:

- New data to combine is at a different memory address.
- **FBWriteMode** is updated.
- **SuspendReads**, **Render**, **Sync**, **WaitForCompletion** or **SuspendUntilFrameBlank** commands received.
- The write mask or block color registers are loaded.

When the WCB buffer is written to memory the address is tested to determine the target memory region. If the most significant bit is set (corresponds to bit 31 for a byte address) the target is the PCI bus. The data is written to the PCI bus via the Memory Controller.

The write data FIFOs to both memory controllers are 8 deep

There is no coherency between the four write combine units so if the WCB are all combining to the same memory address (i.e. the buffers overlap) then the final contents of memory are unpredictable.

4.5 Suspend and Swap on Frame Blank

In a high frame rate animation system the time waiting for frame blanking (before switching the displayed frames over and proceeding with a new frame's worth of drawing) can be a very high percentage of the time each frame takes to process. During this idle time Permedia4 could be doing useful work such as clearing down the depth buffer, but obviously it cannot do any writes to the framebuffer[13].

The **SuspendUntilFrameBlank** message solves this problem by allowing all non framebuffer rendering to proceed while waiting for a frame blank. If any writes to the framebuffer are attempted during this time the graphics core simply stalls until the frame blank occurs. This happens without any host involvement so the host software can ignore having to synchronise with frame blank before issuing rendering commands.

The ScreenBase field in the **SuspendUntilFrameBlank** message indicates where the new display frame is to be found.

4.6 Downloading Data

4.7 Controlling the VTG or RAMDAC

The VTG and RAMDAC can be read and written via the PCI bypass, but sometimes it is useful to control then synchronously with Permedia4 rendering activities. This can be done by using the **VTGAddress** and **VTGData** commands. The address is loaded first

[13] If triple buffering is used rather than double buffering then the framebuffer can be cleared in preparation for the next frame without having to wait for frame blanking.

followed by the data. The address and data are the same as would be used if the VTG, Ramdac or any other device on the PCI bypass were accessed via the bypass.

4.8 Texture Mapping

Texturing is enabled in the **Render** register. Texture maps can be stored in physical memory or in logical/virtual memory. If the texture map is stored in physical memory then it must be physically contiguous and present before that texture is used.

Programming aspects of texture use are discussed in volume II, chapter 4 - Texture Mapping.

4.8.1 Texture Memory Layouts

The Layout field in the **TextureMapWidth** register selects how the texture data is to be laid out in memory for each mip map level. The options are the same as for the Framebuffer functions:

- Linear. Here the rows are stored one after another in memory. This is typically used for small texture maps (less than 32 x 32 x 32bpp which fit into one page) and are always accessed along a row. This matches up with most 2D use of texture maps for font, icon and stipple pattern storage. Video data will also fall into this category.
- Patch64. In this layout the pixel data is arranged into 64x16 patches for 32 bpp, 128x16 for 16 bpp and 256x16 for 8 bpp. This is the preferred layout for the color buffer (desktop) so will only be used when the texture units need to operate on this data directly, for example to stretch blit a window.
- Patch32_2. The texture data is arranged into 32x32 patches, but also patched to a finer level so that one read always returns a 2x2 block of texel data (for 32 bit texels), a 2x4 block for 16 bit texels or a 2x8 block for 8 bit texels.
- Patch2. The texture data is arranged into 2x2 patches. This is used for texture maps where the total number of texels is less than 1K so it all fits into a page.

For textures even more than for color and alpha data, patched formats deliver better performance than linear:

- Performance should be independent of the traversal direction, especially for 'large' texture maps (i.e. > 32x32). Storing the texture map as linear data gives very good access times in the u direction but poor access times in the v direction due to the page organisation of SDRAMS. Storing the texture maps in a patch form (32x32 in our case for 32 bit texels) equalises the access times.
- The memory width is very much wider than the texel width so each memory read returns multiple texels. If the texel data in a memory word are all for the same row then all the data is used when traversing in u (along a row) but very little is used in the v direction (along a column). The 2x2 patch organisation ensures that at least 2 texels can be used from each memory read for all traversal directions.

Texture maps are preferably stored in memory in one of the 2x2 patched formats to give the best overall performance for general 3D use, but this is not always possible. For example if the texture data originates from an external source or is used to drive an external device (i.e. a monitor) the layout of the data may be fixed and not in 2x2 format.

Alternatively the traversal direction may be known to always be in the u direction - examples of this are video scaling, fonts and general 2D use.

When the texture map is stored in memory in a non 2x2 layout it is formatted into the 2x2 layout expected by the Filter Unit during read in.

4.8.1.1 Origin Orientation

Linear or Patch64 texture formats can choose between top left and bottom left origins, but the texture map must start on the natural boundary for the texel size. For 8 bit texels this is on a byte boundary, for 16 bit texels this is on a 2 byte boundary and for 32 bit texels this is on a 4 byte boundary.

4.8.1.2 Patch layout rules

The preferred layout for texture maps (1D or 2D) for use by 3D rendering is Patch32_2 or Patch2 as this gives the minimum number of reads for an arbitrary orientation of the map, but for this to work the following rules must be followed:

- Texture maps are stored with the top left corner as the origin, i.e. texels at increasing u and/or v coordinates are at increasing memory addresses.
- The texture map must start on the natural patch boundary for the texel size. For 8 bit texels this is on a 4 byte boundary, for 16 bit texels this is on a 8 byte boundary and for 32 bit texels this is on a 16 byte boundary.
- Patch32_2 layout only make sense when the width of the texture map is greater than the patch width (128 bytes). Using Patch32_2 on texture maps which are less than 128 bytes wide wastes storage and may increase the number of page breaks. If the width of a texture map is less than or equal to 128 bytes it will be changed from Patch32_2 to Patch2 automatically. This allows mip maps to be Patch32_2 for the high resolution levels and Patch2 for the low resolution levels.

It is the software's responsibility to set the layout to Patch32_2 or Patch2 as appropriate when the texture map is downloaded. The hardware writes the texel data into the correct place. It does not, as a rule, switch layouts automatically.

4.8.1.3 Texture Border Width

The minimum width and height of a texture map (in any layout) is 2 texels. If the width and/or height of a texture map is 1 (such as the lowest resolution map in a set of mip maps) then the texels must be replicated to expand the offending dimension(s) to 2 texels. If a 1x1 texture map has a border then the resulting 3x3 map is stored as a 4x4 map.

4.8.2 Address Calculation

If it is a 3D texture map the base address is set from **TextureBaseAddr[0]** register, the layout and texel size are taken from **TextureReadMode0** register and the width from **TextureMapWidth0**.

If the texture is *not* a 3D texture map then the layout, texel size and width parameters are taken from the appropriate texture registers (these registers should be loaded in the same way as for per pixel mip mapping). The **TextureMapSize** should be set to a value greater than or equal to the product of the width and height for a slice. The width is divided by 2 (map level) so the correct mip map width is used.

Note *The width does not have to be a power of 2 so the divide may have a remainder (which is ignored) which would fail past some map level. This is not a problem as mip maps will always be a power of two in size and non mip maps will always have a map level of 0.*

The *MapBaseRegister* field of the **TextureReadMode** register defines which **TextureBaseAddr**(n) register should be used to hold the address for map level 0 when mip mapping, or the texture map when not mip mapping. Successive map levels are at increasing **TextureBaseAddr** registers up to and including the *MapMaxLevel*. The actual one used depends on the map level, the map base level and map max level associated with this texture as given by:

offset into base registers = min (texture map level + map base level, map max level)

...so the allocation of the base registers between the two possible textures is up to software.

The maximum width is 4095, but the minimum width depends on the layout as Patch2 and Patch32_2 have minimum requirements. If the mip mapping forces the width below these minimum requirements then the width is forced to be the minimum allowed for the texel size:

- The minimum texel widths are 8, 4 and 2 for 8, 16 and 32 bits per texel respectively.
- The minimum width is one memory word (i.e. 16 bytes).
- If the width falls below 128, 64 or 32 texels for 8, 16 or 32 bits per texel respectively.

Any textures with a Patch32_2 layout are automatically set to Patch2.

As with Framebuffer and Localbuffer operations, direct access to patched texel layouts should be avoided due to the complexity of the calculations required.

4.8.3 Primary Cache

The primary cache holds the texel data in 8, 16 or 32 bits per texel format. This is converted into a standard internal ABGR format for processing. The cache is divided up into 8 banks, giving two independent texture maps (banks 0-3 and 4-7). This also allows two levels of a mip map or slices of a 3D texture.

When a single non-mip mapped texture is used the two caches can be joined together to render a larger texture map or polygon while still maintaining scanline coherency.

There is a fixed relationship between a texel's position in the texture map and which bank of cache is must be stored in. The 8 banks are assigned depending on the type of texture mapping being done:

Filter Type	Texture maps
Single bilinear	The texture map is stored in both banks of the cache. This is achieved by connecting the output of the second bank's register files to the corresponding register files in bank 0. This is controlled by the *CombineCaches* bit in the **TextureFilterMode**. This arrangement allows the full size of the cache to be used on a single texture, so a larger texture map can be handled before scanline coherency starts to break down with consequential loss of performance

Filter Type	Texture maps
Dual Bilinear	Texels from texture map 0 are stored in banks 0…3 and texels from texture map 1 are stored in banks 4…7.
Mip mapping	Even mip maps are stored in banks 0…3, odd mip maps are stored in banks 4…7.
3D texture maps	Texels with an even k coordinate (i.e. the third coordinate) are in banks 0…3 and maps with an odd k coordinate are in banks 4…7.

An efficient texture cache is vital for texture performance, particularly when mip mapping[14]. Ideally the entire texture map would fit into cache, but this is not currently possible except for the smallest texture maps (32x32 at 16 bits per texel). As a result, the cacheing mechanism adopts various strategies to optimise scanline coherence and consistency of performance for different traversal directions through the texture map(s).

Span processing does not use the primary cache when the pixel mask (as part of the **SpanStep** register) is modified by the texel data.

The cache is always enabled. User control is limited to invalidating the cache when cached data become stale (e.g. when a new texture map is selected or when the current texture map's data is edited in memory).

The cache is divided into two parts: a ***data part*** and a ***directory part***.

4.8.3.1 Cache Data Part

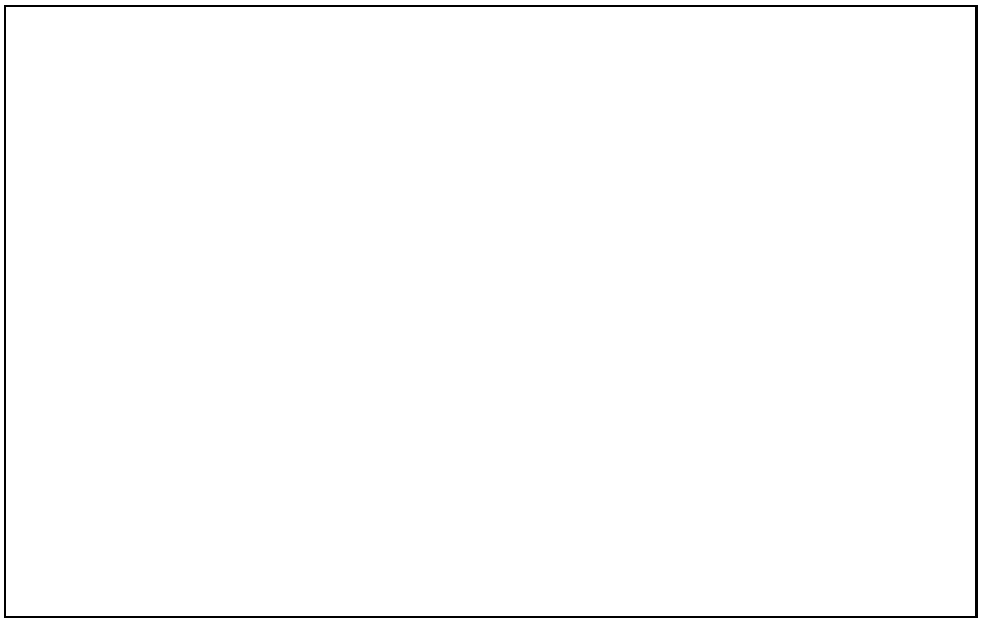
The data part of cache holds the texel data. Data formatting is controlled by the **TextureFilterMode** register. Each cache line holds 128 bits of data. There are 64 cache lines in each bank for Permedia4.[15] Each cache line holds a 2x2, 4x2 or 8x2 patch of texels for 32, 16 and 8 bits per texel respectively. In the 2x2 case the cache's performance is independent of the traversal direction through the texture map. Otherwise the 'u' direction is more efficient than the 'v' direction.

The patch has a fixed relationship to the origin of the texture map: the origin of the patch is always an integer multiple of the patch size from the origin of the texture map.

The following diagram shows the 2x2 patch arrangement within a texture map. The numbers in the brackets show how the texel coordinates within the texture map vary and the T0…T3 are the corresponding filter registers to which each texel is assigned. The grey areas show the texels held in a memory word (16 bytes) for each size of texel.

[14] ...where the zoom ratio means limited re-use of texel data when moving from one pixel to the next

[15] These sizes are for illustration only and may be changed later.



4.8.3.2 Directory Part

The directory part of the primary cache is controlled by the **TextureReadMode** register and is searched to find out if a texel is already in the primary cache and if so where.

A bank of the cache cannot hold texels from different texture maps (but texels from the different levels in a mip map or from different slices in a 3D texture can be held in the same bank). This means that the cache *must* be invalidated whenever a new texture map is selected.

Caution: The InvalidateCache command must complete before any further texture operations can be performed or the pipeline may lock up[16]

The size of the cache is a compromise - the larger the better, but it follows the law of diminishing returns. Each bank has 1K bytes of storage so for 16 bit textures the cache works best when fewer than 128 texels are used for mip maps or 256 texels for a single texture map using both cache banks.

4.8.3.3 Combining Both Cache Banks

When a single texture map is being used, Permedia4 can be put into a mode where the register files from bank 1 extend the corresponding register files in bank 0.

The **TextureReadMode0**.*CombineCaches* bit is used to enable the mode. When set, texels are loaded into each bank alternately. The texture 0 indices are used and are checked in both banks for their presence.

Caution: CombineCaches should not be used for texturing operations when clock speeds in excess of 100MHz are anticipated.[17]

---

16 Use **WaitforCompletion** or another workaround - refer to *Permedia4 Errata and Alerts* PEREN0016.

17 Refer to *Permedia4 Errata and Alerts*, ALERT002

4.9 Virtual Texture Management

The management of physical textures is complicated by the fact that an application can request more textures than can fit into on-card memory so textures need to be dynamically swapped:

- The need to swap and usage are decoupled in time by the DMA buffers.
- The memory granularity is controlled by the texture map size so is continually changing.
- Memory gets fragmented.
- There is no clear replacement policy

Possible solutions include:

- Increase the amount of physical memory to hold texture maps.
- Allow textures to be executed out of host memory via the AGP or PCI bus.
- Treat texture addresses as logical or virtual addresses.

Permedia4 implements virtual texture management, which allows texture maps to be stored in non-contiguous physical 4KB pages and allows demand paging of textures out of host or system memory with or without assistance from the host CPU.

Host textures can also be managed. Permedia4 supports this on AGP (via the *HostTexture* bit in the **TextureMapWidth** registers) although it is bandwidth limited and raises latency issues. The main difference is that no texture data is downloaded but is accessed 'in situ' using the side band addressing capability of the AGP texture execute mode.

4.9.1 Mapping an Address

When a texel causes a primary cache miss the following events take place in the virtual texture management system:

- The texel has its logical byte address calculated from it's integer coordinates, base address of the texture, texture map width, etc.
- The logical page the logical address resides in is calculated and the Translation Look aside Buffer (TLB) checked to see if the physical page assigned to the logical page is present in order to determine a physical address to post to the memory controller.
- If the logical page is not found in the TLB then the system regenerates the physical addressing data locally or via Host DMA.

4.9.2 Logical Page Mapping

The size of each page is always 4K bytes so the bottom 12 bits of a texel byte address give the byte within a page while the next 16 bits give the page number (the remaining 4 most significant bits are ignored). This gives a maximum virtual texture size of 65536 pages or 256MBytes. The working set can be any number of pages in size. Each logical page has 8 bytes of overhead (in the Logical Page Table) and each physical page has 8 bytes of overhead (in the Physical Page Allocation Table). Some typical sizes for these tables are:

Managed Memory (pages / MBytes)	Table Size
256 / 1MByte	1KBytes
512 / 2MByte	2KBytes
1024 / 4MByte	4KBytes
2048 / 8MByte	8KBytes
4096 / 16MByte	16KBytes
8192 / 32MByte	32KBytes

The Logical Page Table is typically much bigger than the Physical Page Allocation Table. The Logical Page Table must be physically contiguous and in allocated in local buffer memory. The Physical Page Allocatoin Table must be physically contiguous and is allocated in local buffer memory.

4.9.3 Translation Look-aside Buffer (TLB)

The Translation Look-aside buffer (TLB) caches the recent logical to physical page mappings. It is checked first to see if the mapping for a page is present - this is much faster than having to query the Logical Page Table in memory. The TLB runs whenever virtual texture is enabled. It can be invalidated by using the **InvalidateCache** command with the *TLB* bit set. The TLB should be invalidated whenever the host changes the Logical Page Table directly through the bypass. Changes to the Logical Page Table via the **UpdateLogicalTextureInfo** command will automatically invalidate those logical pages which are updated, if present in the TLB.

4.9.4 Logical Page Table

The Logical Page Table has one entry per logical page and each entry has the following format:

Bit No.	Name[number of bits if not 1]	Description
0…15	PhysicalPage[16]	These bits hold the physical page number relative to the start of the working set where this logical page is held. If the page is not resident (next field) then these bits are ignored (but will frequently be set to zero). This field is normally maintained by P4, except when the page is marked as a HostTexture.
16	Resident	This bit, when set, marks this logical page as resident in the working set. This field is normally maintained by P4, except when the page is marked as a HostTexture.
17	HostTexture	This bit, when set, marks this logical page as resident in the host memory and it should be accessed using AGP texture execute mode rather than downloading it. The Length field should also be set to zero.
18…31	reserved	This field is not used but is set to zero whenever the Resident bit is updated.
32…40	Length[9]	This field holds the number of 128 bit words to transfer when a page fault occurs. This allows a page to hold a texture map smaller than 4K without spending the extra download time on the unused words. There is no way to download to unused portion without overwriting the used part. When the physical page is in host memory the length field must be set to zero. This field is maintained by the host.
41…42	MemoryPool[2]	This field holds the memory pool this logical page should be allocated out of. This field is maintained by the host.
43	VirtualHostPage	This bit, when set, indicates the HostPage (next field) is a virtual page in host memory so cannot be accessed directly. Setting this bit will generate an interrupt and involve the host in providing this page of texture data. When this bit is 0 the HostPage is the physical page and will be read directly with no host intervention. This field is maintained by the host.
44…63	HostPage[20]	This field holds the page in host memory where the texture data is held. This is a virtual host page or a physical host page as indicated by the VirtualHostPage bit (previous field). This field is maintained by the host.

The first word in each entry is basically read and written during the memory management activities unless the page is a host texture in which case the host is responsible for the first word as well. The second word is written by the host (either directly via the bypass or via the core using messages) and just read by Permedia4.

The base address of the table is held in the **LogicalTexturePageTableAddr** register and is aligned to a 64 bit boundary. The number of entries in the table is held in the **LogicalTexturePageTableLength** register and each logical page number is tested against this limit. If the logical page number is out of range then the address is always mapped into page 0 of the working set[18] and will never cause a texture download. The **LogicalTexturePageTableLength** is initialised to zero during reset which effectively disabled the logical and virtual texture management.

The table can be updated by the host directly via the bypass once the chip has been synced to make sure there are no conflicting accesses. The **PhysicalPageAllocationTable** must also be updated to remove the reference (if any) to the logical page being updated. The TLB should also be invalidated.

The table can also be updated via the normal command stream using the **SetLogicalTexturePage** command to set the first page to update. The data for bits 32…63 is supplied with the **UpdateLogicalTextureInfo** command. This updates the Logical Page Table at the previously set page and does any necessary housekeeping. The logical page to update is auto-incremented so several consecutive table entries are updated. Updates beyond the number of entries in the table (as set by **LogicalTexturePageTableLength**) are discarded and leave the memory untouched.

The logical table is updated by:

- Memory Allocator to mark a logical page as non-resident when its allocated physical page is reclaimed and assigned to another logical address.
- The Download Controller to update the resident bit and physical page field once the download is complete.

4.9.5 Memory Allocation

When there is a new page of non host texture data to load into the working set a physical page needs to be allocated to it from the specified pool of memory. The least recently used page in the specified pool is used.

Keeping track of the least recently used page is done by a queue. Whenever a page is first accessed (easily identified by a TLB miss on the page) it is moved to the head of the queue. The page at the tail of the queue is therefore the least recently used and can be allocated to the new texture page. This physical page may already be assigned to a logical page so that logical page is marked as non-resident in the Logical Page Table and removed from the TLB.

The queue used to track the physical pages is held in the Physical Page Allocation Table. This table has one entry per physical page and each entry has the following format:

Bit No.	Name *[number of bits if not 1]*	Description
0…15	LogicalPage[16]	These bits hold the logical page number this physical page has been assigned to. If no assignment has been made (or it has been removed) then the valid bit (next field) will be zero and these bits are ignored (but will frequently be set to zero).
16	Valid	This bit, when set, marks this logical page as resident in the working set. This field is normally maintained by P4.

[18] As a debug aid page 0 of the working set can be missed out of the Physical Page Allocation Table and initialised to some distinctive texture map so any out of range texture mappings cause a distinctive visual effect.

17…31	reserved	This field is not used but is set to zero whenever the Resident bit is updated.
32…47	NextPage[16]	This field holds the page number of the next page in the pool - i.e. the next recently used page.
48…63	PrevPage[16]	This field holds the page number of the previous page in the pool - i.e. the previous recently used page.

The Physical Page Allocation Table is not normally accessed by the host. The two exceptions are during power-on initialisation and if pages are to be locked down.

The *NextPage* and *PrevPage* fields are used to form a double linked list of the pages assigned to a memory pool. In this application a deletion can occur from any queue entry but insertions only occur at the head. The head entry is the most recently used physical page.

A traditional linked list suffers from a linear search time, but by combining it with an array (i.e. table) a constant search time to find a given physical page is guaranteed - you just use the physical page number to index into the table. This is important as a frequent operation is to make a specific physical page the most recent. This involves searching for this page and updating the head (and maybe the tail) pointer to move this page to the head of the queue.

Each memory pool has a head and tail page. These are held in the **HeadPhysicalPageAllocation[0…3]** and **TailPhysicalPageAllocation[0…3]** registers respectively and the index relates to each memory pool. These registers are initialised by software at the start of day, but there after are read and written by the hardware.

The *PrevPage* field for the head page is ignored and holds links which should be ignored. Similarly for the *NextPage* field for the tail page.

The maximum size the Physical Page Allocation Table needs to be[19] is the amount of buffer memory (LB+FB) in Mbyes, divided by 4096. This gives one entry for each 4K page on the card. Many of these pages are not available for virtual texture storage because they hold:

- color buffers.
- Z, stencil, etc. buffer.
- overlay buffers.
- video overlay buffers.
- non logical textures, icons, fonts, bitmaps, etc.
- the Logical Page Table.
- the Physical Page Allocation Table.
- run length encoded window ID information.
- logical textures which have been locked down.

These pages are not included in any of the four linked lists so are ignored by the memory allocation hardware.

[19] There is no reason why the Physical Page Allocation Table could not be smaller and just cover the contiguous region set aside for dynamic texture management. Having it cover all the on card memory helps to illustrate some points.

4.9.6 Programming Notes for Non-Host Textures

This section looks at some general programming information on how the virtual texture management hardware is used.

4.9.6.1 Start of Day Initialisation

Before any logical or virtual texture management can be done there are a number of areas which need to be initialised (in addition to the usual mode, etc. register initialisation):

- Space for the Logical Texture Page Table must be reserved in the local buffer and the table initialised to zero. The **LogicalTexturePageAddr** and **LogicalTexturePageTableLength** must be set up.
- Space for the working set must be reserved in the local buffer and/or framebuffer. This need not be physically consecutive pages. The **BasePageOfWorkingSet** register is set up.

If virtual texture management is to be used then the following additional initialisation is required:

- Space for the Physical Page Allocation Table is reserved in the local buffer and **PhysicalPageAllocationTableAddr** register is set up to point to it.
- Bits 0…31 of each entry in the Physical Page Allocation Table is set to zero - to clear the valid bit.
- Each page entry in the Physical Page Allocation Table is associated to one of the four pools based on which bank of memory it resides in. All the pages in a pool are linked together as a double linked list by setting the *NextPage* and *PrevPage* fields. The order is unimportant, but sequential is simplest. The *PrevPage* field for the first entry in the double linked list and the *NextPage* field for the last entry can be set to any value as they are not used. Finally the **HeadPhysicalPageAllocation** and **TailPhysicalPageAllocation** registers for this memory pool are updated with first and last page numbers.

Each memory pool (to a maximum of 4) is set up like this. (Unused memory pools must not be referenced in the Logical Texture Page Table). The texture management hardware is ready to be used once logical textures have been created. The texture management can be done on a global basis so all contexts/APIs share the same mechanisms, or can be done on a context by context basis.

4.9.6.2 Creating and Loading Texture Maps

The sequence of events when the application asks for a texture to be loaded are as follows:

- Host memory to hold the texture map is allocated and locked down. This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the texture management hardware will do any necessary housekeeping

to prevent aliasing of physical pages to the same logical page (thereby degrading the performance although still functioning correctly).

- The host physical page (or host virtual page when host virtual addressing is used) of each page reserved for the texture is found and the *HostPage* field for each corresponding entry in the Logical Texture Page Table is updated with it.
- The memory pool this texture is to be stored in is determined[20] and each logical entry has its *MemoryPool* field set appropriately.
- The Length field for each logical entry will normally be set to 0x100 (i.e. 4096 bytes), however as an optimisation if only part of the 4K page is used (must be the lower part) then the number of 64 bit words used can be used instead.
- The application's texture is copied into the previously allocated host memory and during the copy the texture map is patched and aligned as required by the setting the texture map will be invoked with[21].

The preferred way to update the Logical Texture Page Table is to use the **SetLogicalTexturePage** and **UpdateLogicalPageInfo** commands. The **SetLogicalTexturePage** command takes the logical page to update in the least significant bits. The **UpdateLogicalPageInfo** command sets bits 0…31 to zero and updates bits 32…63 with the given data. The entry to update was set by **SetLogicalTexturePage** command and this is auto incremented after the update. All the necessary housekeeping is done.

Alternatively the Logical Texture Page Table can be edited by software by reading and/or writing it directly to the table in memory using bypass memory accesses. In this case it is the software's responsibility to do the necessary housekeeping to remove any references to the updated logical pages in the Physical Page Allocation Table.

After this set up has been done the texture map can be bound and used. Note that the texture map (or pages of it) are not loaded until it is actually used.

4.9.6.3 PreLoading Texture Maps

As mentioned in the previous section the texture map is only downloaded when it is used, but it is sometimes useful to ensure it is downloaded when it is created. This can be done by using the Load mode to load each logical page in the texture map. Alternatively when a texture map is bound (to a context) you may want to ensure it is resident at that time, rather than wait for a page fault. If the page is already resident then there is no need to load it (as the Load mode would do) so the Touch mode can be used instead. These can be done using the command **TouchLogicalPages**. This command has the following data fields:

[20] This may be difficult to determine as the usage of the texture maps is not available. Ideally texture maps to be used simultaneously should be in different pools, unless they can both fit into the same 4K page.

[21] It is impossible to do any patching or aligning while the page of texture is downloading as the download mechanism has no knowledge of the dimensions of the texture map, its base address, layout or texel size.

Bit No.	Name	Description
0…15	Page	This field set the first Logical Page to touch.
16…29	Count	This field holds the number of pages to touch.
30…31	Mode	This field is set to 3 to touch a page(s) or to 1 to load a page(s).

As each page is touched the corresponding texture data is downloaded.

4.9.6.4 Editing Texture Maps

To edit the texture map (for example as part of a TexSubImage operation in OpenGL) the host's copy is edited. The texture management hardware is notified that the texture pages (if resident) are stale by using **TouchLogicalPages** to mark these pages as non resident. This command has the following data fields:

Bit No.	Name	Description
0…15	Page	This field set the first Logical Page to mark as stale.
16…29	Count	This field holds the number of pages to mark as stale.
30…31	Mode	This field is set to 0 to mark the pages as stale (i.e. non resident).

The primary texture cache is invalidated (using the **InvalidateCache** command) to ensure it doesn't hold any stale texel data for the texture map just edited.

4.9.6.5 Deleting Texture Maps

Texture maps do not need to be deleted. Simply reusing the logical address achieves the same thing. If you really want to delete the pages then the **TouchLogicalPages** command can be used to mark them non resident[22].

4.9.6.6 Locking Down Texture Maps

The best way to have locked down texture maps (i.e. they don't get swapped out) is to avoid using logical/virtual management and have them as physical textures. If a texture is to be locked down *after* is has been created as a logical texture then the software must edit the Physical Page Allocation Table (and possibly the **HeadPhysicalPageAllocation** and/or **TailPhysicalPageAllocation** registers for the affected pools). The system must be idle before these edits can take place to avoid texture downloads in mid-edit with unpredictable results.

4.9.6.7 Virtual Host Textures

Virtual host textures are textures which live in virtual host memory so do not need to be locked down into physical memory. As a result they are not guaranteed to be present when a corresponding page fault occurs, and in any case the Logical Texture Page Table only holds the virtual page address and not the physical page address.

The Logical Texture Page Table will have the *VirtualHostPage* bit set for these logical pages. Setting or clearing the bit does not otherwise change the setup from Permedia4's point of view.

[22] This does not mean that these pages are made the least recently used pages so they get reused sooner - they will percolate to this status subsequently just through inactivity.

On a page fault the DMA controller cannot go and fetch the page information directly but raises an interrupt.

On receiving this interrupt the **TextureAddr** PCI register is read. This holds the 20 bit virtual address page for the faulting texture page. When the data is available in locked memory the physical address where the data is located is written in to the **TextureAddr** PCI register. This wakes up the texture download DMA controller which performs the download and finishes any necessary house keeping.

4.9.6.8 Using Logical Mapping without Virtual Management

Logical texture mapping can be used without virtual management. This allows textures to be mapped over non-contiguous physical memory without automatic loading. Set up this way, textures are managed similarly to the GLINT MX, but memory management is simpler because physical memory allocation is now done on pages rather than variable-size texture maps.

To do this all current logical textures must be resident so a page fault will never occur. When a texture is created the software needs to do two things:

- Allocate the physical memory and update the Logical Texture Page Table with the logical to physical mappings.
  The physical page for each corresponding logical page is stored in bits 0…15 and the resident bit (bit 16) is set. The second word in each entry is not used (it would only be accessed on a page fault). The Logical Texture Page Table can be modified directly via the bypass (after syncing) or can be updated via the command stream. The **DownloadAddress** register and **DownloadData** commands can be used to update an arbitrary region of memory. This allows them to be used to update the logical entries in the Logical Texture Page Table[23].
- The texture map must be downloaded into the physical pages. This can be done via the bypass mechanisms or through the command stream. In either case it is the software's responsibility to do any patching and alignment consistent with how the texture map will be used.

Note *The texture download mechanism which can do the patching doesn't have any method of remapping the addresses so cannot work with non-contiguous physical memory. The DownloadAddress register and DownloadData commands can be used to download each page of texture (pre-patched, if necessary) into its corresponding physical page.*

4.9.7 Programming Notes for Host Textures

Texture maps stored in host memory can be managed by the virtual management hardware. This allows a texture map to be split over non-contiguous pages of host memory (without relying on the AGP GART table to do the logical to physical mapping) and texture maps to be paged in and out of this memory.

The host pages are not part of the physical memory pool managed by the hardware so all host pages are allocated (or reallocated) by host software.

---

[23] The **UpdateLogicalPageInfo** command cannot be used as it zeros the physical page field and updates the fields concerned with page faults. Also this command does housekeeping work on the Physical Page Allocation Table, which presumably will not have been set up if the virtual texture management is not being used.

4.9.7.1 Start of Day Initialisation for Host Textures

Assuming the range of logical pages reserved for host texture management is already included in the length of the Logical Page Table then no further initialisation is needed other than to set up the **BasePageOfWorkingSetHost** register with the address of the region to manage. This is a 256MByte region and can be positioned anywhere in the 4G host address range.

No changes to the Physical Page Allocation Table are needed.

4.9.7.2 Creating Logical Texture Maps for Host Textures

The sequence of events when the application asks for a texture to be loaded are as follows:

- Host memory to hold the texture map is allocated and locked down[24]. This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the TLB should be invalidated to prevent it from holding stale addresses.
- Each logical page has its physical page, resident and host texture fields in the Logical Page Table updated with the corresponding host physical page where the texture is located. The length field must be set to zero (to disable a download from occurring). The pool field and the *HostPage* field are not used (but are available to software to hold information about this page).
- The application's texture is copied into the previously allocated host memory. During the copy the texture map is patched and aligned as required by the setting the texture map will be invoked with[25].

The preferred way to update the Logical Texture Page Table is to use the **DownloadAddress** and **DownloadData** commands. The **DownloadAddress** command takes the byte address in memory of the Logical Page Table Entry to update. The **DownloadData** command writes its data to memory and then auto increments the address. Two words are written per logical page entry. After the Logical Page Table has been updated the TLB must be invalidated to prevent it holding stale data (use the InvalidateCache command with bit 2 set) and **WaitForCompletion** used to ensure the table in memory has been updated before any rendering can start[26].

Alternatively the Logical Texture Page Table can be edited by software by reading and/or writing it directly to the table in memory by using bypass memory accesses methods. In this case it is the software's responsibility to Sync with the chip first to ensure no outstanding rendering is going to use a logical page about to be updated. The TLB still

---

[24] Virtual host memory could be used, however the driver will need to respond to every page fault and make the textures available in locked physical memory before starting the DMA off to download them.

[25] It is impossible to do any patching or aligning while the page of texture is downloading as the download mechanism has no knowledge of the dimensions of the texture map, its base address, layout or texel size.

[26] The writes to the Logical Page Table are done as Framebuffer Writes so may still be queued up on the subsequent TLB miss, hence stale page data will be read from the Logical Page Table. The **WaitForCompletion** command ensures this cannot happen.

needs to be invalidated after the bypass updates have been done. After this set up has been done the texture map can be bound and used.

4.9.7.3 PreLoading Texture Maps for Host Textures

This is not meaningfull unless they are virtually managed, in which case they can be touched like non-host textures. This is because the texels are read on demand and not downloaded as pages.

4.9.7.4 Editing Texture Maps for Host Textures

The procedure is identical to that for non-host textures (above).

4.9.7.5 Deleting Texture Maps for Host Textures

It is unnecessary to delete texture maps. Reusing the logical page has the same effect.

4.9.7.6 Virtual Host Textures

Virtual host textures are textures which live in virtual host memory so do not need to be locked down into physical memory. As a result they are not guaranteed to be present when a corresponding page fault occurs, and in any case the Logical Texture Page Table only holds the virtual page address and not the physical page address.

The Logical Texture Page Table will have the *VirtualHostPage* bit set, the resident bit clear, the host texture bit set and length field zero for these logical pages.

The DMA controller raises an interrupt (even though no download is needed the DMA controller is involved so the same software interface can be used).

On receiving this interrupt the **TextureAddr**, **LogicalPage** and **TextureOperation** PCI registers are read to identify the faulting texture page. When the data is available in locked memory the Logical Page Table is updated via the bypass and the **TextureAddr** PCI register is written (the data is not used). The write to the **TextureAddr** register will wake up the texture download DMA controller but because the length field is zero no download is done or physical page (from the Physical Page Allocation Table) allocated. The TLB is automatically invalidated.

In servicing the interrupt a physical page (or pages if the interrupt is used to allocate a whole texture rather than just a page) must be allocated by software. If these physical pages are already assigned then the corresponding logical pages must be marked as non resident in the Logical Texture Page Table. If these newly non resident logical pages are subsequently accessed (maybe by a queued texture operation) they themselves will cause a page fault and be re assigned. Hence no knowledge of what textures are waiting in the DMA buffer to be used is necessary. The physical pages are allocated from the host working set whose base page is given by **BaseOfWorkingSetHost** register.

4.10 3D and Other Textures

4.10.1 3D Textures

A 3D texture map is one where the texels are indexed by a triplet of coordinates: (u, v, w) or (i, j, k) depending on the domain and is typically used for volumetric rendering.

The texture map is stored as a series of 2D slices. Each slice is stored in an identical fashion to all other 2D texture maps. The first slice (at k = 0) is held at the address given

by **TextureBaseAddr0** and the remaining slices are held at integral multiples of *TextrueMapSize* (measured in texels) from **TextureBaseAddr0**.

3D texture mapping is in this unit is enabled by setting the *Texture3D* bit in **TextureReadMode0** (the same bit in **TextureReadMode1** is always ignored). The layout, texel size, texture type and width should be set up the same for texture 0 and texture 1.

When 3D texture is enabled then any bits to control dual textures or mip mapping are ignored. The *CombinedCache* mode bit should not be set when 3D textures are being used.

4.10.2 Bitmaps

Bitmap data can be stored in memory and accessed via the texture mapping hardware. The resulting 'texel' data is treated as a bitmap and used to modify the pixel or color mask used in a span operation.

The bitmap data can be held at 8, 16, 32 or 64 bit texels and is zero extended (when necessary) to 64 bits before being optionally byte swapped, optionally mirrored, optionally inverted and ANDed with the pixel mask or the color mask. Bitmaps use the secondary texture cache, not the primary texture cache.

The bitmap data can only be held in Linear or Patch64 layouts - Patch32_2 or Patch2 formats are not supported, however no interlocks prevent their use - the results are just not interesting or useful. The bitmap data can be stored as logical or physical textures.

The bitmap data can be held as packed 8, 16, 32 or 64 bit data, usually with one scanline of the glyph held per texel. Glyphs wider than 64 bits will take multiple texels to cover the width. Packing multiple scanlines together reduces the waste of memory (in MX the texel size was limited to 32 bits for spans), and makes the cacheing more efficient.

Before the texel can be used it is processed as follows:

- The bitmap texel is zero extended up to 64 bits.
- The texel is byte swapped according to TextureReadMode0.ByteSwap field. If the 64 bit word has bytes labelled: ABCDEFGH then the three bits swap the bytes as follows:

Bit 2 (long swap)	Bit 1 (short swap)	Bit 0 (byte swap)	Swapped ABCDEFGH
0	0	0	ABCDEFGH
0	0	1	BADCFEHG
0	1	0	CDABGHEF
0	1	1	ABCDEFGH
1	0	0	EFGHABCD
1	0	1	FEHGBADC
1	1	0	GHEFCDAB
1	1	1	HGFEDCBA

- Next the texel is optionally mirrored. This is controlled by the **TextureReadMode0** *Mirror* bit. The mirror swaps bits:
  (0,. 63), (1, 62), (2, 61),…(31, 32).
- The texel is next optionally inverted under control of the **TextureReadMode0** *Invert* bit.
- When **TextureReadMode0** field *OpaqueSpan* is zero the texel is ANDed with the pixel mask to remove pixels from the mask. When **TextureReadMode0**.*OpaqueSpan*

is 1 the texel is ANDed with the color mask to control foreground/background color selection (which must be preloaded into the **FBBlockColor** and **FBBlockColorBack** registers). This is more fully described in section 4.3.3 and 4.3.4 above.

Windows normally supplies its bitmasks as a byte stream with successive bytes controlling 8 pixel groups at increasing x (i.e. towards the right edge). Bit 7 within a byte controls the leftmost pixel (for that group) and bit 0 the right most pixel. To match up the pixel mask order (bit 0 controls the left most pixel, bit 63 the right most pixel) the three byte swap bits are all set and the mirror bit set.

4.10.3 Indexed Textures

Indexed textures are a special case because they are stored as 8 bit texels and expanded to 32 bit texels when loaded. This makes the addressing and cache management slightly more complicated as addressing uses 8 bit texels while cache management uses 32 bit texels.

The secondary cache holds the texture data in its 8 bit format which reduces the number of memory reads when the access path is mainly in u across the texture map.

4.10.4 YUV 422 Textures

YUV textures are a special case because two texels are stored in a 32 bit word (so in this sense they are 16 bit texels), however the U and V components are shared so the 32 bit word represents two 24 bits texels (the spare 'alpha' byte is set to 255). If the input bytes in the 32 bit word are labelled:

Y1 V0 Y0 U0 (U0 in the ls byte)

then the two output words are formed (in the internal format):

255 V0 U0 Y0 and 255 V0 U0 Y1 (Y in the ls byte)

This arrangement of the YUV pixels in memory is called YVYU, but an alternative memory format (called VYUY) is also supported. In this case the bytes are labelled:

V0 Y1 U0 Y0 (Y0 in the ls byte)

4.10.5 Borders

Borders (in the OpenGL sense) are used only when the filter mode is bilinear and the wrapping mode is clamp. In this case when one of the filter points goes outside the texture map the border texel is read or (if it is not present) the border color is used. The border, if present, still needs to be skipped over and this will have already been done by incrementing the i, j indices before they arrive.

The width of a texture map is (2n + 2b) where b is 0 (no border) or 1 (border). Unfortunately the texture map width cannot simply be set to this value because the lower resolution mip map levels 'divide out the border' as the width is divided by 2 for each successive level. The **TextureMapWidth0** and **TextureMapWidth1** registers hold the width of the texture map without the border (in bits 0…11). If a border is present the border bit (bit 12) in **TextureMapWidth0** or **TextureMapWidth1** is also set.

If a 1x1 texture map has a border then the 3x3 map is stored as a 4x4 map as shown:

b0	b1	b2
b3	t0	b4
b5	b6	b7

b0	b1	b2	b2
b0	b1	b2	b2
b3	t0	b2	b4
b5	b6	b7	b7

Texels which fall into the border when no border is present are flagged - these texels are not checked in the cache and no texels are read from memory. The **T0BorderColor**… **T7BorderColor** flags used for this purpose select the **BorderColor0** (T0…T3) or **BorderColor1** (T4…T7) registers instead of the primary cache to provide the texture data. The **BorderColor0** and **BorderColor1** registers would normally be set to the same value for OpenGL when mip mapping.

4.11 Texture Implementation

Having looked at the memory management aspects of Texture Cacheing we can now examine the real-time sequence of events.

4.11.1 Overview

Texture functionality is organised into three groups: The *Primary Cache Manager*, *Address Generator* and *Dispatcher* form the core and work in a similar way to the other read units. The logical address translation is handled by the Address Mapper and TLB. Dynamic texture loading is handled by the Memory Allocater and the Download Controller.

Interfaces between all the units are FIFOs most of which are simply registers with full/empty flags for handshaking. The two shared resources which are managed in this way are the TLB and Memory Allocater. The TLB is mainly queried by the Address Mapper but the Memory Allocater needs to invalidate pages when a physical page is reassigned. The Memory Allocater allocates pages when requested by the Download Controller, but also needs to mark pages as 'most recently used' when requested by the Address Mapper.

There are two read/write ports to the Memory Controller used to access the Logical Page Table and the Physical Page Allocation Table - these are 64 bit ports and are not FIFO buffered. (It is unnecessary to queue reads or writes on these ports as the texture process stalls until these are satisfied.)

The texture data read port to the Memory Controller has a deep address FIFO and return data FIFO to absorb as much latency as possible.

The write port to the Memory Controller is used by the Download Controller to write texture data into memory during a download.

All the controlling registers (**TextureReadMode**, **TextureMapWidth**, **TextureBaseAddr**, etc.) are held in the Primary Cache Manager so the responsibility for loading them from the message stream, context dumping and readback is concentrated in one place. As a result, before any of them can be updated all outstanding work which may depend on them must be allowed to complete.

The sequence of events when a texture read command arrives under various conditions is as follows:

a) All the texel data is in the primary cache

The texels: (i0, j0, map), (i1, j0, map), (i0, j1, map), (i1, j1, map) for texture 0 and for texture 1 are checked in parallel in the Primary Cache Manager to see if they are in the primary cache.

b) One texel from texture 0 and one texel from texture 1 miss the primary cache

The cache allocation for both banks is checked simultaneously and the missing texels passed to the Address Generator via the AG0 and AG1 FIFOs for the corresponding banks. The step message, with the address of each texel filled in, is written to the M FIFO and the texel read count field on this step set to two. This part of the processing all happens in the same cycle so the fragment throughput is maintained.

The Address Generator processes the texel reads one at a time. It calculates the address for the texel in memory using the i, j and map values together with the appropriate **TexelReadMode** and **TextureMapWidth** values. The address is checked to see if it is in the secondary cache, and if it is then instructions to load the primary cache from the secondary cache are sent down the T FIFO. A more common case (for Patch32_2 or Patch2 layout) is that the secondary cache doesn't hold the texel so the Address Mapper is given the address and its type (logical or physical) via the AM FIFO.

For a physical texture the Address Mapper passes the address through to the Memory Controller via the Tx Addr FIFO. For a logical address the TLB unit is asked if this logical page is present and what the corresponding physical page is. If the TLB is hit the physical memory address is derived from the physical page and low order bits of the logical address and passed to the Memory Controller. If the TLB misses then the Address Mapper reads the Logical Texture Page Table entry for this logical page and, in this case, the page is resident so the corresponding physical page is now available. The physical memory address is derived from the physical page and low order bits of the logical address and passed to the Memory Controller. The TLB is updated so this logical page is the most recent one and its corresponding physical page recorded.

When the outstanding texel data (as shown by the texel read count field) has been loaded into the primary Texture Filter cache data is passed on as soon as the following unit can accept it. If, however the outstanding texel data has not been loaded then the step message is stalled until it has.

c) Two texels (from different texture maps) are not in the primary cache but are in physical memory.

The texels: (i0, j0, map), (i1, j0, map), (i0, j1, map), (i1, j1, map) for texture 0 and for texture 1 are checked in parallel in the Primary Cache Manager to see if they are in the primary cache.

d) One texel from texture 0 and one texel from texture 1 miss the primary cache

The cache allocation for both banks is checked simultaneously and the missing texels passed to the Address Generator via theAG0 and AG1 FIFOs for the corresponding banks. The step message, with the address of each texel filled in, is written to the M FIFO and the texel read count field on this step set to two. This part of the processing all happens in the same cycle so the fragment throughput is maintained.

The Address Generator processes the texel reads one at a time. It calculates the address for the texel in memory using the i, j and map values together with the appropriate TexelReadMode and TextrueMapWidth values. The address is checked to see if it is in the secondary cache, and if it is then instructions to load the primary cache from the secondary cache are sent down the T FIFO. A more common case

(for Patch32_2 or Patch2 layout) is that the secondary cache doesn't hold the texel so the Address Mapper is given the address and its type (logical or physical) via the AM FIFO.

The Address Mapper checks the TLB to see if the logical page is present and, if so, what its corresponding physical page is. When the logical page is not in the TLB the Address Mapper reads the entry in the Logical Texture Page Table for this logical page. The entry returns a resident bit and a physical page number. The resident bit is set so the physical page number is now known. The physical memory address is derived from the physical page and low order bits of the logical address and passed to the Memory Controller. The TLB is updated so this logical page is the most recent one and its corresponding physical page recorded.

When the outstanding texel data (as shown by the texel read count field) has been loaded into the primary Texture Filter cache the data is passed on as soon as the following unit can accept it. If, however the outstanding texel data has not been loaded then the step message is stalled until it has.

e) Two texels (from different texture maps) are not in the primary cache or in physical memory

The texels: (i0, j0, map), (i1, j0, map), (i0, j1, map), (i1, j1, map) for texture 0 and for texture 1 are checked in parallel in the Primary Cache Manager to see if they are in the primary cache.

f) One texel from texture 0 and one texel from texture 1 miss the primary cache

The cache allocation for both banks is checked simultaneously and the missing texels passed to the Address Generator via theAG0 and AG1 FIFOs for the corresponding banks. The step message, with the address of each texel filled in, is written to the M FIFO and the texel read count field on this step set to two. This part of the processing all happens in the same cycle so the fragment throughput is maintained.

The Address Generator processes the texel reads one at a time. It calculates the address for the texel in memory using the i, j and map values together with the appropriate **TexelReadMode** and **TextureMapWidth** values. The address is checked to see if it is in the secondary cache, and if it is then instructions to load the primary cache from the secondary cache are sent down the T FIFO. A more common case (for Patch32_2 or Patch2 layout) is that the secondary cache doesn't hold the texel so the Address Mapper is given the address and its type (logical or physical) via the AM FIFO.

The logical page is not in the TLB and the resident bit in the Logical Texture Page Table is clear so the Address Mapper writes to the host physical address (read from the page table) into the PCI **HostTextureAddress** register, the logical page into the PCI **LogicalTexturePage** register and the transfer length, memory pool and address type (set to host physical for this description) into the PCI **TextureOperation** register. Finally the PCI *TextureDownloadRequest* bit is set. The Address Mapper waits for the Texture Download Complete signal to be asserted by the Download Controller.

The Texture DMA Controller responds to the *TextureDownloadRequest* bit being set. It writes the logical address, transfer length and memory pool into the Texture Input FIFO and follows this data with the page of texture map data.

The Download Controller on receiving the logical page and pool information in the Texture Input FIFO, requests the Memory Allocator for the physical page to use for the download about to start. The Memory Allocator uses the Physical Page Allocation Table to allocate a physical page and asks the TLB to invalidate the logical page previously occupying (if any) the newly allocated physical page. The Memory

Allocator also updates the Logical Texture Page Table to mark the logical page as resident at the new physical page. The physical page is returned back to the Download Controller via the MAD FIFO.

The Download Controller on receiving the physical page transfers the texture data in the Texture Input FIFO to the given physical page. Once this is done the TextureDownloadComplete signal is asserted which releases the Address Mapper to complete its task.

The Address Mapper reads the Logical Texture Page Table entry for this logical page. Now that the page is resident the physical page is read from the Logical Texture Page Table. The physical memory address is derived from the physical page and low order bits of the logical address and passed to the Memory Controller. The TLB is updated so this logical page is the most recent one and its corresponding physical page recorded.

When the outstanding texel data (as shown by the texel read count field) has been loaded into the primary Texture Filter cache the data is passed on as soon as the following unit can accept it. If, however the outstanding texel data has not been loaded then the step message is stalled until it has.

4.11.2 Memory Interfaces

The Texture Read Unit has connections to four ports in the Memory Interface. The four ports are (in priority order from highest to lowest):

1. Memory Allocator Port
2. Address Mapper Port
3. Texture Write Port
4. Texture Read Port

This is an absolute priority and not based on any page break considerations

Note: The first two ports are not FIFO buffered and block subsequent texture processing until their read or write requests have been serviced.

4.11.2.1 Texture Read Port

This port is used to read texel data from memory. The addresses (after any necessary translation) are written into the Tx Addr FIFO and sometime later the 128 bits worth of data are returned via the Tx Data FIFO.

The following information is passed to the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0…1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2…29	Addr	28	The read address of the 128 bits of memory data.

The following information is passed back from the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0…127	Data	128	The data to be read from the memory.

4.11.2.2 Texture Write Port

This port is used by the Download Controller to write texture data into its allocated physical page. It is also used to update the Logical Texture Page Table to mark the page as being resident once it has been downloaded.

The following information is passed to the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0…1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2…29	Addr	28	The write address of the 128 bits of memory data.
30…45	ByteEnables	16	A high on a bit enables that byte to be written. The ls byte enable corresponds to data bits 0…7.
46… 173	Data	128	The data to be written to the memory.

The following information is passed back from the Memory Controller:

Bit No.	Name	Width	Description
0	TxWrComplete	1	This signal is asserted by the memory controller when the FIFO is empty and all writes from this port, the Memory Allocator Port and the Address Mapper Port have been written to memory so can be read from another port.

4.11.2.3 Memory Allocator Port

This port is used to update the Logical Texture Page Table with information from the host and to remove references from a physical page to a logical page in the Physical Page Allocation Table. The port is 64 bits wide (to save routing a 128 bit data bus from the Memory Controller). The read and write operations are buffered by a single level FIFO (to provide a simple interface) so will stall until their operations are satisfied.

The following signals are passed to the Memory Controller (MC):

Bit No.	Name	Width	Description
0…1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2	Command	1	0 = Write, 1 = Read
3…31	Addr	29	The write address of the 64 bits of memory data.
32…39	ByteEnables	8	A high on a bit enables that byte to be written. The ls byte enable corresponds to data bits 0…7.
40… 103	WrData	64	The data to be written to the memory.

The following signals are passed from the Memory Controller (MC):

Bit No.	Name	Width	Description
0	RdData	64	The data read from memory

4.11.2.4 Address Mapper Port

This port is used to update the Physical Page Allocation Table as pages are allocated or made the most recent accessed page. It is also used to mark logical pages in the Logical Page Table as non resident when the associated physical page is re-used. The port is 64 bits wide (to save routing a 128 bit data bus from the Memory Controller). The read and write operations are buffered by a single level FIFO (to provide a simple interface) so will stall until their operations are satisfied.

The following signals are passed to the Memory Controller (MC):

Bit No.	Name	Width	Description
0…1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2	**Command**	**1**	**0 = Write, 1 = Read**
3…31	Addr	29	The write address of the 64 bits of memory data.
32…39	ByteEnables	8	A high on a bit enables that byte to be written. The ls byte enable corresponds to data bits 0…7.
40… 103	WrData	64	The data to be written to the memory.

The following signals are passed from the Memory Controller (MC):

Bit No.	Name	Width	Description
0	RdData	64	The data read from memory

4.11.3 Translation Look-Aside Buffer (TLB)

The TLB responds to two command streams (serviced in round robin order):

- The Memory Allocator requests a logical page be invalidated if it is present.
- The Address Mapper checks if the logical to physical page mapping is already known before it takes the slower route of reading the Logical Texture Page Table. The TLB is fully associative and can provide the physical page (if present) in a single cycle. The update time can take longer if necessary as this will only occur after a Logical Texture Page Table read.

The TLB holds 16 entries for Permedia4. The TLB can report a maximum of one match for a given logical page

4.11.4 Memory Allocater

The Memory Allocator responds to two command streams (serviced in round robin order):

- The Download Controller asks for a physical page at the start of a new texture download. The tail page for the requested memory pool is allocated. The Physical Page Allocation Table is then updated to move the tail page to the head of the pool. The previous logical page assigned to the allocated physical page is marked as non resident in the Logical Texture Page Table and invalidated in the TLB. The physical page is returned to the Download Controller via the Memory Allocator FIFO.
- The Memory Allocator then reads the first word to find out about the texture which is just about to be received. It asks the Memory Allocator for a suitable physical page and once it has received this it copies the texture data into the memory. If the logical page number of the texture matches up with the one the Address Mapper was waiting for, the Address Mapper is notified it can continue by the **TextureDownloadComplete** signal and **TextureDownloadRequest** is cleared.

This subunit interacts with the Address Mapper via the following signals:

Name	Width	Description
PciTextureDownload Request	1	Asserted by the Address Mapper when it hits a page fault and needs a texture page downloaded and that page is not currently being downloaded. This is cleared by the Download Controller. This signal tells the Texture Download Controller a download is needed.
pciLogicalTexturePage	16	This is set by the Address Mapper to show what logical page it is requesting.
TextureDownload Request	1	This is asserted by the Address Mapper when it hits a page fault and need a texture page downloaded. This is cleared by the Download Controller when this page has been downloaded and the Logical Texture Page Table updated. This signal tells the Download Controller the pciLogicalTexturePage register holds a valid page number so it can inform the Address Mapper the download is complete (assuming the page matches).

TextureDownloadIn Progress	1	This is asserted by the Download Controller as is used to validate the DownloadLogicalPage value. The Address Mapper uses this to check if the download it want is currently being done.
DownloadLogicalPage	16	This is set by the Download Controller to identify the logical page it is in the process of downloading.
TextureDownload Complete	1	This is asserted by the Download Controller when it has finished downloading a texture the Address Mapper is waiting on.

4.11.5 Dispatcher

The Dispatcher holds the data part of the secondary cache and forwards texel data to the primary cache (in the Filter Unit). Texel data is allowed to flow through whenever it arrives from the Memory Controller, but under control from commands received via the T FIFO. A count of the texel data loaded for each filter bank (i.e. texture map) is maintained so that an active step message can be delayed until all the texel data it requires is present in the Filter Unit. In normal operation this delay should not be invoked very often.

The Dispatcher also handles span processing. This involves zero extending the texel data to a 64 bit bitmask, byte swapping, mirroring and inverting when necessary and finally anding the pixel mask in the span step message.

4.12 Texture DMA Controller

The P4 Texture DMA Controller handles a single request at a time. The following hardware signals are used to communicate between the Texture Read Unit and the Texture DMA Controller

- **pciTextureDownloadRequest**. This signal is asserted by Texture Read Unit to request a texture download. It is deserted once the texture download has started.
- **TextureFIFOFull**. This signal is asserted by the Texture Read Unit when it is not able to accept any more data being written into the TextureInput FIFO.

When the Texture DMA Controller has detected a download request it reads three PCI registers from the requester. These registers are:

- **HostTexturePage**. This register holds the host page (in bits 0…19) where the texture resides. This is either a physical page or a virtual page. A bit in the TextureOperation register identifies the type of page. If the page is a virtual page then an interrupt is generated and the host will read the page and initiate the DMA once the data has been made available. The conversion from page to address is done by multiplying by 4096.
- **LogicalTexturePage**. This register holds the logical page for the texture data and is returned back to the Texture Read Unit in bits 0…15[27] of the first entry written to the Texture Input FIFO (the FIFO is 128 bits wide) as a header preceding the actual texture data.
- **TextureOperation**. This register holds the following information:

[27] All 32 bits of the register are returned in bits 0…31 to allow for future capabilities.

Bit No.	Name	Description
0…8	Length	Transfer length in multiples of 128 bit words, maximum being 256
9…10	MemoryPool	Identifies which memory pool the physical page is to be allocated from.
11	HostVirtualAddress	download interrupt is generated, if enabled.

This data (and bits 12…31) are returned in bits 32…64 of the first entry written to the Texture Input FIFO (the FIFO is 128 bits wide) as a header preceding the actual texture data.

If the texture download request results in a **TextureDownload** interrupt being generated the **TextureAddr** PCI register is loaded with the virtual address and the **TextureOperation** PCI register is loaded with the TextureOperation data read from Texture Read before the interrupt is generated. The host services the interrupt, reads these two registers and provides the data. When the data is available in memory the physical address where the data is located is written in to the **TextureAddr** PCI register. This wakes up the texture download DMA controller and it executes the download.

5

Video System

The Permedia4 video system comprises three units; Video Unit, Video Overlay Unit, and RAMDAC. They work together to generate the display signal sent to the monitor, or flat-panel display. The Video Unit and RAMDAC are the key components of the system, responsible for displaying framebuffer images generated by the PERMEDIA 3 graphics core.

The Video Unit reads the raw framebuffer image data from memory, and forwards it to the RAMDAC, along with the video timing signals (HSync, VSync and Blank). The RAMDAC takes the raw image data from the Video Unit, extracts the individual pixels, and converts them into the 3 analogue outputs to the monitor.

The Video Overlay Unit compliments the functionality provided by Video Unit and RAMDAC, by providing a scaleable overlay region which the RAMDAC can combine with the framebuffer image. One application of the Video Overlay Unit is the display of MPEG video data, either in a window, or fullscreen, with the Video Overlay Unit automatically scaling the image to fit the destination region.

Both standard analogue output to a monitor, and digital output to a flat-panel display are supported by the RAMDAC.

To properly configure the video system to drive the display, both the Video Unit, and the RAMDAC must be initialised with suitable video timing parameters. The RAMDAC acts as the master, strobing data from the Video Unit at the correct frequency. It is driven by either an externally, or internally generated DClk signal.

Note: *When the SVGA Unit is active, it takes over the video system, and the Video Unit is disabled. The SVGA Unit must be programmed directly to set up the video timing when in this mode.*

5.1 Video Unit

The video unit is responsible for generating the video timing signals (HSync, VSync, and Blank), and reading the raw framebuffer image data from memory. This data is then forwarded to the RAMDAC for display on a suitable monitor. In addition, the external DDC2 compatible I2C bus is controlled through this unit.

5.1.1 Programming The Video Unit Timing Registers

The Video Unit must be programmed with the required video timing information and memory image parameters. The timing information controls the size and position of the image on the display, while the memory image parameters control the area of memory to be displayed.

All horizontal parameters in the Video Unit are expressed in 128-bit memory units. This restricts the granularity of these parameters to 4, 8 or 16 pixels, depending on the pixel depth.

e.g. If the pixel depth of the framebuffer image is 8-bits, then each 128-bit memory location holds 128/8 = 16 pixels, and hence all horizontal parameters are expressed as multiples of 16 pixels.

If the pixel depth of the framebuffer image is 32-bits, then each word of data holds only 4 pixels, and hence all horizontal parameters are expressed as multiples of 4 pixels.

If 8 pixel granularity is essential when using 8-bit pixel formats (e.g. for VESA timing compliance), the Video Unit can be programmed to use image byte doubling. However this requires the RAMDAC to be driven at twice the required dot clock, and so imposes a limit on the refresh rate. See section 5.1.4.6 for full details of this mode.

5.1.2 Setting the display memory region

The display memory region is defined by the **ScreenBase**, **ScreenStride**, and **ScreenBaseRight** registers.

All three registers are in 128-bit memory units.

5.1.2.1 ScreenBase

The **ScreenBase** register specifies the address in memory of the first visible pixel of the image. In stereo display mode, this register specifies the address in memory of the left hand image (See section 5.1.4.3).

5.1.2.2 ScreenStride

The **ScreenStride** register specifies the number of words between consecutive scanlines in memory.

e.g. If the image in the framebuffer is 1024 pixels wide, and the pixel depth is 32, then ScreenStride = (1024 * 32) / 128 = 256

Note: ScreenStride is not necessarily the same as screen width. If the image in memory is 1024 pixels wide, but the display width is 800 pixels, then ScreenStride must reflect the 1024 pixel figure, not the 800 pixel figure.

5.1.2.3 ScreenBaseRight

The **ScreenBaseRight** register is used only when stereo display mode is enabled in the **VideoControl** register. It specifies the address in memory of the first visible pixel of the right hand image (See section 5.1.4.3).

5.1.3 Setting Video Timing Parameters

The video timing parameters are controlled by 9 registers; **HTotal**, **HgEnd**, **HbEnd**, **HsStart**, **HsEnd**, **VTotal**, **VbEnd**, **VsStart**, and **VsEnd**.

The diagram below shows the relationship of each of these parameters.

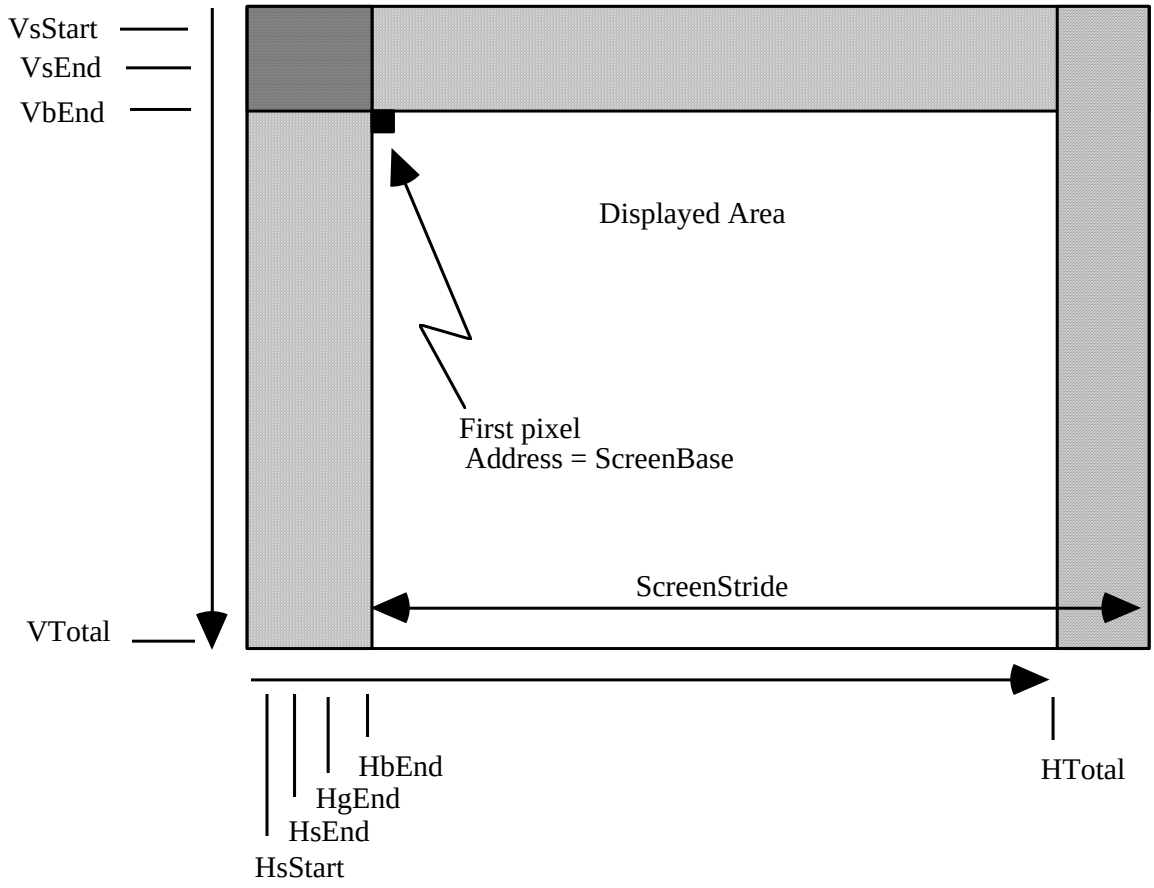


Figure 5.1 Video Timing Parameters

5.1.3.1 HTotal

This register holds the total number of horizontal memory cclocks per display scanline. It should be set to :

((Horizontal Total Period in pixels * Depth) / 128) - 1

5.1.3.2 HgEnd

This register indicates the horizontal cycle on which the Video Unit must start issuing video data to the RAMDAC. It is usually set to the same value as HbEnd. In some circumstances (such as when panning is activated) the RAMDAC requires video data one or more cycles before the end of horizontal blanking. In these situations HgEnd will be slightly lower than HbEnd.

5.1.3.3 HbEnd

Indicates the total width of the horizontal blank region, including the front porch and HSync. It should be set to :

(Horizontal Blank Period in pixels * Depth) / 128

5.1.3.4 HsStart

Indicates the start of the HSync period. It should be set to :

(Horizontal Front Porch in pixels * Depth) / 128

5.1.3.5 HsEnd

Holds the value of the first horizontal memory clock after the horizontal sync pulse. It should be set to :

((Horizontal Front Porch in pixels + Horizontal Sync Width in pixels) * Depth) / 128

5.1.3.6 VTotal

Holds the count of the last scanline in the display, including the vertical blank period. It should be set to :

Vertical Total Period - 1

5.1.3.7 VbEnd

Holds the scanline index of the first scanline out of vertical blank. It should be set to :

Vertical Blank Period

5.1.3.8 VsStart

Holds the line index of the last scanline before the vertical sync pulse. It should be set to :

Vertical Front Porch - 1

5.1.3.9 VsEnd

Holds the line index of the last scanline within the vertical sync pulse. It should be set to :

(Vertical Front Porch + Vertical Sync Height) - 1

5.1.4 Configuring the VideoUnit

The **VideoControl** register provides various control bits which affect the behaviour of the VideoUnit timing parameters, or enable/disable various special modes, such as framebuffer un-patching, and stereo support.

Bit 0 of this register is the unit enable toggle. This bit must be set to enable the VideoUnit output to the RAMDAC.

The *DisplayDisable* bit of the **VideoControl** register blanks the video display, and suppresses all reads from video memory. All video timing data is generated as normal, and the RAMDAC may still display its hardware cursor, and/or Video Overlay Image, if enabled. This is particularly useful for fullscreen video overlays, where none of the framebuffer image will be visible on the display, and hence reading from memory would waste bandwidth.

5.1.4.1 Video Sync Control

The *HSyncCtl*,*VSyncCtl* and *BlankCtl* fields of the **VideoControl** register configure the polarities of the timing signals sent to the RAMDAC and permit manual override of the HSync and VSync outputs.

The RAMDAC only supports active high polarity on its input signals, and so these registers should never be programmed to supply active low signals to the RAMDAC. The RAMDAC provides it's own control over the output polarities for VSync and HSync.

The *HSync* and *VSync* fields can also be used to directly set the value of the video sync outputs. This is used to control certain special monitor features, such as DDC (See section 5.1.4.7)

5.1.4.2 Buffer Swap Control

The *BufferSwap* field of the **VideoControl** register controls the time at which writes to the ScreenBase and HgEnd registers take effect. There are three possible modes of operation:

FreeRunning:

In this mode, writes to the ScreenBase and HgEnd registers take immediate effect. This can cause tearing of the video image if the registers are updated part way through a frame.

SyncOnFrameBlank:

In this mode, writes to the ScreenBase and HgEnd registers do not take effect, until the next vertical blank period. If the image is already in vertical blank, the writes have immediate effect.

LimitToFrameRate:

In this mode, writes to **ScreenBase** and **HgEnd** have immediate effect only if they have not already been written on the current frame. If two or more writes occur within the same frame, then subsequent writes are delayed until the next vertical blank period.

This avoids animation frame-rate errors which may occur if the GP Core is unable to keep up with the video frame-rate, but can cause tearing of the image.

The *BypassPending* field of the **VideoControl** register indicates whether the last write to **ScreenBase** or **HgEnd** has taken effect or not. If this bit is 1, the last write is still pending. If this bit is 0, all previous writes have taken effect.

5.1.4.3 Displaying Stereo Images

The VideoUnit and RAMDAC support the display of stereo video images. This is achieved by alternating display between two buffers, and driving the VidRightEye pin, to indicate which frame is currently being displayed.

To enable stereo output, the *Stereo* bit of the **VideoControl** register is set, and the required pin output polarity is set in the *RightEyeCtl* field of the **VideoControl** register.

Two video base addresses are required for stereo output. The address of the left image should be loaded into the **ScreenBase** register, and the address of the right image should be loaded into the **ScreenBaseRight** register. Both buffers must be the same size, so only one **ScreenStride** register is required.

The *RightFrame* field of the **VideoControl** register indicates which frame is currently being displayed by the VideoUnit. If this bit is 0, then the left frame is being displayed. If this bit is 1, then the right frame is being displayed. This bit is read only.

5.1.4.4 Displaying Patched Framebuffers

The GP Core can be configured to render into framebuffer memory in a 64 pixel wide by 16 pixel high patched format (patch64 mode). In some circumstances this can improve

memory efficiency by reducing the number of page breaks. If patch64 mode is being used by the GPCore, then the VideoUnit must be configured to reflect this, so that it can un-patch the image before displaying it.

Patching is enabled by the *PatchEnable* bit of the **VideoControl** register. In addition, the *PixelSize* field of this register must also be set, to reflect the pixel depth of the framebuffer. The *PixelSize* field only affects patched mode in the VideoUnit. If patching is not being used, then *PixelSize* has no effect.

Note: Patched formats are not supported when using line doubling (see below).

When patching is enabled, the **ScreenBase** and **ScreenBaseRight** registers must be aligned to the start of a patch. If a non patch-aligned screen origin is required, then the *PatchOffsetX* and PatchOffsetY fields of the **VideoControl** register must be adjusted to reflect the start offset within the patch.

5.1.4.5 Driving Very Low Resolution Images

Permedia4's video system has no practical lower bound on the display resolutions it can support. However many modern monitors are not able to lock on to low frequency signals such as that required for 320x200 resolution framebuffers. To help overcome this problem, the VideoUnit and RAMDAC can be configured to double the horizontal and vertical resolutions by pixel and line replication. The horizontal and vertical frequencies seen by the monitor are then twice the internal frequency.

Enabling the *LineDouble* bit in the **VideoControl** register causes the Video Unit to duplicate every scanline sent to the RAMDAC. All vertical timing information must be set up for the internal (un-doubled) frequency. i.e. if the framebuffer resolution is 200 lines, and this is being line doubled to 400 lines, the vertical timing parameters must be set to reflect the 200 line resolution.

Horizontal pixel doubling is performed by the RAMDAC. Once enabled (By setting the *PixelDouble* bit in the RAMDAC **RDMiscControl** register), each pixel is replicated before being transmitted to the monitor. Enabling pixel doubling also causes the horizontal timing information to be replicated. This effectively means that the horizontal timing parameters must be set to reflect the lower resolution frequencies. i.e. if the framebuffer resolution is 320 pixels, and this is being pixel doubled to 640 pixels, the horizontal timing parameters must be set to reflect the 320 pixel resolution.

When using horizontal, or vertical doubling, the RAMDAC dot clock frequency must be set to match the monitor frequency. i.e. for a 320x200 source image with doubling in both axes, the dot clock frequency must be set for a 640x480 display.

Note: Patched formats are not supported when using line doubling - refer to Peremdia3 Errata and Alerts, PEREN0012..

5.1.4.6 Driving 8-bit Images With 8 Pixel Timing Granularity

All horizontal timing has a granularity of 128-bit memory units. This is usually fine for 16-bit and 32-bit framebuffer depths, because the VESA standard specifies timings to 8-pixel granularity. For 8-bit color depths however, this imposes a minimum granularity of 16-pixels.

If VESA requirements must be met with 8-bit displays, then the Video Unit can be configured to perform byte doubling. This has a similar effect as RAMDAC pixel doubling described in section 5.1.4.5, in that the horizontal frequency is doubled. Each byte of image data is replicated before sending it to the RAMDAC, hence doubling the external

dot-clock frequency. Each 128-bit data packet sent to the RAMDAC now contains only 64-bits of image data, and therefore the horizontal timing parameters also represent 64-bit (8 pixel) spans.

Byte doubling is enabled by setting the **ByteDouble** bit in the Video Unit **MiscControl** register. When using byte doubling, the horizontal timing parameters should be adjusted to 64-bit units, instead of the usual 128-bit units, and the dot clock multiplied by 2.

5.1.4.7 Display Data Channel

The DDC interface allows PERMEDIA 3 to read timing information from a compatible monitor. Both DDC1 and DDC2 protocols are supported.

For DDC1, the data is read one bit at a time, and is clocked from the monitor by the vertical sync signal. The vertical sync should be controlled directly from software using the **VideoControl** register (See section 5.1.4.1)

Vertical sync should be driven high and the data will become valid 30 microseconds later; when the data has been read the vertical sync should be driven low for at least 20 microseconds before it is driven high again. Accurate timing can be derived from the MemCounter register in the memory controller register group. The input data is read from the DataIn bit of the DisplayData register.

For DDC2, an I2C bus interface is used. The I2C bus is monitored and controlled via the **DisplayData** register. The software can detect an external request, or data phase by enabling the DDC Interrupt in the **IntEnable** register. When a DDC2 start condition is detected, the Start bit of the **DisplayData** register is set. Similarly a Stop condition causes the Stop bit to be set. When data is transferred, it is latched into the *LatchedData* field, and the *DataValid* field is set.

The host may insert wait states to slow down a data transfer by setting the Wait bit of the **DisplayData** register. This effectively holds the clock output low until the bit is cleared again.

The *Start*, *Stop* and *DataValid* bits within the **DisplayData** register may be cleared by writing to this register with the respective bit positions set to 1.

The *ClkIn* and DataIn fields of the **DisplayData** register indicate the current state of the respective lines on the I2C bus.

The host may control the bus, causing Permedia4 to become a bus master, by writing to the *ClkOut* and *DataOut* fields of the **DisplayData** register.

5.1.5 Video FIFO control

The Video Unit framebuffer image returns data from the memory controller in a 32 entry FIFO. To ensure efficient use of memory, read requests will usually be low priority requests, causing several consecutive requests to queue up in the memory controller, and then complete in a burst. The priority of the requests is controlled by the number of returned data items in the video FIFO. If there is not enough data queued up in the FIFO, the Video Unit will assert high priority to the memory controller. The thresholds for high and low priority are controlled by the FifoControl register. If the number of spaces in the FIFO is greater than or equal to the HighThreshold field of this register, the Video Unit asserts high priority. If the number of spaces is less than or equal to LowThreshold, high priority will be deasserted.

5.1.5.1 Interrupt Generation

The Video Unit provides three interrupt outputs; Vertical Retrace Interrupt, Scanline Interrupt, and Video DDC Interrupt. These are visible in the Region 0 IntFlags register, and are enabled and disabled in the Region 0 IntEnable register. The interrupts are cleared by writing to the IntFlags register with the corresponding bits set to 1.

The Vertical Retrace Interrupt is signalled at the rising edge of VSync, if enabled.

The Scanline Interrupt is generated at the start of the video scanline specified in the InterruptLine register.

The VidDDC Interrupt indicates that the chip has detected either an DDC2 start condition, or a DDC2 data phase.

In addition, FIFO underflow sets the VideoFifoUnderflow error flag in the Region 0 **ErrorFlags** register. Error conditions in the **ErrorFlags** register can be made to signal an interrupt, by setting the *ErrorFlag* bit in the **IntEnable** register. The source of the error can then be determined, by examining the **ErrorFlags** register.

5.1.6 Example Timing Values

5.1.6.1 Example 1 - Timing Values for 640x480 16 BPP 75Hz

VESA Timings for this mode are:

HSync front porch	24	pixels	
HSync width	64	pixels	
HSync back porch	88	pixels	
Horizontal blank width	176	pixels	(24 + 64 + 88)
Horizontal total width	816	pixels	(640 + 176)

Scanline(480 + 22)

HsStart = HSync front porch in 128-bit words
= (24 * 16) / 128
= 3
HsEnd = (HSync front porch + HSync width) in 128-bit words
= ((24 + 64) * 16) / 128
= 11
HbEnd = Horizontal blank width in 128-bit words
= (176 * 16-bits) / 128
= 22
HgEnd = HbEnd
= 22
HTotal = (Horizontal total width in 128-bit words) - 1
= ((816 * 16) / 128) - 1
= 101
VsStart = VSync front porch - 1
= 1 - 1
= 0
VsEnd = (VSync front porch + VSync width) – 1
= (1 + 3) - 1
= 3

VbEnd = Vertical blank period
= 22
VTotal = Vertical line total – 1
= 502 – 1
= 501

5.1.6.2 Example 2 - Timing Values for 800x600 32 BPP 75Hz

VESA Timings for this mode are:

HSync front porch	40	pixels	
HSync width	80	pixels	
HSync back porch	120	pixels	
Horizontal blank width	240	pixels	(40 + 80 + 120)
Horizontal total width	1040	pixels	(800 + 240)

VSync front porch	1	scanline	
VSync width	3	scanlines	
VSync back porch	23	scanlines	
Vertical blank period	27	scanlines	(1 + 3 + 27)
Vertical total period	627	scanlines	(600 + 27)

HsStart = HSync front porch in 128-bit words
= (40 * 32) / 128
= 10
HsEnd = (HSync front porch + HSync width) in 128-bit words
= ((40 + 80) * 32) / 128
= 30
HbEnd = Horizontal blank width in 128-bit words
= (240 * 32) / 128
= 60
HgEnd = HbEnd
= 120
HTotal = (Horizontal total width in 128-bit words) - 1
= ((1040 * 32) / 128) - 1
= 259
VsStart = VSync front porch - 1
= 1 - 1
= 0
VsEnd = (VSync front porch + VSync width) – 1
= (1 + 3) - 1
= 3
VbEnd = Vertical blank period
= 27
VTotal = Vertical line total – 1
= 627 – 1
= 626

5.2 RAMDAC

The primary purpose of the RAMDAC is to convert the internal digital video data, from the Video Unit, into an analogue output suitable for driving a monitor. Data is supplied from the Video Unit as raw 128-bit packed words, so the RAMDAC must extract the individual pixel values from this data.

The RAMDAC can overlay, or blend, pixel data from the Video Overlay Unit, and overlay a hardware cursor before the final data is send to the display. An alternative form of overlay is also available, in addition to the Video Overlay. This feature allows an 8-bit overlay image, stored in the alpha-channel of the framebuffer, to be applied over the framebuffer image.

PERMEDIA 3 provides two internal PLL's for clock generation, in addition to the external clock sources. All clock selection is controlled through the RAMDAC.

5.2.1 Programming The RAMDAC registers

The RAMDAC registers are either accessed directly or indirectly. The direct registers are accessed by reading or writing the appropriate offset. The indirect registers are accessed by writing their respective 16-bit index to the Index Registers (**RDIndexLow** and **RDIndexHigh**) and then either reading from, or writing to, the **RDIndexedData** Register.

For example, to write to the **RDColorFormat** register (index 004h), the index is first set by writing 04h to **RDIndexLow**, and 00h to **RDIndexHigh**. The data can then be written to the **RDIndexedData** register.

Accesses to several consecutive indirect registers can be accelerated by enabling auto-increment mode. This is achieved by setting bit 0 of the **RDIndexControl** register. Once enabled, the indirect address registers are automatically advanced after every access, permitting consecutive registers to be accessed with a single read/write cycle.

The following tables show the direct and the indirect registers.

Offset (hex)	Mode	Register Name
000	R/W	RDPaletteWriteAddress
008	R/W	RDPaletteData
010	R/W	RDPixelMask
018	R/W	RDPaletteReadAddress
020	R/W	RDIndexLow
028	R/W	RDIndexHigh
030	R/W	RDIndexedData
038	R/W	RDIndexControl

Table 5.1 RAMDAC Direct Register Map

Index (hex)	Mode	Register Name
000	R/W	RDMiscControl
001	R/W	RDSyncControl
002	R/W	RDDACControl
003	R/W	RDPixelSize
004	R/W	RDColorFormat

Index (hex)	Mode	Register Name
005	R/W	RDCursorMode
006	R/W	RDCursorControl
007	R/W	RDCursorXLow
008	R/W	RDCursorXHigh
009	R/W	RDCursorYLow
00A	R/W	RDCursorYHigh
00B	R/W	RDCursorHotSpotX
00C	R/W	RDCursorHotSpotY
00D	R/W	RDOverlayKey
00E	R/W	RDPan
00F	R	RDSense
018	R/W	RDCheckControl
019	R	RDCheckPixelRed
01A	R	RDCheckPixelGreen
01B	R	RDCheckPixelBlue
01C	R	RDCheckLUTRed
01D	R	RDCheckLUTGreen
01E	R	RDCheckLUTBlue
01F	R/W	RDScratch
020	R/W	RDVideoOverlayControl
021	R/W	RDVideoOverlayXStartLow
022	R/W	RDVideoOverlayXStartHigh
023	R/W	RDVideoOverlayYStartLow
024	R/W	RDVideoOverlayYStartHigh
025	R/W	RDVideoOverlayXEndLow
026	R/W	RDVideoOverlayXEndHigh
027	R/W	RDVideoOverlayYEndLow
028	R/W	RDVideoOverlayYEndHigh
029	R/W	RDVideoOverlayKeyR
02A	R/W	RDVideoOverlayKeyG
02B	R/W	RDVideoOverlayKeyB
02C	R/W	RDVideoOverlayBlend
02D -1EF		Reserved
1F0 - 215	R/W	PLL Setup Registers (See separate table)
300 - 302		Reserved
303 - 32F	R/W	RDCursorPalette0..44
330 - 3FF		Reserved
400 - 7FF	R/W	RDCursorPattern0..1023

Table 5.2 RAMDAC Indirect Register Map

5.2.2 Basic RAMDAC Configuration

The RAMDAC must be configurared to match the layout of the framebuffer data being sent from the Video Unit, and the DClk frequency set according to the video timing requirements. See section 5.2.5.5 for details on setting the system clocks.

5.2.2.1 Defining the framebuffer pixel format

The framebuffer pixel format is defined by the **RDPixelSize** and **RDColorFormat** indirect registers.

The **RDPixelSize** register should be initialised to reflect the depth of the framebuffer. Write 0 for 8-bit pixel depth, 1 for 16-bit depth, or 2 for 32-bit depth.

The **RDColorFormat** register has 3 fields; *ColorFormat*, *RGB*, and *LinearColorExtension*.

Set the ColorFormat field to match the GP Core framebuffer format. Note that the RAMDAC ColorFormat values do not have the same mapping as those in the GP Core. For example, the value required for RGB565 format in the RAMDAC **RDColorFormat** register is 16, but the GP Core **DitherMode** register value for the same mode is 3.

The RGB field of the **RDColorFormat** register indicates the pixel color order. If the value in this field is 1, the color order is RGB, if the value is 0, the color order is BGR.

The *LinearColorExtension* field of the **RDColorFormat** register controls the scaling algorithm used to convert non 8-bit color components into 8-bits. Color format RGB 565, for example, uses 5-bits for red and blue, and 6-bits for green. These components must be scaled to 8-bits internally. If the *LinearColorExtension* bit is 0, then color components are scaled by shifting left, and filling the low bits with zero. If this bit is 1, then color components are scaled by shifting left, and then copying the high order bits into the newly exposed low bits.

5.2.2.2 Configuring Sync Signals

The RAMDAC **RDSyncControl** register is used to control the output sync signals to the display. The *VSyncCtl* and *HSyncCtl* fields select the output polarities of the VSync and HSync signals, and also provide options to directly drive these signals high, low, or tri-state. The *VSyncOverride* and *HSyncOverride* fields provide a second mechanism for forcing the VSync and HSync signals to their high states. The *VSyncOverride* and *HSyncOverride* fields take priority over the *VSyncCtl* and *HSyncCtl* fields, and hence provide a convenient means to temporarily pull the syncs high (for instance to trigger monitor standby mode), without affecting any assigned output polarity settings.

5.2.2.3 Controlling the DAC output signals

The three analogue color output signals can be individually controlled via the **RDDACControl** register.

The *BlankPedestal* field of the **RDDACControl** register enables pedestal blanking, in which the color channels are 'propped-up' by a pedestal voltage when in active video. This permits a suitable monitor to differentiate between blank and active by the voltage level of the color outputs. If this bit is 1, pedestal blanking is enabled. If this bit is 0, pedestal blanking is disabled.

The three color outputs may be individually disabled by enabling one or more of the *BlankRedDAC*, *BlankGreenDAC*, or *BlankBlueDAC* fields in the **RDDACControl** register. When these bits are set to 1, the corresponding output is forced low, effectively removing the associated color from the display.

The *SyncOnGreen* field enables sync-on-green mode, in which the VSync and HSync signals are combined with the green output signal. This functionality may not be available on all versions of Permedia4.

The *DACPowerCtl* field of the **RDDACControl** register enables low power shutdown of the DAC.

5.2.2.4 Enabling CheckSums

You can use the **RDCheckControl** register to tell the RAMDAC to sum the R, G and B values for a scan line. Typically, wait for Vblank, enable checksum before or after LUT, wait for RAMDAC to sum the first active scanline (after which enable bits are Reset) then read the **RDCheckLUT*** or **RDCheckPixel*** registers for the corresponding RGB component values..

5.2.3 Color Palette RAM

Video Image color values are usually translated by indexing each color components red, green, and blue values into a 3x256 entry color palette. This functionality may be disabled by writing 1 into the *DirectColor* field of the **RDMiscControl** register. The hardware cursor does not use the color palette, since it has it's own dedicate cursor pallete (See section 5.2.5.3 for details of the cursor palette).

The color palette RAM is addressed by the **RDPaletteWriteAddress** and **RDPaletteReadAddress** registers. These registers are automatically incremented following a RAM transfer, allowing the entire palette to be accessed with one write to the appropriate address register. When an address register increments beyond the last location in the RAM it is reset to the first location.

Color data may be transferred either as full width 8-bit values, or as 6-bit values. The *HighColorResolution* bit of the **RDMiscControl** indirect register controls selection of these two modes. Setting this bit to 1 enables 8-bit mode, clearing this bit selects 6-bit mode.

Internally, the color palette RAM is 8 bits wide for each color component even when 6-bit mode is chosen. If 6-bit mode is chosen and the color data is written into the palette, the 6 LSB bits will be shifted to the 6 MSB positions and the 2 LSBs filled with 0's. In addition, if they are read back in the 6 bit mode, the 6 MSB bits will be shifted to the 6 LSB positions and the 2 MSBs filled with 0's.

To load the color palette, the CPU first writes to the **RDPaletteWriteAddress** register with the index of the first entry to be modified. The selected palette RAM location is loaded a byte at a time by writing a sequence of three bytes (red, green and blue) to the RDPaletteData register. After the blue write cycle, the **RDPaletteWriteAddress** register increments to the next location.

To read from the color palette, the CPU first writes to the **RDPaletteRedAddress** register (Direct register: 18h) with the index of the entry to be read. Three successive reads from the palette RAM data register supplies red, green, and blue color data for the specified location. Following the blue read cycle, BOTH the **RDPaletteReadAddress** and the **RDPaletteWriteAddress** registers are incremented.

The **RDPixelMask** register is an 8-bit register used to enable or disable a bit plane from addressing the color-palette RAM in the SVGA mode. Each palette address bit is logically ANDed with the corresponding bit from the read-mask register before going to the palette page register and addressing the palette RAM

5.2.4 Panning The Video Display

The Video Unit can only align the framebuffer image to whole 128-bit memory words. If finer grain positioning is required (for virtual desktop panning for instance), the RAMDAC must be used to fine tune the position to 32-bit accuracy. The pixel granularity supported is determined by the framebuffer pixel depth; If the framebuffer depth is 32-bits, the screen can be positioned to single pixel accuracy, but if the depth is 8-bits, only 4 pixel accuracy is possible.

The first stage in positioning is to adjust the Video Unit base address to the start of the 128-bit memory address of the target start pixel. If the start pixel is 128-bit aligned, then nothing more is required, and RAMDAC panning must be disabled. If the start pixel is not 128-bit aligned, then RAMDAC panning must be enabled.

Panning in the RAMDAC is achieved in two stages[28]; The first shifts the 128-bit aligned image right 64-bits, and the second stage shifts the image right 32-bits. In addition, the Video Unit must be configured to supply the video data one cycle before the end of horizontal blanking, by setting **HgEnd** to (HbEnd – 1). Doing this effectively shifts the image left by 128-bits, before the RAMDAC right shift is applied. The combined effect of these two shifts is a left shift by 128 – (Total right shift).

The two shift modes are controlled by the **RDPan** register. To enable right shifts by 64-bits, set the Gate flag in this register. To enable 32-bit shifts, enable the Pan flag. The possible combinations are illustrated in the following table :

Gate (64)	Pan (32)	Adjust HgEnd ?	Total Left Pan In Bits
NO	NO	NO	0
NO	YES	YES	96
YES	NO	YES	64
YES	YES	YES	32

The following algorithm achieves 32-bit alignment of the framebuffer, using the technique described above :

```
ScreenBaseReg = (Address32 >> 2);
if (Address32 & 3) {
  int right_shift = 4 - (Address32 & 3);
  RDPanReg.Gate = right_shift & 2;
  RDPanReg.Pan  = right_shift & 1;
  HgEndReg      = HbEndReg - 1;
}
else {
  RDPanReg.Gate = 0;
  RDPanReg.Pan  = 0;
  HgEndReg      = HbEndReg;
    }
```

[28] Refer to *Permedia4 Errata and Alerts*, PEREN0013 about 8 bit panning. Use the byte double mode (**Misccontrol** register), to achieve 4-pixel resolution - each 8 bit pixel is issued twice so to pan to 4 pixels only requires 64 bit accuracy.

5.2.5 Configuring The Cursor

Permedia4 supports a user-defineable hardware cursor with up to 15 colors, and a maximum size of 64x64 pixels. The cursor is configured via the **CursorMode** register. To make the cursor visible the *CursorEnable* bit must be set, and a cursor format selected. The format defines the size of the cursor and which part of the cursor pattern RAM should be used.

When the cursor is disabled it takes effect immediately, but when it is enabled it does not take effect until the next frame blank. This allows easy update of the cursor pattern RAM; the cursor can be disabled, the new pattern loaded, and then the cursor re-enabled, but the new pattern does not take effect until the next blank which prevents tearing of the cursor.

5.2.5.1 Selecting the Cursor Format

Four types of cursor are supported; the Windows cursor, the X cursor, the 3-color cursor, and the 15-color cursor. These are defined by the *Type* field of the **CursorMode** register.

The table below lists the supported cursor types, and shows how each of the possible cursor pattern values are interpreted :

Pattern Value	Windows	X	3 Color	15 Color
0	Color 1	Transparent	Transparent	Transparent
1	Color 2	Transparent	Color 1	Color 1
2	Transparent	Color 1	Color 2	Color 2
3	Highlight	Color 2	Color 3	Color 3
4..15	-	-	-	Color n

Highlight refers to the bitwise inversion of the pixel color.

The cursor can be either 32 pixels, or 64 pixels square. This is selected by the *Format* field of the **RDCursorMode** register. If a 64 pixel square cursor is selected, then its depth is 2-bits, providing a maximum of 3 colors (plus transparent). If a 32 pixel square cursor is selected, then the cursor pattern RAM may be partitioned into as many as four different cursor patterns. The number of cursor patterns that can be stored is governed by the pixel depth required. If 4 bits pixel depth is required, only two patterns are available, but for 2-bit depths, four patterns may be stored. See section 5.2.5.4 for more details on cursor patterns.

The *ReversePixel* field in the **RDCursorMode** register is used to reverse the order in which a row of pixels is read from the cursor RAM. If it is enabled the incrementing locations are placed on the screen from right to left.

The cursor may also be pixel doubled in X or Y independently. This function is normally used in conjunction with line and pixel doubling used to support low-resolution screens (See section 5.1.4.5). Cursor doubling is enabled by the *DoubleX* and *DoubleY* fields of the **RDCursorControl** register.

5.2.5.2 Positioning The Cursor

The cursor position is set by the **RDCursorXLow**, **RDCursorXHigh**, **RDCursorYLow**, and **RDCursorYHigh** registers. The XY position defined by these registers is the position of the origin of the cursor with respect to the top left visible pixel of the screen. The **RDCursorHotSpotX** and **RDCursorHotSpotY** registers specify the position of the origin of the cursor within its defined size.

Reading the cursor position registers returns either the current cursor position, or the last assigned cursor position. These may not be the same because a new cursor position is not used until the **RDCursorYHigh** value is written, to prevent the cursor flickering due to a partially formed address. Which value is returned is governed by the *ReadbackPosition* field in the **RDCursorControl** register. If this bit is 1, the current visible screen position of the cursor is returned, otherwise the currently assigned cursor position is returned.

5.2.5.3 Cursor Color Registers

The registers for the 15 cursor colors are accessed through the 45 **RDCursorPalette** indirect registers. The palette is organized into RGB triplets. **RDCursorPalette0** holds the red value for color 1, **RDCursorPalette1** holds the green value for color 1, e.t.c.

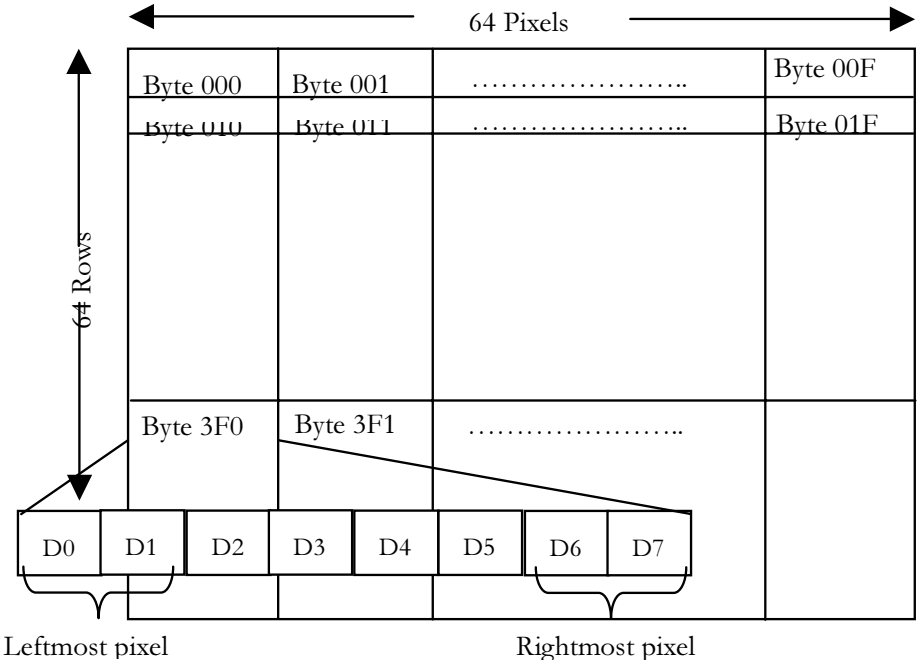
5.2.5.4 Cursor RAM

The cursor is controlled through the **CursorMode** register. To make the cursor visible it must be enabled, and its format selected. The format defines the size of the cursor, and which part of the cursor RAM should be used. When the cursor is 64x64 2-bit pixels, the entire RAM is used to hold a single pattern, but if the cursor is smaller, more than one pattern will fit in the RAM, so a cache of cursor patterns can be maintained.

The format field selects both the size and the part of the RAM that holds the pattern. The cursor RAM is divided into quadrants, numbered 0 and 1 on the bottom row, and 2 and 3 on the top row. The register description defines the area of the RAM to use in terms of these quadrants.

The cursor RAM is loaded by writing to the **RDCursorPattern** registers; each register holds 8 bits which make up either 4 or 2 consecutive pixels along a row depending on the number of bits per pixel. **RDCursorPattern0** holds the top left bits of the first cursor.

The diagram below illustrates the format of the cursor pattern RAM for the 64x64 4-color cursor.



5.2.5.5 RAMDAC Overlay

The RAMDAC overlay feature allows an overlay image to be stored in the alpha-channel of the framebuffer, and displayed on top of the framebuffer image. This feature is distinct from the Video Overlay, which uses a completely separate memory buffer, and allows arbitrary scaling to be applied. See section 5.3 for details on programming the Video Overlay Unit.

To enable RAMDAC overlay, set the **Overlay** bit in the **RDMiscControl** register, and define an overlay transparency color in **RDOverlayKey**. Once enabled, the overlay channel is displayed in place of the framebuffer color channel (and Video Overlay). Wherever the overlay color matches the **RDOverlay** value, the overlay will be transparent, and the framebuffer (and Video Overlay, if enabled) will be visible.

When the overlay is selected, all three color channels are assigned the same overlay pixel value, and these are always passed through the color palette, regardless of the setting of the **DirectColor** bit in **RDMiscControl**. By programming the color palette for the overlay region, 256 color overlays can be displayed. If direct color is enabled (by setting the **DirectColor** bit in **RDMiscControl**), full 64K or 16M color framebuffer images can still be achieved.

5.2.6 Digital Flat Panel Display Output

In addition to driving standard analogue monitors, the RAMDAC provides a digital output port for driving flat panel displays. This port is routed through the Video Streams B Unit, and is enabled by setting the **VSBOutput** bit in the **RDMiscControl** register to 1.

5.2.7 Programming The Clocks

There are 7 clock domains in PERMEDIA 3; DClk, IClk, JClk, KClk, MClk, PClk and SClk. PClk is the external PCI clock, and so can not be adjusted inetrnally. IClk and JClk are the

Video Streams A and B clocks respectively, and like PClk, are driven externally. The remaining four clock domains can be configured to use either an external reference clock, or one of two internal Phase-Locked Loops (PLL's). All clock configuration (including PLL setup) is through registers in the RAMDAC.

The D,M, S, and K Clocks are controlled by indirect RAMDAC registers. These are listed in the table below.

Index (hex)	Mode	Register Name
1F0	R/W	RDDClkSetup1
1F1	R/W	RDDClkSetup2
1F2	R/W	RDKClkSetup1
1F3	R/W	RDKClkSetup2
200	R/W	RDDClkControl
201	R/W	RDDClk0PreScale
202	R/W	RDDClk0FeedbackScale
203	R/W	RDDClk0PostScale
204	R/W	RDDClk1PreScale
205	R/W	RDDClk1FeedbackScale
206	R/W	RDDClk1PostScale
207	R/W	RDDClk2PreScale
208	R/W	RDDClk2FeedbackScale
209	R/W	RDDClk2PostScale
20A	R/W	RDDClk3PreScale
20B	R/W	RDDClk3FeedbackScale
20C	R/W	RDDClk3PostScale
20D	R/W	RDKClkControl
20E	R/W	RDKClkPreScale
20F	R/W	RDKClkFeedbackScale
210	R/W	RDKClkPostScale
211	R/W	RDMClkControl
212-214		Reserved
215	R/W	RDSClkControl

Table 5.3 RAMDAC Clock Register Map

5.2.7.1 PLL Programming

There are two internal PLL's; One may used to drive DClk, and the other KClk. MClk and SClk can be locked to the KClk frequency[29], so may also be affected by the KClk PLL. The DClk PLL has four sets of control registers, so that several alternate frequencies can

[29] The memory clock should be tied to the graphics processor clock - refer to *Permedia4 Errata and Alerts*, PEREN003. If slow speed memories are used, the memory clock may be run from, for example, an external clock with a frequency lower than the graphics processor clock. Care should be taken when using the power saving mode of the graphics processor clock as it may result in it having a lower frequency than the memory clock.

be configured, and the required frequency selected without reconfiguring the PLL. The KClk PLL has only one set of control registers.

Each set of PLL control registers comprise 3 registers :-

ClkPreScale, ClkFeedbackScale and **ClkPostScale**

These are used to control the output frequency, according to the following formula :-

Output Frequency = (Frequency of reference clock * ClkFeedbackScale) /

(ClkPreScale * (1 << ClkPostScale))

The first two sets of DClk control registers are configured at reset to generate 25.057MHz (Set 0), and 28.278 MHz (Set 1). The KClk PLL is configured at reset to generate 50MHz. The second pair of DClk control registers are un-initialised at reset. These frequencies assume that the external reference clock is 14.31818MHz.

5.2.7.2 DClk Programming

DClk (or Dot Clock) is used to control the frequency of the video output from the RAMDAC, and must be set to a frequency suitable for the display resolution and refresh rate.

The DClk frequency may be locked to one of four sources; the PLL's, the external reference clock, Video Stream A (IClk), or Video Stream B (JClk). The source is selected by the **RDDClkControl** indirect RAMDAC register.

If DClk is driven by the PLL, then the frequency is controlled by one of four sets of PLL registers: 0, 1, 2 and 3. Only one of them can be selected at a time. The selection is controlled by the VClkCtl register (bits 1 and 0) located at offset 0000.0040h of Memory Region Zero.

5.2.7.3 KClk Programming

The KClk is used to control the operation of the GP Core. It can be locked to one of 2 sources; the KClk PLL, or PClk. When using the PLL, the frequency is controlled in much the same way as for DClk. However, only one set of PLL control registers is provided for KClk. It is also possible to drive KClk at half the PClk frequency.

5.2.7.4 MClk Programming

The MClk is used to control the operation of the memories. It is usually set to the highest frequency compatible with the timings of the memory parts fitted. It can be sourced from one of 3 sources; PClk, KClk, or the external MClk. An option is also provided to drive MClk at half the selected clock source.

5.2.7.5 SClk Programming

The SClk is used to control the operation of the graphics core setup units (Host-In, and Delta). It can be sourced from PClk, KClk, the external SClk, or half these frequencies.

5.3 Video Overlay

The Video Overlay unit provides a second source of video data, which can be displayed in addition to, or instead of, the conventional framebuffer image from the Video Unit. Like the Video Unit, the Video Overlay Unit sources its data from the local memory, but rather than passing this data straight through to the RAMDAC, it extracts the individual pixels, and converts these to RGB888 format before feeding them to the RAMDAC. The RAMDAC can

display this image on top of the framebuffer image from the Video Unit, or use blending, or color keying to combine the two images.

The Video Overlay Unit can also apply an arbitrary scale factor, to resize the source image to fit the destination region. When scaling, the unit can optionally apply a bi-linear filter, to reduce the aliasing effects of pixel replication.

Source data may be RGB8888, RGB4444, RGB5551, RGB332, YUV422, YUV444 or CI8 (8-bit monochrome) format.

Note: *If full-screen Video Overlay is used and no blending is to be used in the RAMDAC (i.e. the image will completely obscure the framebuffer image), then the Video Unit should be programmed to disable framebuffer reads (See section 5.1.4). This will reduce the demand on the memory, and could improve the throughput of the GP Core. Timing information will still be required from the Video Unit, so it should not simply be disabled.*

5.3.1 Programming The Video Overlay Unit Registers

The Video Overlay Unit registers are located in the Video Unit region, at address offset 3100H. All registers are direct registers.

5.3.2 Basic Video Overlay Configuration

To correctly display video overlay data, it is necessary to configure both the Video Overlay Unit and the RAMDAC. The Video Overlay must be configured to indicate the layout of the source image in memory, together with required scaling, filtering, and buffer-swap synchronization parameters. The RAMDAC must be configured to reflect the final scaled image dimensions, required screen origin, and blend mode.

5.3.2.1 Setting The Video Overlay Source Image Parameters

The Video Overlay units supports triple-buffering of input frames, by providing three memory base address registers. The addresses of the three buffers are stored in the **VideoOverlayBase0**, **VideoOverlayBase1**, and **VideoOverlayBase2** registers. These addresses are pixel addresses, rather than memory addresses. For example, if the byte address of the buffer in memory is 40000H, and the pixel format is RGB8888, the pixel address is :-

40000H / 4 BytesPerPixel = 10000H

The **VideoOverlayBase** registers actually contain two fields; Bits 30-31 specify the memory type of the source buffer. On Permedia4 the *MemoryType* field should always be 0. If single or double buffering is used, only one, or two buffer registers need be used.

In addition to setting the base addresses, the **VideoOverlayIndex**[30] register must be set to select the initial source buffer. This simply involves writing the buffer index (0, 1 or 2) to bits 0-1 of this register. Bit 31 of this register controls even and odd field selection for interlaced video (see section 5.3.4).

The *Flip* field of the **VideoOverlayMode** register should be set to 0. This causes the unit to update its internal base address when the Video Unit signals a vertical retrace event. all other values for *Flip* are reserved on Permedia4.

[30] After updating the VideoOverlayIndex, or VideoOverlayOrigin registers, the VideoOverlayUpdate register must be written to trigger the update. See section 5.3.2.8

The *BuffserSync* field of the **VideoOverlayMode** register must be set to 0 to specify manual buffer swapping. All other values for *BufferSync* are reserved.

VideoOverlayStride specifies the scanline stride, in pixels.

The source image dimensions are stored in the **VideoOverlayWidth**, and **VideoOverlayHeight** registers. These are the dimensions of the visible portion of the image, not necessarily the dimensions of the entire source buffer. It is possible to display a portion of the source buffer, in which case, the visible image is smaller than the source buffer.

The **VideoOverlayOrigin** register controls the start pixel of the source image region, relative to the buffers base address. This can be used to pan a window around a larger internal buffer.

5.3.2.2 Defining The Video Overlay Color Format

The color format is selected by the *YUV* and *ColorFormat* fields of the **VideoOverlayMode** register. For RGB formats, set *YUV* to 0, and then select the required RGB format in the **ColorFormat** register. For YUV422 or YUV444, set the *YUV* field to 1 or 2, respectively. When YUV is selected, the **ColorFormat** field is ignored.

The *PixelSize* field of the **VideoOverlayMode** register must be set to match the pixel depth of the source buffer. Set 0 for 8-bit images, 1 for 16-bit images, or 2 for 32 bit images.

If the pixel format is not RGB8888, or one of the YUV formats, the *LinearColorExtension* bit of the **VideoOverlayMode** register is used to control color channel scaling. If this bit is 0, each color channel is scaled to 8-bits by left shifting, and padding the lower bits with 0. If this bit is 1, then the color channels are scaled by shifting, and then copying the high order bits into the low order bits.

If the video overlay image is to be indexed through the color palette, then the *DirectColor* bit of the **RDVideoOverlayControl** register should be cleared. If the video overlay image should be displayed directly, without tranlating thtough the palette, this bit should be set.

If the video overlay image is stored in Patched64 format (See section 5.1.4.4), the *PatchMode* bit of the **VideoOverlayMode** register should be set, otherwise this bit should be cleared.

5.3.2.3 Setting The Video Overlay Destination Region

Once the source image parameters have been assigned in the Video Overlay Unit, the RAMDAC must be configured to map the output onto the display. Three pieces of information are required for this; The screen position at which to display the overlay image, the screen position of the bottom right corner of the overlay image, and finally, video overlay output must be enabled by setting the *Enable* bit of the **RDVideoOverlay** register.

The **RDVideoOverlayXStartLow**, **RDVideoOverlayXStartHigh**, **RDVideoOverlayYStartLow**, and **RDVideoOverlayYStartHigh** registers define the X and Y screen coordinates at which to start displaying the video overlay data.

The **RDVideoOverlayXEndLow**, **RDVideoOverlayXEndHigh**, **RDVideoOverlayYEndLow**, and **RDVideoOverlayYEndHigh** registers define the X and Y coordinates at which to stop display overlay data. In other words, they hold the end X/Y screen location + 1.

Together, the **RDVideoOverlayX/YStart** and **RDVideoOverlayX/YEnd** registers indicate the dimensions of the video overlay region on the display. These dimensions should be used to derive the video overlay scaling parameters. It is essential that the dimensions

used in the RAMDAC match the scaled output dimensions of the Video Overlay Unit. Any mismatch will result in image skew. See section 5.3.2.9 for information on setting scaling parameters.

5.3.2.4 Setting The Video Overlay Application Mode

The RAMDAC supports 3 basic modes for applying video overlay data to the framebuffer image; Opaque, Blend, and Color-key. The required mode is selected by the *Mode* field of the **RDVideoOverlayControl** register.

5.3.2.5 Opaque Video Overlay Regions

The opaque mode of operation causes video overlay pixels to completely obscure the framebuffer. This mode is selected by assigning the *Mode* field of the **RDVideoOverlayControl** register to 2 (Always).

5.3.2.6 Blended Video Overlay Regions

Blending causes the video overlay, and framebuffer images to be blended in 0:1, 1:3, 3:1, or 1:1 ratios. The ratio selection can either be constant, if *BlendSrc* in R**DVideoOverlay Control** is 1 (Register), or per-pixel, if *BlendSrc* is 0 (Main). Blending is enabled by setting the *Mode* field in the **RDVideoOverlayControl** register to 3 (Blend).

If constant blending is selected, the blend factor is taken from the **RDVideoOverlayBlend** register. If per-pixel blending is enabled, the blend factor for each pixel is taken from the framebuffer alpha channel. In either mode, the blend factors are stored as 8-bit integers, but currently only the top 2-bits are used to give a 2-bit value. These values are interpreted as follows :-

Blend Factor	Video Overlay Contribution	Framebuffer Contribution
0	0%	100%
1	25%	75%
2	75%	25%
3	100%	0%

5.3.2.7 Color-keyed Video Overlay Regions

When color-key mode is selected, a transparency color is defined either for the framebuffer image, making the video overlay image visible only where the framebuffer pixel is transparent, or for the video overlay image, making the framebuffer image visible only where the video overlay color is transparent. These two modes are referred to as MainKey, and OverlayKey respectively.

Color-keying through the framebuffer is enabled by setting the *Mode* field of the **RDVideoOverlayControl** register to 0 (MainKey). In this mode, the test for transparency can either be made against the framebuffer color channel, or the framebuffer alpha channel. To enable color channel keying, set the *Key* field in the **RDVideoOverlayControl** register to 0 (Color), and set the transparency color in the **RDVideoOverlayKeyR/G/B** registers. To enable alpha channel keying, set the *Key* field to 1 (Alpha), and set the transparent alpha key in the **RDVideoOverlayKeyR** register.

Color-keying through the video overlay image is enabled by setting the *Mode* field of the **RDVideoOverlayControl** register to 1 (OverlayKey). There is no alpha-channel in the video overlay image sent to the RAMDAC, so the *Key* field of the **RDVideoOverlayControl** register is ignored. The transparent color is set in the **RDVideoOverlayKeyR/G/B** registers.

When keying from either the framebuffer, or video overlay color channel, the transparent color specified in the **RDVideoOverlayKeyR/G/B** registers should be set to the RGB888 color (before LUT translation), matching the required transparent color. This is determined by the pixel format, and the **LinearColorExtension** mode of the corresponding buffer, e.g. if the buffer is in RGB565 pixel format, LinearColorExtension is disabled, and the required transparent color is full intensity blue, then **RDVideoOverlayKeyB** would be set to (0x1F << 3) = 0xF8.

If LinearColorExtension is enabled, **RDVideoOverlayKeyB** would be (0x1F << 3) | (0x1F >> 2) = 0xFF.

When keying from the framebuffer alpha channel, the transparent color is a single 8-bit value. In this case, since the framebuffer value originates from the alpha-channel, no scaling of the color is performed before the transparency test. Therefore, the value which should be written to **RDVideoOverlayKeyR** is unaffected by the **LinearColorExtension** mode and should not be prescaled to 8-bits.

So for example if the framebuffer format is RGB5551, then a 1-bit transparency key should be written to bit0 of **RDVideoOverlayKeyR.** If the framebuffer format is RGB4444, then the key is a 4-bit value written to bits 0..3 of **RDVideoOverlayKeyR.**

5.3.2.8 Triggering Parameter Updates

The Video Overlay Unit implements a register double buffering scheme, in which updates to certain registers are delayed until the start of the next video frame. This prevents image tearing occuring when a change is made part way through a frame. For the update to occur, bit 0 of the **VideoOverlayUpdate** register must also be set. Once written, the *VideoOverlayUpdate* bit will remain set, until the update takes effect at the start of the next frame.

If interrupt triggered updates are required, since the Video Overlay Unit run synchronously to the Video Unit, the vertical retrace, or programmable scanline interrupt can be used, to signal the start of a new frame, or a particular scanline. The vertical retrace interrupt will be generated at the start of each frame, and any previous update will take effect. The Video Overlay Unit can then be setup ready for the next frame, and **VideoOverlayUpdate** rewritten, to prime then next update. See section 5.1.5.1 for details of these interrupts.

The registers which are delayed are **VideoOverlayIndex**, **VideoOverlayZoomXDelta, VideoOverlayShrinkXDelta,VideoOverlayYDelta** and **VideoOverlayOrigin.**

5.3.2.9 Mirroring Video Overlay Images

Source images can be mirrored in both X and Y, before displaying. This is controlled through the *MirrorX*, and *MirrorY* bits in the **VideoOverlayMode** register.

5.3.3 Scaling Images Through The Video Overlay Unit

The Video Overlay Unit can perform arbitrary scaling of video data, about both axes, and can optionally filter subsequent aliasing affects by applying a bilinear filter. It is essential that the scaling parameters are correctly set, to reflect the relative source and destination region sizes, otherwise image skewing will occur.

Scaling is controlled by specifying X and Y deltas in three Video Overlay registers. The deltas control the ammount by which the unit advances its input coordinates for each horizontal or vertical output pixel. For example, if an X delta of 0.5 is specified, then the unit will advance by ½ a source pixel for every output pixel, resulting in horizontal magnification by a factor of 2. If an X delta of 2 is specified, then the unit will advance by two source pixels for every output pixel, resulting in a reduction by a factor of 2.

Scaling in X is decomposed into two stages. The first applies a reduction delta (In **VideoOverlayShrinkXDelta**) with a value greater than or equal one. This reduces the internal frequency of the unit during reduction, which would otherwise need to execute faster than the dot clock frequency of the display. The second stage applies a magnification delta (in **VideoOverlayZoomXDelta**) with a value less than or equal to one.

Generally the two X deltas are mutually exclusive; If zooming is required, **VideoOverlayZoomXDelta** is used and **VideoOverlayShrinkXDelta** is set to 1, otherwise **VideoOverlayShrinkXDelta** is used and **VideoOverlayZoomXDelta** is set to 1. In certain applications, such as displaying very large source images, it may be necessary to apply both stages; Reducing the image first, to reduce the internal clock frequency, and then magnifying the image to the required size.

Scaling in Y requires just one register; **VideoOverlayYDelta.**

The required deltas for X and Y can be determined from the source and destination region dimensions as follows :-

XDelta = (Width of source image in memory – 1) / Width of image on display

YDelta = (Height of source image in memory – 1) / Height of image on display

The deltas are stored in unsigned fixed point format, either in 12.12 format (YDelta, and ShrinkXDelta), or 1.12 format (ZoomXDelta). The fixed point register value corresponding to a floating point delta can be calculated in C as follows :-

```
Register = ((unsigned long) (Delta * 0x10000)) & mask
```

Where mask is either 0x0FFFFFF0 (for 12.12 results), or 0x0001FFF0 (for 1.12 results).

Two stages of filtering are provided; Partial and Full. Partial filtering is applied in the horizontal direction during zooming in X. Full filtering is applied for all scaling, in both axes. Filtering is enabled by setting the *FilerMode* field of the **VideoOverlayMode** to 1 (Full), or 2 (Partial). Setting this field to 0 disables filering.

5.3.4 Interlaced Video With The Video Overlay Unit

There are two approaches to displaying interlaced video. The first involves combining each pair of fields into a single buffer and displaying this at half the interlaced frequency. The second approach is to retain the interlaced properties and display each field as soon as it has been geneated.

The first approach requires no special Video Overlay Unit setup but does require more memory since each buffer is now twice as large. A second drawback to this approach is that it can introduce jitter because the image is being updated less frequently.

The second approach is usually the best option, and in its simpliest form, also requires no special Video Overlay configuration. Each video field can be treated as a separate frame, one being written to **BaseAddress0**, and the other to **BaseAddress1**. After each field has been completed, the **VideoOverlayIndex** register should be switched to point to the new buffer.

The Video Overlay Unit provides a Bob de-interlacing algorithm[31]. This allows odd video fields to be offset slightly in the Y axis, to simulate a true interlaced display. This mode is controlled by the **DeInterlace** field in the **VideoOverlayMode** register. Setting this field to 1 enables Bob de-interlacing. Setting this field to 0 disables de-interlacing. All other values are reserved.

When using Bob de-interlacing, the index of the current field should be written to bit 31 of the **VideoOverlayIndex** register. This is used to enable, or disable the Y axis offset. For even fields, set this bit to 0. For odd field set this field to 1.

The odd field Y Offset should be written to the **VideoOverlayFieldOffset** register. This register is in unsigned 12.12 fixed point format. The fixed point value corresponding to a floating point offset can be calculated in C as follows :-

VideoOverlayFieldOffset = ((unsigned long) (Offset * 0x10000)) & 0x0FFFFFF0

Normally, the odd fields are offset by **VideoOverlayFieldOffset** when Bob de-interlacing is enabled. Setting the **FieldPolarity** bit of the **VideoOverlayMode** regoster reverses this, so that the even fields are offset instead.

5.3.5 Video Overlay Unit Fifo Control

The Video Overlay Unit stores image return data from the memory controller in two 16 entry fifos. To ensure efficient use of memory, read requests will usually be low priority requests, causing several consecutive requests to queue up in the memory controller, and then complete in a burst. The priority of the requests is controlled by the number of returned data items in the video fifos. If there is not enough data queued up in either fifo, the Video Overlay Unit will assert high priority to the memory controller. The thresholds for high and low priority are controlled by the **VideoOverlayFifoControl** register. If the number of spaces in either fifo is greater than or equal to the **High** field of this register, the Video Overlay Unit asserts high priority. If the number of spaces in both fifos is less than or equal to **Low**, high priority will be deasserted.

If the memory system fails to supply the Video Overlay Unit fast enough to prevent image loss, the Video Overlay Unit will set the *FifoUnderflow* bit of the **VideoOverlayStatus** register. Once set, this bit will remain set, until it is cleared by writing **VideoOverlayStatus** with this bit set to 1. This bit can be monitored, to determine whether the fifo threshold settings are sufficient to prevent underflow.

Note: *It will not always be possible to prevent underflow conditions, even with very stringent threshold settings. If an underflow problem persists, with tight thresholds, it probably indicates that the source image is too large for the available memory bandwith. The resolution of the source image should either be reduced, or alternatively, it may be possible to read the image if it is reduced within the video overlay unit, and then magnified again, by using both* ***VideoOverlayShrinkXDelta****, and* ***VideoOverlayZoomXDelta*** *simultaneously.*

[31] But see *Permedia4 Errata and Alerts PEREN006*