3D*labs*®

PERMEDIA® 2

Programmer's Reference Manual

Issue 5

Email: info@3Dlabs.com
WWW: http://www.3Dlabs.com

3D*labs Inc.*
181 Metro Drive, Suite 520,
San Jose, CA 95110
United States
Tel: (408) 436 3455
Fax: (408) 436 3458

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

Change History

Document	Issue	Date	Change
147.2.0	1	18 March 97	First Issue
147.2.0	2	30 April 97	Minor textual changes.
147.2.0	3	31 May 97	Added missing registers RectangleOrigin and RectangleSize. Minor typographical fixes.
147.2.0	4	30 July 97	Added TexelLutIndex register to appendix A. Corrected section 1.1.
147.2.0	5	30 Nov 97	Correct text on perspective correction in 5.7.1. Change V0, V1 and V2Float[0..15] to be [0..14] and fog range to be –1.0 to +1.0 in Appendix A. Change V0, V1 and V2Float[14] to be [15] in Appendix E. Correct diagrsm and text for TexelLUTAddress in Appendix A. Minor typographical changes.

Contents

1. Introduction 1
1.1 How to use this manual 1
1.2 Further Reading 2

2. Overview 3
2.1 PERMEDIA 2 Key Features 3
2.2 Functional Overview 4

3. Programming Model 7
3.1 PERMEDIA as a Register file 8
3.2 PERMEDIA I/O Interface 10
3.3 Interrupts 20
3.4 Synchronization 20
3.5 Host Memory Bypass 21
3.6 DMA Controller 22
3.7 Register Read back 22
3.8 Byte Swapping 23
3.9 Red and Blue Swapping 23

4. Memory I/O and Organization 24
4.1 Patched Data 24
4.2 Localbuffer 24
4.3 Framebuffer 26
4.4 Double Buffering 31
4.5 Texture Buffer 35

5. Graphics Programming 37
5.1 The Graphics HyperPipeline 37
5.2 Delta Unit 39
5.3 Rasterizer Unit 44
5.4 Scissor/Stipple Unit 60
5.5 Localbuffer Read and Write Units 65
5.6 Stencil/Depth Test Unit 68
5.7 Texture Address Unit 76
5.8 Texture Read Unit 79
5.9 YUV Unit 86
5.10 Framebuffer Read and Write Units 89
5.11 Color DDA Unit 96
5.12 Texture/Fog/Blend 99
5.13 Color Format Unit 108
5.14 Logical Op Unit 111
5.15 Host Out Unit 113

6. Initialization 119
6.1 Initializing PERMEDIA 119
6.2 System Initialization 119

6.3 Window Initialization 123
6.4 Application Initialization 125
6.5 Bypass Initialization 126

7. Programming Tips 127
7.1 PCI Bus Issues 127
7.2 Graphics Hyperpipeline 129
7.3 Area Filling Techniques 130
7.4 Copies and Downloads 131
7.5 Multi Buffering 133
7.6 Overlays 133
7.7 Memory Organization 134
7.8 Chroma Test 134
7.9 Configuration for 2D 135

8. Delta Programming Examples 136

Appendix A. Graphics Register Reference 146

Appendix B. Pseudocode Definitions 257

Appendix C. Screen Widths Table 259

Appendix D. A Gouraud Shaded Triangle without using the Delta Unit 260

Appendix E. Register Tables 266

Appendix F. PERMEDIA 1 and PERMEDIA 2 Differences 276

Glossary 282

Index 287

Table of Figures

Figure 2.1 External Interfaces 4
Figure 3.1 DMA Tag Description Format 14
Figure 3.2 Indexed Format 15
Figure 5.1 Hyperpipeline 38
Figure 5.2 Triangle Mesh. 40
Figure 5.3 Triangle Fan. 40
Figure 5.4 Rasterizing a triangle. 45
Figure 5.5 Polyline 47
Figure 5.6 Relationship between Bitmask and Scanning Directions 50
Figure 5.7 Copy Operation 53
Figure 5.8 Real Coordinate Representation 55
Figure 5.9 Screen Scissor and User Scissor Tests 61
Figure 5.10 Scissor Mode Register 62
Figure 5.11 AreaStippleMode Register 62
Figure 5.12 LBReadMode Register 67
Figure 5.13 LBWriteMode Register 67
Figure 5.14 LBReadFormat / LBWriteFormat Register 67
Figure 5.15 Depth Interpolation 72
Figure 5.16 Depth Derivative Format 72
Figure 5.17 StencilMode Register 72
Figure 5.18 StencilData Register 73
Figure 5.19 DepthMode Register 73
Figure 5.20 Window Register 74
Figure 5.21 Texture Address Interpolation 76
Figure 5.22 Fixed Point S and T Format 77
Figure 5.23 Fixed Point Q Format 77
Figure 5.24 TextureAddressMode 78
Figure 5.25 TextureReadMode Register 81
Figure 5.26 TextureMapFormat Register 82
Figure 5.27 TextureDataFormat Register 82
Figure 5.28 TexelLUTMode Register 83
Figure 5.29 TexelLUTAddress register 83
Figure 5.30 YUVMode Register 87
Figure 5.31 ChromaUpperBound and ChromaLowerBound Registers RGB Format 88
Figure 5.32 ChromaUpperBound and ChromaLowerBound Registers YUV Format 88
Figure 5.33 FBReadMode Register 94
Figure 5.34 FBWriteMode Register 94
Figure 5.35 FBReadPixel Register 94
Figure 5.36 PackedDataLimits Register 95
Figure 5.37Color Representation 96
Figure 5.38 Color Interpolation 97
Figure 5.39 Fixed Point Color Format 97
Figure 5.40 ColorDDAMode Register 98
Figure 5.41 Fog Interpolation Over A Triangle 102
Figure 5.42 Fog Interpolant Fixed Point Format 102
Figure 5.43 Fogging 103
Figure 5.44 TextureColorMode Register 105
Figure 5.45 Texel0 Register - RGB and YUV formats 105
Figure 5.46 FogMode Register 106

Figure 5.47 AlphaBlendMode Register 106
Figure 5.48 Dither Mode Register 109
Figure 5.49 LogicalOpMode Register 113
Figure 5.50 FilterMode Register 116
Figure 5.51 StatisticMode Register 117
Figure 5.52 PickResult Register 117
Figure 8.1 Geometry of the Mesh and Clip regions. 136

List of Tables

Table 2.1 Standard VGA Modes 5
Table 2.2 VESA SVGA Modes 6
Table 3.1 Memory Regions 7
Table 3.2 Region 0 Address Map 7
Table 4.1 Supported Color Formats 29
Table 5.1 Vertex Parameters 39
Table 5.2 Draw Command Bit Field Assignments Affecting Delta 41
Table 5.3 DeltaMode Register Bit Field Assignments. 43
Table 5.4 Rasterizer Command Registers 56
Table 5.5 Rasterizer Control Registers 57
Table 5.6 Render Command Register Fields 58
Table 5.7 Rasterizer Mode Register 59
Table 5.8 Localbuffer Read/Write Modes 66
Table 5.9 Stencil Comparison Modes 69
Table 5.10 Possible Update Operations for Stencil Planes 69
Table 5.11 Stencil Operations 70
Table 5.12 Stencil Sources 70
Table 5.13 Depth Comparison Modes 71
Table 5.14 Depth Sources. 71
Table 5.15 Depth Interpolation Registers 74
Table 5.16 Texture Interpolation Registers 77
Table 5.17 Chroma Test Modes 87
Table 5.18 Framebuffer Read/Write Modes 91
Table 5.19 Color Interpolation Registers 98
Table 5.20 Logical Operations 111
Table 5.21 Filter Modes 114
Table 7.1 Memory Organization 134

1. Introduction

PERMEDIA 2 is a high performance PCI/AGP graphics processor that balances high quality 3D polygon and textured graphics acceleration, windows acceleration and state-of-the-art MPEG1/MPEG2 playback with a fast integrated SVGA core, integrated RAMDAC and video ports. This document provides a high level overview of the architecture of the PERMEDIA 2 graphics processor and is intended as an introduction for design engineers and project managers planning the implementation of PERMEDIA 2 based systems.

PERMEDIA 2 sets the standard for 3D and multimedia acceleration, making it the ideal solution to meet the increasingly pervasive need for balanced 3D and multimedia acceleration - and all in a single, low cost PCI device.

This document has been written as the primary reference for programmers and system designers who wish to develop software to drive PERMEDIA 2. Information on programming the I/O registers can be found in the *PERMEDIA 2 Hardware Reference Manual.*

PERMEDIA 2 is the second generation PERMEDIA device. Compared with PERMEDIA 1, it provides greater flexibility, additional features and enhanced performance. Throughout this manual the terms PERMEDIA 2 and PERMEDIA are used interchangeably.

An understanding of the principles of 2D and 3D graphics programming will be useful in reading this document.

1.1 How to use this manual

Chapter 2 gives an overview of PERMEDIA.

Chapter 3 details the programming model for the chip.

Chapter 4 describes the data formats that PERMEDIA supports in the framebuffer, localbuffer and texture buffer.

Chapter 5 describes how to use PERMEDIA for graphics rendering.

Chapter 6 describes the initialization of PERMEDIA.

Chapter 7 provides tips for programming PERMEDIA.

Chapter 8 provides examples of Delta programming.

Appendix A details the PERMEDIA registers.

Appendix B gives the format used in the pseudocode examples throughout the document.

Appendix C gives a table used to set-up common screen widths.

Appendix D describes how a Gouraud shaded triangle can be rendered without using the Delta Unit. This is helpful in understanding how the chip works and also when dealing with PERMEDIA 1 legacy.

Appendix E tabulates the PERMEDIA 2 registers.

Appendix F describes the differences between PERMEDIA 1 and 2

A Glossary of technical terms follows the Appendices.

An extensive index is included.

1.2 Further Reading

- *PERMEDIA 2 Hardware Reference Manual, 3Dlabs*
- *PERMEDIA 2 Architecture Overview, 3Dlabs*
- *OpenGL Programming Guide, Jackie Neider et al, Reading MA: Addison-Wesley*
- *Microsoft WIN32 Software Development Kit 3.1, Microsoft*
- *Windows NT 3.1 Graphics Programming, Emeryville CA, Ziff-Davis Press*
- *Computer Graphics: Principles and Practice, James D. Foley et al, Reading MA: Addison-Wesley*
- *Programmer's Guide to the EGA, VGA and Super VGA Cards, Richard F. Ferraro, Reading MA: Addison-Wesley, ISBN 0-201-62490-7*

2. Overview

2.1 PERMEDIA 2 Key Features

- Full support for Intel's Accelerated Graphics Port (AGP) and PCI
  - 66 MHz operation
  - DMA and Execute mode support
  - Sideband addressing
- Enhanced 3D graphics features and performance (at 83MHz)
  - 83M perspective correct, bilinear filtered, texture mapped pixels/sec
  - 42M perspective correct, bilinear filtered, texture mapped, depth buffered pixels/sec
  - 800K texture mapped polygons/sec
  - True-color 3D graphics
  - Polygon based with Z buffer
  - Texture decompression
  - Full scene anti-aliasing
- Enhanced GUI acceleration
  - Ultra-fast BLT engine and 2D rasterizer
  - Stretch BLTs, monochrome/color expansion and logic ops
  - 8, 16, 24 and 32-bit packed framestore
- MPEG2 compatible Video playback acceleration
  - YUV 4:4:4, YUV 4:2:2 and YUV 4:2:0 (native MPEG2 format)
  - Unlimited multiple playback windows (occluded)
  - Independent XY scaling and mirroring
- Integrated geometry pipeline set-up processor
- Integrated true-color 230 MHz RAMDAC
  - 320x200 to 1600x1200 screen resolution
  - DPMS, DDC1 and DDC2AB+
  - Clock synthesizer and Hardware cursor
- Multi-mode video streams
  - Simultaneous input and output video
  - Optional scaling and filtering
  - Optional color space conversion and gamma correction
- Fast on-chip SVGA
- Flexible multi-function SDRAM or SGRAM memory (2, 4, 6 or 8 Mbytes)
- Microsoft PC97 and Intel GPC97 compliance
- Comprehensive suite of optimized software drivers
- Reference board designs and manufacturing kits

2.2 Functional Overview

2.2.1 Memory Subsystem

PERMEDIA provides flexible support for the memory subsystem (Fig. 2.1). This allows the system designer a wide choice of price/performance tradeoffs.

The same physical memory holds all data used by PERMEDIA. Internally the data types are divided into texture, localbuffer and framebuffer. The localbuffer holds depth and stencil data; the framebuffer holds color data for display.

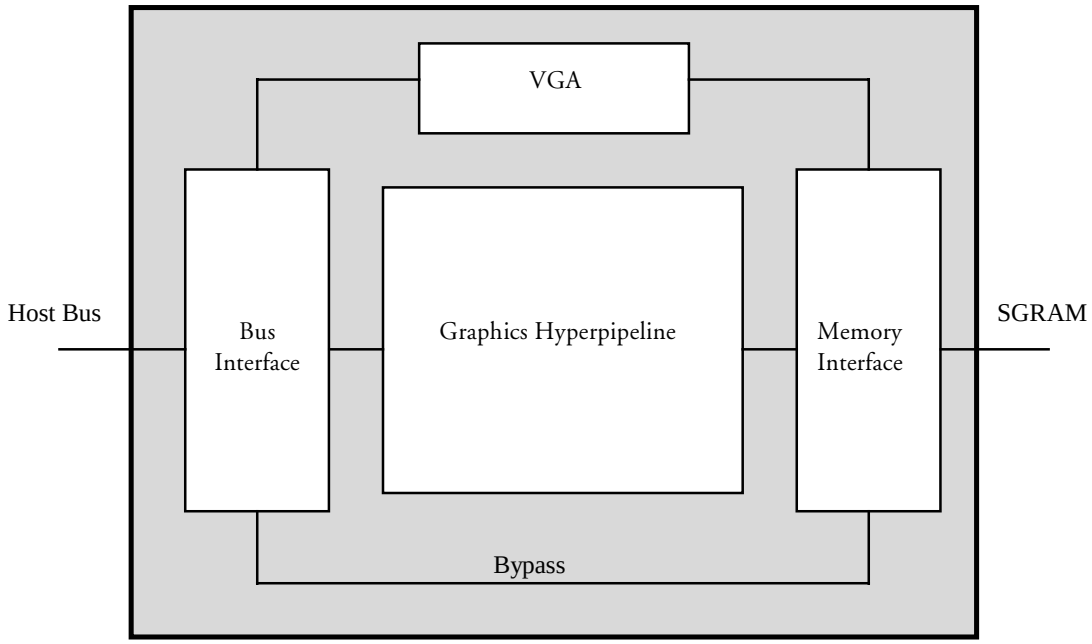


Figure 2.1 External Interfaces

2.2.2 Host Interface

Conceptually PERMEDIA can be viewed as a register file. Control registers are primed with the information required for a primitive, and then to start the chip drawing, a write is made to a Command register

PERMEDIA registers can be accessed directly through the memory map. Registers can be accessed either individually or in groups.

The chip also supports a bypass route to the memory to allow direct read/write of pixels, and implementation of algorithms not directly supported by PERMEDIA.

2.2.3 Task Switching

Where multiple applications wish to make simultaneous access to PERMEDIA, it is the responsibility of the software driving the chip to handle the loading of correct state. PERMEDIA has been designed to support a number of different software architectures.

- Synchronous operation means that a new task can load its context without waiting for current rendering to complete
- All loadable state can be read back
- A Sync command is provided to flush all rendering. This can be polled or it can return an interrupt

2.2.4 SVGA

PERMEDIA contains a fast VGA core. The PERMEDIA SVGA is used for DOS VGA applications and during boot time before switching to use the Graphics Hyperpipeline. This document does not cover VGA programming. Specific information on PERMEDIA's VGA can be found in the *PERMEDIA 2 Hardware Reference Manual*. VGA information, such as standard registers, is described in the "Programmer's Guide to the EGA, VGA and Super VGA Cards" by Richards F. Ferraro.

The following standard VGA modes are supported:

Mode (hex)	Alpha Format	Char Size	Colors	Max Page	Type Format	Resolution
00 0	40 by 25	8 by 8	16/256K bw	8	Alpha	320 by 200
0*	40 by 25	8 by 14	16/256K bw	8	Alpha	320 by 350
0+	40 by 25	9 by 16	16/256K bw	8	Alpha	360 by 400
01 1	40 by 25	8 by 8	16/256K	8	Alpha	320 by 200
1*	40 by 25	8 by 14	16/256K	8	Alpha	320 by 350
1+	40 by 25	9 by 16	16/256K	8	Alpha	360 by 400
02 2	80 by 25	8 by 8	16/256K bw	8	Alpha	640 by 200
2*	80 by 25	8 by 14	16/256K bw	8	Alpha	640 by 350
2+	80 by 25	9 by 16	16/256K bw	8	Alpha	720 by 400
03 3	80 by 25	8 by 8	16/256K	8	Alpha	720 by 200
3*	80 by 25	8 by 14	16/256K	8	Alpha	640 by 350
3+	80 by 25	9 by 16	16/256K	8	Alpha	720 by 400
04 4	40 by 25	8 by 8	4/256K	1	Graph	320 by 200
05 5	40 by 25	8 by 8	4/256K bw	1	Graph	320 by 200
06 6	80 by 25	8 by 8	2/256K bw	1	Graph	640 by 200
07 7	80 by 25	9 by 14	bw	8	Alpha	720 by 350
7+	80 by 25	9 by 16	bw	8	Alpha	720 by 400
0D D	40 by 25	8 by 8	16/256K	8	Graph	320 by 200
0E E	80 by 25	8 by 8	16/256K	4	Graph	640 by 200
0F F	80 by 25	8 by 14	bw	2	Graph	640 by 350
10 10	80 by 25	8 by 14	16/256K	2	Graph	640 by 350
11 11	80 by 30	8 by 16	2/256K	1	Graph	640 by 480
12 12	80 by 30	8 by 16	16/256K	1	Graph	640 by 480
13 13	40 by 25	8 by 8	256/256K	1	Graph	320 by 200

Table 2.1 Standard VGA Modes

The following VESA SVGA modes are supported:

Mode (hex)	Pixels	Colors
100	640 by 400	256
101	640 by 480	256

Table 2.2 VESA SVGA Modes

ModeX is also supported.

3. Programming Model

This chapter describes the programming model for PERMEDIA. It describes the interface conceptually rather than detailing specific registers and their exact usage. In-depth descriptions of how to program PERMEDIA for specific drawing operations can be found in later chapters.

PERMEDIA is divided into the following memory regions:

Region	Address Space	Bytes	Description	Comments
Config	Configuration	256	PCI Configuration	PCI special
Zero	Memory	128K	Control Registers	relocatable
One	Memory	8M	Memory Region One	relocatable
Two	Memory	8M	Memory Region Two	relocatable
ROM	Memory	64K	Expansion ROM	relocatable
SVGA	Memory & I/O	-	SVGA Addresses	optional & fixed

Table 3.1 Memory Regions

Address Range	Description	Byte Swap
0000.0000 -> 0000.0FFF	Control & Status	No
0000.1000 -> 0000.1FFF	Memory Control	No
0000.2000 -> 0000.2FFF	GP FIFO access	No
0000.3000 -> 0000.3FFF	Video Control	No
0000.4000 -> 0000.4FFF	RAMDAC	No
0000.5000 -> 0000.57FF	Video Streams General Purpose Bus	No
0000.5800 -> 0000.5FFF	Video Streams Control	No
0000.6000 -> 0000.6FFF	SVGA Control	No
0000.7000 -> 0000.7FFF	Reserved	No
0000.8000 -> 0000.FFFF	GP Registers	No
0001.0000 -> 0001.0FFF	Control & Status	Yes
0001.1000 -> 0001.1FFF	Memory Control	Yes
0001.2000 -> 0001.2FFF	GP FIFO access	Yes
0001.3000 -> 0001.3FFF	Video Control	Yes
0001.4000 -> 0001.4FFF	RAMDAC	Yes
0001.5000 -> 0001.57FF	Video Streams General Purpose Bus	No
0001.5800 -> 0001.5FFF	Video Streams Control	No
0001.6000 -> 0001.6FFF	SVGA Control	Yes
0001.7000 -> 0001.7FFF	Reserved	Yes
0001.8000 -> 0001.FFFF	GP Registers	Yes

Table 3.2 Region 0 Address Map

3.1 PERMEDIA as a Register file

The simplest way to view the interface to the PERMEDIA Graphic Processor is as a flat block of memory-mapped registers (*i.e.* a register file). This register file appears as part of the address map for PERMEDIA.

When a PERMEDIA host software driver is initialized it can map 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 most straightforward way to load a value into a register is to write the data to its mapped address. In reality the chip interface comprises a 256 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 PERMEDIA to draw a primitive consists of writing values to the appropriate registers followed by a write to a command register. This last write triggers the start of drawing.

PERMEDIA has approximately 200 registers. All registers are 32 bits wide and should be 32-bit addressed. Many registers are split into bit fields, and it should be noted that 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. Software developers should ensure that only defined registers are written to and that undefined bits in registers are always written as zeros. The only exception to this rule is that in certain registers it is convenient to allow unmasked values to be written to registers which hold numeric data. These fields are marked as "not used" in Appendix A and elsewhere.

Register Types

PERMEDIA has three main types of register:

- Control Registers
- Command Registers
- Internal Registers

Control Registers are updated only by the host - the chip effectively uses them as read-only registers. Examples of control registers are the scissor clip min and max registers. Once initialized by the host, the chip only reads these registers to determine the scissor clip extents. Most registers are control registers.

Command Registers are those which, when written to, cause some action to occur. Typically, the host will initialize the appropriate control registers and then write to a command register to initiate drawing. Some command registers such as **ResetPickResult** or **Sync** do not initiate rendering. Apart from these, there are two types of command registers: begin-draw and continue-draw. Begin-draw

commands cause rendering to start with those 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. Making use of continue-draw commands can significantly reduce the amount of data that has to be loaded into PERMEDIA when drawing multiple connected objects such as polylines. Examples of command registers include the **Render** and **ContinueNewLine** registers.

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

Internal Registers are not accessible to 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 and before rendering starts, the internal registers are updated with the values in the corresponding control registers. 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, if a line is being drawn then 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.

For the most part internal registers can be ignored. It is helpful to appreciate that they exist in order to understand the continue-draw commands.

Efficiency Issues and Register Types

Software developers wishing to write device drivers for PERMEDIA should become familiar with the different types of registers. Some control registers such as the **StartXDom** and **StartY** registers have to be updated for almost every primitive whereas other control registers such as those for scissor clip or logical ops 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 tables in Appendix D lists the graphics registers according to their type, name and address.

3.2 PERMEDIA I/O Interface

There are four ways of loading PERMEDIA registers:

- The host writes a value to the mapped address of the register
- The host writes address-tag/data pairs to the FIFO.
- The host writes address-tag/data pairs to the FIFO via DMA.
- The host writes to raw memory mapped GP FIFO addresses.

In cases where the host writes data values directly to the chip via the register file, consideration has to be given to FIFO overflow (unless PCI Disconnect is enabled). The **InFIFOSpace** register indicates how many free entries remain in the FIFO. Before writing to any register, the host must ensure that there is enough space left in the FIFO. The values in this register can be read at any time. When using DMA, the DMA controller will automatically ensure that there is room in the FIFO before it performs further transfers. Thus a buffer of any size up to 64K, 32 bit words, can be passed to the DMA controller. The FIFO and DMA controller are described in more detail below.

3.2.1 PCI Disconnect

The PCI bus protocol incorporates a feature known as PCI Disconnect, which is supported by PERMEDIA. PCI Disconnect is enabled by writing a one to bit zero of the *DisconnectControl* register which is at offset 0x68 in PCI Region 0. Once the PERMEDIA is in this mode, if the host processor attempts to write to the full FIFO then instead of the write being lost, the PERMEDIA chip will assert PCI Disconnect which will cause the host processor to keep retrying the write cycle until it succeeds.

This feature allows faster download of data to PERMEDIA, since the host need not poll the *InFIFOSpace* register but should be used with care since whenever the PCI Disconnect is asserted the bus is effectively hogged by the host processor until such time as the PERMEDIA frees up an entry in its FIFO. In general this mode should only be used either for operations where it is known that the PERMEDIA can consume data faster than the host can generate it, or where there are no time critical peripherals sharing the PCI bus.

3.2.2 Idle bit

In some systems, PCI Disconnect may cause interrupts to be lost if it used too often or for too long. It is normal to only rely on this feature when it is known that the data to be sent to PERMEDIA will be absorbed quickly enough that the disconnect will seldom be used. It also advisable to check that the Graphics Processor is not processing a large primitive before transferring data of this sort, and this may be done by checking the Graphics Processor Active bit in the PCI Disconnect register. Disconnect should not normally be enabled if this bit is set.

3.2.3 FIFO ControlFIFO Control

The description in section §3.1 above considered the PERMEDIA interface to be a register file. More precisely, when a data value is written to a register, this value and the address tag for that register are combined and put into the FIFO as a new entry. The actual register is not updated until PERMEDIA processes this entry. In the case where PERMEDIA is busy performing a time consuming operation (*e.g.* drawing a large texture mapped 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 FIFO and that write is effectively lost.

The input FIFO is 256 entries deep and each entry consists of a tag/data pair; an address word which addresses the register to be updated, followed by the data to be sent to the register. The **InFIFOSpace** register can be read to determine how many entries are free. The value returned by this register will never be greater than 256.

An example of loading PERMEDIA 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 5 words (including the **Render** command).

```
dXDom(0x0); // common set-up
dXSub(0x0);
dY(1);

for (i = 0; i < nrects; ++i) {
      while (*InFIFOSpace < 5)
            ;     // wait for room

      StartXDom (rect->x1);
      StartXSub (rect->x2);
      Count (rect->y2 - rect->y1);
      YStart(rect->y1);
      Render (PERMEDIA_TRAPEZOID_PRIMITIVE);
}
```

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 in the FIFO. Before writing to the input FIFO, the user must check that there is sufficient space by reading the **InFIFOSpace** register.

The Graphics Core FIFO interface provides a port through which both GC register addresses and data can be sent to the input FIFO. A range of 4 Kbytes of host space is provided although all data may be sent through one address in the range. ALL accesses go directly to the FIFO; the range is provided to allow for data transfer schemes which force the use of incrementing addresses.

Note that the GC registers cannot be read through this interface. Command buffers generated to be sent to the input FIFO interface, may be read directly by PERMEDIA by using the DMA controller.

A data formatting scheme is provided to allow for multiple data words to be sent with one address word where adjacent or grouped registers are being written, or where one register is to be written many times.

Note. The FIFO interface can be accessed at 32 bit boundaries. This is to allow a direct copy from a DMA format buffer.

3.2.4 The DMA Interface

Loading registers directly via the FIFO is often an inefficient way to download data to PERMEDIA. Given that the FIFO can accommodate only a small number of entries, PERMEDIA has to be frequently interrogated to determine how much space is left. Also, consider the situation where a given API function requires a large amount of data to be sent to PERMEDIA . If the FIFO is written directly then a return from this function is not possible until almost all the data has been consumed by PERMEDIA. This may take some time depending on the types of primitives being drawn.

To avoid these problems PERMEDIA 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. In its simplest form the host software has to prepare a host buffer containing register address tag descriptions and data values. It 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. Meanwhile, in parallel, PERMEDIA is reading data from the host buffer and loading it into its FIFO. FIFO overflow never occurs since 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. It is valid to load the **DMAAddress** register while the previous DMA is in progress since the address is latched internally at the start of the DMA transfer. Many display driver functions can be implemented using the following skeleton structure:

```
do any pre-work
DMAAddress(address of dma_buffer);
while (TRUE) {
   count = *DMACount;  // note this is volatile
   if (count) {
       while (--count)
           ;    // wait for count to expire
   }
   else
       break;    // DMA completed
}
copy render data into DMA buffer
DMACount(number of words in DMA buffer)
return
```

Using DMA leaves the host free to return to the application, while in parallel, PERMEDIA is performing the DMA and drawing. This can increase performance significantly over loading a FIFO directly. In addition, some algorithms require that data be loaded multiple times (e.g. drawing the same object across multiple clipping rectangles). Since the PERMEDIA 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.

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 PERMEDIA 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 (TRUE) {
   count = *DMACount;  // note this is volatile
   if (count) {
       while (--count)
           ;    // wait for count to expire
   }
   else
       break;    // DMA completed
}
DMACount(number of words in new buffer)
mark the old buffer as free
return
```

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.

DMA Tag Description Format

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

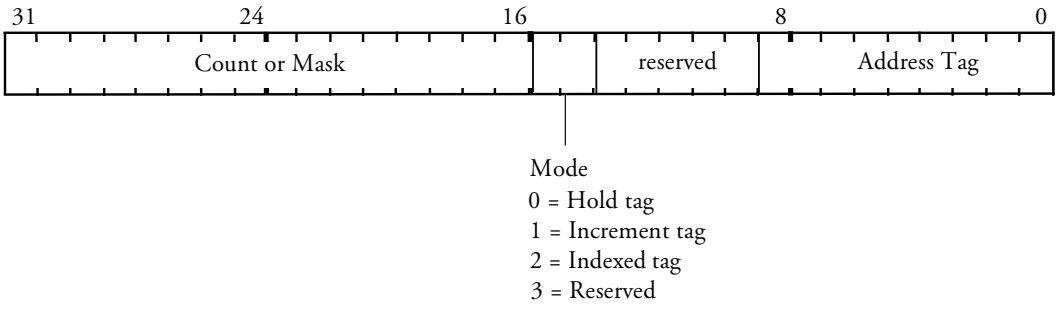


Figure 3.1 DMA Tag Description Format

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. Each of these is described in the following sections. Each row in the following diagrams represents a 32-bit value in the DMA buffer. The address tag for each register is given in the Graphics Register Reference Appendix D.

Hold Format

```
address-tag with Count=n-1, Mode=0
value 1
...
value n
```

This is commonly used for image download by setting the SyncOnHostData bit in the **Render** command.. 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. For example, this is useful for image download where pixel data is continuously written to the **Color** register. The bottom 9 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 that 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
StartXSub12 << 16
Count
1
Render
(trapezoid render command)
```

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; PERMEDIA updates an internal copy). Thus, this mode allows contiguous PERMEDIA registers to be loaded by specifying a single 32-bit tag value followed by a data word for each register. The low-order 9 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 PERMEDIA register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 8 **AreaStipplePattern** registers can be loaded as follows:

```
AreaStipplePattern0, Count=7, Mode=1
row 0 bits
row 1 bits
...
row 7 bits
```

Indexed Format

PERMEDIA address tags are 9 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 5 bits give the major group number. Appendix D Register Table, lists the individual registers with their Major Group and Offset.

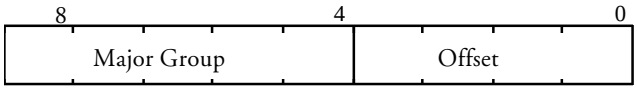


Figure 3.2 Indexed Format

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 16 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. These are updated with value 1, value 2 and value 3 respectively.

DMA Example

The following pseudo-code shows the previous example of drawing a series of rectangles but this time using the DMA controller. This example uses a single DMA buffer and the simplest 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++ = PERMEDIATagdXDom;
*pbuf++ = 0;
*pbuf++ = PERMEDIATagdXSub;
*pbuf++ = 0;
*pbuf++ = PERMEDIATagdY;
*pbuf++ = 1 << 16;
for (i = 0; i < nrects; ++i) {
    *pbuf++ = PERMEDIATagStartXDom;
    *pbuf++ = rect->x1 << 16;  // Start dominant edge
    *pbuf++ = PERMEDIATagStartXSub
    *pbuf++ = rect->x2 << 16;  // Start of subordinate edge
    *pbuf++ = PERMEDIATagCount;
    *pbuf++ = rect->y2 - rect->y1;
    *pbuf++ = PERMEDIATagYStart;
    *pbuf++ = rect->y1 << 16;
    *pbuf++ = PERMEDIATagRender;
    *pbuf++ = PERMEDIA_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”. It is worth noting that significantly less data would be required if indexed tags were used in this example.

DMA Buffer Addresses

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

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.

DMA Interrupts

PERMEDIA provides interrupt support, as an alternative means of determining when a DMA transfer is complete. This can provide considerable speed advantage. 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.

To enable the DMA interrupt, the DMAInterruptEnable bit must be set in the **IntEnable** register. The interrupt handler should check the DMAFlag bit in the **IntFlags** register to determine that a DMA interrupt has actually occurred. To clear the interrupt a word should be written to the **IntFlags** register with the DMAFlag bit set to one.

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 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.

3.2.5 Output FIFO and Graphics Processor FIFO Interface

To read data back from PERMEDIA an output FIFO is provided. Each entry in this FIFO is 32-bits wide and it can hold tag or data values. Thus its format is unlike the input FIFO whose entries are always tag/data pairs (we can think of each entry in the input FIFO as being 41 bits wide – 9 bits for the tag and 32 bits for the data). The type of data written by PERMEDIA to the output FIFO is controlled by the **FilterMode** register. 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 data for the **FilterMode** register consists of 2 bits per category. If the least significant of these two bits is set (0x1) then output of the register tag for that category is enabled; if the most significant bit is set (0x2) then output of the data for that category is enabled. Both tag and data output can be enabled at the same time. In this case the tag is written first to the FIFO followed by the data. The **FilterMode** register is described in more detail in section §5.15.

For example, to perform an image upload from the framebuffer, the **FilterMode** register should have data output enabled for the Color category. Then, the rectangular area to be uploaded should be described to the Rasterizer. Each pixel that is read from the framebuffer will then be placed into the output FIFO. If the output FIFO becomes full, then PERMEDIA will block 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 this many from the FIFO.

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

NB all expected data must be read back. PERMEDIA will block if the output 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 PERMEDIA is waiting for the host to read data from the output FIFO.

Graphics Processor FIFO Interface

PERMEDIA has a sequence of 1K x 32 bit addresses in the PCI Region 0 address map called the Graphics Processor FIFO Interface. To read from the output FIFO any address in this 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. Again, the first address is normally chosen. 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.

Writing to the input FIFO by this method is different from writing to the memory mapped register file. Since the register file has a unique address for each register, writing to this unique address allows PERMEDIA to determine the register for which the write is intended. This allows a tag/data pair to be constructed and inserted

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 the data that follows is identical to that described above for DMA buffers. Instead of using the PERMEDIA DMA it is possible to transfer data to PERMEDIA 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 PERMEDIA constructs tag/data pairs to enter as real FIFO entries. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

Note, that when writing to the raw FIFO address the FIFO full condition must still be checked by reading the *InFIFOSpace* register. However, writing tag descriptions does not cause any entries to be entered into the FIFO – such a write simply establishes a set of tags to be paired with the subsequent data. Thus, free space need be ensured only for actual data items that are written (not the tag values). For example, in the simplest case where each tag is followed by a single data item, assuming that the FIFO is empty, then 32 writes are possible before checking again for free space.

See the *PERMEDIA 2 Hardware Reference Manual* for more details of the Graphics Processor FIFO Interface address range.

3.3 Interrupts

All interrupts can be individually enabled and disabled. Refer to the *PERMEDIA 2 Hardware Reference Manual* for more details.

3.4 Synchronization

There are two main cases where the host must synchronize with PERMEDIA:

- before reading back from PERMEDIA registers
- before directly accessing the memory via the bypass mechanism

Also the host must synchronize with PERMEDIA for framebuffer management tasks such as double buffering, though this may be better handled using the **SuspendUntilFrameBlank** command. Synchronizing with PERMEDIA 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:

```
PERMEDIAData    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 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 PERMEDIA 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 PERMEDIA will generally empty the FIFO more quickly than it takes to set-up and handle the interrupt.

3.5 Host Memory Bypass

Normally, the host will access memory indirectly via commands sent to the PERMEDIA FIFO interface. However, PERMEDIA does provide the whole memory as part of its address space so that it can be memory mapped by an application. Access to the memory via this route is independent of the PERMEDIA FIFO.

Drivers may choose to use direct access to memory for algorithms which are not supported by PERMEDIA or for better performance in some specific cases. This may be so, for example, when multiple pixels can be written simultaneously and there is minimal host software overhead.

A driver making use of the bypass mechanism should synchronize memory accesses made through the FIFO with those made directly through the memory map. If data is written to the FIFO and then an access is made to the memory, it is possible that the memory access will occur before the commands in the FIFO have been fully processed. This lack of temporal ordering is generally undesirable.

There are two windows through which the memory can be accessed. Each window can have its own data formatting control that allows for different forms of byte swapping and data packing. If the framebuffer is set to use the 5:5:5:1Front and 5:5:5:1Back color modes, two pixels are packed into each 32 bit word, but each pixel belongs to a different buffer. Adjacent pixels in the same buffer are separated by 16 bits. As some software has difficulty with pixels that are not packed together, the memory windows can be configured to remap the data so that only the front or back buffer is visible, and it appears packed.

3.6 DMA Controller

A DMA controller is provided to allow transfer of data from the PCI bus to PERMEDIA memory. This controller is independent of the DMA controller which feeds the Graphics Processor FIFO, and has support for rectangular data structures and data formatting.

3.7 Register Read back

Under some operating environments, multiple tasks will want access to the PERMEDIA chip. Sometimes a server task or driver will want to arbitrate access to PERMEDIA on behalf of multiple applications. In these circumstances, the state of the PERMEDIA chip may need to be saved and restored on each context switch. To facilitate this, the PERMEDIA registers can be read back. For details of which registers are readable, see Appendix D Register Tables. Internal and command registers cannot be read back.

To perform a context switch the host must first synchronize with PERMEDIA. This means sending a **Sync** command and waiting for the sync output data to appear in the output FIFO. After this the registers can be read back.

To read a PERMEDIA 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.

Note that since internal registers cannot be read back 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. This state will be destroyed by any rendering work done by a new task. To prevent this, continue-draw commands should be performed via DMA 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.

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

3.8 Byte Swapping

Internally PERMEDIA operates in little-endian mode. However, PERMEDIA 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, PERMEDIA provides facilities to handle byte swapping. See the *PERMEDIA 2 Hardware Reference Manual* for more details of byte-swapping via the PCI bus.

Additional support is provided within the graphics core of the chip to byte swap images and bitmasks as they are transferred to and from the host. These are documented in the relevant sections of chapter §5.

3.9 Red and Blue Swapping

For a given graphics board the RAMDAC and/or API will usually force a given interpretation for true color pixel values. For example, 32-bit pixels will be interpreted as either RGB (red at byte 2, green at byte 1 and blue at byte 0) or BGR (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.

PERMEDIA provides three registers to specify the byte positions of blue and red internally. In the Texture/Fog/Blend unit the **AlphaBlendMode** register contains a 1-bit field called ColorOrder. If 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 in the Alpha Blend unit, it must also be set in the Color Format unit and the Texture Read unit via the **DitherMode** and **TextureDataFormat** registers.

4. Memory I/O and Organization

This section describes the arrangement of data stored in memory. Although PERMEDIA has a single unified memory space for ease of reference, this is divided into three buffers: the localbuffer, framebuffer and texture buffer. Any of these buffers can be any size at any position in the memory.

For 3D operation, associated with the framebuffer there would normally be a localbuffer to hold depth and/or stencil information. A texture buffer may be present if needed. For 2D operation the localbuffer would not generally be used, but the texture buffer may be used to store pixmaps.

4.1 Patched Data

PERMEDIA supports an optional scheme for organizing memory, known as "patching". Data is normally stored linearly in memory such that incrementing addresses move from left to right along a scanline of the appropriate buffer. The type of memory supported by PERMEDIA uses a page structure which allows fast accesses within a 2 Kbyte region, but imposes a penalty for moving to a new 2 Kbyte region. This page structure favors access patterns that move along a scanline but is inefficient for moving vertically as the large change in address may cause a page break.

Patched data is organized so that there is less penalty for moving vertically in a buffer at the expense of a decrease in performance for moving horizontally. This is done by organizing memory such that a two dimensional region or patch in the buffer corresponds to a linear sequence in memory. A buffer will comprise lots of patches.

Two patch modes are supported which differ in the detail of how the data is organized within the patch. Normal patch mode is used for localbuffer and framebuffer data. Subpatch mode is used for texture and framebuffer data. Patched data cannot be displayed, so patching of framebuffer data is normally only done for off-screen bitmaps or when processing localbuffer or texture data through the framebuffer units.

4.2 Localbuffer

The localbuffer holds the Depth and Stencil information corresponding to each displayed pixel. The Depth field can be either 15 or 16 bits wide and the Stencil field either 1 or 0 bits wide. The total width of the localbuffer data cannot be greater than 16 bits. If a Stencil field is defined then it occupies bit 15; the depth field always starts at bit 0.

The format of the localbuffer is specified in two places: the **LBReadFormat** register and the **LBWriteFormat** register.

4.2.1 Localbuffer Coordinates

The translation from the internal coordinate system to the external address map involves setting the base address of the window (or screen if coordinates are screen relative) and positioning the origin in either the top left or bottom left corner. The origin is specified in the **LBReadMode** register.

The actual equations used to calculate the localbuffer address to read and write are:

Bottom left origin

Destination address = LBWindowBase - Y * W + X

Source address = LBWindowBase - Y * W + X + LBSourceOffset

Top left origin

Destination address = LBWindowBase + Y * W + X

Source address = LBWindowBase + Y * W + X + LBSourceOffset

where:

X	is the pixel's X coordinate.
Y	is the pixel's Y coordinate.
LBWindowBase	holds the base address in the localbuffer of the current window.
LBSourceOffset	is normally zero except during a copy operation where data is read from one address and written to another address. The offset between source and destination is held in the **LBSourceOffset** register.
W	is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **LBReadMode**register. See the table in Appendix C for more details.

This produces the localbuffer address in pixels. For PERMEDIA, the localbuffer data is always 16 bits so the physical byte address is two times the pixel address. The destination address is the address that data will be written to; data may also be read from this address if read-modify-write operations are needed such as depth testing. The source address is mainly used for copy operations and is only used for reading data.

4.3 Framebuffer

The framebuffer holds color data produced by PERMEDIA. The framebuffer may hold both displayed and non-displayed data. Color buffers can be placed anywhere in memory, there is no restriction on areas that can be displayed from.

There may be several buffers, such as the front and back buffers of a double buffered system, or the left and right buffers of a stereo system. No restrictions are placed on the number or organization of the buffers other than the total amount of memory fitted.

To access alternative buffers either the **FBPixelOffset** register can be loaded, or the base address of the window held in the **FBWindowBase** register can be redefined.

4.3.1 Framebuffer Coordinates

Coordinate generation for the framebuffer is similar to that for the localbuffer except for the addition of **FBPixelOffset**. The WindowOrigin bit in the **FBReadMode** register selects top left or bottom left as the origin for the framebuffer.

The actual equations used to calculate the framebuffer address to read and write are:

Bottom left origin

Destination address = FBWindowBase - Y * W + X + FBPixelOffset

Source address = FBWindowBase - Y * W + X + FBPixelOffset + FBSourceOffset

Top left origin

Destination address = FBWindowBase + Y * W + X + FBPixelOffset

Source address = FBWindowBase + Y * W + X + FBPixelOffset + FBSourceOffset

where:

X	is the pixel's X coordinate,
Y	is the pixel's Y coordinate,
FBWindowBase	holds the base address in the framebuffer of the current window.
FBPixelOffset	is normally zero except when multi-buffer writes are needed when it gives a way to access pixels in alternative buffers without changing the **FBWindowBase** register. This is useful as the window system may be asynchronously changing the window's position on the screen. It is held in the **FBPixelOffset** register.

FBSourceOffset	is normally zero except during a copy operation where data is read from one address and written to another address. The FBSourceOffset is held in the **FBSourceOffset** register.
W	is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **FBReadMode** register. See the table in Appendix C for more details.

These address calculations translate a 2D address into a linear address so non power of two framebuffer widths (e.g. 640) are economical in memory. The address is in pixels; this is translated to a physical byte address by multiplying by the number of bytes in the pixel.

The width is specified as the sum of selected partial products which are selected by the fields PP0, PP1 and PP2 in the **FBReadMode** register. This is the same mechanism as is used to set the width of the localbuffer, however the widths may be set independently. The range of widths supported are tabulated in Appendix C, together with the values for each of the PP fields. This table holds all the common screen widths.

For arbitrary screen sizes, for instance when rendering to 'off screen' memory such as bitmaps the next largest width from the table must be chosen. The difference between the table width and the bitmap width will be an unused strip of pixels down the right hand side of the bitmap.

Note that such bitmaps can be copied to the screen only as a series of scanlines rather than as a rectangular block, unless the Texture Read unit is used. In this case the stride for the read can be set differently to the write by means of the partial products However, windowing systems often store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used

4.3.2 Framebuffer Color Formats

The contents of the framebuffer can be regarded in two ways:

- As a collection of fields of up to 32 bits with no meaning or assumed format as far as PERMEDIA is concerned. Bit planes may be allocated to control cursor, color look up tables (LUTs), multi-buffer visibility or priority functions. In this case PERMEDIA will be used to set and clear bit planes quickly but not perform any color processing such as interpolation or dithering. All the color processing can be disabled so that raw reads and writes are done and the only operations are writemasking and logical ops. This allows the control planes to be updated and modified as necessary.

- As a collection of one or more color components. All the processing of color components, except for the final writemask and logical ops are done using the internal color format . The final stage before writemask and logical ops processing converts the internal color format to that required by the physical configuration of the framebuffer and video logic. The range of supported formats are given in table 4.1. The nomenclature *n@m* means this component is *n* bits wide and starts at bit position *m* in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode.

Some important points to note:

- The alpha channel, when present, is always associated with the RGB color channels rather than being a separate buffer. This allows it to be moved in parallel and to work correctly in multi-buffer updates and double buffering.
- For the Front and Back modes the data value is duplicated in both buffers. In general, if the data format does not take 32 bits the data is repeated in the empty bit planes. If the data format requires 8 bits, the same value is repeated in all four bytes of the word. The pixel size then determines how many of the bytes are written to memory. If a 16 bit format is chosen (e.g. 5:5:5:1) then the data is repeated in the upper and lower halves of the word. If the pixel size is set to 16 bits then only half the word is written to memory; if the pixel size is set to 32 bits then both halves are written, with the same data in each. A writemask can be used to select which bits are written. This is used for certain types of double buffering. The front and back modes are used in the alpha blend unit to extract the appropriate buffer.
- The offset modes (10 and 11) format the colors into a 7 bit value and then add 64 to the result. This avoids reserved entries in window system color tables.
- YUV formats are only available as textures. PERMEDIA can convert YUV textures to RGB and apply them to polygons; it cannot convert RGB to YUV for storage. If a YUV texture is being loaded into the chip it should be done as raw data or converted to RGB as it is loaded.
- The CI4 format is only available as a texture.
- When reading the framebuffer, RGBA components are scaled to their internal width if needed for alpha blending.
- The color format of the framebuffer is independent of the color format of the texture buffer; the texture buffer supports the same formats as the framebuffer plus some for YUV color formats

Color information is stored as values of red, green and blue (RGB) with or without alpha values. Alternatively, it can be stored as a color index value (CI) where each value references an entry in a color look up table that contains RGB values.

The color format information needs to be stored in three places: the **DitherMode** register[1], the **AlphaBlendMode** register[2] and the **TextureDataFormat** register.

				Internal Color Channels			
	Format	**Color Order**	**Name**	**R/Y**	**G/U**	**B/V**	**A**
BGR	0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
	1	BGR	5:5:5:1Front	5@0	5@5	5@10	1@15
	2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
	5	BGR	3:3:2Front	3@0	3@3	2@6	0
	6	BGR	3:3:2Back	3@8	3@11	2@14	0
	9	BGR	2:3:2:1Front	2@0	3@2	2@5	1@7
	10	BGR	2:3:2:1Back	2@8	3@10	2@13	1@15
	11	BGR	2:3:2FrontOff	2@0	3@2	2@5	0
	12	BGR	2:3:2BackOff	2@8	3@10	2@13	0
	13	BGR	5:5:5:1Back	5@16	5@21	5@26	1@31
	16	BGR	5:6:5Front	5@0	6@5	5@11	0
	17	BGR	5:6:5Back	5@16	6@21	5@27	0
YUV	18	BGR	YUV444	8@0	8@8	8@16	8@24
	19	BGR	YUV422	8@0	8@8	8@8	0
RGB	0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
	1	RGB	5:5:5:1Front	5@10	5@5	5@0	1@15
	2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
	5	RGB	3:3:2Front	3@5	3@2	2@0	0
	6	RGB	3:3:2Back	3@13	3@10	2@8	0
	9	RGB	2:3:2:1Front	2@5	3@2	2@0	1@7
	10	RGB	2:3:2:1Back	2@13	3@10	2@8	1@15
	11	RGB	2:3:2FrontOff	2@5	3@2	2@0	0
	12	RGB	2:3:2BackOff	2@13	3@10	2@8	0
	13	RGB	5:5:5:1Back	5@26	5@21	5@16	1@31
	16	RGB	5:6:5Front	5@11	6@5	5@0	0
	17	RGB	5:6:5Back	5@27	6@21	5@16	0
YUV	18	RGB	YUV444	8@16	8@8	8@0	8@24
	19	RGB	YUV422	8@8	8@8	8@0	0
CI	14	-	CI8	8@0	0	0	0
	15	-	CI4	4@0	0	0	0

Table 4.1 Supported Color Formats

[1] Note: the DitherMode register does not support the YUV444, YUV422 or CI4 formats.

[2] Note: the AlphaBlendMode register does not support the YUV444, YUV422 or CI4 formats.

4.3.3 Special Memory Modes

PERMEDIA uses SGRAM to store data. SGRAM devices usually have special features that are particularly useful for graphics.

Hardware Writemasks.

These allow writemasking in the framebuffer without incurring a performance penalty. If hardware writemasks are not available, PERMEDIA 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 should be set to all 1's, and the number of framebuffer reads is set to 0 (for normal rendering). This is achieved by clearing the ReadSource and ReadDestination enables in the **FBReadMode** register.

To use software writemasking (if hardware masks are not available), the required writemask is written to the **FBSoftwareWriteMask** register and the number of framebuffer reads is set to 1 (for normal rendering). This is achieved by setting the ReadDestination enable in the **FBReadMode** register.

Block Writes

Block writes cause consecutive pixels in the framebuffer to be written simultaneously. This is useful when filling large areas but does have some restrictions:

- No depth or stencil testing can be done
- All the pixels must be written with the same value so no color interpolation, alpha 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, but not software writemasking. The scissor tests and extent checking operate correctly with block writes, and bitmask patterns can be applied.

The **FBBlockColor** register holds the value to write to each pixel. Note that this register should not be updated immediately after a **Render** command which performs a block write.

Sending a **Render** command with the PrimitiveType field set to "trapezoid" and the FastFillEnable field set will then cause block filling of the area. Note that during a block fill any inappropriate state is ignored so even if stippling, color interpolation, depth testing and/or logical ops, for example, are enabled they have no effect. However, scissor clipping does function correctly with block writes.

PERMEDIA always writes 32 pixels per block fill. It takes care of any partial blocks at the beginning or end of spans.

4.4 Double Buffering

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

For further details see section §5.12.6, §5.12.7 and §5.13 of this manual, and refer to the *PERMEDIA 2 Hardware Reference Manual.*

4.4.1 BitBlt Double Buffering

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

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

The BLT can be performed using the texture units to allow arbitrary scaling and filtering of data.

4.4.2 Full Screen Double Buffering

This section describes how to implement full-screen double buffering with PERMEDIA when using the video timing generator. To perform full-screen double buffering, the available display RAM must be partitioned into two parts – buffer 0 and buffer 1 – each of which contains enough memory to display a full screen of pixel information. The partitioning consists of deciding the offset into RAM at which a given buffer starts. This offset is used to program various PERMEDIA registers. For a given resolution and pixel depth there must be enough RAM configured on the display adapter for this to be possible. For example, with 32 bit deep pixels and 4MB of RAM it is possible to implement full-screen double buffering at 800x600 resolution, but not at 1024x768.

There are two factors to consider for full-screen double buffering. Firstly, the video *output* hardware must be configured to display the pixels from the correct buffer. Secondly, the PERMEDIA chip must be programmed to render into the correct buffer. To achieve smooth animations, the buffer being rendered into is usually different from the buffer being displayed.

Video Output

To display a given buffer, the video output hardware must be programmed with the offset of that buffer in RAM. In the PERMEDIA internal timing generator this is

controlled by the *ScreenBase* register located in the PERMEDIA control space at offset 0x3000.

PERMEDIA Rendering

When determining the memory location of a pixel being rendered, PERMEDIA operates in screen coordinates.

To simplify the calculation of pixel coordinates that are loaded into PERMEDIA, this value may be loaded into the **FBPixelOffset** register. The last thing PERMEDIA does before passing a pixel address to the framebuffer interface is to add the value in the **FBPixelOffset** register to its address. Thus it is possible to move the rendering origin to any pixel location in memory. When swapping buffers it is normal to move this position to be the pixel at which a given buffer starts.

These values can be pre-calculated at system start-up ready to be loaded as required.

Synchronization

Double buffering allows the displaying of one buffer (the front buffer) whilst rendering into the other (the back buffer). When the rendering has been completed to the back buffer, the buffers are swapped and rendering continues into the new back buffer. As a general rule, buffers should not be swapped until all rendering to the back buffer has completed so that the buffer swap does not result in visible tearing, or screen break-up.

PERMEDIA reads the *ScreenBase* register at the end of each vertical blanking period to determine the starting pixel for the next frame to be displayed. Thus, in principle, this register can be written at any time to swap buffers and will only take effect on the next frame. The same is not true of loading the **FBPixelOffset** register. This register gets updated as soon as the command to load it works its way through the input FIFO. Hence, any rendering that takes place after the **FBPixelOffset** has been loaded will occur in the new buffer. If care is not taken, this can result in rendering being seen before the buffers have been swapped. The following scheme would probably produce picture break-up:

```
ScreenBase = Buf0_Addr              // display buffer 0
FBPixelOffset = Buf1_Offset         // draw to buffer 1 now
Render Commands                     // draw next frame
ScreenBase = Buf1_Addr              // display buffer 1
FBPixelOffset = 0                         // draw to buffer 0 now
Render Commands                     // draw next frame
```

There are two problems here. Firstly, even though the write to the *ScreenBase* register happens immediately, PERMEDIA does not actually swap the buffers till the end of the next vertical blanking period. Thus the start of rendering of the next frame may be seen in the front buffer prior to the buffer swap. Secondly, once a command has been loaded into the input FIFO the host is free to continue with other work, while PERMEDIA executes the command. Accesses to the *ScreenBase*

register bypass the FIFO so it is possible for the host to update it, and for the buffer swap to happen, before PERMEDIA has completed rendering the last frame.

The PERMEDIA includes the **SuspendUntilFrameBlank** command to solve these problems without the need for the host synchronizing with PERMEDIA. Here is the preferred version of the above example:

```
SuspendUntilFrameBlank(parameters)  // display buffer 0
FBPixelOffset = Buf1_Offset         // draw to buffer 1 now
Render Commands                     // draw next frame
SuspendUntilFrameBlank(parameters)  // display buffer 1
FBPixelOffset = 0                         // draw to buffer 0 now
Render Commands                     // draw next frame
```

The **SuspendUntilFrameBlank** command will flush all outstanding reads and writes to the framebuffer, and will prevent any further framebuffer memory accesses until after the buffers have been swapped.

The data that is loaded into the **SuspendUntilFrameBlank** command enables PERMEDIA to swap the buffers automatically when the VBLANK occurs by loading a new buffer offset into the *ScreenBase* register as discussed above. For full details, see the detailed description in the register reference, Appendix A.

Thus a single command register access ensures that:

- all rendering has completed to the back buffer
- the chip will wait for VBLANK before carrying out the swap
- the host can continue sending rendering commands to PERMEDIA without risk of them affecting the displayed buffer.

As a general performance note, it is best to send non-framebuffer related commands to PERMEDIA following the **SuspendUntilFrameBlank** command. This allows better overlap between the host and PERMEDIA. In general any commands that will not cause rendering to the framebuffer to occur can be queued in the PERMEDIA FIFO before waiting on VBLANK.

Eventually more framebuffer rendering commands will be sent by the host, and the PERMEDIA will then stall its hyperpipeline until the buffer swap completes. Ideally the host should use this time to perform non-rendering operations e.g. prepare additional DMA buffers

Using this scheme the host will not normally ever need to wait for VBLANK, unless it is making framebuffer memory accesses through the bypass.

To wait for VBLANK, the *LineCount* register can be polled. There is also a VBLANK interrupt available (see *PERMEDIA 2 Hardware Reference Manual* for details). The *LineCount* register is reset at the start of the VBLANK period and is incremented by one for each scanline as the video scanner moves down the screen. Thus polling for this register to have a value of less than the value held in the *VbEnd* register indicates that PERMEDIA is in the VBLANK period.

4.4.3 Bitplane Double Buffering

Bitplane double buffering is of use at 32 bits per pixel framebuffer depth using 32768 colors in 5:5:5:1 true color mode. It relies on the RAMDAC selecting between the high and low 16 bits of its input stream based on whether bit 31 is set or clear. Effectively the front and back buffer for each pixel, become interleaved within the same 32 bit word in the framebuffer, i.e. buffer 0 becomes the lower 16 bits and buffer 1 becomes the upper 16 bits.

The buffer swap is thus implemented as a block fill of bit 31 of the interior of a window with either one or zero. While this is not as quick as full screen double buffering which just requires a single register *ScreenBase* to be updated, it is many times quicker than BitBlt double buffering, and like the BitBlt case allows any number of windows to be hardware double buffered simultaneously..

Note that when rendering GUI data (such as window borders, titles etc.) bit 31 must always be set to the same value so that these pixels are always displayed from the same buffer. The hardware writemask can then be used to write to only the high, or only the low, 16 bits when rendering the animating contents of a window.

The features are:

- "almost instantaneous" buffer swap
- no offscreen buffer required (e.g. 1152x900 would be the maximum resolution on a 4MB framebuffer at 32bpp depth)
- Multiple windows can be double buffered. GUI can write with no performance penalty.
- Only useful at 5:5:5:1 RGB color depth.
- No triple buffering or other advanced buffer operations

In order to allow the Microsoft Windows 95 DIB engine to render direct to the framebuffer in the 5:5:5:1 format, a special framebuffer bypass option is supported which presents the front and back buffers uninterleaved, i.e. as a 5:5:5:1 16bpp packed framebuffer. This allows rarely used complex primitives to be rendered by software.

4.4.4 Panning

Display panning can be achieved by setting the ScreenBase and ScreenStride registers appropriately. The ScreenBase register defines where in the framebuffer the image is to start. For panning to work, the image in the framebuffer must be larger than that to be displayed. The ScreenStride holds this difference in terms of 64 bit units per scanline. For example, with a screen width of 640 pixels and a framebuffer image width of 660, 32 bit pixels, the ScreenStride needs to be set to 10.

4.5 Texture Buffer

The texture buffer is very similar to the framebuffer. Textures are stored in the formats the framebuffer supports, and loaded into memory through the Framebuffer Write unit. If the texture format is different to the framebuffer format, the **DitherMode** register should be temporarily set to the texture format during texture loads. Textures are read through the Texture Read unit.

If the texture is already in the correct format then a fast texture load can be used. This is done by writing raw texture data to the **TextureData** register. Raw data is 32 bits wide, with the correct bit pattern to be stored in memory. No data formatting or packing is done, so the texture must be pre-processed if this is required. The texture is stored linearly in memory from the address specified in **TextureDownLoadOffset** which is automatically incremented; no patching is done, so if the texture is to be patched it must be done by the host. This method avoids setting up the Rasterizer and changing the state of the pipeline.

4.5.1 Texture Load Through Bypass

Alternatively, a texture map may be loaded through the bypass, either directly by the CPU or by the DMA controller. This mechanism supports patching of data, but not general data formatting. The only data formatting supported is conversion of YUV420 to YUV422. Refer to the *PERMEDIA 2 Hardware Reference Manual* for more details.

4.5.2 Texture Buffer Co-ordinates

Texture co-ordinates are formed by the Texture Address unit and passed to the Texture Read unit. In place of the Rasterize X and Y coordinate system, the Texture Address unit generates S and T values.

The actual equations used to calculate the texture buffer address are:

Bottom left origin

Texture address = TextureBaseAddress - T * W + S

Top left origin

Texture address = TextureBaseAddress + T * W + S

where:

S	is the texel's S coordinate,
T	is the texel's T coordinate,
TextureBaseAddress	holds the base address in the framebuffer of the current window.
W	is the texture map width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **TextureReadMode** register. See the table in Appendix C for more details.

These address calculations translate a 2D address into a linear address so non power of two texture widths (e.g. 640) are economical in memory. Note that the width of the texture map used for these calculations is independent of the width and height used for texture effects such as repeat or clamp. The address is in texels; the physical byte address is calculated by multiplying the texel address by the number of bytes in the texel.

4.5.3 Texture Color Formats

Texture maps have the same choice of formats as the framebuffer plus YUV and 4 bit Color Index formats (see section §4.3.2 for details). The formats of the texture map and framebuffer do not have to be the same.

5. Graphics Programming

PERMEDIA provides a rich variety of operations for 2D and 3D graphics supported by its Hyperpipelined architecture. Section §5.1 shows the units in the HyperPipeline. Sections §5.2 to §5.15 describe each unit.

5.1 The Graphics HyperPipeline

The Graphics Hyperpipeline, or Graphics Processor, supports:

- Point, Line, Triangle Rectangle and Bitmap primitives.
- Flat and Gouraud shading
- Texture Mapping, Fog and Alpha blending
- Scissor and Stipple
- Stencil test, Depth (Z) buffer test
- Dithering
- Logical Operations

The units in the HyperPipeline are:

- **Delta Unit** calculates parameters.
- **Rasterizer** scan converts the primitive into a series of fragments.
- **Scissor/Stipple** tests fragments against a scissor rectangle and a stipple pattern.
- **Localbuffer Read** loads localbuffer data for use in the Stencil/Depth unit.
- **Stencil/Depth** performs stencil and depth tests.
- **Texture Address** generates addresses of texels for use in the Texture Read unit.
- **Texture Read** accesses texture values for use in the texture application unit.
- **YUV** converts YUV to RGB and applies chroma test.
- **Localbuffer Write** stores localbuffer data to memory.
- **Framebuffer Read** loads data from the framebuffer.
- **Color DDA** generates color information.
- **Texture/Fog/Blend** modifies color.
- **Color Format** converts the color to the external format.
- **Logic Ops** performs logical operations.
- **Framebuffer Write** stores the color to memory.
- **Host Out** returns data to the host.

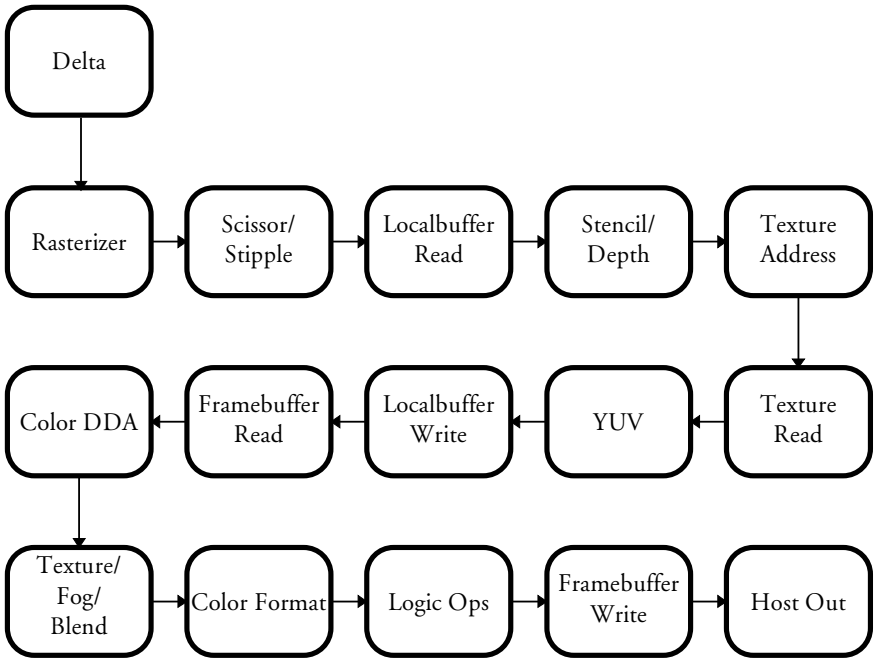


Figure 5.1 Hyperpipeline

The order of the Hyperpipeline shows the order in which operations are performed. The Scissor/Stipple unit is before the texture address generator, so any fragments that fail a stipple test will not cause a texture access. This makes best use of the processing capacity of the pipeline. An awareness of the pipeline is important when programming PERMEDIA; all units in the pipeline can be thought of as independent. For example, enabling the XOR logic op will not automatically enable reading from the framebuffer; this must be done explicitly.

5.2 Delta Unit

For best performance, the Delta unit in PERMEDIA should be used to calculate the edge deltas used by the Graphics Processor.

The Delta Unit accepts the following vertex parameters:

Offset	Category	Parameter	Fixed Point Format	IEEE Single Precision Floating Point Range
0	Texture	s	2.30 s [footnote 1]	-1.0…1.0 [footnote 2]
1		t	2.30 s	-1.0…1.0
2		q	2.30 s	-1.0…1.0
3		Ks	2.22 us	0.0…2.0
4		Kd	2.22 us	0.0…1.0
5	Color	red	1.30 us	0.0…1.0
6		green	1.30 us	0.0…1.0
7		blue	1.30 us	0.0…1.0
8		alpha	1.30 us	0.0…1.0
9	Fog	f	10.22 s	-512.0…512.0
10	Coordinate	x	16.16 s	-32K…+32K [footnotes 3,4]
11		y	16.16 s	-32K…+32K
12		z	1.30us	0.0…1.0
14	PackedColor	PackedColor	8888	8888

Table 5.1 Vertex Parameters

While values may be written to the vertex store in either floating or fixed point formats, any values returned via the readback mechanism will be the clamped floating point (IEEE single precision) version of the value written. The returned value of a parameter may be different from the value written if any of the following conditions has occurred:

- Any clamping has occurred;
- The input number was a NaN or Denormalized IEEE number;
- The input value has exceeded the internal range (approximately $\pm 2^{32}$).

No parameters are corrupted by the calculations so parameter sharing between primitives is simply achieved by not re-loading those parameters. For example if

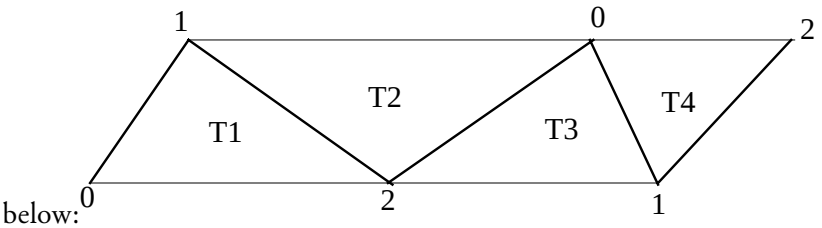
[1]This is the range when Normalise is not used. When Normalise is enabled the fixed point format can be anything, providing it is the same for the s, t and q parameters. The numbers will be interpreted as if they had 2.30 format for the purpose of conversion to floating point. If the fixed point format (2.30) is different from what the user had in mind then the input values are just pre-scaled by a fixed amount (i.e. the difference in binary point positions) prior to conversion.

[2]This is the range when Normalise is not used. When Normalise is enabled the range is extended to $2^{\pm 32}$ approximately. This also applies to the t and q values as well.

[3]The normal range here is limited by the size of the screen.

[4]K = 1024.

the first triangle in a triangle strip is loaded into V0, V1 and V2, then the next triangle will load V0, the next V1, etc.. This is shown below.



below:

Figure 5.2 Triangle Mesh.

The vertices are automatically sorted so any vertex can be associated with any vertex store.

Similarly a triangle fan may be implemented initially loading V0, V1 and V2 and then cycling through loading V1 and V2 as shown below (note that T1 and T5 share a vertex which is loaded first in V1 and then in V2):

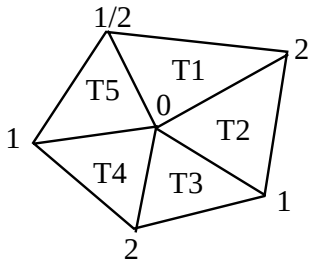


Figure 5.3 Triangle Fan.

Individual triangles, strips, or fans may be backface culled such that triangles that face away from the viewer are not drawn. Detection of backfacing triangles is done by the sign of the area of the triangle, but whether positive or negative areas should be rejected depends on the definition of the triangle format (whether the vertices are considered to go clockwise or counter-clockwise). It may also vary when meshed primitives are drawn, such a strip where the sign of the area alternates triangle by triangle. When backface culling is enabled in the Delta Unit, the sign to reject may be set for each triangle as it is drawn.

Lines are handled slightly differently in that only V0 and V1 are used. The direction of the line is defined as part of the command. Hence a line may run either from V0 to V1 or from V1 to V0. A polyline may be drawn by loading the first vertex into V0, the second vertex into V1, the third vertex into V0, the fourth vertex into V1 etc..

The texture parameters (S, T and Q) are handled differently to the other parameters as their range must be constrained to get the best results from the finite precision DDA and perspective division hardware available in the Graphics Processor. Any operation on the texture parameters before they are used is

controlled by the TextureParameterMode in the DeltaMode register. The options are NoClamp, Clamp or Normalize. The NoClamp and Clamp options work the same as for the other parameters. The Normalize option finds the maximum absolute value of the texture S, T and Q values for the primitive and normalizes all the value to lie in the range -1.0 … 1.0 inclusive prior to being used in the set-up calculations. Note that the texture values in the vertex store are *not* changed by the Normalize option to allow normalization to work on a triangle by triangle basis across a triangle mesh.

5.2.1 Drawing Commands

The Delta Unit responds to five drawing commands: DrawTriangle, RepeatTriangle, DrawLine01, DrawLine10 and RepeatLine. When using Delta, these drawing commands replace the Render command, and have the same data field.

The Draw and Repeat commands cause Delta to calculate the required data for the rendering devices and update the Start, dX and dyDom registers in the Rasterizer, Color, Depth, Texture and Fog Units of the Graphics Processor. Any additional registers in the Rasterizer Unit are also loaded (N.B. the RasterizerMode register is not updated). Finally the Render and ContinueNewSub commands are sent to the rendering devices.

The data field accompany the DrawTriangle or DrawLine command is used to control some aspects of the Delta's operation in conjunction with the DeltaMode register. The relevant bits in the Draw command, and their effect in the Delta Unit are described in Table 5.2. Note that the values in the remaining bits must be compatible with the desired operation.

Bit No.	Name	Description
13	TextureEnable	When set (and qualified by the TextureEnable bit in the DeltaMode register) causes the texture values (S, T and Q) to be calculated.
14	FogEnable	When set (and qualified by the FogEnable bit in the DeltaMode register) causes the fog values to be calculated.
16	SubPixelCorrectionEnable	When set (and qualified by the SubPixelCorrectionEnable bit in the DeltaMode register) enables the sub pixel correction of any value interpolated in the Y direction. The rendering devices will perform the sub pixel corrections in the X direction.
20	RejectNegativeFace	Qualified by the BackFaceCull field in the DeltaMode register. If set rejects triangles with a negative area. If clear, rejects triangles with a positive area.

Table 5.2 Draw Command Bit Field Assignments Affecting Delta

5.2.2 DrawLine Commands

The command DrawLine01 causes Delta to draw a line from vertex 0 - V0 to vertex 1 - V1. Conversely DrawLine10 causes Delta to draw a line from V1 to V0. These two commands allow polylines to be drawn by updating V0 and V1 alternately. The alternate use of DrawLine01 and DrawLine10 allows the line stipple pattern to continue correctly across segments in a polyline.

Note, that due to the DDA algorithm, drawing direction may affect the rendered pixels. Hence, with the same data in V0 and V1, the two DrawLine commands may render different pixels. This may be important for operations such as XOR lines or patterned lines.

5.2.3 Repeat Commands

The RepeatTriangle and RepeatLine commands allow the previously set-up triangle or line to be repeated again. This is useful when some rendering state has changed and the primitive must be redrawn. An example of this is when the scissor region is updated and the primitive redrawn to implement window clipping.

A RepeatTriangle command should only follow a DrawTriangle command and not a DrawLine command. Mixing the incorrect Repeat and Draw commands will cause undefined visual effects.

5.2.4 DeltaMode Register

The DeltaMode register is used to hold 'long term' state information. The per primitive control information is taken from the Draw command as already outlined. The following table lists the DeltaMode register bit field assignments and describes their function.

Bit No.	Name	Description
0, 1	Reserved	
2, 3	DepthFormat	The following options apply: 0 15 bit depth 1 16 bit depth 2 Reserved 3 Reserved
4	FogEnable	When set enables the fog calculations. This field is qualified by the FogEnable bit in the Draw command.
5	TextureEnable	When set enables the texture calculations. This field is qualified by the TextureEnable bit in the Draw command.
6	SmoothShadingEnable	When set enables the color calculations.
7	DepthEnable	When set enables the depth calculations.
8	SpecularTextureEnable	When set enables the specular texture calculations.
9	DiffuseTextureEnable	When set enables the diffuse texture calculations.
10	SubPixelCorrectionEnable	When set provides the subpixel correction in Y. This is qualified by the SubPixelCorrectionEnable in the Draw command.

11	DiamondExit	When set enables the application of the OpenGL 'Diamond-exit' rule to modify the start and end coordinates of lines.
12	NoDraw	When set prevents a Render command from being sent to the rendering devices. This field only affects the Draw commands. This field allows the host to alter the set-up parameters before sending a Render command.
13	ClampEnable	When set causes the input values to be clamped to a parameter specific range. Note that the texture parameters are not affected by this field.
14, 15	TextureParameterMode	These field causes the texture parameters to be: 0: Used as given 1: Clamped to lie in the range -1.0 to 1.0 2: Normalize to lie in the range -1.0 to 1.0
16	Reserved	
17	BackFaceCull	When set enables backface culling of triangles. Rejection is based on the sign of the area of the triangle, whether +ve or -ve is controlled by the draw command.
18	ColorOrder	Specifies order of colors in V*PackedColor messages. Bit 31 Bit 0 0 = Alpha, Blue, Green, Red 1 = Alpha, Red, Green, Blue Each color component is 8 bits.

Table 5.3 DeltaMode Register Bit Field Assignments.

Any unused bits in the DeltaMode register should be set to zero.

Note that any Repeat commands will use the DeltaMode values which were in effect when the corresponding Draw command was issued.

5.2.5 Rasterizer Modes

The only Delta specific requirement for the rendering modes in the Rasterizer Unit is that the BiasCoordinates bits in the RasterizerMode register (bits 4 and 5) are set to zero to select a zero bias for addition to the start X and Y values.

5.3 Rasterizer Unit

The Rasterizer decomposes a given primitive into a series of fragments for processing by the rest of the HyperPipeline.

PERMEDIA can directly rasterize:

- aliased screen aligned trapezoids
- aliased single pixel wide lines
- aliased single pixel points
- rectangles

All other primitives are treated as one or more of the above.

5.3.1 Trapezoids

PERMEDIA's basic area primitive is the screen aligned trapezoid. This is characterized by having top and bottom edges parallel to the X axis. The side edges may be vertical (a rectangle), but in general will be diagonal. The top or bottom edges can degenerate into points in which case we are left with either flat topped or flat bottomed triangles. Any polygon can be decomposed into screen aligned trapezoids or triangles. Usually, polygons are decomposed into triangles because the interpolation of values over non-triangular polygons is ill defined. The Rasterizer does handle flat topped and flat bottomed 'bow tie' polygons which are a special case of screen aligned trapezoids.

To render a triangle, the approach adopted to determine which fragments are to be drawn is known as 'edge walking'. Suppose the aliased triangle shown in Fig. 5.5 was to be rendered from top to bottom and the origin was bottom left of the window. Starting at (X1, Y1) then decrementing Y and using the slope equations for edges 1-2 and 1-3, the intersection of each edge on each scanline can be calculated. This results in a span of fragments per scanline for the top trapezoid. The same method can be used for the bottom trapezoid using slopes 2-3 and 1-3.

It is usually required that adjacent triangles or polygons which share an edge or vertex are drawn such that pixels which make up the edge or vertex get drawn exactly once. This may be achieved by omitting the pixels down the left or the right sides and the pixels along the top or lower sides. PERMEDIA has adopted the convention of omitting the pixels down the right hand edge. Control over whether the pixels along the top or lower sides are omitted depends on the start Y value and the number of scanlines to be covered. With the example, if **StartY** = Y1 and the number of scanlines is set to Y1-Y2, the lower edge of the top half of the triangle will be excluded. This excluded edge will get drawn as part of the lower half of the triangle.

To minimize delta calculations, triangles may be scan converted from left to right or from right to left. The direction depends on the dominant edge that is the edge which has the maximum range of Y values. Rendering always proceeds from the dominant edge towards the relevant subordinate edge. In the example above, the dominant edge is 1-3 so rendering will be from right to left.

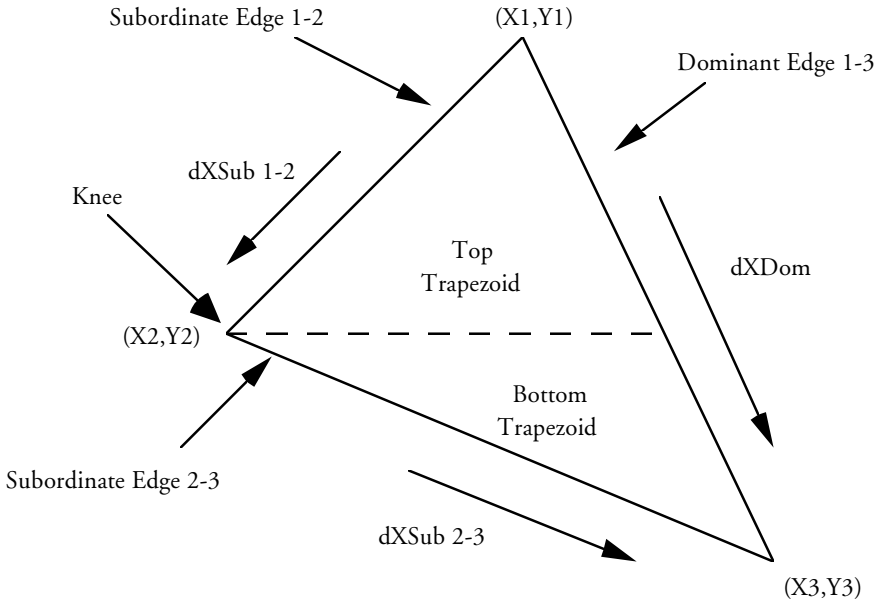


Figure 5.4 Rasterizing a triangle.

The sequence of actions required to render a triangle (with a 'knee') are:

- Load the edge parameters and derivatives for the dominant edge and the first subordinate edges in the first triangle.
- Send the **Render** command. This starts the scan conversion of the first triangle, working from the dominant edge. This means that for triangles where the knee is on the left we are scanning right to left, and vice versa for triangles where the knee is on the right.
- Load the edge parameters and derivatives for the remaining subordinate edge in the second triangle.
- Send the **ContinueNewSub** command. This starts the scan conversion of the second triangle.

Pseudocode for the above example is:

```
// Set the Rasterizer mode to the default, see
// §5.3.11

RasterizerMode (0)

// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format

StartXDom (X1<<16)
dXDom (((X3- X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2- X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<<16)                          // Down the screen
Count (Y1 - Y2)

// Set the render mode to aliased primitive with
// subpixel correction.  See §5.3.7

render.PrimitiveType = PERMEDIA_TRAPEZOID_PRIMITIVE
render.SubpixelCorrectionEnable = PERMEDIA_TRUE

// Draw top half of the triangle

Render (render)

// Set the start and delta for the second half of the
// triangle.

StartXSub (X2<<16)
dXSub (((X3- X2)<<16)/(Y3 - Y2))

// Draw lower half of triangle

ContinueNewSub (abs(Y2 - Y3))
```

After the **Render** command has been sent, the registers in PERMEDIA can immediately be altered to draw the second half of the triangle. For this, note that only two registers need be loaded and the command **ContinueNewSub** be sent. Once drawing of the first triangle is complete and PERMEDIA has received the **ContinueNewSub** command, drawing of this sub-triangle will start. The **ContinueNewSub** command register is loaded with the remaining number of scanlines to be rendered.

A **Continue** command can be used instead of the **ContinueNewSub** command in certain situations where it is beneficial to avoid reloading the Rasterizer's edge DDAs. However, accumulation of rasterization errors can occur which may result in imprecise rendering.

The **ContinueNewDom** command can be used to draw complex 2D shapes as a series of trapezoids. Since this command only affects the Rasterizer DDA and not that of any other units, it is not suitable for 3D operations.

5.3.2 Lines

Single pixel wide aliased lines are drawn using a DDA algorithm, so all PERMEDIA needs by way of input data is **StartX**, **StartY**, **dX**, **dY** and length. The algorithm calculates:

```
while (length--)
{
   X = X + dx
   Y = Y + dy
   plot ((int)X,  (int)Y)
}
```

Consider rendering a two segment polyline from (X_1, Y_1) to (X_2, Y_2) to (X_3, Y_3)

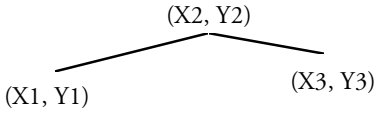


Figure 5.5 Polyline

Both segments are X major so:

abs $(X_{n+1} - X_n)$ > abs $(Y_{n+1} - Y_n)$

The pseudocode to render this line is shown below.

```
// Set the Rasterizer mode to the default, see
// §5.3.11

RasterizerMode (0)

// Load the delta values for the first segment.

StartXDom (X1<<16)
dXDom (1.0<<16)
StartY (Y1<<16)
dY (((Y2- Y1)<<16)/(X2 - X1))
Count (abs (X2 - X1))

// Set the render mode
render.PrimitiveType = PERMEDIA_LINE_PRIMITIVE

// Start rendering

Render (render)

// The first segment is complete, load delta
// for the second

dXDom (1.0<<16)
dY (((Y3- Y2)<<16)/(X3 - X2))

// Continue with the second segment

ContinueNewLine (abs (X3 - X2))
```

Note that the mechanism to render the second segment with the **ContinueNewLine** command is analogous to the **ContinueNewSub** command used at the knee of a triangle. Care must be taken when a continue command is being used for lines. Incorrect rendering can occur with operations such as alpha blending and logical ops if a segment draws back over the previous line segment thus attempting to reuse pixels that have just been updated. The solution is to send a **Sync** prior to the **ContinueNewLine.** This will ensure pending writes are flushed before the framebuffer reads for the new line segment. Note that there is no need to poll for the Sync here; the act of loading this command register is sufficient.

When a **Continue** command is used rather than a **ContinueNewLine**, some error will be propagated along the line so this is rarely used for lines. To minimize these errors, a choice of actions are available as to how the DDA units are restarted on the receipt of a **ContinueNewLine** command, see section §5.3.11.

It is recommended that for OpenGL rendering, the **ContinueNewLine** command is not used and individual segments are rendered.

5.3.3 Points

PERMEDIA supports a single pixel aliased point primitive. For points larger than one pixel, trapezoids should be used. The fields in the **Render** command register are described in detail later, however, in this case the PrimitiveType field in the **Render** command should be set to equal PERMEDIA_POINT_PRIMITIVE. The pseudocode portion to render an aliased unity sized point is:

```
// Set the Rasterizer mode to the default, see
// §5.3.11

RasterizerMode (0)

// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format

StartXDom (X<<16)
StartY (Y<<16)

// Set-up the render command.
render.PrimitiveType = PERMEDIA_POINT_PRIMITIVE

// Render the point

Render (render)
```

5.3.4 Rectangles

The rectangle primitive is restricted to integer pixel positions only; rectangles requiring sub-pixel positioning should use the trapezoid primitive. The rectangle is defined with two registers, RectangleOrigin which defines the X and Y start point, and RectangleSize which defines the width and height. The direction in which the

rectangle is filled can be controlled by the Render command, with separate control of fill direction in X and Y making the primitive suitable for copy operations.

5.3.5 Spans

Shapes more complex than points, lines or trapezoids may be drawn as a series of spans. Each span may be drawn as a horizontal line or as a single pixel high trapezoid. Both are special cases of 5.3.2 and 5.3.3 in that the loading of certain registers may be omitted e.g. dXDom, dXSub and dY. However, trapezoids can optionally use block writes for constant color spans and so may be preferable.

5.3.6 Block Write Operation

PERMEDIA supports SGRAM block writes with block sizes of 32 pixels. Any screen aligned trapezoid can be filled using block writes, not just rectangles. The SGRAM hardware writemasks can be used in conjunction with block writes.

The use of block writes is enabled by setting the FastFillEnable field in the **Render** command register.

Note only the Rasterizer and Framebuffer Write units are involved in block filling. The other units will ignore block write fragments, so it is not necessary to disable them.

5.3.7 Sub Pixel Precision and Correction

As the Rasterizer has fractional precision of 15 bits in X and Y, and the maximum screen width is 2048 pixels wide a number of bits, called subpixel precision bits, are available. The extra bits are required for a number of reasons:

- when using an accumulation buffer (where scans are rendered multiple times with jittered input vertices)
- for correct interpolation of parameters to give high quality shading as described below

PERMEDIA supports subpixel correction of interpolated values when rendering trapezoids. Subpixel correction ensures that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment's center. This correction is required to ensure consistent shading of objects made from many primitives. It should generally be enabled for all rendering which uses interpolated parameters.

5.3.8 Bitmaps

A Bitmap primitive is a trapezoid or line of ones and zeros which control which fragments are generated by the Rasterizer. Only fragments where the corresponding Bitmap bit is set are submitted for drawing. The normal use for this is in drawing characters, although the mechanism is available for all primitives. The Bitmap data is packed contiguously into 32 bit words so that rows are packed

adjacent to each other. Bits in the mask word are by default used from the least significant end towards the most significant end and are applied to pixels in the order they are generated in. The relationship between bits in the mask and the scanning order is shown in Fig. Figure 5.6.

Instead of rejecting fragments which fail the bitmask, they may be set to the background color. This is controlled by the **RasterizerMode** register. The background color comes from the **Texel0** register, which may be static or dynamically loaded through the Texture Read unit.

The Rasterizer scans through the bits in each word of the Bitmap data and increments the X,Y coordinates to trace out the rectangle of the given width and height. By default, any set bits (1) in the Bitmap cause a fragment to be generated, any reset bits (0) cause the fragment to be rejected.

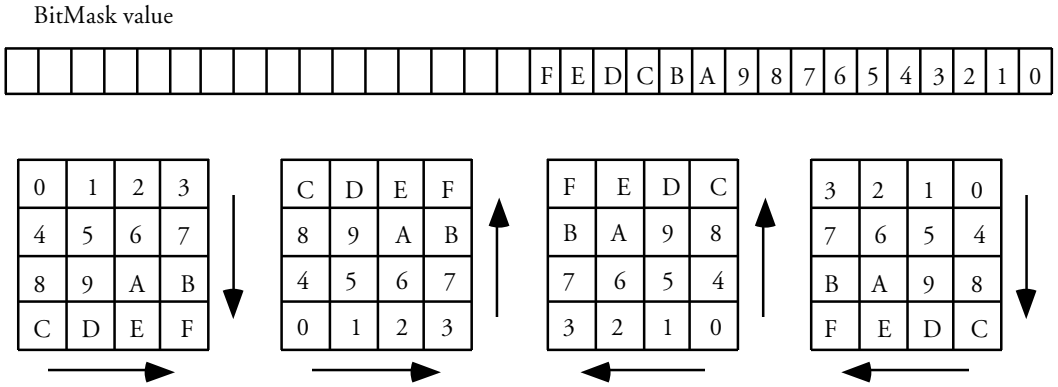


Figure 5.6 Relationship between Bitmask and Scanning Directions

The selection of bits from the **BitMaskPattern** register can be mirrored, that is, the pattern is traversed from MSB to LSB rather than LSB to MSB. Also, the sense of the test can be reversed such that a set bit causes a fragment to be rejected and vice versa. This control is found in the **RasterizerMode** register, described in section §5.3.11.

When one Bitmap word has been exhausted and pixels in the rectangle still remain then rasterization is suspended until the next write to the **BitMaskPattern** register, or the bitmask can be reused. If the bitmask is still valid when a new line is started it can continue to the next line or be discarded and a new one started; the start position of the mask can be specified to allow the first bits to be ignored. It is also possible to index into the mask using the X position of the Rasterizer. This allows 32 bit wide window aligned bit pattern; used with a new mask for every scanline a 32x32 stipple pattern can be supported.

For example a 5 pixel wide, 8 pixel high bitmap requires a register set-up as follows:

```
// Set the Rasterizer mode to the default, see
// §5.3.11

RasterizerMode (0)

// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format

StartXDom (X<<16)
dXDom (0)
StartXSub ((X + 5)<<16)       // Right hand edge pixels get
                              // missed off.

StartY (Y<<16)
dY (1<<16)
Count (8)

// At least the following bits require setting for the
// Render command.

render.PrimitiveType = PERMEDIA_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = PERMEDIA_TRUE
render.ReuseBitMask = PERMEDIA_FALSE

// Issue render command. First fragment will be
// generated on receipt of the BitMaskPattern

Render (render)

// 8x5 pixel bitmap requires 40 bits, and so 2
// 32 bit words.

BitMaskPattern (patternWord0)
BitMaskPattern (patternWord1)
```

Rendering will start as soon as the first patternWord is loaded into the **BitMaskPattern** register.

5.3.9 Block Writes and Bitmaps

The fastest way to render downloaded bitmap data, not requiring logical op processing, is to use block fills. The Rasterizer is set-up as normal setting the FastFillEnable bit. If it is necessary to also plot the background color then, the operation should be repeated for the background color but with the InvertBitMask bit set in the **RasterizerMode** register.

Since the downloaded bitmask data will be ANDed with masks generated by the Rasterizer without any re-alignment being performed, it is up to the host software to ensure that the masks match up. This can be achieved in two ways. First, the host software can align the bits that it downloads to match the alignment of the Rasterizer. A faster way is to use the User Scissor. This is the recommended method. Note that this is a general algorithm. In the special case where the data to

be downloaded is already aligned to 32 bits on both the left and right edges then the scissor need not be used.

For example, suppose that we want to download data to fill a rectangle with left edge at 10 and right edge at 200. And further, assume that the host bitmap data is to be loaded from an offset of 35 within the bitmap. Our goal is to match the bit at offset 35 with the pixel at offset 10.

Since we want to do the least amount of work on the host by avoiding shifting the data, we will actually download the host bitmap data at the previous 32-bit boundary. This means that we must set PERMEDIA up to discard the first 3 bits of data. We achieve this by rasterizing a rectangle whose left edge is 3 pixels less than that required, in this case we would rasterize the left edge to start at pixel 7. This causes the source bitmap data to be correctly aligned with the mask data produced by the Rasterizer. But, in order to protect the 3 pixels that we would otherwise overwrite, we use the scissor clip and set its bounds to be those of the original rectangle.

When using a block write operation like this, the Rasterizer will wait for new bitmask data to be downloaded at the start of each scanline. So we do not have to perform the alignment operation on the right hand edge.

A similar algorithm can be used to implement fast text rendering. For example, for fonts where each line fits into 32 bits, each line of a glyph can be downloaded as a mask.

Block writes can be used in combination with bitmasks with InvertBitMask and/or MirrorBitMask options but not BitMaskOffset or BitMaskPacking.

5.3.10 Copy/Upload/Download

PERMEDIA supports three "pixel rectangle" operations: copy, upload and download. These can apply to all buffer types.

Typically, a PERMEDIA copy moves *raw* blocks of data around buffers. To zoom or re-format data, either external software must upload the data, process it and then download it again, or the texture part of the Texture/Fog/Blend unit should be used.

To copy a rectangular area, the Rasterizer would be configured to render the destination rectangle, thus generating fragments for the area to be copied. PERMEDIA copy works by adding a linear offset to the destination fragment's address to find the source fragment's address. The calculation of the offset value is as shown in the diagram below:

Note that the offset is independent of the origin of the buffer or window, as it is added to the destination address. Care must be taken when the source and destination overlap to choose the source scanning direction so that the overlapping area is not overwritten before it has been moved. This may be done by swapping

the values written to the **StartXDom** and **StartXSub**, or by changing the sign of **dY** and setting **StartY** to be the opposite side of the rectangle.

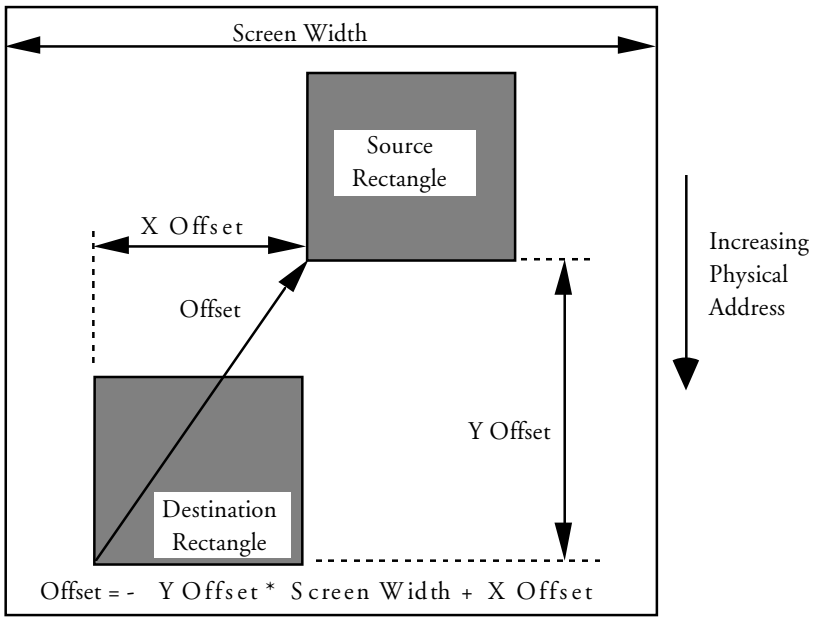


Figure 5.7 Copy Operation

PERMEDIA buffer upload/downloads are very similar to copies in that the region of interest is generated in the Rasterizer. However, the localbuffer and framebuffer are generally configured to read *or* to write only, rather than both read *and* write. The host out unit should be set to output data to the FIFO for image uploads. For downloads, the Rasterizer should be set to sync on the appropriate data type. This means that the Rasterizer will not generate the next fragment address until data is supplied from the host processor.

Units which can generate fragment values, the Color DDA unit for example, should generally be disabled for any copy/upload/download operations.

Warning: During image upload, all the returned fragments must be read from the Host Out FIFO, otherwise the PERMEDIA pipeline will stall. In addition it is strongly recommended that any units which can discard fragments (for instance the following tests: bitmask, user scissor, screen scissor, stipple, depth, stencil), are disabled otherwise a shortfall in pixels returned may occur, also leading to deadlock.

Note that because the area of interest in copy/upload/download operations is defined by the Rasterizer, it is not limited to rectangular regions.

Color formatting can be used when performing image copies, uploads and downloads. This allows data to be formatted from, or to, any of the supported PERMEDIA color formats, section §5.12.6 fully describes this operation.

5.3.11 Rasterizer Mode

A number of long-term modes can be set using the **RasterizerMode** register, these are:

- Mirror BitMask: This is a single bit flag which specifies the direction that bits are checked in the **BitMaskPattern** register. If the bit is reset, the direction is from least significant to most significant (bit 0 to bit 31), if the bit is set, it is from most significant to least significant (from bit 31 to bit 0).
- Invert BitMask: This is a single bit which controls the sense of the accept/reject test when using a Bitmask. If the bit is reset then when the BitMask bit is set the fragment is accepted and when it is reset the fragment is rejected. When the bit is set the sense of the test is reversed.
- Fraction Adjust: These 2 bits control the action taken by the Rasterizer on receiving a **ContinueNewLine** command. As PERMEDIA uses a DDA algorithm to render lines, an error accumulates in the DDA value. PERMEDIA provides for greater control of the error by doing one of the following:
    - leaving the DDA running, which means errors will be propagated along a line.
    - or setting the fraction bits to either zero, a half or almost a half (0x7FFF).
- Bias Coordinates: Only the integer portion of the values in the DDAs are used to generate fragment addresses. Often the actual action required is a rounding of values. This can be achieved by setting the bias coordinate bit to true which will automatically add almost a half (0x7FFF) to all input coordinates.
- ForceBackgroundColor: When set, if a fragment fails the bitmask test it is not discarded, but it is made to use the contents of the **Texel0** register in place of the normal color. This is used to provide foreground/background color selection.
- BitMaskByteSwapMode. This controls how or whether the bitmask is byte swapped a it is loaded. Four different byte orders are supported.
- BitMaskPacking. Controls whether a bitmask is discarded at the end of a scanline or continued onto the next. Not supported for block writes.
- BitMaskOffset. Sets the position of the first bit in the bitmask to test. Not supported for block writes.
- HostDataByteSwapMode. Controls byte swapping of host data being sent to the chip. This applies to any operation using the SyncOnHostData in the **Render** register. Four different byte orders are supported.
- LimitsEnable. When enabled, this allows quick rejection of fragments outside the defined area.
- BitMaskRelative. If enabled, this specifies that the bitmask should be accessed by an index made up of the lower 5 bits of the X coordinate of the current fragment.

5.3.12 Synchronization

For most circumstances PERMEDIA will automatically synchronize between primitives so that data for the first primitive is written before data for the second primitive is read. This is handled by data type, so localbuffer reads and writes are synchronized as are framebuffer reads and writes, but localbuffer reads are not synchronized with framebuffer writes.

If a unit is used to modify data that is not its normal type, then it may be necessary to explicitly synchronize the pipeline. If the Framebuffer Write unit is used to clear the localbuffer with block fills then the pipeline must be synchronized before localbuffer data is read. If the Framebuffer Write unit is used to download a texture

map, the pipeline must be synchronized before the Texture Read unit accesses the texture.

Explicit synchronization of the pipeline is done by the **WaitForCompletion** command. This has no data field, and may be inserted into a stream of commands; there is no need to wait for PERMEDIA to report that synchronization has taken place.

Alternatively, synchronization must be done with the **Sync** command, but this does require the host processor to poll the chip until it reports that the pipeline is idle (see the section on the Host Out unit).

5.3.13 X and Y limits clipping

The Rasterizer will normally rasterize all pixels on every scanline, generating a fragment per pixel. If large numbers of scanlines are subsequently clipped out by, for example, the scissor unit, then a lot of time can be wasted. The **Ylimits** register has been added to provide a way of quickly eliminating whole scanlines for a given primitive. This register effectively provides a Y scissor clip in the Rasterizer.

If limits testing has been enabled in the **RasterizerMode** register, and if a scanline being rasterized falls outside the Y limits bounds, then the Rasterizer will move directly onto the next scanline without rasterizing in X.

The **Xlimits** register has been added to avoid unnecessary rasterization, but does not act as a true X scissor clip. This is to ensure correct interpolation of color, fog etc. The limits registers are provided for efficiency reasons.

Both X and Y Limits clipping are automatically disabled when SyncOnHostData or SyncOnBitMask is used.

5.3.14 Registers

Real coordinates with fractional parts are provided to the Rasterizer in 2's complement fixed point. The point is kept consistent with a 16.16 format even though some of the integer and fractional bits may not be significant. The integer portion should be sign extended to fill unused bits; unused bits in the fraction should be set to zero.

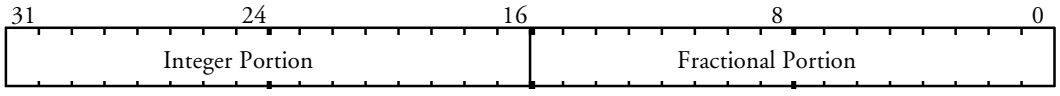


Figure 5.8 Real Coordinate Representation

When reference is made to "Signed Fixed Point Format", the sign bit is included in the integer section. For example, a signed fixed point format of 12.15 implies 1 sign bit followed by 11 integer bits and 15 fraction bits.

Register Name	Data Field	Description
Render	See below	Starts the rasterization process
ContinueNewDom	12 bit integer	Allows the rasterization to continue with a new dominant edge The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids, with continuity maintained across boundaries. Since this command only affects the Rasterizer DDA and not that of any other units, it is not suitable for 3D operations. The data field holds the number of scanlines to fill. Note this count does not get loaded into the Count register.
ContinueNewSub	12 bit integer	Allows the rasterization to continue with a new subordinate edge. The subordinate DDA is reloaded with the new parameters. The dominant edge is carried on from the previous trapezoid. This is useful when scan converting triangles with a 'knee' (i.e. two subordinate edges). The data field holds the number of scanlines to fill. Note this count does not get loaded into the Count register.
Continue	12 bit integer	Allows the rasterization to continue after new delta value(s) have been loaded, but does not cause either of the primitive's edge DDAs to be reloaded. This can result in the accumulation of rasterization errors causing imprecise rendering. The data field holds the number of scanlines to fill. Note this count does not get loaded into the Count register.
ContinueNewLine	12 bit integer	Allows the rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, however the fraction bits in the DDAs can be: kept, set to zero, half, or nearly one half, under control of the RasterizerMode. The data field holds the number of pixels in a line. Note this count does not get loaded into the Count register. The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments.
WaitForCompletion	Not used	This is used to suspend the PERMEDIA core until all outstanding reads and writes in framebuffer memory units have completed. This is intended to prevent a new primitive from starting to be rasterized before the previous primitive is *completely* finished. It would be used, for example, to separate texture downloads from the surrounding primitives. The same functionality can be achieved using the Sync command and waiting for it in the Host Out FIFO. However, using WaitForCompletion doesn't involve the host and can be inserted into a DMA buffer.

Table 5.4 Rasterizer Command Registers

RasterizerMode	See below	Defines the long term mode of operation of the Rasterizer.
StartXDom	Signed fixed point 12.15 format	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing.
dXDom	Signed fixed point 12.15 format	Value added when moving from one scanline to the next for the dominant edge in trapezoid filling. Also holds the change in X when plotting lines so for Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0, depending on the required scanning direction.
StartXSub	Signed fixed point 12.15 format	Initial X value for the subordinate edge.
dXSub	Signed fixed point 12.15 format	Value added when moving from one scanline to the next for the subordinate edge in trapezoid filling.
StartY	Signed fixed point 12.15 format	Initial scanline in trapezoid filling, or initial Y position for line drawing.
dY	Signed fixed point 12.15 format	Value added to Y to move from one scanline to the next. For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0, depending on the required scanning direction.
Count	12 bit integer	Number of pixels in a line. Number of scanlines in a trapezoid.
Xlimits	Xmax: 2's complement 12 bit value in the upper word. Xmin: 2's complement 12 bit value in the lower word.	Defines the X extents that the Rasterizer should fill between. A span is rasterized if its X value satisfies: Xmin £ X < Xmax
Ylimits	Ymax: 2's complement 12 bit value in the upper word. Ymin: 2's complement 12 bit value in the lower word.	Defines the Y extents that the Rasterizer should fill between. A scanline is filled if its Y value satisfies: Ymin £ Y < Ymax
RectangleOrigin	Y: 2's complement 12 bit value in the upper word. X: 2's complement 12 bit value in the lower word.	Defines the origin of a rectangle primitive. The corner of the rectangle this refers to is controlled by the rectangle fill direction fields in the Render command.
RectangleSize	Height: 2's complement 12 bit value in the upper word. Width: 2's complement 12 bit value in the lower word.	

Table 5.5 Rasterizer Control Registers

For efficiency, the **Render** command register has a number of bit fields that can be set or cleared per render operation, and which qualify other state information within PERMEDIA. These bits are AreaStippleEnable, TextureEnable, FogEnable, ReuseBitMask and SubpixelCorrection.

One use of this feature can occur when a window is cleared to a background color. For normal 3D primitives, stippling and fog operations may have been enabled, but these are to be ignored for window clears. Say that initially the **FogMode** and **AreaStippleMode** registers are enabled through the unit Enable bits. Now bits need only be set or cleared within the **Render** command to achieve the required result, removing the need to load the **FogMode** and **AreaStippleMode** registers for every Render operation.

The bit fields of the **Render** command register are detailed as follows:

Bit No.	Name	Description
0	AreaStippleEnable	Enable area stippling.
1, 2	Reserved	
3	FastFillEnable	Enable fast fill using VRAM block mode.
4, 5	Reserved	
6, 7	PrimitiveType	Set type of primitive: 0 = line 1 = trapezoid 2 = point 3 = rectangle
8, 9, 10	Reserved	
11	SyncOnBitMask	Enable bitmask test. Wait for new bitmask when current one expires unless SyncOnHostData or ReuseBitmask enabled.
12	SyncOnHostData	Wait for host data before sending step message.
13	TextureEnable	Enable texturing.
14	FogEnable	Enable fog.
15	Reserved	
16	SubPixelCorrectionEnable	Enable sub-pixel correction.
17	ReuseBitMask	Reuse bitmask when last bit used.
18, 19	Reserved	
20	RejectNegativeFace	Used by Delta unit.
21	IncreaseX	Direction of fill for rectangle
22	IncreaseY	Direction of fill for rectangle

Table 5.6 Render Command Register Fields

Several long-term Rasterizer modes are stored in the **RasterizerMode** register as shown below:

Bit No	Name	Description
0	MirrorBitMask	When this bit is set the bitmask bits are consumed from the most significant end towards the least significant end. When this bit is reset the bitmask bits are consumed from the least significant end towards the most significant end.
1	InvertBitMask	When this bit is set the bitmask is inverted first before being tested.
2,3	FractionAdjust	These bits are for the ContinueNewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted: 0: No adjustment is done 1: Set the fraction bits to zero 2: Set the fraction bits to half 3: Set the fraction to *nearly half*, i.e. 0x7fff
4,5	BiasCoordinates	These bits control how much is added onto the StartXDom, StartXSub and StartY values when they are loaded into the DDA units. The original registers are not affected: 0: Zero is added 1: Half is added 2: *Nearly half*, i.e. 0x7fff is added
6	ForceBackgroundColor	This bit, when set, causes the color to be taken from the Texel0 register instead of the normal color if the bitmask test fails.
7,8	BitMaskByteSwapMode	Controls byte swapping of the bitmask. If input is ABCD, 0: ABCD 1: BADC 2: CDAB 3: DCBA
9	BitMaskPacking	If enabled, the current bitmask is discarded at the end of every scanline even if it has not been finished. 0: Enabled 1: Disabled
10..14	BitMaskOffset	Position of first bit to test in bitmask.
15,16	HostdataByteSwapMode	Controls byte swapping of host data. If input is ABCD, 0: ABCD 1: BADC 2: CDAB 3: DCBA
17	Reserved	
18	LimitsEnable	If enabled, quickly reject areas of primitive outside defined area. 0: Disabled 1: Enabled
19	BitMaskRelative	Controls whether bitmask is indexed by counter or by lower 5 bits of X value. 0: Disabled 1: Enabled

Table 5.7 Rasterizer Mode Register

The register **BitMaskPattern** simply holds the 32-bit mask for bit mask stippling.

5.4 Scissor/Stipple Unit

Two scissor tests are provided in PERMEDIA, the User Scissor test and the Screen Scissor test. The user scissor checks each fragment against a user supplied scissor region; the screen scissor checks that the fragment lies within the screen. The stipple test checks each fragment against an 8x8 pattern.

5.4.1 User Scissor Test

The user scissor test, tests each fragment as follows:

XMin <= X < XMax

YMin <= Y < YMax

Where X and Y are the coordinates for the fragments, and XMin, XMax, YMin and YMax define the user supplied scissor region. If a fragment fails the test it is discarded. The test may be screen or window relative. This test applies to normal pixels and block fill operations.

5.4.2 Screen Scissor Tests

This test ensures that a pixel lies within the screen boundaries. For each fragment the XY origin stored in the **WindowOrigin** register is added to the fragment coordinates and this is tested against the screen boundaries stored in the **ScreenSize** register. Since the X and Y coordinates are held as 2's complement numbers, the window origin can be moved off the edges of the screen.

The following test is made:

0 <= (X + WX) < SW

0 <= (Y + WY) < SH

Where:

X = Fragment X coordinate WX = Window origin X coordinate

Y = Fragment Y coordinate WY = Window origin Y coordinate

SW = Screen Width

SH = Screen Height

The diagram below shows a simple scenario of a screen with a single window which has a user defined scissor region. The shaded area shows the region where fragments pass the user and screen scissor tests and so can progress in the pipeline. Fragments outside this region are culled from the pipeline. This test applies to normal pixels and block fill operations.

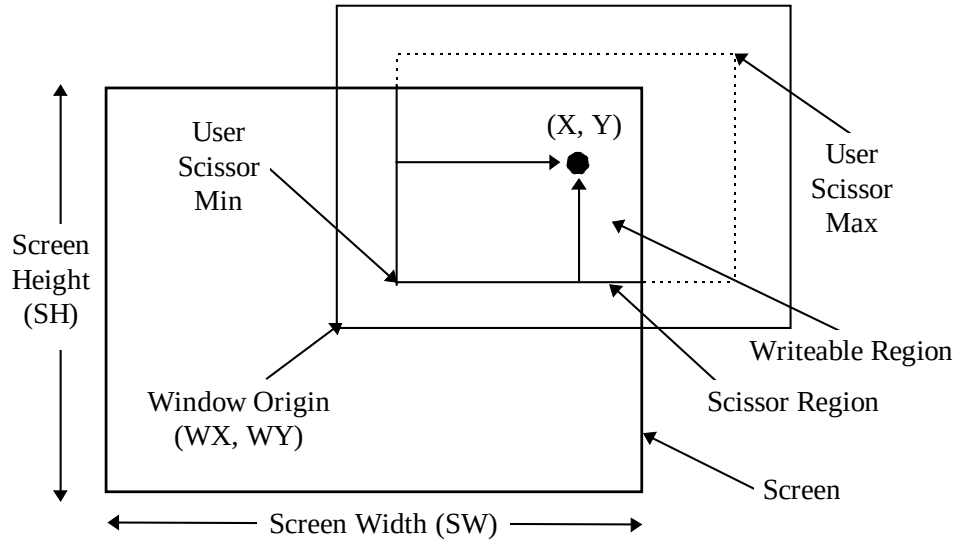


Figure 5.9 Screen Scissor and User Scissor Tests

This test may reject fragments if some part of a window has been moved off the screen. It will not reject fragments if part of a window is simply overlapped by another window.

The screen scissor would normally be enabled. The most common exception is during image upload.

5.4.3 Area Stippling

An 8 x 8 bit area stipple pattern can be applied to fragments. The least significant 3 bits of the fragment's (X,Y) coordinates, index into a 2D stipple pattern. If the selected bit in the pattern is set, then the fragment passes the test, otherwise it is rejected. In addition the bit pattern can be inverted or mirrored. Inverting the bit pattern has the effect of changing the sense of the accept/reject test. If the mirror bit is set the most significant bit of the pattern is towards the left of the window, the default is the converse.

In some situations window relative stippling is required but coordinates are only available screen relative. To allow window relative stippling, an offset is available which is added to the coordinates before indexing the stipple table. X and Y offsets can be controlled independently.

If the **ForceBackgroundColor** bit is set in the **AreaStippleMode** register, fragments which fail the area stipple test are not discarded. Instead, the contents of the **Texel0** register are used in place of the normal color for that pixel.

Area stippling is enabled using the **AreaStippleMode** register and must be qualified by the AreaStippleEnable bit in the **Render** command register. Area stippling may be used with block fills, but in this case the background color is not available.

5.4.4 Registers

The scissor operation is controlled by the **ScissorMode** register:

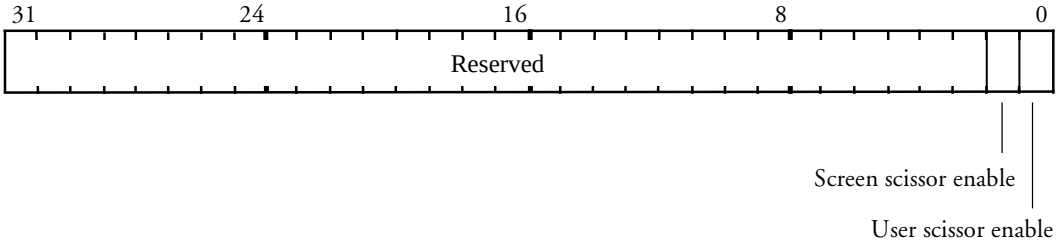


Figure 5.10 Scissor Mode Register

The screen scissor test would normally always be enabled. The most common exception is during image upload.

The user scissor region is specified by two registers **ScissorMinXY** and **ScissorMaxXY** the X values are stored in the least significant 16 bits of the register, the Y values in the most significant 16 bits of the register.

The **WindowOrigin** register has the X coordinate of the origin stored in the least significant 16 bits of the register, and the Y coordinate in the most significant 16 bits of the register. As each fragment is generated by the Rasterizer unit, this origin is added to the coordinates of the fragment to generate its screen coordinates.

The **ScreenSize** register specifies the screen width and height, with the width in the least significant 16 bits and the height in the most significant 16 bits.

The area stipple operation is controlled by the **AreaStippleMode** register:

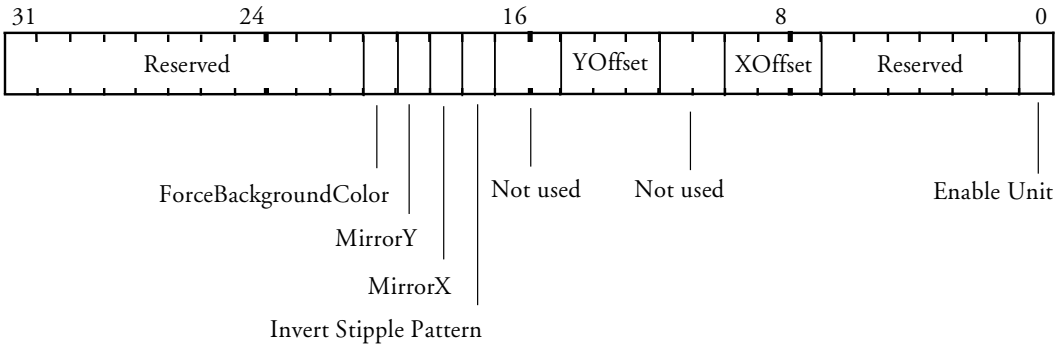


Figure 5.11 AreaStippleMode Register

The EnableUnit bit is qualified by the AreaStippleEnable bits in the **Render** command register. The area stipple is set-up in the **AreaStipplePattern** *n* register, where *n* represents an integer between 0 and 7.

5.4.5 Scissor Example

To enable screen scissor for a region: 10 <= X < 500, 100 <= Y < 200 with a screen size of 1280x1024 and the window origin at (100,100).

```
// Set the screen size
screenSize.Width = 1280
screenSize.Height = 1024

ScreenSize(screenSize)

// Set the window origin
windowOrigin.X = 100
windowOrigin.Y = 100

// Set-up the user scissor values
minXY.X = 10
minXY.Y = 100
maxXY.X = 500
maxXY.Y = 200
ScissorMinXY(minXY)          // Load the registers
ScissorMaxXY(maxXY)

// Enable the unit
scissorMode.UserScissorEnable = PERMEDIA_ENABLE
scissorMode.ScreenScissorEnable = PERMEDIA_ENABLE

ScissorMode(scissorMode)

// Render primitives
```

5.4.6 Area Stipple Example

A repeating area stipple pattern of 2x2 pixels producing a 50% grey area:

```
AreaStipplePattern0(0xAA)
AreaStipplePattern1(0x55)
AreaStipplePattern2(0xAA)
AreaStipplePattern3(0x55)
AreaStipplePattern4(0xAA)
AreaStipplePattern5(0x55)
AreaStipplePattern6(0xAA)
AreaStipplePattern7(0x55)

// Set-up mode register
areaStippleMode.UnitEnable = PERMEDIA_ENABLE
areaStippleMode.XOffset = 0
areaStippleMode.YOffset = 0
areaStippleMode.Invert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0

// Load mode register
AreaStippleMode(areaStippleMode)

// When issuing a Render command, the AreaStippleEnable bit
// should be set in addition to the area stipple test being
// enabled:
// render.AreaStippleEnable = PERMEDIA_TRUE
```

5.5 Localbuffer Read and Write Units

The localbuffer holds the Stencil and Depth data associated with a fragment. Although separate units in the Hyperpipeline, the localbuffer read and write units are best considered as a pair.

5.5.1 Localbuffer Read

The **LBReadMode** register can be configured to make 0, 1 or 2 reads of the localbuffer. The following are the most common modes of access to the localbuffer:

- Normal rendering without depth or stencil testing. This requires no localbuffer reads or writes.
- Normal rendering with depth and/or stencil testing required which conditionally requires the localbuffer to be updated. This requires localbuffer reads and writes to be enabled.
- Copy operations. Operations which copy all or part of the localbuffer. This requires reads and writes enabled.
- Upload/download operations. Operations which download depth or stencil information to the localbuffer, or read back depth or stencil values from the localbuffer to the host.

The address calculation implements the following equations:

Bottom left origin -

Destination address = LBWindowBase - Y * W + X

Source address = LBWindowBase - Y * W + X + LBSourceOffset

Top left origin -

Destination address = LBWindowBase + Y * W + X

Source address = LBWindowBase + Y * W + X + LBSourceOffset

where:

Destination address	is the address any write will be made to and any destination read will be made from.
Source address	is the address a source read will be made from.
X	is the pixel's X coordinate.
Y	is the pixel's Y coordinate.
LBWindowBase	holds the base address in the localbuffer of the current window.

LBSourceOffset — is normally zero except during a copy operation where data is read from one address and written to another address. The offset from destination to source is held in the **LBSourceOffset** register.

W — is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **LBReadMode** register. See the table in Appendix C for more details.

The localbuffer can be read in three formats: LBDefault, LBStencil or LBDepth. These tell PERMEDIA which areas of the localbuffer is required. LBDefault is used for all copy and rendering operations, LBStencil and LBDepth are used for image upload of the Stencil and Depth planes. The table below summarizes the common rendering operations and the read modes required for them:

ReadSource	ReadDestination	Writes	Data Type	Rendering Operation
Disabled	Disabled	Disabled	-	Rendering with no Depth or Stencil enabled.
Disabled	Disabled	Enabled	LBStencil LBDepth	Download to localbuffer from host
Disabled	Enabled	Disabled	LBStencil LBDepth	Upload from localbuffer to host
Disabled	Enabled	Enabled	LBDefault	Rendering with depth and/or stencil updates enabled.
Enabled	Disabled	Enabled	LBDefault	Localbuffer copy operations .

Table 5.8 Localbuffer Read/Write Modes

5.5.2 Localbuffer Write

Writes to the localbuffer must be enabled to allow any update of the localbuffer to take place. The **LBWriteMode** register is a single bit flag which controls updating of the buffer.

5.5.3 Localbuffer Data Formats

The Depth field can be either 15 or 16 bits wide and the Stencil field either 1 or 0 bits wide. The total width of the localbuffer data should not be greater than 16 bits. If a Stencil field is defined it occupies bit 15; the depth field always starts at bit 0.

The **LBReadFormat** and **LBWriteFormat** registers must be configured to the appropriate values, see Fig. 5.15. The format can be different for different windows.

5.5.4 Registers

The **LBReadMode** register is as shown below:

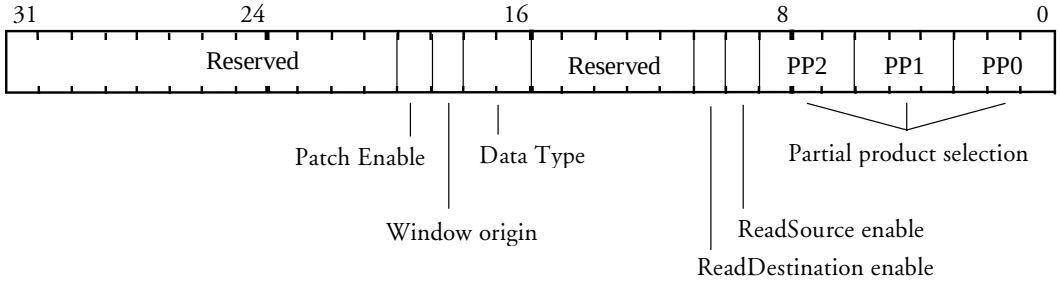


Figure 5.12 LBReadMode Register

PatchEnable, when set, enables normal patch addressing of the localbuffer. This typically results in more efficient memory bandwidth utilization.

The Partial Product fields PP0, PP1, and PP2 define the width of the localbuffer. They are described in Appendix C.

ReadSourceEnable and ReadDestinationEnable control localbuffer reads of the destination address and source address respectively. DataType controls the format of localbuffer data, and WindowOrigin specifies if the window origin is Top Left or Bottom Left.

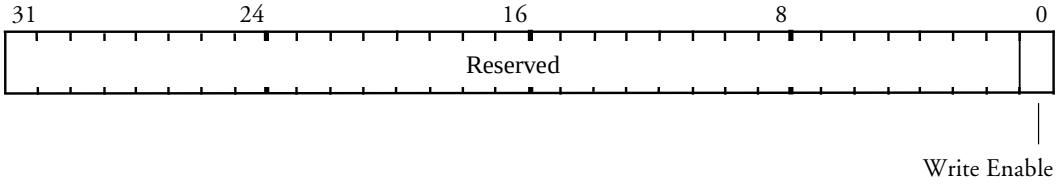


Figure 5.13 LBWriteMode Register

The localbuffer format must be specified for both reads and writes using the **LBReadFormat** and **LBWriteFormat** registers. Normally these registers are set to identical values. It may be useful to set them to different values when, say, copying between two windows using different depth widths.

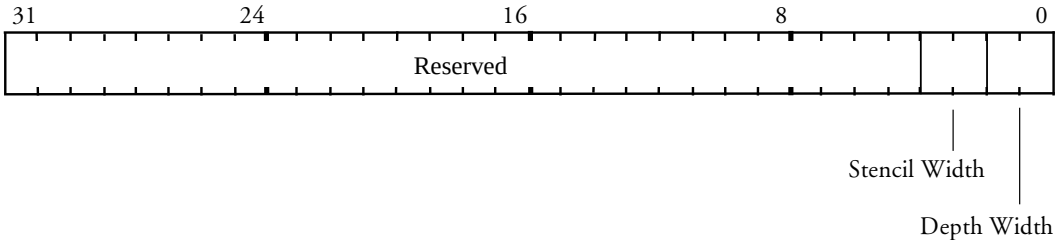


Figure 5.14 LBReadFormat / LBWriteFormat Register

LBWriteMode is a single bit register. When the least significant bit is set, writes to the localbuffer are enabled.

LBSourceOffset holds a 24 bit 2's complement value used in copy operations.

LBWindowBase updates the base address of the localbuffer.

The relative positions of the depth and stencil fields within the localbuffer are fixed. If a Stencil field is defined then it occupies bit 15. The depth field always commences at bit 0.

5.5.5 Localbuffer Example

The following is an example of a rendering operation with localbuffer read and write. PERMEDIA is configured with a 16 bit localbuffer such that 15 bits are used for depth and 1 bit for stencil with a screen size of 800x600.

```
lbReadFormat.DepthWidth = 3          // 15 bit
lbReadFormat.StencilWidth = 3        // 1 bit

LBReadFormat(lbReadFormat)           // Load read format
LBWriteFormat(lbReadFormat)          // Write is same as read

// Set the localbuffer write mode
LBWriteMode(PERMEDIA_ENABLE)

// Set the localbuffer read mode

// Partial products for 800 : 512 + 256 + 32

lbReadMode.PP0 = 5     // 512 (<< 9)
lbReadMode.PP1 = 4     // 256 (<< 8)
lbReadMode.PP2 = 1     // 32  (<< 5)

lbReadMode.ReadSource = PERMEDIA_DISABLE
lbReadMode.ReadDestination = PERMEDIA_ENABLE
lbReadMode.DataType = PERMEDIA_LBDEFAULT
lbReadMode.WindowOrigin = as appropriate
lbReadMode.PatchMode = PERMEDIA_DISABLE
LBReadMode(lbReadMode)


// Now ready to render with localbuffer read and write
// suitable for stencil and depth buffering operations.
```

5.6 Stencil/Depth Test Unit

The stencil test conditionally rejects fragments based on the outcome of a comparison between the value in the stencil buffer and a reference value. The

stencil buffer is updated according to the current stencil update mode which depends on the result of the stencil test and the depth test. Stencil testing can be used in many different ways, e.g. hidden line removal, decals, masking areas of the screen, stippling.

The depth (Z) test, if enabled, compares a fragment's depth against the corresponding depth in the depth buffer. If the test fails, the fragment will be rejected.

5.6.1 Stencil Test

This test only occurs if all the preceding tests (bitmask, scissor, stipple) have passed. The stencil test is controlled by the *stencil function* and the *stencil operation*. The stencil function controls the test between the reference stencil value and the value held in the stencil buffer. If the test is LESS and the result is true then the fragment value is less than the source value. The stencil operation controls the updating of the stencil buffer, and is dependent on the result of the stencil and depth tests.

The table below shows the stencil functions available:

Mode	Comparison Function
0	Never
1	Less
2	Equal
3	Less or Equal
4	Greater
5	Not Equal
6	Greater or Equal
7	Always

Table 5.9 Stencil Comparison Modes

Some of these comparison modes are effectively redundant as PERMEDIA only uses 1 bit stencil values. They have been included to ease software compatibility with GLINT and possible future devices.

If the stencil test is enabled then the stencil buffer will be updated depending on the outcome of *both* the stencil and the depth tests (if the depth test is disabled the depth result is set to pass). Refer to the tables below and the definition of the **StencilMode** register in section §5.6.4 to fully understand their relationship.

		Stencil Test	
		Pass	Fail
Depth Test	Pass	*dppass*	*sfail*
	Fail	*dpfail*	*sfail*

Table 5.10 Possible Update Operations for Stencil Planes

The entries dppass, dpfail and sfail are set to one of the update operations below, source stencil is the value in the stencil buffer:

Update Method	Mode	Stencil Value
Keep	0	Source stencil
Zero	1	0
Replace	2	Reference stencil
Increment	3	Clamp (Source stencil + 1) to $2^{\text{stencil width}} - 1$
Decrement	4	Clamp (Source stencil -1) to 0
	5	~Source stencil

Table 5.11 Stencil Operations

In addition a comparison bit mask is supplied in the **StencilData** register. This is used to establish which bits of the source and reference value are used in the stencil function test.

The source stencil value can be from a number of places as controlled by a field in the **StencilMode** register:

LBWriteData Stencil	Use
Test logic	This is the normal mode.
Stencil register	This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the **Stencil** register. It is used when a constant stencil value is needed, for example when clearing the stencil buffer .
LBSourceData: (stencil value read from the localbuffer)	This is used, for instance, in the OpenGL copy pixels function when the stencil planes are to be copied to the destination. The source is offset from the destination by the value in **LBSourceOffset** register.
Source stencil value read from the localbuffer	This is used, for instance, in the OpenGL copy pixels function when the stencil planes in the destination are **not** to be updated. The stencil data will come from the localbuffer data.

Table 5.12 Stencil Sources

See *The OpenGL Reference Manual* and *The OpenGL Programming Guide* from Addison-Wesley for more details of the stencil operations and examples of its use.

5.6.2 Depth Test

This test is only performed if all the preceding tests (bitmask, scissor, stipple) have passed. The comparison tests available are:

Mode	Comparison Function
0	Never
1	Less
2	Equal
3	Less Than or Equal
4	Greater
5	Not Equal
6	Greater Than or Equal
7	Always

Table 5.13 Depth Comparison Modes

The test compares the fragment's depth against a source depth value. If the compare function is LESS and the result is true then the fragment value is less than the source value. The source value can be obtained from a number of places as controlled by a field in the **DepthMode** register.

Source	Use
DDA (see below)	This is used for normal Depth buffered 3D rendering.
Depth register	This is used, for instance, in the OpenGL draw pixels function where the host supplies the depth values through the **Depth** register. Alternatively this is used when a constant depth value is needed, for example, when clearing the depth buffer or 2D rendering where the depth is held constant.
LBSourceData: Source depth value from the localbuffer	This is used, for instance, in the OpenGL copy pixels function when the depth planes are to be copied to the destination.
Source Depth	This is used, for instance, in the OpenGL copy pixels function when the depth planes in the destination are *not* updated. The depth data will come from the localbuffer.

Table 5.14 Depth Sources.

For a depth buffered trapezoid, PERMEDIA interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in the diagram below. The rendering direction chosen here is bottom to top.

ZStart = Start Z value

dZdyDom = Increment along dominant edge

dZdx = Increment along the scan line.

The dZdx value is not required for Z-buffered lines.

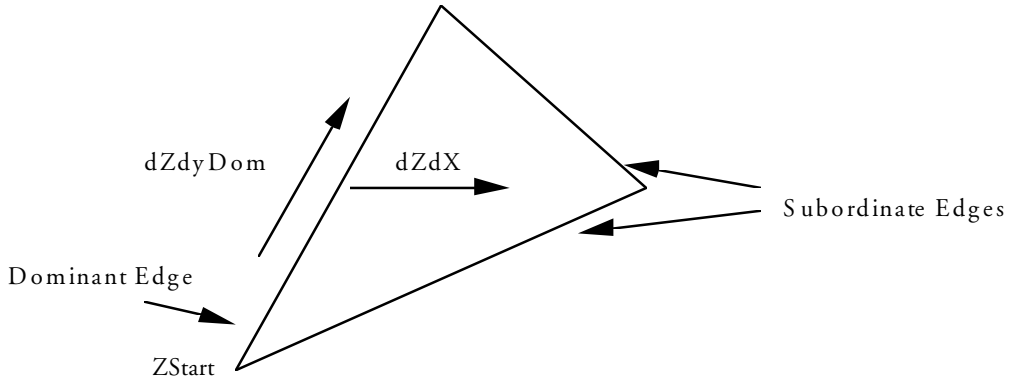


Figure 5.15 Depth Interpolation

The number format for the increment values is 2's complement fixed point integer: 16 bits integer and 11 bits fraction. All the start, derivative and internal data is in this format. This is mapped into the Upper and Lower registers (U and L) as shown below:

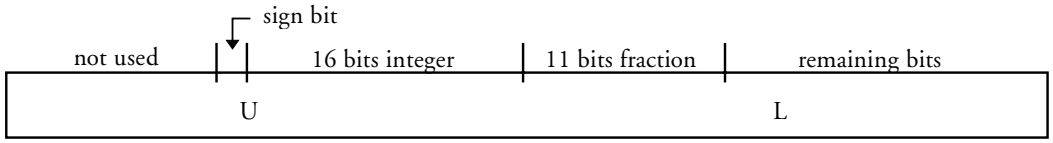


Figure 5.16 Depth Derivative Format

This data format is compatible with GLINT 300SX and GLINT 500TX graphics processors. In many instances, the fractional part can be left containing zero, avoiding the need to continually update ZStartL, dZdxL and dZdyDomL.

The Depth unit must be enabled to update the depth buffer. If it is disabled then the depth buffer will only be updated if ForceLBUpdate is set in the **Window** register. If no updates of the localbuffer are required, setting DisableLBUpdate in the **Window** register may improve performance.

5.6.3 Registers

Stencil test is controlled by the **StencilMode** register:

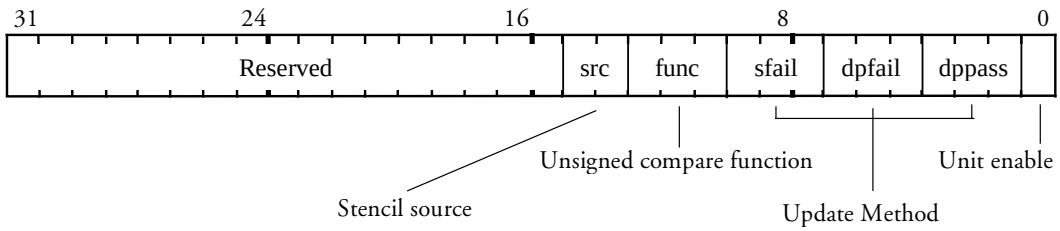


Figure 5.17 StencilMode Register

The **StencilData** register holds the other data associated with the test.

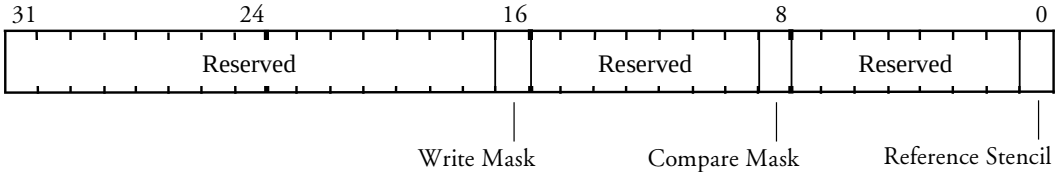


Figure 5.18 StencilData Register

The stencil writemask is used to control which stencil planes are updated as a result of the test. The **Stencil** register holds an externally sourced stencil value. It is a 32bit register of which only the least significant bit is used. The unused bits should be set to zero.

The Stencil unit must be enabled to update the stencil buffer. If it is disabled then the stencil buffer will only be updated if ForceLBUpdate is set in the **Window** register.

Operation of the Depth unit is controlled by the **DepthMode** register:

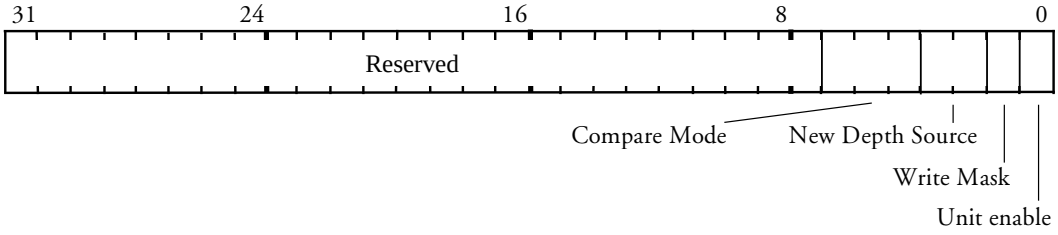


Figure 5.19 DepthMode Register

The single bit writemask is used to control updating all the bits in the depth buffer.

The **Depth** register holds an externally sourced 16 bit depth value. If the depth buffer holds 15bits then the user supplied depth value is right justified to the least significant end of the register. The unused most significant bit should be set to zero.

The DDA and other registers are shown below (note the increment values are split into two registers):

Register	Description
ZStartU	Depth start value
ZStartL	
dZdxU	Depth derivative per unit X
dZdxL	
dZdyDomU	Depth derivative per unit Y, dominant edge or along a line.
dZdyDomL	

Table 5.15 Depth Interpolation Registers

The Window register is used to control the update of the localbuffer.

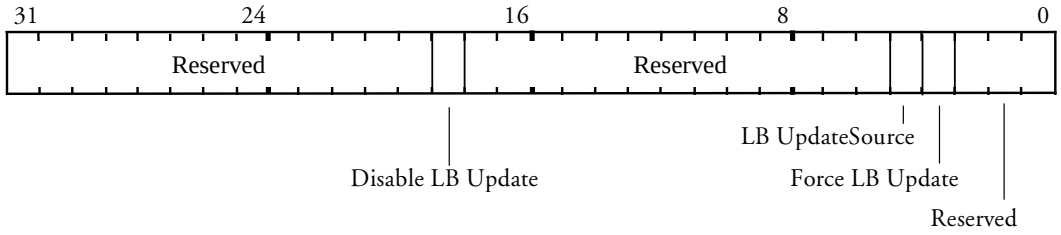


Figure 5.20 Window Register

5.6.4 Stencil Example

This example sets the Stencil unit to use a supplied reference value (0x1) and to test fragments to be LESS than this value. It also sets the stencil planes update function to be Decrement if the test passes *and* the depth test passes (or is not enabled), otherwise it sets the update function to Keep. Because Decrement is the selected mode, this example does not require that the Stencil register be loaded.

```
// Set the localbuffer read and write modes
// See section §5.5

// Set the stencil modes

stencilMode.UnitEnable = PERMEDIA_ENABLE
stencilMode.DPPass = PERMEDIA_STENCIL_METHOD_DECREMENT
stencilMode.DPFail = PERMEDIA_STENCIL_METHOD_KEEP
stencilMode.SFail = PERMEDIA_STENCIL_METHOD_KEEP
stencilMode.CompareFunction = PERMEDIA_STENCIL_COMPARE_LESS
stencilMode.StencilSource = PERMEDIA_SOURCE_TEST_LOGIC
StencilMode(stencilMode)

// Set the reference stencil value and set the
// compare and writemasks to 0x1

stencilData.ReferenceStencil = 0x1
stencilData.CompareMask = 0x1
stencilData.StencilWriteMask = 0x1

StencilData(stencilData)

// Enable the depth test here if required, if not enabled
// the result of the depth test is set to pass.
```

5.6.5 Depth Example

This example does the required set-up for drawing a depth buffered primitive.

```
// Set the localbuffer read and write modes
// See section §5.5

depthMode.UnitEnable = PERMEDIA_ENABLE
depthMode.WriteMask = 1
depthMode.NewDepthSource = PERMEDIA_NEW_DEPTH_SOURCE_DDA
depthMode.CompareMode = PERMEDIA_DEPTH_COMPARE_MODE_LESS

DepthMode(depthMode)

// Load the depth start values and deltas for the dominant edge
// and the body of the trapezoid

ZStartU()    // Load upper and lower start values
ZStartL()
dZdxU()      // Load upper and lower dZdx deltas
dZdxL()
dZdyDomU()   // Load upper and lower dominant edge deltas
dZdyDomL()

// Render primitive
```

5.7 Texture Address Unit

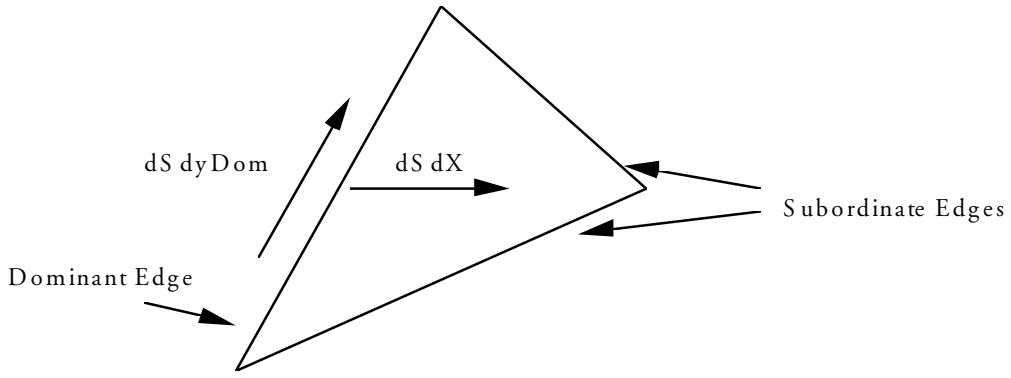
The Texture Address unit calculates the address of the texel that maps to the current fragment XY position. Perspective correction can be applied as part of the operation.

The texture coordinates are referred to as S and T where S is analogous to X and T to Y. The S and T values are generated by interpolation; a third component, Q, may also be interpolated and is used in perspective correction.

5.7.1 Texture Interpolation

The DDA units perform linear interpolation given a set of start and increment values.

PERMEDIA interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required per texture component, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated, for the S component, in the diagram below:



SStart = Initial S value
dSdyDom = S gradient in the Y direction along the dominant edge
dSdx = S gradient in the X direction

Figure 5.21 Texture Address Interpolation

The calculation for the delta values is the same as other parameters such as depth values see Appendix D6.

If perspective correction is not enabled then the S and T values are the texture coordinates of the appropriate vertex. If perspective correction is enabled the texture coordinates are divided by the homogenous coordinate W, and Q is formed from 1/W. S, T and Q are then normalized. A Q value of zero will be handled in a reasonable manner.

If perspective correction is enabled each interpolated S and T value is divided by the interpolated Q value. The result is passed to the Texture Read unit which reads the texel from memory.

If subpixel correction has been enabled for a primitive, then any correction required will be applied to the texture coordinates.

5.7.2 Registers

The S and T values are in 30 bit 2's complement format:

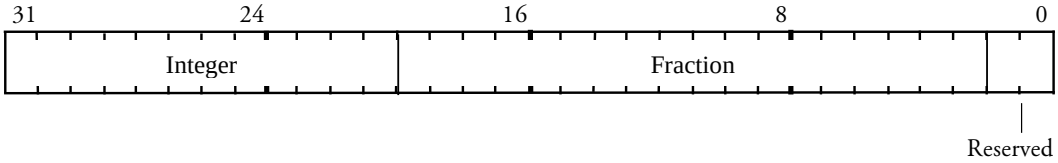


Figure 5.22 Fixed Point S and T Format

The Q values are in 29 bit 2's complement format:

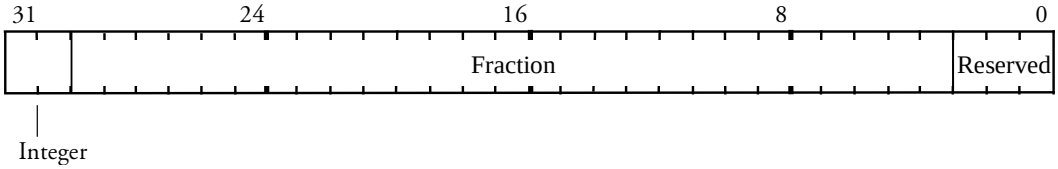


Figure 5.23 Fixed Point Q Format

The registers to set-up Texture interpolation are:

Register	Data Field	Description
Sstart	30 bit 2's comp fix pt	S start value
dSdx	30 bit 2's comp fix pt	S derivative per unit X
dSdyDom	30 bit 2's comp fix pt	S derivative per unit Y, dominant edge
Tstart	30 bit 2's comp fix pt	T start value
dTdx	30 bit 2's comp fix pt	T derivative per unit X
dTdyDom	30 bit 2's comp fix pt	T derivative per unit Y, dominant edge
Qstart	29 bit 2's comp fix pt	Q start value
dQdx	29 bit 2's comp fix pt	Q derivative per unit X
dQdyDom	29 bit 2's comp fix pt	Q derivative per unit Y, dominant edge

Table 5.16 Texture Interpolation Registers

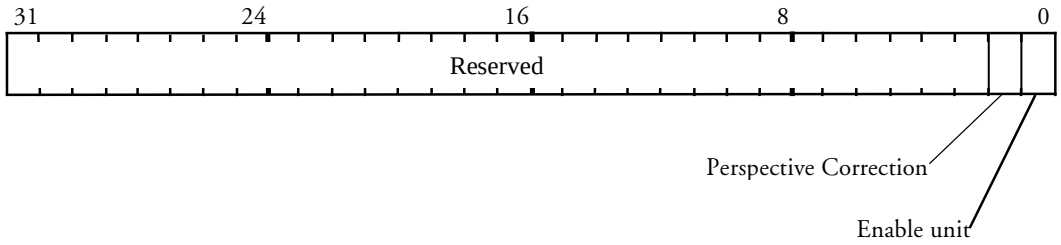


Figure 5.24 TextureAddressMode

5.7.3 Texture Interpolation Example

This example sets up the parameters for 2D texture mapping. 1D texture mapping can be achieved by setting TStart, dTdx and dTdyDom to zero.

```
// Load the start values and deltas for the dominant edge
// and the body of the trapezoid

SStart()    // Load S start value
TStart()    // Load T start value
QStart()    // Load Q start value

dSdx()      // Load S delta for X
dTdx()      // Load T delta for X
dQdx()      // Load Q delta for X

dSdyDom()   // Load S dominant edge delta
dTdyDom()   // Load T dominant edge delta
dQdyDom()   // Load Q dominant edge delta

// Render primitive
```

5.8 Texture Read Unit

The texture buffer holds texture data. The buffer shares the same memory as the localbuffer and framebuffer; texture maps are normally written to memory through the framebuffer write unit in a similar manner to image download.

The Texture Read unit receives texture addresses from the Texture Address unit and reads data from memory. If bilinear filtering is enabled, several accesses may be done to collect the correct number of texels.

5.8.1 Read Unit

The address calculation implements the following equations:

Bottom left origin -

Address = TextureBaseAddress - T* W + S

Top left origin -

Address = TextureBaseAddress + T * W + S

where:

Address	is the address any read will be made from.
S	is the texel's S coordinate.
T	is the texel's T coordinate.
TextureBaseAddress	holds the base address of the current texture.
W	is the texture width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **TextureReadMode** register. See the table in Appendix C for more details.

The **TextureMapFormat** register specifies how the texture map is held in memory. This includes the width of the texture map using partial product codes and the size of the texel. The **TextureReadMode** register specifies how the texture map should be handled internally. This sets the width (maximum S) and height (maximum T) that should be used when accessing the texture. There are three ways that the address can be modified if it exceeds either the width or height (or goes negative):

Clamp	clamp the coordinate to 0 or the maximum value.
Repeat	access the map modulo the width or height. This results in the texture map being repeated.

Mirror	access the map modulo the width or height and mirror alternate texture maps.

The width used to repeat or clamp can be different to the width used to set the stride of the texture in memory. This allows a texture to be selected from part of a larger image.

5.8.2 Texture Base Address

The base address of the texture map is set in the **TextureBaseAddress** register. The lower 24 bits of this field specify the address of the map in texels. Bit 30 is used to specify that the texture is held in system memory instead of local memory and the texture should be 'executed' directly across the PCI bus without first copying the texture to local memory. Refer to the *PERMEDIA 2 Hardware Reference Manual* for more details.

The base address of the texture may be loaded indirectly from memory using the **TextureID** register. The value loaded into this register should be the address in memory of the base address of the texture (specified in 32 bit units). Loading the **TextureID** register causes the real base address to be loaded from memory. If bit 31 of the value loaded is set, the value is interpreted as invalid, the graphics processor halted, and an interrupt issued to the CPU. This mechanism is normally used to indicate that the required texture is not resident in local memory and should be copied in. Once the copy has been completed and the texture base address in memory is updated with its invalid bit clear, the graphics processor re-reads this value and restarts. Refer to the *PERMEDIA 2 Hardware Reference Manual* for details on loading textures while the Graphics Processor has stalled.

5.8.3 Texture Filtering

A bilinear filter is available which combines the values of the 4 texels surrounding the index into the texture map to produce a single value. This filter will reduce pixelation effects when textures are enlarged, and reduce aliasing effects when textures are shrunk.

5.8.4 Texture Formatting

The texture map can be held in memory in a variety of formats that correspond to the formats supported by the framebuffer. Two additional formats are provided to allow texture maps to be stored in YUV color format. When a texel is read into PERMEDIA it is converted to the internal color format. External color formats are shown in table 4.1. Note: the color format value is made up of the 4 bits of the TextureFormat field and the 1 bit TextureFormatExtension field in the **TextureDataFormat** register.

If the selected format has no alpha buffer, a default value of 0xFF, which is the maximum is used. If the NoAlphaBuffer bit is set in the **TextureDataFormat** register then 0xFF is used even if the format has an alpha buffer.

If the texture is in Color Index mode (either 4 or 8 bits) the single value is repeated for all color components. If the framebuffer format is also Color Index, the single value is used as the pixel color; if the framebuffer is RGBA, then the texture value becomes grey scale.

The texture values may be indexed through a 256 entry look-up table. Each entry of the table holds a 32 bit RGBA value. If the CI8 texture is used, then the whole LUT is used for each texture; if the CI4 texture format is used each texture uses 16 entries, so 16 separate LUTs may be loaded and the appropriate one indexed (the upper 4 bits of the index are supplied by the upper 4 bits of TexelLUTIndex).

If an RGB or RGBA texture format is used (as opposed to CI8 or CI4) the individual R, G, B, and A components are indexed separately which allows remapping functions such as gamma correction.

5.8.5 Registers

The **TextureReadMode** register controls the way that textures are read from memory.

The S and T wrap modes can be set to clamp, repeat or mirror as described earlier.

With Filter Mode disabled, nearest-neighbor texture mapping will be performed. With this bit set, bilinear filtering is enabled.

The Packed Data bit is used to define how texels are read from memory. If this bit is cleared, each texel is read one at a time; if set several texels can be read simultaneously improving efficiency. The actual number of texels read in this case is dependent on the texel size. See section §5.10.4 for how this can be used for packed copies.

The **TextureReadMode** register controls the way that textures are read from memory. With Filter Mode disabled, nearest-neighbor texture mapping will be performed. With it set, bilinear filtering is enabled.

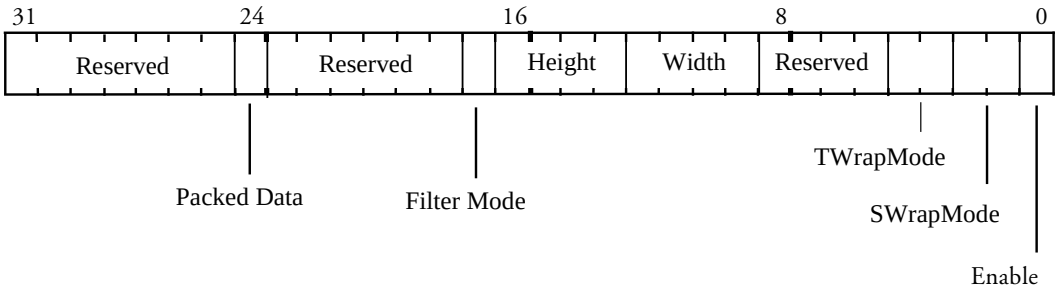


Figure 5.25 TextureReadMode Register

The **TextureMapFormat** register specifies the way that the texture map is held in memory. The partial product codes are detailed in Appendix C. The window origin specifies the origin as being top left or bottom left. SubPatchMode when enabled, improves the performance of typical texture mapping.

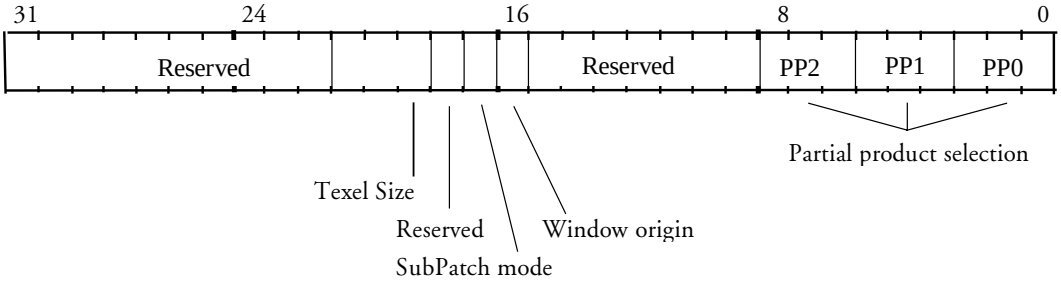


Figure 5.26 TextureMapFormat Register

The **TextureDataFormat** register specifies the color format of the texture. The TextureFormat combined with the TextureFormat Extension contain one of the modes described in table 4.1. The color order specifies whether the texture is in RGB or BGR color format.

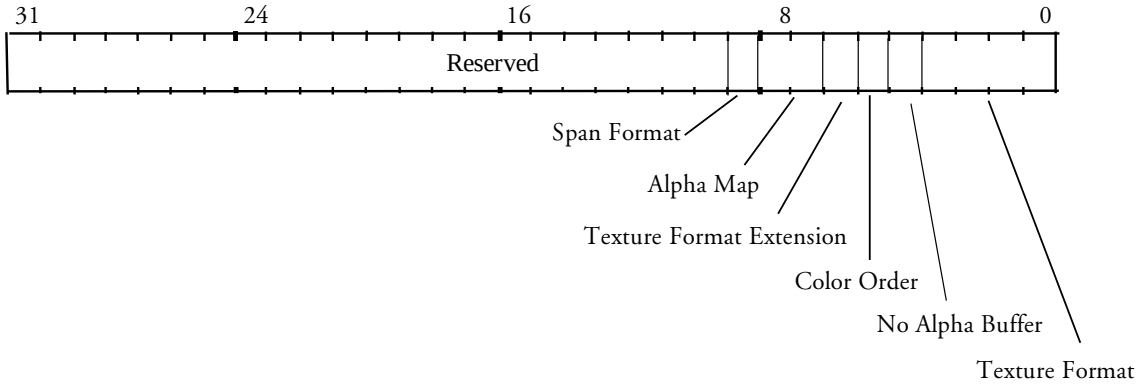


Figure 5.27 TextureDataFormat Register

5.8.6 Using the Texel LUT

The **TexelLUT0** to **15** registers contain the texture color look-up table. Each register contains 8 bit fields for red, green, blue and alpha color components. The **TexelLUTMode** register allows use of the **TexelLUT0** to **15** registers. When enabled, the texel value becomes an index into this look-up table.

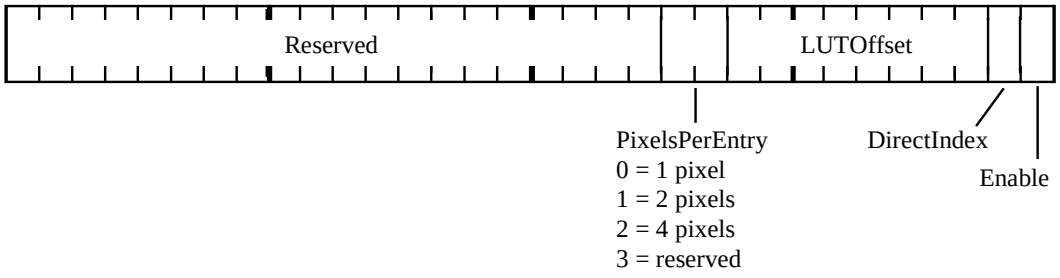


Figure 5.28 TexelLUTMode Register

The LUT must be enabled before a look-up will be done. The other fields of this register are used to control use of the LUT for 2D operations. Enabling DirectIndex causes the LUT to be indexed by the address of the fragment, not by data read from memory. If block fills are used the LUT is indexed at the start of every scanline based on the lower 3 bits of the Y value (X is ignored), the LUTOffset which is added to the index, and the PixelsPerEntry field; two consecutive entries in the LUT are used to fill the upper and lower halves of the 64 bit block color register.

If block fills are not used the lower 3 bits of the X and Y values of each fragment are used to index the LUT; the PixelsPerEntry field scales the X and Y values so that an 8 pixel by 8 pixel pattern is supported, and the LUTOffset field is added to the index before it is used.

If the LUT is used for 2D operations, the texture application unit should be enabled and set to copy mode so that the texture color generated by the look-up table is converted to a color that can be plotted on the screen.

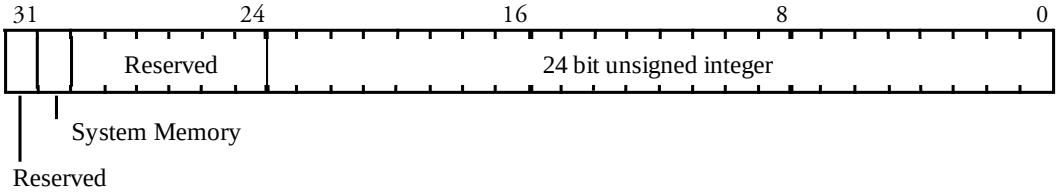


Figure 5.29 TexelLUTAddress register

If all 256 entries in the LUT need to be filled, the **TexelLUTData** and **TexelLUTOffset** registers should be used. The offset into the LUT for the first entry to be loaded should be written to the **TexelLUTOffset** register, then a succession of LUT entries written to the **TexelLUTData register**. The offset into the LUT will be automatically incremented after each entry is written.

It is also possible to load the LUT directly from memory. This is initiated by loading the **TexelLUTAddress** register with the address of the LUT in memory (in 32 bit units) and the **TexelLUTTransfer** register:

The Index field specifies the first entry in the LUT to load, while the Count field specifies the number of entries to load. Bit 30 of the **TexelLUTAddress** register specifies that the LUT is resident in system memory and should be read across the PCI bus.

The **TextureLUTAddress** register may be loaded indirectly by the **TexelLUTID** register. This operates in an identical manner to the **TexelID** register. There may be some latency between the register value being written to PERMEDIA and the interrupt being asserted, and it is possible that both registers will have been loaded before the interrupt is received. To determine which register caused the interrupt, they may be read back and will hold the value read from memory.

To read back the LUT entries, first read from the **TexelLUTOffset** register which resets the read back index to zero, then from the **TexelLUTData** register as many times as necessary.

5.8.7 Block Fill Textures

If texture mapping is enabled (and DirectIndex disabled) when a block fill is done the mask for the block fill is read from memory as a texture map. The texture address unit must be set appropriately so that the S value increments or decrements by one for each block of 32 pixels while T stays at zero. The texture address calculated is used to index a texture map and data returned is used as a mask to control which pixels are plotted during a block fill. This feature might be used to draw text for which the font has been previously loaded into a font cache in memory.

The layout of the data in memory should be byte aligned, so if the character is up to 8 pixels wide specify a texel size of 8 bits, up to 16 use 16, up to 24 use 24, and up to 32 use 32. If the character is wider than 32 pixels change to a word aligned bitmask and keep the pixel size to 32 bits. To match the normal data format for fonts, set the SpanFormat field in the **TextureDataFormat** register which allows the data to be stored with the bits in each byte mirrored.

5.8.8 Alpha Mapping

Alpha mapping performs a color key test before bilinear filtering, and prevents any of the red, green, or blue, components of a rejected pixel taking part in the filtering. The alpha channel is treated differently, and if a pixel fails the color test its alpha value is set to zero, but if it passes it is left at the original value. The alpha channel of all pixels, whether rejected or accepted, are filtered. This results in an alpha value of zero where all contributing pixels are rejected, an alpha value of one where all contributing pixels are accepted, and a varying alpha value where some are rejected and some accepted. As the magnification factor of the bilinear zoom is

increased the variable alpha is spread across more destination pixels. The range of alpha values rejected by the chroma key test in the YUV unit can be adjusted to allow fine control over the exact size of the cut-out. If blending is enabled then the varying alpha values smooth the transition of the edge of the sprite to the background.

The registers **AlphaMapUpperBound** and **AlphaMapLowerBound** are used to control the range over which the test is done. The test is enabled by the **TextureDataFormat** register.

5.8.9 Texture Download Example

```
fbReadMode.PatchMode = PERMEDIA_TRUE
fbReadMode.SubPatchMode = PERMEDIA_SUBPATCH
FBReadMode(fbReadMode);

fbWriteMode.Enable = PERMEDIA_TRUE
FBWriteMode(fbWriteMode)

// Set format to 8 bits

ditherMode.UnitEnable = PERMEDIA_TRUE
ditherMode.Enable = PERMEDIA_FALSE
ditherMode.ColorMode = PERMEDIA_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)

// Do image download
```

5.8.10 Texture Mapping Example

Texture map a trapezoid:

```
textureAddressMode.Enable = PERMEDIA_TRUE
textureAddressMode.PerspectiveCorrection = PERMEDIA_TRUE
TextureAddressMode(textureAddressMode)

// Load texture address parameters

SStart()
dSdx()
dSdyDom()
TStart()
dTdx()
dTdyDom()
QStart()
dQdx()
dQdyDom()

// Configure texture read

textureReadMode.Enable = PERMEDIA_TRUE
textureReadMode.SWrapMode = PERMEDIA_TEXTURE_WRAP_REPEAT
textureReadMode.TWrapMode = PERMEDIA_TEXTURE_WRAP_REPEAT
textureReadMode.Width = width
textureReadMode.Height = height
textureReadMode.FilterMode = PERMEDIA_FALSE
TextureReadMode(textureReadMode)

textureMapFormat.PP0 = partialProduct0
textureMapFormat.PP1 = partialProduct1
textureMapFormat.PP2 = partialProduct2
textureMapFormat.SubPatchMode = PERMEDIA_TRUE
textureMapFormat.TexelSize = PERMEDIA_8_BITS_PER_TEXEL
TextureMapFormat(textureMapFormat)


textureDataFormat.TextureFormat = PERMEDIA_COLOR_FORMAT_RGB_332
TextureDataFormat(textureDataFormat)

// Enable texture/fog/blend unit, load other parameters and
// render
```

5.9 YUV Unit

The YUV unit converts from YUV color format, also known as YCbCr, to RGB. It also does chroma key testing. This test may be done either before or after the conversion.

The YUV conversion is done on data that is being loaded into the **Texel0** register. The data for this may come from the TextureRead unit or from the host, so YUV conversion can be done either during texture download or on a texture as it is applied to a primitive. The YUV data can be in either 444 format or 422 format. The chroma test may be done with either YUV or RGB data.

5.9.1 Chroma Test

The chroma test specifies upper and lower bounds against which the **Texel0** value is tested. The test may be set to pass if the components of **Texel0** are either all inside or all outside the bounds. This is controlled by the accept/reject TestMode options of the **YUVMode** register. If the test passes, the **Texel0** data may be used in the Texture/Fog/Blend unit as normal. If the test fails, then the fragment to which the texture data maps, may be rejected (not plotted). This is useful for cut-outs and sprites. Alternatively, on test failure, the **Texel0** value may be rejected and the texture operation on the fragment suppressed. This is achieved by setting the RejectTexel bit in the **YUVMode** register. In this case the underlying color provided by PERMEDIA is used without being modified by the texture color. This is useful for applying a logo to a shaded polygon where the underlying color is provided by the Color DDA unit.

The test modes available are:

Mode	Test Mode
0	No test
1	Accept
2	Reject

Table 5.17 Chroma Test Modes

Chroma key testing can be done without texture mapping by setting the TexelDisableUpdate field in the **YUVMode** register. This allows fragment rejection during a copy operation. If chroma testing is required against the destination color of a copy (i.e. only overwrite pixels of the specified color), then the destination region of the screen is used as the texture map and the framebuffer units are set-up to do a normal copy. The texels are read in and tested. Fragments are rejected if the colors do not match. The copy operation for that pixel will not take place if the fragment has been rejected. Setting the TexelDisableUpdate bit discards the texel as soon as the test has been done which improves performance.

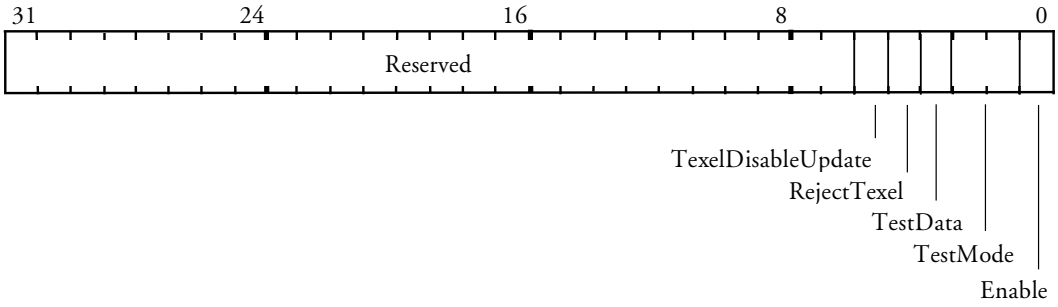


Figure 5.30 YUVMode Register

The TestData bit controls when the chroma test occurs in relation to the color conversion. Setting this bit causes the chroma test to occur on the output of the unit; clearing it causes the chroma test to occur on the input i.e. after or before color conversion respectively, assuming the Enable bit is set.

The TestMode can be set to:

Accept, i.e. pass test if (upper bound <= color >= lower bound

Reject, i.e. fail test if (upper bound <= color >= lower bound

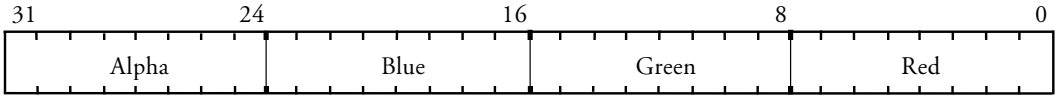


Figure 5.31 ChromaUpperBound and ChromaLowerBound Registers RGB Format

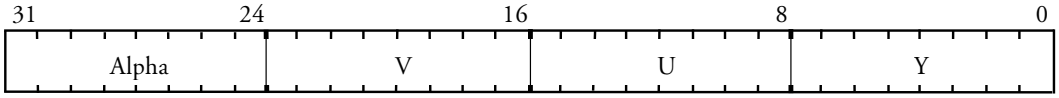


Figure 5.32 ChromaUpperBound and ChromaLowerBound Registers YUV Format

5.10 Framebuffer Read and Write Units

Before drawing can take place, PERMEDIA must be configured to perform the correct framebuffer read and write operations. Framebuffer read modes affect the operation of alpha blending, logic ops, software writemasks, image upload and image copy operations. Framebuffer write modes are relevant to all drawing in the framebuffer.

5.10.1 Framebuffer Read

The **FBReadMode** register allows PERMEDIA to be configured to make 0, 1 or 2 reads of the framebuffer. The following are the most common modes of access to the framebuffer:

- Rendering operations with no logical operations, software writemasking or alpha blending. In this case no read of the framebuffer is required and framebuffer writes should be enabled. Framebuffer reads should be disabled for maximum efficiency.
- Rendering operations which use logical ops, software writemasks or alpha blending. In these cases the destination pixel must be read from the framebuffer and framebuffer writes must be enabled.
- Image copy operations. Here set-up depends on whether logical ops, software writemasks and/or alpha blending are occurring with the copy. If any of these are, the framebuffer needs two reads, one for the source and one for the destination. Otherwise, only one read is required.
- Image upload. This requires reading of the destination framebuffer pixels to be enabled and framebuffer writes to be disabled.
- Image download. This case requires no framebuffer reads (as long as software writemasking, alpha blending and logic ops are disabled) but writes must be enabled.

Note: Avoiding unnecessary additional reads will enhance performance.

For both the read and the write operations, an offset is added to the calculated address. The source offset (**FBSourceOffset**) is used for copy operations. The pixel offset (**FBPixelOffset**) can be used to allow multi-buffer updates[1]. The offsets

[1] The OpenGL specification, for example, allows any combination of the Front, Back, Left and Right color buffers to be updated 'simultaneously'. In this case a scene would be rendered multiple times changing the FBPixelOffset as appropriate. When using this mode it is important to ensure that the buffers which affect the rendering are updated only once. For example, when rendering with depth buffering enabled, localbuffer writes should only be enabled for the last buffer updated.

should be set to zero for normal rendering. The address calculation implements the following equations:

Bottom left origin

Destination address = FBWindowBase - Y * W + X + FBPixelOffset

Source address = FBWindowBase - Y * W + X + FBPixelOffset + FBSourceOffset

Top left origin

Destination address = FBWindowBase + Y * W + X + FBPixelOffset

Source address = FBWindowBase + Y * W + X + FBPixelOffset + FBSourceOffset

where:

Destination Address	is the address in the framebuffer which is written to if writes are enabled, and is also the address read when ReadDestination is enabled.
Source Address	is the address in the framebuffer which is read from when ReadSource is enabled.
X	is the pixel's X coordinate,
Y	is the pixel's Y coordinate,
FBWindowBase	holds the base address in the framebuffer of the current window.
FBPixelOffset	is normally zero except when multi-buffer writes are needed when it gives a way to access pixels in alternative buffers without changing the **FBWindowBase** register. This is useful as the window system may be asynchronously changing the window's position on the screen. It is held in the **FBPixelOffset** register.
FBSourceOffset	is normally zero except during a copy operation where data is read from one address and written to another address. The FBSourceOffset is held in the **FBSourceOffset** register and is the offset from destination to source.
W	is the screen width. Only a subset of widths are supported and these are encoded into the PP0, PP1 and PP2 fields in the **FBReadMode** register. See the table in Appendix C for more details.

The calculation of FBSourceOffset can be avoided by using the **FBSourceDelta** and **FBSourceBase** registers. For screen to screen copies FBSourceBase should be

set to the same value as **FBWindowBase** (this is done automatically whenever FBWindowBase is loaded) and FBSourceDelta should hold the distance from the destination area to the source area in X and Y. If the copy is from an offscreen bitmap, FBSourceBase should hold the base address of the bitmap, and FBSourceDelta should hold the offset in X and Y into the bitmap to where the source area begins.

The data read from the framebuffer may be either FBDefault (data which may be written back into the framebuffer or used in some manner to modify the fragment color) or **FBColor** (data which will be uploaded to the host). The table below summarizes the framebuffer read/write control for common rendering operations:

ReadSource	ReadDestination	Writes	Read Data Type	Rendering Operation
Disabled	Disabled	Enabled	-	Rendering with no logical operations, software writemasks or alpha blending.
Disabled	Disabled	Enabled	-	Image download.
Disabled	Enabled	Disabled	**FBColor**	Image upload.
Enabled	Disabled	Enabled	FBDefault	Image copy with hardware writemasks.
Disabled	Enabled	Enabled	FBDefault	Rendering using destination-only logical operations, software writemasks or alpha blending.
Enabled	Enabled	Enabled	FBDefault	Image copy with logical operations, software writemasks or alpha blending.

Table 5.18 Framebuffer Read/Write Modes

Incorrect data can be read if reads are enabled but the same data had just been written with reads disabled. To avoid this problem, a **WaitForCompletion** command should be sent after enabling reads, but prior to the next primitive.

5.10.2 Framebuffer Write

Framebuffer writes must be enabled to allow the framebuffer to be updated. A single 1 bit flag controls this operation.

The Framebuffer Write unit is also used to control the operation of fast block fills, if supported by the framebuffer. Fast fill rendering is enabled via the FastFillEnable bit in the **Render** command register. The block color is 64 bits wide; normally the same values are used in the upper and lower halves of the register so they are both set with one register, FBBlockColor. If different data is required in both halves of the register, use the FBBlockColorUpper and FBBlockColorLower registers. The data put in the color registers should be of the raw framebuffer format. When using the framebuffer in 8 bit packed mode, the data should be repeated in each byte. When using the framebuffer in packed 16 bit mode, the data should be repeated in the top 16 bits.

Note that due to restrictions in the way that the memory devices implement block fills, a packed 24 bit RGB framestore may only use block fills for colors that have all bytes in the pixel set to the same value.

When uploading images the UpLoadData bit can be set to allow color formatting. See sections §5.12.6 for more details.

5.10.3 Patching

Data in the framebuffer can use patched addressing to improve performance under certain circumstances. However, only non-visible data is normally patched. Patch mode organizes data for efficient drawing of scanline primitives; it also helps line drawing. This form is typically used in the localbuffer, see §5.5.4, for patching the depth buffer. The SubPatch mode re-organizes data for efficient texture operations; see section §5.8.5. SubPatchPack mode is used when 4 bit textures are loaded as 8 bits i.e. the subpatch packing takes into account the 2 texels per byte.

5.10.4 Packed Copies

Packed copies move 32 bits at a time even though the real pixel size may be 8, 16, or 24 bits. The **PackedDataLimits** register holds the left and right X coordinates for the destination area of the screen in the native pixel format. Any pixels outside this area are not plotted. The relative offset field in the **FBReadMode** register specifies the number of pixels that the source data has to be adjusted to align with the destination data. The relative offset field is also available in the **PackedDataLimits** register, the value from the last register loaded takes effect.

5.10.5 Image Downloads

An image download can be performed in one of four ways. It can be achieved by loading the data in standard color format into the **Color** register and using the Color Format unit to organize it into the framestore format. Or it can be achieved by loading the data in raw framebuffer format either into the **Color** register or the **FBData** register. The former requires that the Color Format unit is disabled whilst the latter ignores this unit. Alternatively, the data can be loaded as some other raw format into the **FBSourceData** register and have the Texture/Fog/Blend unit convert it into the internal color format. The Color Format unit can then convert it into the arrangement to be stored in the framebuffer. Both techniques require setting up the Rasterizer appropriately.

5.10.6 Fast Texture Download

Normal texture download is done as an image download. This involves setting up the Rasterizer to draw a rectangle and changing the state of a number of units. This is a good way to load the texture if any processing needs to be done on it, such as color format conversion, color space conversion or patching.

If the texture is held on the host in the raw framebuffer format, the fast texture download approach can be used. The **TextureDownloadOffset** register holds the base address of the framebuffer using 32 bit pixel addressing. The **TextureData** register holds the texture data in raw framebuffer format 32 bits at a time. The load of this register is ignored by all other units in the pipeline so no state needs to be saved and restored. Following the receipt of each **TextureData** value, the **TextureDownloadOffset** value is incremented. If this register is read, it returns the current count, not the original value.

If fast download is used, the texture map on the host must be in exactly the format it will be stored in memory, including any color formatting, byte swapping, or address patching. If a texture will be loaded several times, it can be downloaded as an image the first time using all formatting controls, and then uploaded again as a raw image for later use.

Using this technique, framebuffer writes do not need to be enabled.

5.10.7 Hardware Writemasks

Hardware writemasks, if available, are controlled using the **FBHardwareWriteMask** register. If the framebuffer memory devices support hardware writemasks, and they are to be used, then software writemasking should be disabled (by setting all the bits in the **FBSoftwareWriteMask** register). This will result in fewer framebuffer reads when no logical operations or alpha blending is needed.

If the framebuffer is used in 8 bit packed mode, then an 8 bit hardware writemask must be repeated in all 4 bytes of the **FBHardwareWriteMask** register. If the framebuffer is in 16 bit packed mode then the 16 bit hardware writemask must be repeated in both halves of the **FBHardwareWriteMask** register.

As there is no overall enable for this feature, the hardware writemask MUST be set to all 1's, except when hardware writemasking is explicitly required.

5.10.8 Frame Blank Synchronization

The **SuspendUntilFrameBlank** command register may be used to stall the PERMEDIA pipeline until the next frameblank. For double buffering, it is beneficial to synchronize to the monitor blanking. By using this register, full screen double buffering can be controlled through the pipeline and the host does not need to wait for vertical frame blank itself. Instead, once the **SuspendUntilFrameBlank** command register has been loaded, the host can continue to load PERMEDIA registers and issue commands. PERMEDIA will continue processing these as long as they do not involve writing to the framebuffer. The data field of this register is the base address of the buffer to be displayed and is passed to the Internal Video Timing generator.

5.10.9 Registers

The **FBReadMode** register layout is as follows:

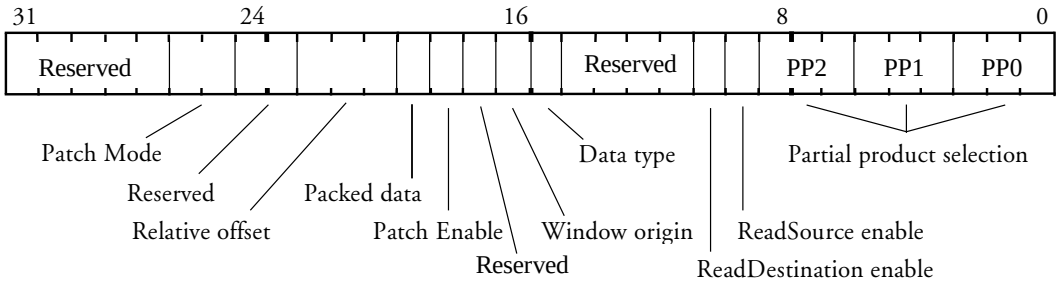


Figure 5.33 FBReadMode Register

See Appendix C for more information on setting partial product codes.

FBWindowBase holds the base address of the window in the framebuffer in 24 bit unsigned format. The **FBPixelOffset** and **FBSourceOffset** registers hold 24 bit 2's complement offsets used in copy operations and multi-buffer updates, as described above.

The **FBWriteMode** controls the framebuffer write operations:

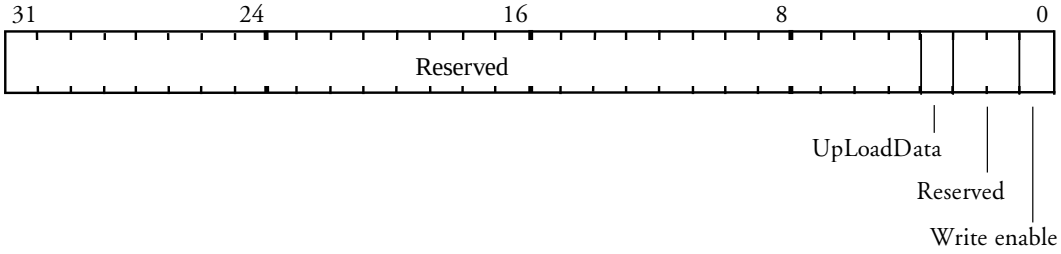


Figure 5.34 FBWriteMode Register

The **FBReadPixel** sets the pixel size.

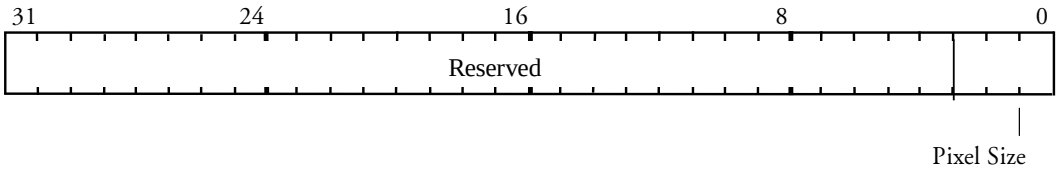


Figure 5.35 FBReadPixel Register

The **PackedDataLimits** register is used to control packed copies.

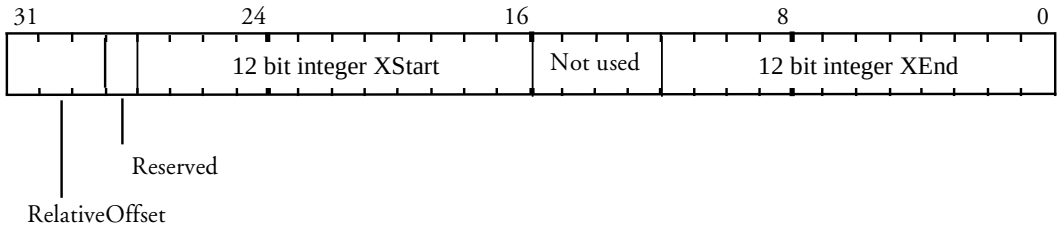


Figure 5.36 PackedDataLimits Register

FBHardwareWriteMask is a 32bit register where each bit acts as a mask. **FBColor** is a read-only register which returns the data to the host during image upload operations.

5.10.10 Image Copy Example

This example copies a rectangular region of the framebuffer, without moving any data in the localbuffer. The region extends from the origin (0,0) to (100,100) and will be shifted right by 200 pixels. The destination rectangle is scan converted.

```
// First set-up the framebuffer read mode
fbReadMode.ReadSource = PERMEDIA_ENABLE
fbReadMode.ReadDestination = PERMEDIA_DISABLE
fbReadMode.DataType = PERMEDIA_FBDEFAULT

FBReadMode(fbReadMode)        // Update register

// Now enable framebuffer writefbWriteMode.WriteEnable =
PERMEDIA_ENABLE
FBWriteMode(fbWriteMode)      // Update register

// Offsets. No Pixel offset, source offset of 200
FBPixelOffset (0x0)
FBSourceOffset (-200)

// All the tests which could remove the fragment must be
// disabled (Stipple, Stencil, Depth) except
// the Scissor test which is still needed for screen
// and possibly window clipping.

// If software writemasks are to be used then they are
// set appropriately, and the framebuffer set-up to do
// extra read operation

// Disable the Color DDA unit, we do not want to
// associate a color with this fragment.
colorDDAMode.UnitEnable = PERMEDIA_FALSE
ColorDDAMode(colorDDAMode)

// Define the region we wish to copy from.
StartXDom (200<<16)
StartXSub (300<<16)
dXSub (0)
dXDom (0)
```

```
StartY (0)
dY (1<<16)
Count (100)

render.PrimitiveType = PERMEDIA_TRAPEZOID

Render (render)        // Start the rasterization
```

5.11 Color DDA Unit

The Color DDA unit is used to associate a color with a fragment produced by the Rasterizer. This unit should be enabled for rendering operations and disabled for pixel rectangle operations (i.e. copies, uploads and downloads).

5.11.1 RGBA and Color-Index(CI) Modes

Two color modes are supported by PERMEDIA, true color RGBA and color index (CI).

PERMEDIA's internal color representation is RGBA with 8 bits per component:

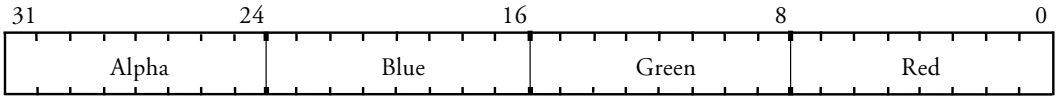


Figure 5.37 Color Representation

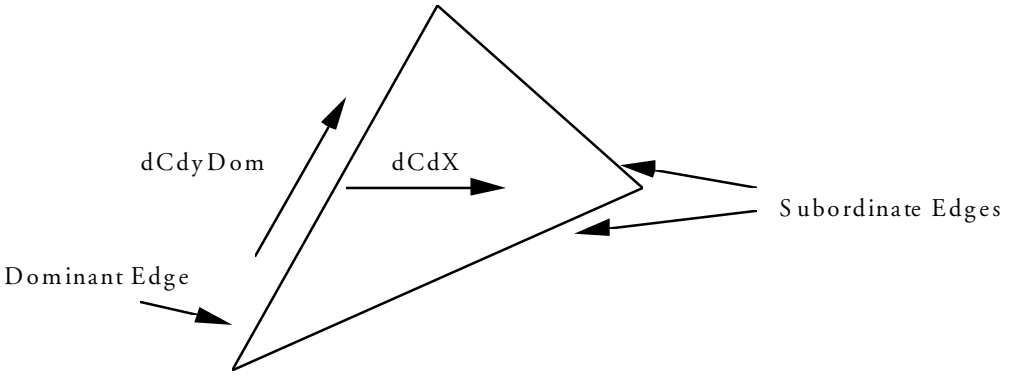
This format is the same for all the different framebuffer configurations supported. If the number of bits in the framebuffer for a color component is less than 8 then the color value is left shifted into the most significant bits of that components field. The unused least significant bits should be set to zero.

In CI mode, the color index is placed in the lower byte of the 32 bit register (i.e., the red component).

5.11.2 Gouraud Shading

When in Gouraud shading mode, the Color DDA unit performs linear interpolation given a set of start and increment values. Clamping is used to ensure that the interpolated value does not underflow or overflow the permitted color range.

For a Gouraud shaded trapezoid, PERMEDIA interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required per color component, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in the diagram below, where C represents a color component (red, green, blue or color index). Alpha is not interpolated and stays at its initial value.



CStart = the initial color value
dCdyDom = color gradient in the Y direction along the dominant edge
dCdx = color gradient in the X direction

Figure 5.38 Color Interpolation

See Appendix D4 Delta values for a Gouraud Shaded Triangle.

For Gouraud shaded lines, each line is treated as the dominant edge of a trapezoid, and so no dCdx increment is required.

To allow accurate interpolation, the increment values are specified in a 17bit fixed point format. The format is 2's complement with 1 bit sign, 5 bits integer and 11 bits fraction:

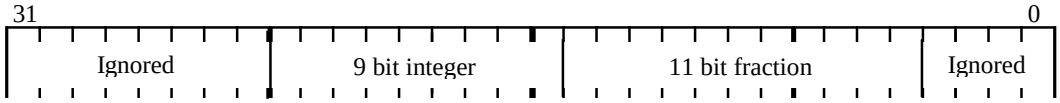


Figure 5.39 Fixed Point Color Format

Note that if you are rendering to multiple buffers and have initialized the start and increment values in the Color DDA unit, then any subsequent **Render** command will cause the start values to be reloaded.

If subpixel correction has been enabled for a primitive, then any correction required will be applied to the color components.

5.11.3 Flat Shading

In flat shading mode, a constant color is associated with each fragment. This color is loaded into the **ConstantColor** register which has the format shown in Fig. 5.36 above.

5.11.4 Registers

The main control register for the Color DDA unit is the **ColorDDAMode** register:

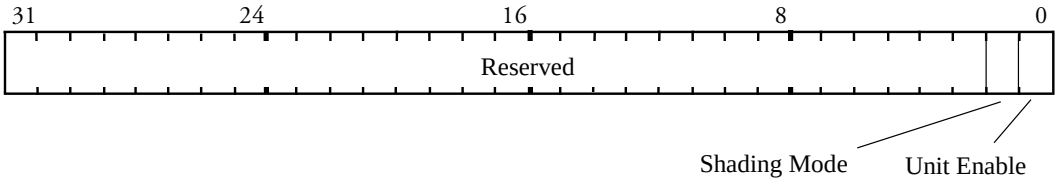


Figure 5.40 ColorDDAMode Register

The registers to set-up Gouraud shading in the Color DDA unit are:

Register	Data Field	Description
RStart	17 bit 2's comp fix pt	Red start value
dRdx	17 bit 2's comp fix pt	Red derivative per unit X
dRdyDom	17 bit 2's comp fix pt	Red derivative per unit Y, dominant edge
GStart	17 bit 2's comp fix pt	Green start value
dGdx	17 bit 2's comp fix pt	Green derivative per unit X
dGdyDom	17 bit 2's comp fix pt	Green derivative per unit Y, dominant edge
BStart	17 bit 2's comp fix pt	Blue start value
dBdx	17 bit 2's comp fix pt	Blue derivative per unit X
dBdyDom	17 bit 2's comp fix pt	Blue derivative per unit Y, dominant edge
AStart	17 bit 2's comp fix pt	Alpha start value

Table 5.19 Color Interpolation Registers

5.11.5 Flat Shading Example

A flat shaded primitive:

```
// Set DDA to flat shade mode
colorDDAMode.UnitEnable = PERMEDIA_ENABLE
colorDDAMode.Shade = PERMEDIA_FLAT_SHADE_MODE

ColorDDAMode(colorDDAMode)

ConstantColor(0xFFFFFFFF)     // Load the flat color
```

5.11.6 Gouraud Shaded Trapezoid Example

See Appendix D for details of how to calculate delta values.

```
// Enable unit in Gouraud shading mode
colorDDAMode.UnitEnable = PERMEDIA_ENABLE
colorDDAMode.Shade = PERMEDIA_GOURAUD_SHADE_MODE

ColorDDAMode(colorDDAMode)

// Load the color start values and deltas for dominant edge
// and the body of the trapezoid

RStart()        // Set-up the red component start value
dRdx()          // Set-up the red component increments
dRdyDom()
GStart()        // Set-up the green component start value
dGdx()          // Set-up the green component increments
dGdyDom()
BStart()        // Set-up the blue component start value
dBdx ()         // Set-up the blue component increments
dBdyDom ()
```

5.11.7 Gouraud Shaded Line Example

See Appendix D for details of how to calculate delta values.

```
// Set DDA for Gouraud shaded mode
colorDDAMode.UnitEnable = PERMEDIA_ENABLE
colorDDAMode.Shade = PERMEDIA_GOURAUD_SHADE_MODE
ColorDDAMode(colorDDAMode)

// For lines we need only start values and dominant edge
// deltas

RStart()        // Set-up the red component start value
dRdyDom()       // Set-up the red component increment
GStart()        // Set-up the green component start value
dGdyDom()       // Set-up the green component increment
BStart()        // Set-up the blue component start value
dBdyDom ()      // Set-up the blue component increment
```

5.12 Texture/Fog/Blend

The Texture/Fog/Blend unit applies effects to the interpolated color. The effects are applied in the order: texture then fog then blend.

5.12.1 Texture Application

There are two major types of texture application, one suitable for RGB applications and one suitable for Ramp applications; Ramp applications use RGB textures and framebuffer format but are limited to a white light source. The enable bit in the **TextureColorMode** register and the TextureEnable bit in the **Render** register must both be enabled before texture will be applied.

RGB Texture Application

This is referred to elsewhere as the OpenGL type of texture application. It can be done in one of three ways.

In copy mode, the texture color replaces the current fragment color.

In decal mode the texture color is blended with the fragment color using the texture alpha value:

$C_f = C_tA_t + C_f(1-A_t)$

$A_f = A_f$

where: C_f is the fragment color, C_t is the texture color , A_f fragment alpha and A_t is the texture alpha. If the texture alpha value is one, decal becomes the same as copy.

In modulate mode the color components are multiplied together:

$C_f = C_tC_f$

$A_f = A_tA_f$

where: C_f is the fragment color, C_t is the texture color , A_f fragment alpha and A_t is the texture alpha.

Ramp Texture Application

This is referred to elsewhere as the Apple type of texture application because of the approach adopted by QuickDraw3D. This type of texture application is done three stages, where each stage can be independently enabled or disabled. The first stage is decal, which does the operation:

$C_f = C_tA_t + C_f(1-A_t)$

$A_f = A_f$

If decal is not enabled then the following operation is done:

$C_f = C_t$

$A_f = A_tA_f$

The next operation is modulate, which does:

$C_f = K_dC_D$

$A_f = K_dA_D$

where: C_f is the fragment color, K_d is an interpolated parameter which represents the diffuse light intensity, A_t is the texture alpha, C_D is the color after the decal operation and A_D is the alpha value after the decal operation.

The next operation is highlight:

$C_f = C_M + K_s$

$$A_f = A_M + K_s$$

where: C_f is the fragment color, K_s is an interpolated parameter which represents the specular or highlight intensity, A_t is the texture alpha, C_M is the color after the modulate operation and A_M is the alpha value after the modulate operation.

5.12.2 Fog Application

The fog unit is used to combine the incoming fragment's color (generated by the Color DDA unit, and potentially modified by the texture unit) with a pre-defined fog color. Fogging can be used to simulate atmospheric fogging, and also to depth-cue images.

Fog application has two stages; derivation of the fog index for a fragment, and application of the fogging effect. The fog index is a value which is interpolated over the primitive using a DDA in the same way color and depth are interpolated. The fogging effect is applied to each fragment using the equation described below.

Note that although the fog values are linearly interpolated over a primitive the fog values at each vertex can be calculated on the host using a linear fog function (typically for simple fog effects and depth-cueing) or a more complex function to model atmospheric attenuation. This might be an exponential function.

A fog test is supported that will reject a fragment if its fog value is negative. This may be used if the background of the scene has been cleared to the fog color; any pixels that are far enough from the eye to be completely fogged need not be plotted.

The enable bit in the **FogMode** register and the FogEnable bit in the **Render** register must both be enabled before fog will be applied.

5.12.3 Fog Index Calculation - The Fog DDA

The fog DDA is used to interpolate the fog index (F) across a primitive. For a fogged trapezoid, PERMEDIA interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in the diagram below. The rendering direction chosen here is bottom to top.

FStart	= Start fog value
dFdyDom	= Increment along dominant edge.
dFdx	= Increment along the scan line.

The dFdx value is not required for fogged lines.

The mechanics are similar to those of the other DDA units, as the diagram below illustrates:

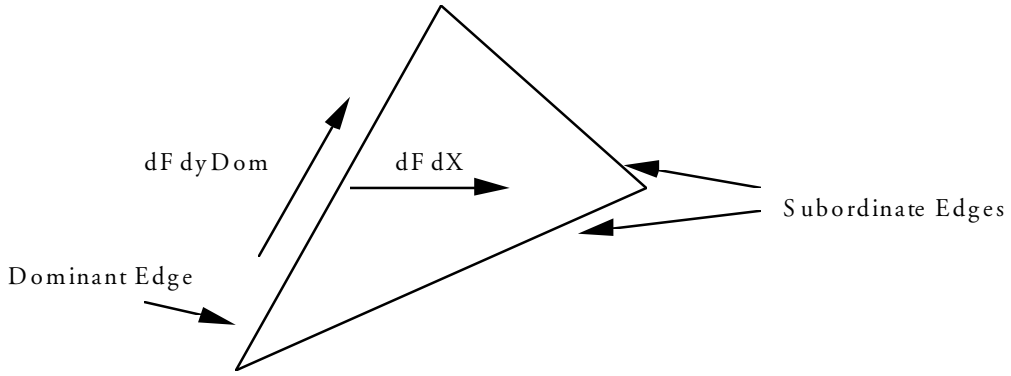


Figure 5.41 Fog Interpolation Over A Triangle

where:

FStart = initial fog value.
dFdx = Fog gradient in the X direction.
dFdyDom = Fog gradient along the dominant edge of a primitive.

Note that for fogged lines the dFdx delta is not required.

The fog index is specified as an 18bit fixed point value. The format is 2's complement with 2 bits integer and 16 bits fraction.

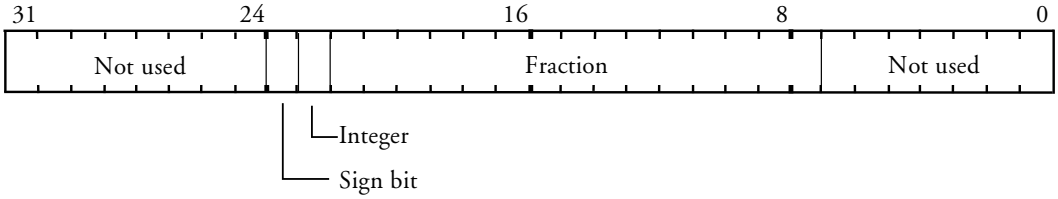


Figure 5.42 Fog Interpolant Fixed Point Format

The fog DDA calculates a fog index value which is clamped to lie in the range 0.0 to 1.0 before it is used in the fogging equations described below.

5.12.4 Fogging Equation

The fogging equation is:

$C = fC_i + (1-f)C_f$

where:

C = outgoing fragment color
C_f = fog color
C_i = incoming fragment color
f = fog index

The equation is applied to the color components, red, green and blue; alpha is not modified. The diagram below shows how the fogging would typically affect a scene. Initially no fogging occurs, f >=1.0, then a region of linear combination of

the fragment color and fog color occurs 1.0 < f > 0.0, followed by a region of constant fog color, f <= 0.0.

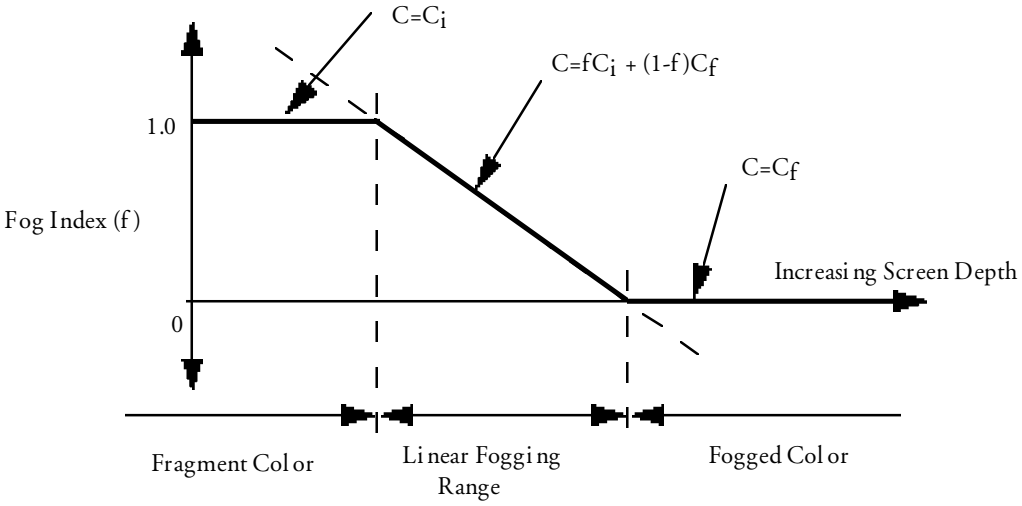


Figure 5.43 Fogging

5.12.5 Alpha Blending

The Alpha Blend Unit supports alpha blending or color formatting. Two types of alpha blending are supported, one that is common for RGB[1] and Ramp[2] applications, and one that is specific to Ramp applications. Alpha blending combines the fragment's color, potentially after texture and fog have been applied, with that stored in the framebuffer.

Data from the framebuffer is in the raw format so must be converted to the internal format before the blend can be done. This is achieved by setting the ColorFormat and ColorFormat Extension fields in the **AlphaBlendMode** register.

In some situations blending is desired when no retained alpha buffer is present. In this case the alpha value which is considered to be read from the framebuffer will be set to 1.0. The NoAlphaBuffer bit in the **AlphaBlendMode** register controls this.

Common Blend Mode

The common blend operation is defined as:

$$C_o = C_s A_s + C_d(1-A_s)$$

where: C_o is the output color, C_s is the source color , A_s is the source alpha and C_d is the destination color read from the framebuffer. Setting the Operation field to "Blend" in the **AlphaBlendMode** register will achieve this.

[1] RGB is also referred to as OpenGL mode.
[2] Ramp is also referred to as Apple mode.

See *The OpenGL Reference Manual* and *The OpenGL Programming Guide* from Addison-Wesley for more details of this style of alpha blending.

Ramp Blend Mode

The alternative blend mode is called PreMult and does the operation:

$$C_o = C_s + C_d(1-A_s)$$

For correct operation of Apple PreMult blending, the BlendType needs to be set to Ramp.

5.12.6 Image Formatting

The Alpha Blend and Color Format units can be used to format image data into any of the supported PERMEDIA framebuffer formats.

Consider the case where the framebuffer is in RGBA 5.5.5.1 mode, and an area of the screen is to be uploaded and stored in an 8 bit RGB 3:3:2 format. The sequence of operations is:

- Set the Rasterizer as appropriate see section §5.3.10
- Enable framebuffer reads
- Disable framebuffer writes and set the UpLoadData bit in the FBWriteMode register
- Enable the Alpha Blend unit, set the operation to "Format" (assuming no alpha blending is needed) and set the color mode to RGBA 5.5.5.1. This can all be achieved by setting the appropriate fields in the **AlphaBlendMode** register.
- Set the Color Format unit to format the color of incoming fragments to an 8 bit RGB 3:3:2 framebuffer format.

The upload now proceeds as normal. This technique can be used to upload data in any supported format.

The same technique can be used to download data which is in any supported framebuffer format. In this case the Rasterizer is set to synchronize with FBData (rather than Color), framebuffer writes are enabled and the UpLoadData bit cleared.

Normally internal color and alpha values require scaling if they are less than 8 bits. However there are situations where the least significant bits should be zeroed. This is needed for multi-pass rendering to prevent dithering occurring multiple times. This option can be independently applied to color and alpha values by setting the ColorConversion and/or AlphaConversion bits in the **AlphaBlendMode** register to Shift rather than Scale.

5.12.7 Registers

The **TextureColorMode** register is used to enable and disable texturing (qualified by the texture application bit in the **Render** command register). The KsDDA and

KdDDA bits enable the internal DDAs and should be set for modulate or highlight Ramp texture application modes. The Texture Type field differentiates between Ramp and RGB application modes. Combinations of decal, modulate and highlight are supported with Ramp Application Mode.

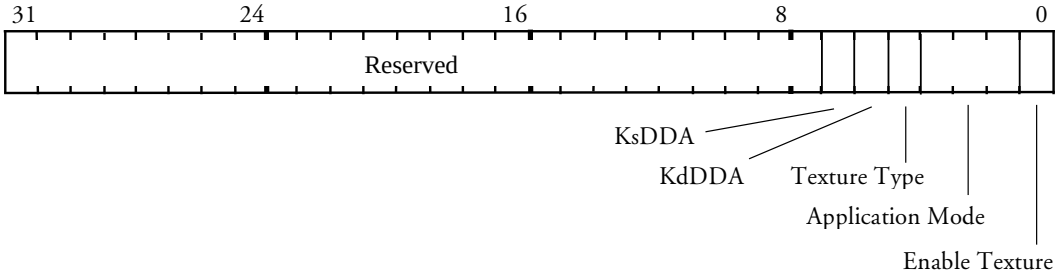


Figure 5.44 TextureColorMode Register

The **Texel0** register holds the texture value. This may be loaded automatically by the Texture Read unit, or supplied from the host for a procedural texture. Fig 5.44 and 5.45 show texture values in RGB and YUV formats respectively. This register is also used to hold the background color for the bitmask and stipple tests. If the tests fail then this color can be used in place of that from the Color DDA unit.

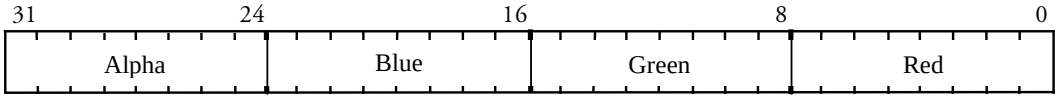


Figure 5.45 Texel0 Register - RGB and YUV formats

The six registers: **KsStart**, **dKsdx**, **dKsdyDom**, **KdStart**, **dKddx** and **dKddyDom** hold the start, dx and dyDom parameters for Ks and Kd. The format is 2's complement 2.16 fixed point format (1 bit sign, 1 bit integer, 16 bits fraction) with an effective range of ±1.999. The values of Ks and Kd at each vertex are used to calculate the gradient values in much the same way as the Z gradients, when interpolating depth see Appendix D.

The **FogMode** register is used to enable and disable fogging (qualified by the fog application bit in the **Render** command register). Setting Fog Test causes fragments with negative fog values to be rejected see section §5.12.2.

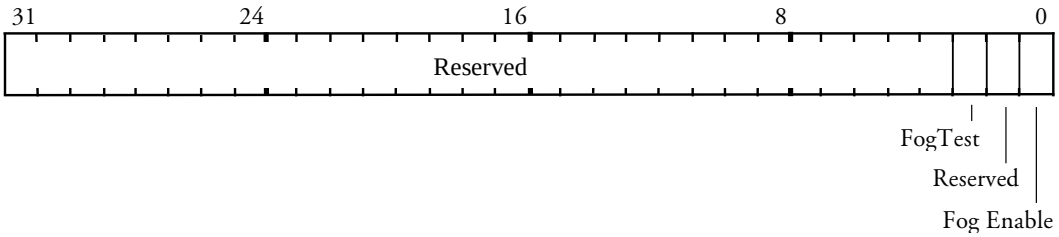


Figure 5.46 FogMode Register

Additional fog registers are, **FogColor**, which holds the fog color in the standard color format. **FStart**, **dFdx** & **dFdyDom** which control the fog DDA and are formatted in 2's complement 2.16 fixed point format as described above.

Blending is controlled by the **AlphaBlendMode** register:

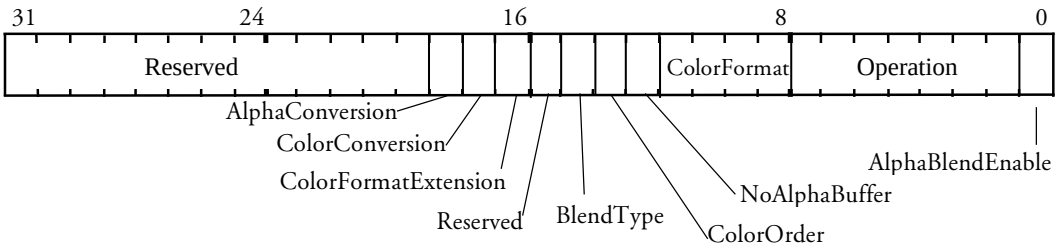


Figure 5.47 AlphaBlendMode Register

The color format and order is needed as the destination color is read from the framebuffer and needs to be converted into the internal PERMEDIA representation, it should therefore be set as appropriate for the framebuffer. The operation can be either format or blend or PreMult.

5.12.8 Texture Application Example

Example of texture mapped trapezoid:

```
// Set-up Texture/Fog/Blend unit

textureColorMode.Enable = PERMEDIA_TRUE
textureColorMode.ApplicationMode = PERMEDIA_TEXTURE_MODULATE
TextureColorMode(textureColorMode)

// Render with texture enabled in render command
// render.TextureEnable = PERMEDIA_TRUE
```

5.12.9 Fog Example

A Gouraud shaded, fogged RGBA trapezoid, with the fog color set to white. See Appendix D for details of how to calculate depth delta values - fog values are calculated in a similar way.

```
// Enable the Color DDA unit in Gouraud shading mode
colorDDAMode.UnitEnable = PERMEDIA_ENABLE
colorDDAMode.Shade = PERMEDIA_GOURAUD_SHADE_MODE

ColorDDAMode(colorDDAMode)

// Enable the Fog unit
fogMode.FogEnable = PERMEDIA_TRUE

FogMode(fogMode)

// Set the fog color to white
FogColor(0xFFFFFFFF)

// Load the color start values and deltas for dominant edge
// and the body of the trapezoid

RStart()        // Set-up the red component start value
dRdx()          // Set-up the red component increments
dRdyDom()
GStart()        // Set-up the green component start value
dGdx()          // Set-up the green component increments
dGdyDom()
BStart()        // Set-up the blue component start value
dBdx ()         // Set-up the blue component increments
dBYDom()

// Load the start value and delta for dominant edge
// and the body of the trapezoid
// Note that the fog deltas are calculated in the same
// way as the color deltas

FStart()        // Set-up the fog component start value
dFdx()          // Set-up the fog component increments
dFdyDom()


// When issuing a Render command the FogEnable bit
// should be set in addition to the fog unit being
// enabled:
// render.FogEnable = PERMEDIA_TRUE
```

5.13 Color Format Unit

The Color Format unit converts from PERMEDIA's internal color representation to a format suitable to be written into the framebuffer. This process may optionally include dithering of the color values. If the unit is disabled then the color is not modified in any way.

5.13.1 Color Formats

The framebuffer may be configured to be RGBA or Color Index (CI). Table 4.1 shows the full list of color modes supported by PERMEDIA. The R, G, B and A columns show the width of each color component. The least significant bit position is 0. For the Front and Back Modes the value is repeated in both buffers, and writemasks may be used to update only one buffer. In CI mode, the index is repeated in all streams.

5.13.2 Color Dithering

PERMEDIA uses an ordered dither algorithm to implement color dithering. It also has a line dither mode which uses a different algorithm which will generally give better results for lines because it is independent of orientation. This mode is not available for trapezoids.

If the Color Format unit is disabled, the color components RGBA are not modified and will be truncated when placed in the framebuffer. In CI mode, the value is truncated to the nearest integer. In both cases the result is clamped to a maximum value to prevent overflow.

PERMEDIA supports 8888 RGBA format for 2d operations only. If this mode is selected and dithering is enabled, it will result in 5551RGBA quality for each 32 bit pixel. This can be used when the window manager needs to be set-up for true color at the same time as 3D windows are required.

In some situations only screen coordinates are available, but window relative dithering is required. This can be resolved by setting up the optional X and Y offsets which get added to the coordinates before the dither tables are indexed. Each offset is a two bit number which is supplied for each coordinate. The XOffset and YOffset fields in the **DitherMode** register control this operation and should be set to zero if window relative coordinates are used.

5.13.3 ForceAlpha

The Color Format unit can force the alpha value to be either 0x0 or the maximum 0xFF, or leave it unchanged. This can be used to implement overlays. See section §7.6 for a detailed description.

5.13.4 Registers

One register controls the operation of this unit, **DitherMode**, and its layout is:

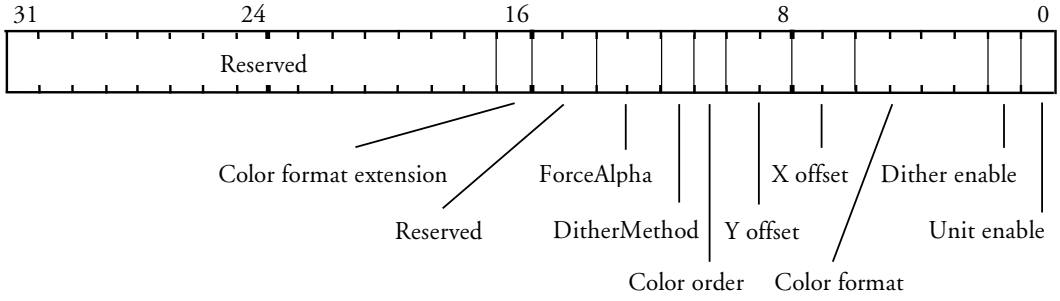


Figure 5.48 Dither Mode Register

The X and Y offset fields are for window relative dithering. Color order species RGB or BGR color order. The Color format and Color format extension fields control color depth and options are given in table 4.1.

5.13.5 Dither Example

To set the framebuffer format to RGB 3:3:2 and enable dithering:

```
// 332 Dithering

ditherMode.UnitEnable = PERMEDIA_TRUE
ditherMode.DitherEnable = PERMEDIA_TRUE
ditherMode.ColorMode = PERMEDIA_COLOR_FORMAT_RGB_332

DitherMode (ditherMode)// Load register
```

5.13.6 Color Format Example

To set the framebuffer format to RGB 3:3:2 and disable dithering:

```
// 332 No Dither

ditherMode.UnitEnable = PERMEDIA_TRUE
ditherMode.DitherEnable = PERMEDIA_FALSE
ditherMode.ColorMode = PERMEDIA_COLOR_FORMAT_RGB_332
DitherMode(ditherMode) // Load register
```

5.13.7 Color Format Example

To set the framebuffer to RGBA 8:8:8:8 and not dithered:

```
// 8888 Dithered (No effect as 8 bit components are
// not dithered)

ditherMode.UnitEnable = PERMEDIA_TRUE
ditherMode.DitherEnable = PERMEDIA_FALSE
ditherMode.ColorMode = PERMEDIA_COLOR_FORMAT_RGBA_8888

DitherMode(ditherMode) // Load register
```

5.14 Logical Op Unit

The Logical Op unit performs three functions:

- logic operations between the fragment color (source color) and a value from the framebuffer (destination color)
- software writemasking
- optional control of a special PERMEDIA mode which allows flat shading rendering.

5.14.1 Logical Operations

The logical operations supported by PERMEDIA are:

Mode	Name	Operation
0	Clear	0
1	And	S & D
2	And Reverse	S & ~D
3	Copy	S
4	And Inverted	~S & D
5	No-op	D
6	Xor	S ^ D
7	Or	S | D
8	Nor	~(S | D)
9	Equivalent	~(S ^ D)
10	Invert	~D
11	Or Reverse	S | ~D
12	Copy Invert	~S
13	Or Invert	~S | D
14	Nand	~(S & D)
15	Set	1

Where: S = Source (fragment) Color, D = Destination (framebuffer) Color

Table 5.20 Logical Operations

For correct operation of this unit in a mode which takes the destination color, PERMEDIA must be configured to allow reads from the framebuffer using the **FBReadMode** register. See section §5.10 for more details.

PERMEDIA makes no distinction between RGBA and CI modes when performing logical operations. However, logical operations are generally only used in CI mode.

5.14.2 Software Writemasks

Software writemasking is normally only implemented when Hardware writemasking is unavailable. It is controlled by the **FBSoftwareWriteMask**

register. The data field has one bit per framebuffer bit which when set, allows the corresponding framebuffer bit to be updated. When reset, it protects the bit from being written. Software writemasking is applied to all fragments and is not controlled by an enable/disable bit. However it may effectively be disabled by setting the mask to all 1's. If the mask is not all 1's, the ReadDestination bit must be enabled in the **FBReadMode** register to correctly use software writemasks. See the Framebuffer Read/Write section for details of how to enable/disable framebuffer reads.

The software writemask MUST be set to all 1's, except when software writemasking is explicitly required.

5.14.3 Flat Shaded Rendering

A special PERMEDIA rendering mode is available which allows rendering of unshaded images.

Note: This method is no longer recommended on PERMEDIA 2. Other methods of flat shading are as least as fast and are simpler to set-up correctly. It has been included here for the benefit of understanding legacy PERMEDIA 1 software.

- Flat shaded primitive
- No dithering required
- No logical ops
- No stencil or depth testing required
- No alpha blending

The following are available:

- Bit masking in the Rasterizer
- Area and line stippling
- User and Screen Scissor test

If all the conditions are met then rendering can be achieved by setting the **FBWriteData** register to hold the framebuffer data (in raw framebuffer format) and setting the UseConstantFBWriteData bit in the **LogicalOpMode** register. All unused units should be disabled.

This mode is most useful for 2D applications or for clearing the framebuffer when the memory does not support block writes. Note that FBWriteData register should be considered volatile when context switching.

5.14.4 Registers

The operation of the unit is controlled by the **LogicalOpMode** register:

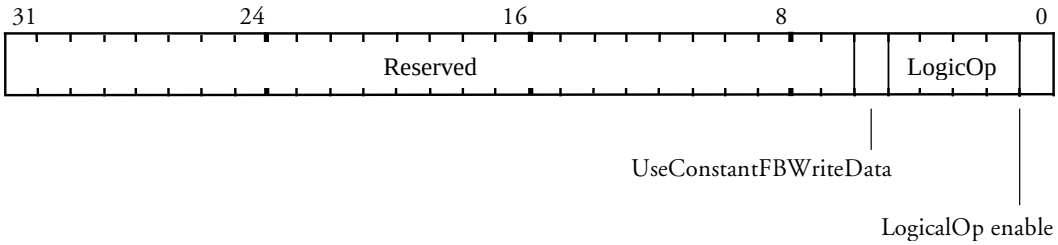


Figure 5.49 LogicalOpMode Register

5.14.5 XOR Example

To set the logical operation to XOR.

```
// Set framebuffer to allow reads
// Not shown

logicalOpMode.UnitEnable = PERMEDIA_ENABLE
logicalOpMode.LogicalOp = PERMEDIA_LOGICOP_XOR

LogicalOpMode(logicalOpMode)        // Load register
```

5.14.6 Software Writemask Example

To set the logical operation to COPY, enable the software writemask, and write to the green component in an 8 bit framebuffer configured in 3:3:2 RGB mode:

```
// Set framebuffer to allow reads
// Not shown

ditherMode.UnitEnable = PERMEDIA_ENABLE
ditherMode.DitherEnable = PERMEDIA_ENABLE
ditherMode.ColorMode = PERMEDIA_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)              // Load register

logicalOpMode.UnitEnable = PERMEDIA_ENABLE
logicalOpMode.LogicalOp = PERMEDIA_LOGICOP_COPY
LogicalOpMode(logicalOpMode)        // Load register

FBSoftwareWriteMask(0xFFFFFFE3)
```

5.15 Host Out Unit

The Host Out Unit controls which registers are available at the output FIFO, gathers statistics about rendering operations (picking and extent testing) and controls synchronization of PERMEDIA with the host.

5.15.1 Filtering

Filtering controls the data made available at the output FIFO. There are the following categories:

- **Depth, Stencil, Color:** These are data values associated with a fragment which has been read from the localbuffer or framebuffer, or generated using the UpLoadData flag in the Framebuffer Write Unit. This category is normally associated with uploading data to the host.
- **Synchronization:** A single register, **Sync** which is used to synchronize PERMEDIA and flush the graphics pipeline.
- **Statistics:** The registers associated with extent checking and picking.

The filtering is controlled by the **FilterMode** register which has 2 bit fields for each category. These fields select whether the register tag and/or register data, are passed to the output FIFO. The format of the **FilterMode** register is shown in the table below.

Register Category	Tag Control Bit	Data Control Bit	Description
Reserved	0	1	
Reserved	2	3	
Depth	4	5	This is the data from image upload of the Depth (Z) buffer.
Stencil	6	7	This is the data from image upload of the Stencil buffer.
Color	8	9	This is the data from image upload of the Framebuffer (FBColor).
Synchronization	10	11	
Statistics	12	13	This is the data generated following a command to read back the results of the statistic measurements: **PickResult**, **MaxHitRegion**, **MinHitRegion**
Reserved	14	15	

Table 5.21 Filter Modes

Note, the filter unit must be set appropriately before any synchronization can take place.

5.15.2 Statistic Operations

There are two statistic collection modes of operation; picking and extent checking. Picking is normally used to select drawn objects or regions of the screen. Typically, extent checking is used to determine the bounds within which drawing has occurred so that a smaller area of the framebuffer can subsequently be cleared.

Statistic collection is controlled using the **StatisticMode** register.

Picking

In picking mode, the active and/or passive fragments have their associated XY coordinates compared against the coordinates specified in the **MinRegion** and **MaxRegion** registers. If the result is true, then the **PickResult** flag is set, otherwise it holds its previous state. The compare function can be either Inside or Outside. Before picking can start, the **ResetPickResult** register must be loaded to clear the PickResult flag.

The **MinRegion** and **MaxRegion** registers are loaded to select the region of interest for picking. A coordinate is inside the region if:

$$X_{min} \le X < X_{max}$$

$$Y_{min} \le Y < Y_{max}$$

where X and Y are from the fragment and the min/max values are from **MinRegion** and **MaxRegion** registers. This comparison is identical to the one used in the scissor tests.

The following stages are required for picking:

1) load **ResetPickResult, MinRegion and MaxRegion** registers

2) Set-up the **FilterMode** to allow statistic commands out of PERMEDIA

3) Draw the primitives.

4) Send a **PickResult** command.

5) Poll the output FIFO waiting for the **PickResult** to have passed through PERMEDIA.

Block fills are ignored by the picking operation.

Extent Checking

In extent mode, active and/or passive fragments have their associated XY coordinates compared to the **MinRegion** and **MaxRegion** registers and if found to be outside the defined rectangular region, then the appropriate register is updated with the new coordinate(s) to extend the region. The Inside/Outside bit has no effect in this mode. Block fills are included in the extent checking if the **StatisticMode** register is set to include spans.

The **MinRegion** and **MaxRegion** registers are loaded to select the maximum value (**MinRegion**) and minimum value (**MaxRegion**) for extent checking. A coordinate is inside the region if:

$$X_{min} \le X < X_{max}$$

$$Y_{min} \le Y < Y_{max}$$

where X and Y are from the fragment and the min/max values are from **MinRegion** and **MaxRegion** registers. This comparison is identical to the one used in the scissor tests.

Once all the necessary primitives have been rendered the results can be found using the **MinHitRegion** and **MaxHitRegion** commands, which cause the contents of the **MinRegion** and **MaxRegion** registers respectively to be written into the output FIFO (under control of the **FilterMode** register).

5.15.3 Synchronization

The **Sync** command register provides a means of ensuring that PERMEDIA has completed all outstanding actions such as localbuffer and framebuffer accesses. **Sync** is filtered and written to the output FIFO in a similar fashion to the other registers. The host can either poll for **Syncs** by reading the output FIFO or await a **Sync** interrupt

If generation of an interrupt is required, then the most significant bit of the **Sync** command register must be set, *and* the filtering must be set-up to at least allow the **Sync** to be written into the FIFO. If the **FilterMode** is set-up so the **Sync** is not written to the FIFO, then **Sync** interrupts will not be generated. The actual interrupt will not occur until the **Sync** data or tag has passed through PERMEDIA and is on the output of the FIFO. This to allow low level resynchronization between the graphics core and PCI clock domains. The FIFO has an extra bit in width to accommodate the interrupt signal. When both the data and tag are written into the FIFO, only the first entry in the FIFO will cause the interrupt (assuming an interrupt was requested).

The remaining bits in the **Sync** data field are free and can be used by the host to identify the reason for the **Sync**.

5.15.4 Registers

Filtering is controlled by the **FilterMode** register:

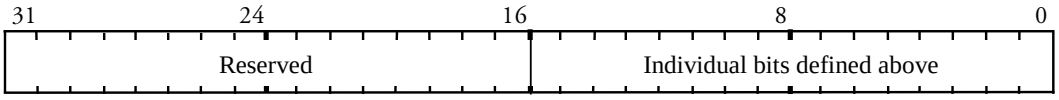


Figure 5.50 FilterMode Register

Statistic collection is controlled by the **StatisticMode** register:

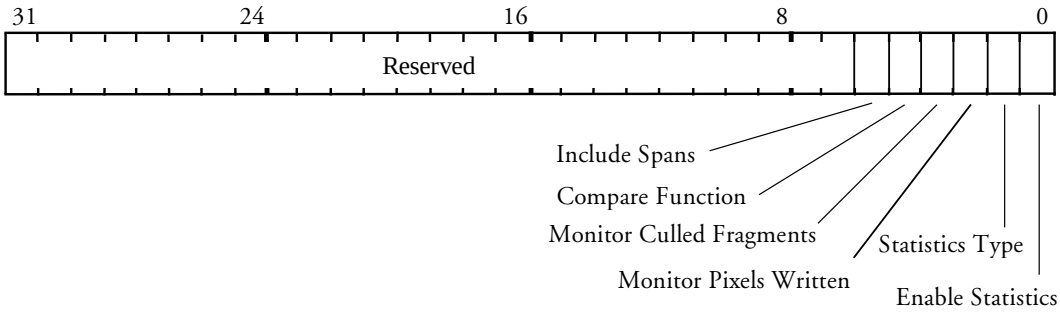


Figure 5.51 StatisticMode Register

The Include Spans bit allows control over whether or not block fills are included in the returned information.

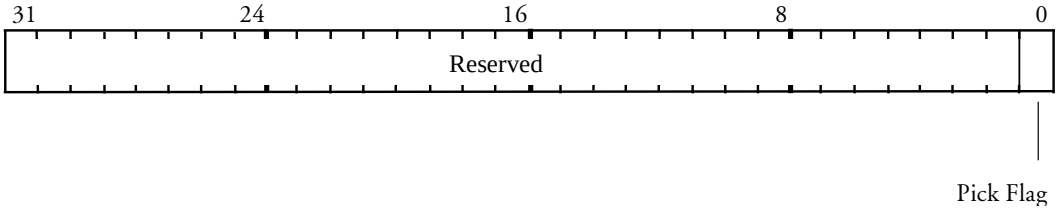


Figure 5.52 PickResult Register

ResetPickResult is used to clear the pick flag. The data field for this register is unused.

MinRegion, **MaxRegion** registers are used to load picking/extent regions, and **MaxHitRegion** and **MinHitRegion** are used to read the registers back. The format is 16 bit 2's complement numbers with Y in the most significant part and X in the least significant part of the word.

Setting the most significant bit of the **Sync** register will request a **Sync** interrupt. Bits 0-30 are available for the user.

5.15.5 Filter Mode Example

```
// Set-up Filter mode to only permit read back of
// synchronization tag and data

FilterMode(0x0C00)     // Set bits 10 & 11
```

5.15.6 Picking Example

Set the statistic mode to picking and detect any active fragments in the region 0x0 <= x < 0x100, 0x0 <= y < 0x100. Render some primitives then read back the results.

```
// Set filter mode as above
FilterMode(0x0C00)     // Set bits 10 & 11

// Set statistic mode
MinRegion(0)
MaxRegion(0x100 | 0x100 << 16)

// Clear the picking flag
ResetPickResult(0x0)      // Data not used

// Now render primitives.... ...

Render (render)            // All units set as appropriate

// All rendering finished.

// Set the filter mode to allow read back of Syncs and
// statistic information (tag and data)
FilterMode(0x3C00)     // Set bits 10 to 13

// Write to the PickResult register
PickResult(0x0)           // Data not used


// Now read the PickResult from the output FIFO (not shown)
```

5.15.7 Sync Interrupt Example

Generate a synchronization interrupt and encode some user defined data (0x34) in the lower 31 bits of the Sync register.

```
// Set-up Filter mode to only permit read back of
// synchronization tag and data
FilterMode(0x0C00)     // Set bits 10 & 11

// Write to the Sync register with the top bit (bit 31) set and
// user data encoded into the lower bits (0-30)

sync = (0x1 << 31) | (0x34 & 0x7FFFFFFF)
Sync (sync)

// Now wait for the sync interrupt. Not shown.
```

6. Initialization

6.1 Initializing PERMEDIA

This section illustrates how to initialize PERMEDIA following reset, prior to carrying out rendering operations.

Initialization falls broadly into three areas, though in different systems precise responsibilities can vary:

- System initialization covers the setting up of the PCI bus, memory and video output. This information typically is only initialized once following reset.
- Window initialization, also referred to as context initialization, covers the setting of the base address of the current rendering window and its color format. This must occur at reset, but will need updating each time PERMEDIA starts drawing to a new window.
- Application initialization covers state that is typically dynamic; enabling and disabling depth testing for example. Again this state must be set at reset, but is likely to be updated relatively frequently.

To make use of the full functionality of PERMEDIA, consult the relevant sections of the Graphics Programming chapter (chapter §5). Examples are given which make use of the pseudocode conventions given in Appendix B.

Note: In general the graphics registers (those listed in Appendix A, as opposed to those documented in the PERMEDIA 2 Hardware Reference Manual) are not hardware initialized to specific values at reset. In the examples below it is assumed that the data structures used to load these registers are initialized to zero. Thus bit fields which are not set explicitly, will default to zero.

6.2 System Initialization

6.2.1 PCI

There are a set of PCI related registers which can be interrogated for information about the chip, for example its revision and device ID. Some of these PCI related registers will need to be set-up at reset, for instance to configure the base addresses of the different memory regions of the chip. For more details refer to the *PERMEDIA 2 Hardware Reference Manual* and the PCI Local Bus Specification Rev2.1.

6.2.2 Memory Configuration

The memory interface control registers should be programmed to reflect the type and amount of memory fitted. The registers are specified in the *PERMEDIA 2 Hardware Reference Manual.*

6.2.3 SVGA and Internal Video Timing Registers

Details for programming the SVGA registers can be found in the *PERMEDIA 2 Hardware Reference Manual.*

The core video timing generator should be programmed to reflect the timings of the monitor being used and the screen resolution and color depth. Note that there is also a SVGA VTG and care must be taken to ensure the correct one is enabled at the right time. To change from SVGA to core display mode, two stages are required. Firstly the core VTG must be set-up and then **VGAControlReg** must be loaded (the EnableVGADisplay bit set to 0).

Details of programming the registers for both VTGs can be found in the *PERMEDIA 2 Hardware Reference Manual.*

6.2.4 Screen Width

The width of the screen is initialized by setting the three partial products fields in the **FBReadMode**, **LBReadMode** and **TextureMapFormat** registers. Note that the width is in pixels, not in bytes, so the same values apply regardless of framebuffer depth, for a given screen resolution. A full list is given in Appendix C.

To initialize the screen to be 1024 pixels wide the registers would be set as follows.

```
fbReadMode.PP0 = 5
fbReadMode.PP1 = 5
fbReadMode.PP2 = 0
FBReadMode(fbReadMode)

lbReadMode.PP0 = 5
lbReadMode.PP1 = 5
lbReadMode.PP2 = 0
LBReadMode(lbReadMode)

textureMapFormat.PP0 = 5
textureMapFormat.PP1 = 5
textureMapFormat.PP2 = 0
TextureMapFormat(textureMapFormat)
```

Note that the PERMEDIA Graphics Core supports a maximum screen resolution of 2048 x 2048[1].

[1] The actual screen resolution obtainable will be limited by the RAMDAC. In the case of the integrated RAMDAC this is 1600 x 1280 at a screen refresh rate of 85 Hz.

6.2.5 Screen Clipping Region

PERMEDIA supports a screen scissor clip which should be set at system initialization, and a user scissor clip which should initially be disabled. Assuming that the **FBWindowBase** and **LBWindowBase** registers are set appropriately, then setting the screen clip prevents writing outside the framebuffer memory (and localbuffer), which could have undesirable results. The following example would be appropriate for a resolution of 1024 by 768 pixels:

```
screenSize.X = 1024
screenSize.Y = 768
ScreenSize(ScreenSize)

scissorMode.ScreenScissorEnable =   PERMEDIA_ENABLE
scissorMode.UserScissorEnable =     PERMEDIA_DISABLE
ScissorMode(ScissorMode)
```

6.2.6 Localbuffer and Framebuffer Configuration

Since PERMEDIA supports a unified memory architecture, it must be decided how the memory is to be partitioned between framebuffer, localbuffer and texture memory. A typical configuration might be to allocate 2 screen sized buffers: one for the visible screen, the other for the 3D back buffer. Then allocate a localbuffer: this is always 16 bits per pixel; and allow the remainder to be used for texture memory. The localbuffer and texture memory can be considered to have different shapes to the front and back buffers. For example, suppose that a screen resolution of 800x600 at 8 bits per pixel is required, then the following offsets could be used. Each offset is a count in pixels from the start of memory.

Front buffer: pixel offset 0
Back buffer: pixel offset 480000 (= 600*800 bytes)
Local buffer: pixel offset 480000 (offset in 16 bit pixels)
Texture memory: byte offset 1920000 (= 2*600*800 + 600*800*sizeof(USHORT))

The size of the pixel depends on the buffer being considered. Hence the offset to the back buffer and the localbuffer appear to be the same but one is measured in bytes, the other in shorts.

These offsets should be saved as software copies to used as required. For example, to select the front buffer for rendering, the **FBPixelOffset** register would be set to 0; to select the back buffer it would be set to the Back buffer pixel offset. The localbuffer offset should be added to the window base offset whenever the **LBWindowBase** register is updated. The value loaded into the **TextureBaseAddress** is a count of the number of texels from the start of memory. Thus the byte offset should be modified to be a texel count when used. In practice, some sort of texture allocation scheme will be needed where textures are allocated starting at the texture memory offset. The final value loaded into the **TextureBaseAddress** register will be the texture memory offset + offset to the

required texture with the final value converted to a texel count from the start of memory.

PERMEDIA supports a range of localbuffer configurations. During initialization, fields in the **LBWriteFormat** and **LBReadFormat** registers should be set to appropriate values. For example:

```
lbReadFormat.DepthWidth = 3       // 15 bit depth buffer
lbReadFormat.StencilWidth  = 3    // 1 bit stencil
LBReadFormat(lbReadFormat)

lbWriteFormat.DepthWidth = 3      // 15 bit depth buffer
lbWriteFormat.StencilWidth  = 3   // 1 bit stencil
LBWriteMode(lbWriteFormat)
```

Note it is possible to dynamically change the number of bits allocated to the depth and stencil buffers, for instance on a per window basis.

Set the framebuffer and localbuffer read units to their default data sources:

```
fbReadMode.DataType =   PERMEDIA_FBDATA
FBReadMode(fbReadMode)

lbReadMode.DataType =   PERMEDIA_LBDEFAULT
LBReadMode(lbReadMode)
```

The following registers are typically only needed for certain specialized operations. Normally their offsets will be zero.

```
FBSourceOffset(0)
FBPixelOffset(0)
LBSourceOffset(0)
```

6.2.7 Host Out Unit

Under some circumstances it is necessary to synchronize with PERMEDIA. This is controlled through the **Sync** command. The host out FIFO should normally be initialized so as to output the **Sync** tag and data (they can be filtered out).

In addition the host out unit should normally be set to filter out all other output data, otherwise the host software must regularly poll the output FIFO to keep it drained and prevent it freezing the pipeline. For example:

```
filterMode.Depth =            PERMEDIA_NULL
filterMode.Stencil =          PERMEDIA_NULL
filterMode.Color =            PERMEDIA_NULL
FilterMode.Synchronization =  PERMEDIA_FILTER_TAG_AND_DATA
                              // Allow Syncs through
filterMode.Statistics =       PERMEDIA_NULL
filterMode.Remainder =        PERMEDIA_NULL
FilterMode(filterMode)
```

6.2.8 Disabling Specialized Modes

Some operations should be disabled until they are need. Refer to the Graphics Programming chapter (chapter §5) for more details on their use.

```
window.LBUpdateSource = PERMEDIA_TRUE
window.ForceLBUpdate =  PERMEDIA_FALSE
window.DisableLBUpdate = PERMEDIA_TRUE
Window(window)
```

6.3 Window Initialization

PERMEDIA supports the concept of a window origin, and makes it relatively simple to implement systems which allow different color formats to coexist in different windows.

6.3.1 Color Format

The Color Format unit and the alpha blend unit should be initialized to an appropriate color format at reset. The units support a variety of different formats, listed in table 4.1.

For example to render in 3:3:2, 8 bit color format, the following would be needed:

```
ditherMode.ColorFormat = PERMEDIA_COLOR_FORMAT_RGB_332_FRONT
DitherMode(ditherMode)

alphaBlendMode.ColorFormat =
PERMEDIA_COLOR_FORMAT_RGB_332_FRONT
AlphaBlendMode(alphaBlendMode)
```

To enable dithering use the following:

```
ditherMode.XOffset =          0
ditherMode.YOffset =          0
ditherMode.DitherEnable =     PERMEDIA_ENABLE
ditherMode.UnitEnable =       PERMEDIA_ENABLE
DitherMode(ditherMode)
```

Note that the Color Format unit is normally always enabled even if dithering itself is not. This is because the unit handles color formatting as well as the dithering operation.

6.3.2 Setting the Window Address and Origin.

PERMEDIA supports the concept of a current window origin. The origin of the window can be specified either as being in the Top Left or Bottom Left corner. This allows the user to pick the most appropriate coordinate system to use; for 3D graphics it would typically be bottom left, whereas for window systems it would be top left. Thus for OpenGL set:

```
fbReadMode.WindowOrigin =  PERMEDIA_BOTTOM_LEFT_WINDOW_ORIGIN
FBReadMode(fbReadMode)

lbReadMode.WindowOrigin =  PERMEDIA_BOTTOM_LEFT_WINDOW_ORIGIN
LBReadMode(lbReadMode)

textureMapFormat.WindowOrigin =
      PERMEDIA_BOTTOM_LEFT_WINDOW_ORIGIN
TextureMapFormat(textureMapFormat)
```

The window origin is set in the Scissor unit. This information usually is provided by the window system. It will need updating if the window moves. As an example if the position of the window is (200, 600) (using a bottom left coordinate system), the origin is specified as follows:

```
windowOrigin.X = 200
windowOrigin.Y = 600
WindowOrigin(windowOrigin)
```

The base address of the window must also be established in the localbuffer read and framebuffer read units. The base address is the physical address that represents the base address of the window. Assuming the base address of the framebuffer represents the pixel in the top left corner of the screen, then for the example above the actual physical address of the bottom left pixel of the window will be set as follows:

```
fbWindowBase = fbBaseAddress +
               (fbWidth * (fbHeight-1-600) + 200)
FBWindowBase(fbWindowBase)

lbWindowBase = lbBaseAddress +
               (lbWidth * (lbHeight-1-600) + 200)
LBWindowBase(lbWindowBase)
```

Where fbBaseAddress, fbWidth and fbHeight are the physical base address, width and height of the framebuffer (in pixels). fbBaseAddress and lbBaseAddress will have been precomputed as described in Section §6.2.6. As with the **WindowOrigin** data, if the window moves, these registers must be updated.

6.3.3 Writemasks

Normally both the hardware (if present) and the software writemasks will initially be set to make all bitplanes writeable:

```
FBSoftwareWriteMask(PERMEDIA_ALL_WRITEMASKS_SET)
FBHardwareWriteMask(PERMEDIA_ALL_WRITEMASKS_SET)
```

6.3.4 Enabling Writing

Which buffers are enabled at any given time is window specific and should be considered for performance reasons. Performance will be improved if unnecessary reads from, and writes to, buffers are disabled. For example if the current rendering does not use depth or stencil testing then reading and writing to the localbuffer may be disabled. The following example initializes the buffers to allow depth buffering and alpha blending:

```
fbWriteMode.UnitEnable =           PERMEDIA_ENABLE
FBWriteMode(fbWriteMode)

lbWriteMode.UnitEnable =           PERMEDIA_ENABLE
LBWriteMode(lbWriteMode)

lbReadMode.ReadSourceEnable =      PERMEDIA_DISABLE
lbReadMode.ReadDestinationEnable = PERMEDIA_ENABLE
```

```
LBReadMode(lbReadMode)

fbReadMode.ReadSourceEnable =      PERMEDIA_DISABLE
fbReadMode.ReadDestinationEnable = PERMEDIA_ENABLE
FBReadMode(fbReadMode)
```

Note that to use software writemasking, the **FBReadMode** register's ReadDestinationEnable field will need to be set if the writemask is set to other than all 1's.

6.3.5 Setting Pixel Size

The size of the pixels must be set so that the memory can be accessed correctly. To do this, use the **FBReadPixel** register e.g.:

```
fbReadPixel.PixelSize = PERMEDIA_16_BIT_PIXEL
FBReadPixel(fbReadPixel)
```

Three framebuffer pixel sizes are possible: 8, 16, 24 and 32 bits. The localbuffer pixel size is fixed at 16 bits.

6.4 Application Initialization

While an application is running, it may dynamically use features of PERMEDIA such as depth buffering, alpha blending, logical operations, etc.. Initially, however, it is recommended that the respective units are disabled, to ensure that they are in a known state:

```
areaStippleMode.UnitEnable =    PERMEDIA_DISABLE
AreaStippleMode(areaStippleMode)

depthMode.UnitEnable =          PERMEDIA_DISABLE
DepthMode(depthMode)

stencilMode.UnitEnable =        PERMEDIA_DISABLE
StencilMode(stencilMode)

textureAddressMode.UnitEnable = PERMEDIA_DISABLE
TextureAddressMode(textureAddressMode)

textureReadMode.UnitEnable =    PERMEDIA_DISABLE
TextureReadMode(textureReadMode)

texelLUTMode.UnitEnable =       PERMEDIA_DISABLE
TexelLUTMode(texelLUTMode)

yuvMode.UnitEnable =            PERMEDIA_DISABLE
YUVMode(yuvMode)

colorDDAMode.UnitEnable =       PERMEDIA_DISABLE
ColorDDAMode(colorDDAMode)

textureColorMode.UnitEnable =   PERMEDIA_DISABLE
TextureColorMode(textureColorMode)

fogMode.UnitEnable =            PERMEDIA_DISABLE
FogMode(fogMode)
```

```
alphaBlendMode.UnitEnable =     PERMEDIA_DISABLE
AlphaBlendMode(alphaBlendMode)

logicalOpMode.UnitEnable =      PERMEDIA_DISABLE
LogicalOpMode(logicalOpMode)

statisticMode.EnableStats =     PERMEDIA_DISABLE
StatisticMode(statisticMode)
```

6.5 Bypass Initialization

The PERMEDIA bypass mechanism gives direct access to memory that PERMEDIA uses to hold the framebuffer, localbuffer and textures. In some situations it is useful for an application to have direct access to this memory without going through the graphics processor. Initialization of PCI registers, in particular the Bypass Writemask register, covers initialization of the bypass mechanism.

The writemask register BypassWriteMask is undefined at boot time and should be set to -1.

Refer to the *PERMEDIA 2 Hardware Reference Manual* for further details.

7. Programming Tips

This chapter covers a variety of programming tips that make best use of PERMEDIA. The topics covered here are not exhaustive.

7.1 PCI Bus Issues

7.1.1 Improving PCI bus bandwidth for Programmed I/O and DMA

The simplest way to program PERMEDIA is by writing data values into the memory mapped registers. i.e. programmed I/O. This is appropriate for primitives which require few set-up parameters such as 2D lines.

For more complex primitives such as Gouraud shaded triangles, where a significant number of registers must be loaded for each primitive, it may be more optimal to write directly to the PERMEDIA FIFO input.

The advantage of this mechanism is that it is then possible to use DMA burst transfers. The disadvantage of this method is that both the address of the register and the data value to be loaded must be written, apparently doubling the amount of data to be loaded.

However, to improve bus bandwidth utilization, the registers have been grouped, into blocks which frequently all need to be updated together, and an indexed addressing mode is supported which allows a single "address" to be loaded, followed by the data for a whole set of registers.

An additional mode is supported which allows a large number of data values to be loaded to the same register. This is useful for image downloads.

For more detail, refer to section §3.2.

7.1.2 PCI burst transfers under Programmed I/O

PCI bus burst transfers typically allow up to four times the bandwidth of individual transfers. However burst transfers are only initiated on the PCI bus when successive addresses are being written to (i.e. the byte address is incremented by 4). When using burst transfers to perform programmed I/O to load the PERMEDIA FIFOs, PERMEDIA multiply maps the FIFO input register throughout the range:

`0x00002000` to `0x00002FFF` in region `0`

Thus when data is being loaded into the FIFO a software loop should be written which starts by writing the first data item at the lower extreme of this address range, and works towards the upper. For further information see section §3.2.

7.1.3 Using PCI Disconnect under Programmed I/O

The PCI bus protocol incorporates a feature known as PCI Disconnect, which is supported by PERMEDIA. Once PERMEDIA is in this mode, if the host processor attempts to write to the full FIFO then instead of the write being lost, the PERMEDIA chip will assert PCI Disconnect. This in turn will cause the host processor to keep retrying the write cycle until it succeeds.

This feature allows faster download of data to PERMEDIA since the host need not poll the *InFIFOSpace* register. But it should be used with care since whenever the PCI Disconnect is asserted, the bus is effectively hogged by the host processor until such time as the PERMEDIA frees up an entry in its FIFO.

7.1.4 Using bus mastership (DMA)

It is expected that most PERMEDIA boards will support PCI bus mastership. This allows the on-board DMA of PERMEDIA to be used to copy data from host memory into the PERMEDIA FIFO.

The use of PCI bus mastership has a number of benefits:

- PCI bus bandwidth utilization is generally much improved.
- PCI bus bandwidth is further improved because the driver software no longer needs to poll the FIFO flags to find how many entries are empty, before loading it.
- Overall system performance may benefit through increased parallelism between PERMEDIA and the host, as the host can often perform useful work preparing the next DMA buffer once it has initiated a DMA transfer.

See section §3.2.4 for more details on using DMA.

7.1.5 Improving performance with DMA

The use of DMA interrupts can significantly improve performance as these allow useful work to be done in time which would be otherwise be used by polling.

Having multiple DMA buffers is usually advantageous. The size and number of buffers is dependent on OS dependent issues such as context switch time.

7.1.6 Improving Texture Mapping performance

The use of interrupts can significantly improve the performance of texture mapping operations. It achieves this by downloading textures 'on demand'. That is during a texture mapping operation, if the required texture map does not exist in local memory, an interrupt is generated so that it can be downloaded. See section §5.8.2 to §5.8.6 for further details.

7.1.7 AGP Support

The Advanced Graphics Port extensions to the PCI protocol are supported by PERMEDIA 2. When in an AGP slot, PERMEDIA 2 will function as a 66MHz PCI

device, and also perform single edge AGP read master transfers, optionally with sideband addressing.

7.2 Graphics Hyperpipeline

7.2.1 Disable Unused Units

Any unit which is not being used should be disabled. This will maximize pixel throughput in the graphics core.

It is important to make sure that data is not being read from the texture buffer, localbuffer or framebuffer unless it is needed. For instance it is perfectly possible to set-up the localbuffer read unit such that PERMEDIA reads per pixel information, such as Z or stencil buffer data, which is then discarded. The effect will be the same visually, but the cost in performance of making the memory accesses will be very high. It is also important to set the LBDisableUpdate bit in the Window register if localbuffer writes are not needed.

For optimal performance, hardware writemasks should be used in preference to software masks.

7.2.2 Avoid Unnecessary Register Updates

PERMEDIA control registers maintain their state between primitives so they do not need to be updated unless the data needs to change. For example, the dY register might be set to +1 for a trapezoid and does not need to be reloaded until a line primitive is drawn.

All delta values and start values are maintained across primitives, so if two triangles share a dominant edge, the start and dominant edge values do not need to be calculated or loaded twice.

Similarly, window clipping need not reload all the registers for each clip rectangle. For example: Load the registers ready for a primitive to be drawn, then enter a loop which repeatedly loads the coordinates for a clip rectangle into the Scissor unit and then sends the **Render** command. Any number of clip rectangles can be processed in this way but PERMEDIA requires only one set-up for each primitive.

7.2.3 Loading Registers in Unit Order

To maximize performance, the control registers for the next primitive should be loaded into the PERMEDIA FIFO in unit order. Thus the registers associated with the Rasterizer unit should be loaded first, then Scissor, Stipple, Localbuffer Read, and so on until the last unit to be loaded is the Host Out unit (if necessary). Then finally the relevant command register should be loaded.

For the order of the units in the hyperpipeline, refer to Fig. 5.1.

7.2.4 Use of Continue Commands

The continue commands provide an efficient method for drawing complex primitives without decomposing them into trapezoids or single lines.

As far as context switching is concerned, each primitive should be treated as atomic. For example, if PERMEDIA context switched after the **Render** command for a triangle, but before it's associated **ContinueNewDom** command, the second part of the primitive may be drawn incorrectly. This is because PERMEDIA relies on internal state set-up by the **Render** command which would have been corrupted by any intervening context.

A second requirement of the continue commands is that data written to the framebuffer or localbuffer before the continue, should not be read after it. This is not a common occurrence, but a possible situation is where two lines are drawn, the second joining the end of the first and being started by **ContinueNewLine**. If these lines are XOR'd they will read the pixel they are about to write to. If the second line is at a sharp angle so that it folds back and overwrites some or all of the first line, the XOR operation is not guaranteed to be correct because the pixels from the first line may not have been written to memory before the second line reads them.

If this situation is likely to occur, a **Sync** command should be sent before the **ContinueNewLine**. This will ensure that all necessary writes complete before the corresponding reads. The software does not have to wait for the **Sync** to be read from the output FIFO, simply sending **Sync** is enough to guarantee correct operation.

7.3 Area Filling Techniques

7.3.1 Clearing Buffers Quickly

Block writes are a feature of SGRAMs. Data written once to a single address can be applied to several addresses at the same time. This is a very fast way of filling areas of the screen, but there are restrictions on when they can be used which are covered elsewhere in this manual.

Block writes are most obviously useful for clearing the screen, but because PERMEDIA has a unified memory buffer it is possible to clear the localbuffer with block writes also.

The extent checking in the host out unit can be used to indicate the area of the screen that has been written, so the screen clear can be limited to the minimum area necessary.

7.3.2 Avoid Clearing Buffers

Although block writes can be used for fast clearing of buffers, it is best not to clear them at all. If all pixels on the screen are drawn at least once per frame then the framebuffer does not need to be cleared. There is no need to clear the localbuffer either if the following procedure is followed.

For even frames, put the viewer at a depth position of zero and draw objects in the lower half of the depth range with the depth test set to 'less than'. For odd frames, put the viewer at the maximum depth value and draw objects into the upper half of the depth range with the depth test set to 'greater than'.

This loses half of the depth range, but avoids the need to clear the depth buffer if every pixel is touched at least once.

7.3.3 Trapezoid Fills

Block writes are most useful when clearing the framebuffer, but can be used to fill any trapezoid.

Block fills, however, are limited to the area defined by the Rasterizer and cannot be changed by the stipple test. A quick filling technique that permits these tests can be achieved by setting the UseConstantFBWriteData bit in the Logic Op unit. When this bit is set, the required color should be loaded into the **FBWriteData** register in the format needed by the memory. All unrequired units should be disabled and the Rasterizer started. The fill can be done up to twice as quickly using this method as opposed to the **ConstantColor** register method.

Also remember that even though the display may be 8 bits per pixel, the chip can be told to draw at 32 bits per pixel. When this is done four pixels are plotted at one time, but the width of the region the Rasterizer covers should be reduced by a factor of four. Use the technique described in the tip about packed copies to get the Framebuffer Write Unit to calculate addresses correctly for 32 bit pixels. The PackedDataLimits register can also be used to mask out unwanted pixels on the left and right edge.

7.4 Copies and Downloads

7.4.1 Copies

If the pixel size is 8 or 16 bits per pixel, the copy speed can be improved by moving more than one pixel at a time. This is achieved by setting the PackedCopy bit in the Framebuffer Read unit. This bit tells PERMEDIA that it should pretend that the pixel size is 32 bits and calculate the addresses accordingly. The screen width does not need to be changed, nor does the base address or source offset value. The Rasterizer should be programmed to rasterize a rectangle that is a factor of four

narrower (for 8 bit pixels) or a factor of 2 narrower (for 16 bit pixels) than the normal size.

The groups of four or two pixels that are copied are all aligned to a 32 bit boundary, but if some of the edge pixels are not needed, the **PackedDataLimits** register can be used to mask them out. If the source and destination pixels have a different alignment then the RelativeOffset field in the **FBReadMode** register can be used to specify how the source needs to be shifted to line up with the destination.

7.4.2 Downloads

The same registers described in the previous tip can also be used to pack data during a download to the framebuffer or localbuffer. If the Rasterizer is set to sync on **FBData**, the data sent to PERMEDIA must be in the raw memory format. Four 8 bit pixels can be written at one time to the chip, and the **PackedDataLimits** register set to mask any unwanted pixels at the left and right edges; the **RelativeOffset** field is used to shift the alignment of the data as it is being stored.

Downloads to the localbuffer can use **LBData,** but the Rasterizer does not support sync on **LBData**, so the data must be explicitly synchronized using the **Sync** command. Alternatively, downloads of stencil and/or depth data can be performed through the framebuffer write unit, allowing **WaitForCompletion** or sync on **FBData** to be used.

7.4.3 Loading Textures

PERMEDIA handles internal synchronization so that all necessary writes complete before reads for a given buffer. If the same data is treated as two different types then the chip must be explicitly synchronized. When a texture is downloaded it is written to memory through the framebuffer write unit, but it is read through the Texture Read unit. This means that the chip must be synchronized between loading the texture and reading it otherwise it is not guaranteed that the writes will have completed before the reads have begun. A **Sync** command can be used to do this, or a **WaitForCompletion** command which does not require the polling of the output FIFO.

Similarly, if the Framebuffer Write unit is used to clear the localbuffer, or the Texture Read unit is used in a copy operation, the chip must be synchronized. The chip will synchronize between localbuffer read and localbuffer write, and between framebuffer read and framebuffer write. Any operations that mix buffers need synchronization.

If a texture is downloaded as a normal image, it can make use of the formatting in the chip to change color format and reorganize the data into rectangular patches. If texture is already in the required format, a fast texture download can be used. To use this, set the TextureDownloadOffset register to point to the start address of the texture (in 32 bit words). Write 32 bit texture data to the TextureData register and

this will be written to memory without changing format. The TextureDownloadOffset will automatically increment following each write. If the texture is 8 bits per texel, then 4 texels must be supplied at a time. This method of texture download avoids the need to set-up the Rasterizer for image download and allows the state of the chip to be left unchanged. Even the framebuffer writes do not have to be enabled.

7.5 Multi Buffering

7.5.1 Fast Double Buffering

PERMEDIA board designs can readily support a variety of double buffering mechanisms depending on the memory configuration and LUT-DAC used, including:

- BLT
- Full Screen
- Bitplane

For further details see section §4.4, §5.12.6, §5.12.7 and §5.13 of this manual.

Note that optimal functionality may be achieved by mixing two or more of the above double buffering techniques.

As a general performance note, it is best to send non-framebuffer related commands to PERMEDIA following a **SuspendUntilFrameBlank** command. This allows better overlap between the host and PERMEDIA. In general any commands that will not cause rendering to the framebuffer to occur can be queued in the PERMEDIA FIFO before waiting on VBLANK.

7.5.2 Triple Buffering

Most 3D systems support double buffering where one frame is displayed while the next frame is being drawn. To avoid display artifacts, the change between old and new buffers must happen during a vertical frame blank, but this imposes a granularity on the frame rate. If a scene takes slightly longer than one frame period to draw, it has to wait for another frame before it can display so the frame rate halves.

If three buffers are used, the quantization is removed and the system can continue to draw at maximum rate.

7.6 Overlays

Overlays are only available with the 5551 color format in a 32 bit pixel. The PERMEDIA 5551 color formats copy the data into both 16 bit halves of the 32 bit pixel. The writemask is used to write either the upper or lower half to memory.

The RAMDAC can be programmed to display a 16 bit pixel from either the upper or lower half of the 32 bit word; which one is displayed is set by bit 31. Bit 31 corresponds to the alpha bit of the 16 bit pixel, and this can be forced to either 1 or 0 by the Color Format unit.

When drawing to the underlay (or main image) set the Color Format unit to force the alpha to zero, set the writemask to allow writes to the lower half of the word. When drawing to the overlay set the Color Format unit to force the alpha value to 1 and writemask to allow writes to the upper half of the word.

If the RAMDAC is set into the appropriate mode, pixels in the overlay half of the word will be drawn where alpha is 1 in the overlay and from the main image where it is zero in the overlay.

7.7 Memory Organization

The amount of memory available to PERMEDIA depends on the board it is fitted to. The most efficient way to allocate memory will depend on the needs of the system, but in general the display should be allocated at one end of the SGRAM and the localbuffer at the other end. This leaves a region between the two buffers in which textures can be stored. For optimal performance, each buffer (front color, back color, texture and depth) should reside in separate memory banks. Memory is organized as follows:

memory size	banks	size per bank
2Mb	2	1Mb
4Mb	4	1Mb
6Mb	4	1 or 2Mb
8Mb	4	2Mb

Table 7.1 Memory Organization

With 6Mb of memory, the first two banks will contain 1Mb and the subsequent two, 2Mb.

7.8 Chroma Test

Chroma key testing can be done without involving texture mapping. This is achieved by setting the TexelDisableUpdate field in the **YUVMode** register. This will allow fragments to be rejected by chroma testing as part of a copy operation. The texels are read in and tested, and fragments rejected if the colors do not match.

Setting the TexelDisableUpdate bit discards the data as soon as the test has been done which improves performance.

This is described in more detail in section §5.9.1.

7.9 Configuration for 2D

Particular fields of several registers can be set by writing to a single register, Config. This groups together fields of registers commonly used in 2D operations together so that PERMEDIA may be configured by fewer accesses. Reading from this register returns invalid data.

8. Delta Programming Examples

The following examples demonstrate how to render a depth buffered, Gouraud shaded triangle mesh using the Delta Unit. The window into which the rendering takes place is partially obscured and hence is clipped by two clip rectangles.

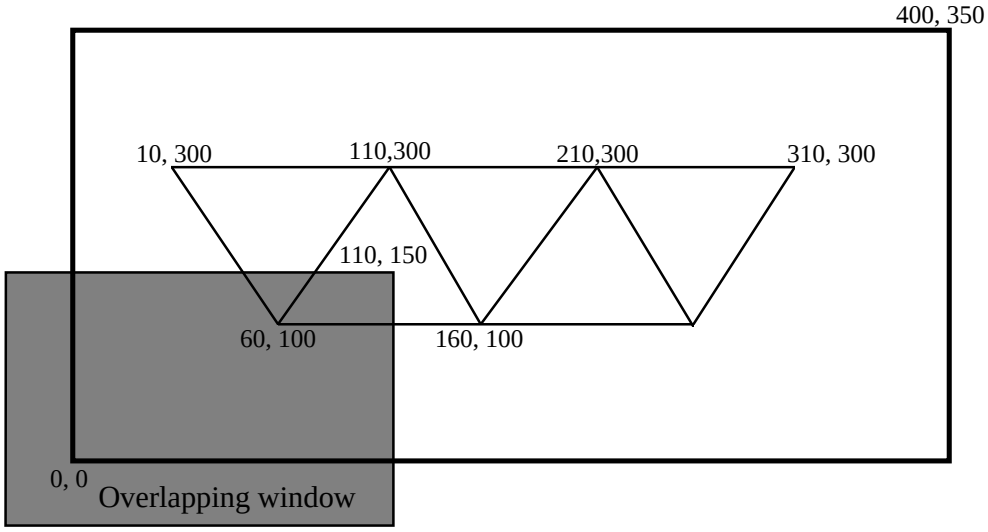


Figure 8.1 Geometry of the Mesh and Clip regions.

The three examples cover drawing the mesh as a set of points at the vertices, as connected line segments and finally as filled triangles. For simplicity, the triangles in these examples are either flat topped or flat bottomed. In practice, triangles are not restricted to these shapes and can have any orientation, size or shape.

```
// This is the header file for the Delta Unit PRM example code.
// It only contains the necessary items to support the examples.

#ifdef BIG_ENDIAN
// The DeltaMode register fields.

typedef struct {
   unsigned int pad:                          14;
   unsigned int ColorOrder:                   1;
   unsigned int BackfaceCallEnable:           1;
   unsigned int TextureParameterMode:         2;
   unsigned int ClampEnable:                  1;
   unsigned int NoDraw:                       1;
   unsigned int DiamondExit:                  1;
   unsigned int SubPixelCorrectionEnable:     1;
   unsigned int DiffuseTextureEnable:         1;
   unsigned int SpecularTextureEnable:        1;
   unsigned int DepthEnable:                  1;
   unsigned int SmoothShadingEnable:          1;
   unsigned int TextureEnable:                1;
   unsigned int FogEnable:                    1;
   unsigned int Reserved:                     4;
   unsigned int TargetChip:                   2;
} __DeltaModeFmat;

// The DrawTriangle and DrawLine command fields.

typedef struct {
   unsigned int pad:                          14;
   unsigned int ReuseBitMask:                 1;
   unsigned int SubPixelCorrectionEnable:     1;
   unsigned int Reserved:                     1;
   unsigned int FogEnable:                    1;
   unsigned int TextureEnable:                1;
   unsigned int SyncOnHostData:               1;
   unsigned int SyncOnBitMask:                1;
   unsigned int Reserved:                     3;
   unsigned int PrimitiveType:                2;
   unsigned int Reserved:                     2;
   unsigned int FastFillEnable:               1;
   unsigned int Reserved:                     2;
   unsigned int LineStippleEnable:            1;
} __DeltaRenderFmat;

#else

// The DeltaMode register fields.

typedef struct {
   unsigned int Reserved:                     4;
   unsigned int FogEnable:                    1;
   unsigned int TextureEnable:                1;
   unsigned int SmoothShadingEnable:          1;
   unsigned int DepthEnable:                  1;
   unsigned int SpecularTextureEnable:        1;
   unsigned int DiffuseTextureEnable:         1;
   unsigned int SubPixelCorrectionEnable:     1;
   unsigned int DiamondExit:                  1;
```

```
   unsigned int NoDraw:                      1;
   unsigned int ClampEnable:                 1;
   unsigned int TextureParameterMode:        2;
   unsigned int BackfaceCallEnable:          1;
   unsigned int ColorOrder:                  1;
   unsigned int pad:                         14;
} __DeltaModeFmat;

// The DrawTriangle and DrawLine command fields.

typedef struct {
   unsigned int AreaStippleEnable:           1;
   unsigned int ReservedC:                   2;
   unsigned int FastFillEnable:              1;
   unsigned int reserved:                    2;
   unsigned int PrimitiveType:               2;
   unsigned int ReservedB:                   1;
   unsigned int SyncOnBitMask:               1;
   unsigned int SyncOnHostData:              1;
   unsigned int TextureEnable:               1;
   unsigned int FogEnable:                   1;
   unsigned int ReservedA:                   1;
   unsigned int SubPixelCorrectionEnable:    1;
   unsigned int pad:                         14;
   unsigned int ReuseBitMask:                1;
} __DeltaRenderFmat;
#endif

// The tag values for the registers.
#define __Delta_V0FloatTag                   0x230
#define __Delta_V1FloatTag                   0x240
#define __Delta_V2FloatTag                   0x250
#define __DeltaTagDeltaMode                  0x260
#define __DeltaTagDrawTriangle               0x261
#define __DeltaTagRepeatTriangle             0x262
#define __DeltaTagDrawLine01                 0x263
#define __DeltaTagDrawLine10                 0x264
#define __DeltaTagRepeatLine                 0x265

// Some temp defines to keep things compiling easily.
#define DrawTriangleTag      __DeltaTagDrawTriangle
#define DrawLine01Tag        __DeltaTagDrawLine01
#define DrawLine10Tag        __DeltaTagDrawLine10
#define RepeatTriangleTag    __DeltaTagRepeatTriangle
#define RepeatLineTag        __DeltaTagRepeatLine
```

```
#include "delta.h"
#include <stdio.h>
extern unsigned long *dmaPtr;
extern DMA *dma;

// Change these macros to what is needed to write the values to Delta
// Unit, or add them to a dma buffer.
#define LD_REG(reg, value)  dmaPtr = dma->Space(2); *dmaPtr++ = reg;\
                                                     *dmaPtr++ = value;
#define LD_PARAM(reg, value) dmaPtr = dma->Space(2); *dmaPtr++ = reg;\
                                *dmaPtr++ = *((unsigned long *) &value);

// Prototypes
void PointMesh (gal &cx);
void LineMesh (gal &cx);
void TriangleMesh (gal &cx);

// Simple structure to use in the example code

typedef struct { float  x, y, z, r, g, b, a; }  Vertex;
typedef struct { short  x, y; }                 XY;
typedef struct { XY scissorMin, scissorMax; }   ClipRectangle;

// Define some test data.

#define verticesInMesh 7
Vertex  mesh[verticesInMesh] = {
             //    x      y      z      r      g      b      a
              {   10,   300,   0.1,   1.0,   1.0,   1.0,   1.0   },
              {   60,   100,   0.2,   1.0,   1.0,   0.0,   1.0   },
              {  110,   300,   0.3,   1.0,   0.0,   1.0,   1.0   },
              {  160,   100,   0.4,   1.0,   0.0,   0.0,   1.0   },
              {  210,   300,   0.5,   0.0,   1.0,   1.0,   1.0   },
              {  260,   100,   0.6,   0.0,   1.0,   0.0,   1.0   },
              {  310,   300,   0.7,   0.0,   0.0,   1.0,   1.0   }};

#define numberClipRectangles 2

ClipRectangle clipRectangles[numberClipRectangles] = {
            { {110, 0}, {400, 150} },
            { {0, 150}, {400, 350} }};

enum {paramS, paramT, paramQ, paramKs, paramKd, paramR, paramG, paramB,
paramA, paramF, paramX, paramY, paramZ};
```

```
// This function draws the vertices in the mesh as points. There is
// no direct support for points in Delta Unit as they do not need
// any set-up calculations. Delta Unit can be used to plot points
// (maybe because you want to always work in floating point) by
// having Delta Unit do the set-up calculations for a line, but tell
// the rendering device to render points.

void PointMesh (gal &cx)
{
    __DeltaModeFmat              deltaMode;
    __DeltaRenderFmat            drawCmd;
    int                          rect, v;

    // Assume the rendering device is already initialized.
   // Note we expect the BiasCoords mode in the RasterizerMode
   // register to be set to add a bias of zero.
   // Set-up the DeltaMode register.

    deltaMode.pad                          = 0;
    deltamode ColorOrder                   = 0;
    deltamode BackfaceCallEnable           = 0;
    deltaMode.TextureParameterMode         = 1;   // Clamp.
    deltaMode.ClampEnable                  = 1;   // Clamp enabled.
    deltaMode.NoDraw                       = 0;   // Do drawing.
    deltaMode.DiamondExit                  = 0;   // Not needed for this
                                                  // example.
    deltaMode.SubPixelCorrectionEnable     = 0;   // No sub pixel
                                                  // correction.
    deltaMode.DiffuseTextureEnable         = 0;   // Disable.
    deltaMode.SpecularTextureEnable        = 0;   // Disable.
    deltaMode.DepthEnable                  = 1;   // Enable.
    deltaMode.SmoothShadingEnable          = 1;   // Enable.
    deltaMode.TextureEnable                = 0;   // Disabled.
    deltaMode.FogEnable                    = 1;   // Enabled, but
                                                  // controlled from the
                                                  // draw command.
    deltaMode.Reserved                     = 0;

   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

   // Set-up the draw command data.
    drawCmd.pad                            = 0;
    drawCmd ReuseBitMask                   = 0;
    drawCmd.SubPixelCorrectionEnable       = 0;   // Enable.
    drawCmd.ReservedA                      = 0;
    drawCmd.FogEnable                      = 0;   // Disable.
    drawCmd.TextureEnable                  = 0;   // Disable.
    drawCmd.SyncOnHostData                 = 0;   // Disable.
    drawCmd.SyncOnBitMask                  = 0;   // Disable.
    drawCmd.ReservedB                      = 0;
    drawCmd.AntialiasEnable                = 0;   // Disable.
    drawCmd.PrimitiveType                  = 2;   // ** Point **
    drawCmd.reserved                       = 0;
    drawCmd.FastFillEnable                 = 0;   // Disable.
    drawCmd.ReservedC                      = 0;
    drawCmd.AreaStippleEnable              = 0;   // Disable.
```

```
    // We need to ensure that the end vertex of the line (in V1)
    // can never be the same as the point vertices.  Any X (or Y)
    // coordinate which is out of the normal range (0.0 to screen
    // width) will do so in this case an X of -1.0 has been used.

    float   tempEndCoord = -1.0;

    LD_PARAM ((__Delta_V1FloatTag + paramX), tempEndCoord);

    for (v = 0; v < verticesInMesh; v++)
    {
        LD_PARAM((__Delta_V0FloatTag + paramR), mesh[v].r);
        LD_PARAM((__Delta_V0FloatTag + paramG), mesh[v].g);
        LD_PARAM((__Delta_V0FloatTag + paramB), mesh[v].b);
        LD_PARAM((__Delta_V0FloatTag + paramA), mesh[v].a);
        LD_PARAM((__Delta_V0FloatTag + paramX), mesh[v].x);
        LD_PARAM((__Delta_V0FloatTag + paramY), mesh[v].y);
        LD_PARAM((__Delta_V0FloatTag + paramZ), mesh[v].z);

        for (rect = 0; rect < numberClipRectangles; rect++)
        {
            // Load in the scissor rectangle.
            LD_REG(ScissorMinXYTag, (clipRectangles[rect].scissorMin.y
                    << 16 |  clipRectangles[rect].scissorMin.x));
            LD_REG(ScissorMaxXYTag, (clipRectangles[rect].scissorMax.y
                    << 16 |  clipRectangles[rect].scissorMax.x));

            if (rect == 0)
            {
                    LD_REG(DrawLine01Tag, *((long *) &drawCmd));
            }
            else
            {
                    LD_REG(RepeatLineTag, 0);  // data field not used.
            }
        }
     }
}




// This array holds the order we are going to visit the
// vertices in to draw each line segment.

Lint lineOrder[12] = {1, 0, 2, 4, 6, 5, 4, 3, 2, 1, 3, 5};
```

```
// This function draws the mesh as a series of lines.  The order the
// lines are drawn in is hardcoded (this is only an example!).

void LineMesh (gal &cx)
{
    __DeltaModeFmat          deltaMode;
    __DeltaRenderFmat        drawCmd;
    int                      vertexStore, rect, i, v;

   // Assume the rendering device is already initialized. Note we
   // expect the BiasCoords mode in the RasterizerMode register to
   // be set to add a bias of zero.

   // Set-up the DeltaMode register.
    deltaMode.pad                          = 0;
    deltamode.ColorOrder                   = 0;
    deltamode.BackfaceCallEnable           = 0;
    deltaMode.TextureParameterMode         = 2;   // Auto normalize.
    deltaMode.ClampEnable                  = 1;   // Clamp enabled.
    deltaMode.NoDraw                       = 0;   // Do drawing.
    deltaMode.DiamondExit                  = 1;   // Not needed for this
                                                  // example.
    deltaMode.SubPixelCorrectionEnable     = 1;   // Enable sub pixel
                                                  // correction.
    deltaMode.DiffuseTextureEnable         = 0;   // Disable.
    deltaMode.SpecularTextureEnable        = 0;   // Disable.
    deltaMode.DepthEnable                  = 1;   // Enable.
    deltaMode.SmoothShadingEnable          = 1;   // Enable.
    deltaMode.TextureEnable                = 0;   // Disabled.
    deltaMode.FogEnable                    = 1;   // Enabled, but
                                                  // controlled from the
                                                  // draw command.
    deltaMode.Reserved                     = 0;

   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

// Set-up the draw command data.
    drawCmd.pad                            = 0;
    DrawCmd.ReuseBitMask                   = 0;
    drawCmd.SubPixelCorrectionEnable       = 1;   // Enable.
    drawCmd.ReservedA                      = 0;
    drawCmd.FogEnable                      = 0;   // Disable.
    drawCmd.TextureEnable                  = 0;   // Disable.
    drawCmd.SyncOnHostData                 = 0;   // Disable.
    drawCmd.SyncOnBitMask                  = 0;   // Disable.
    drawCmd.ReservedB                      = 0;
    drawCmd.AntialiasingQuality            = 0;   // Not used.
    drawCmd.AntialiasEnable                = 0;   // Disable.
    drawCmd.PrimitiveType                  = 0;   // Line.
    drawCmd.reserved                       = 0;
    drawCmd.FastFillEnable                 = 0;   // Disable.
    drawCmd.ReservedC                      = 0;
    drawCmd.AreaStippleEnable              = 0;   // Disable.
```

```
        for (i = 0; i < 12; i++)
        {
              v = lineOrder[i];
              vertexStore = __Delta_V0FloatTag + 16 * (i % 2);

              LD_PARAM((vertexStore + paramR), mesh[v].r);
              LD_PARAM((vertexStore + paramG), mesh[v].g);
              LD_PARAM((vertexStore + paramB), mesh[v].b);
              LD_PARAM((vertexStore + paramA), mesh[v].a);
              LD_PARAM((vertexStore + paramX), mesh[v].x);
              LD_PARAM((vertexStore + paramY), mesh[v].y);
              LD_PARAM((vertexStore + paramZ), mesh[v].z);

              if (i >= 1)
              {
                  // We now have enough vertices to draw a line.
                  for (rect = 0; rect < numberClipRectangles; rect++)
                  {
                     // Load in the scissor rectangle.
                     LD_REG(ScissorMinXYTag,
                            (clipRectangles[rect].scissorMin.y << 16 |
                            clipRectangles[rect].scissorMin.x));

                     LD_REG(ScissorMaxXYTag,
                            (clipRectangles[rect].scissorMax.y << 16 |
                            clipRectangles[rect].scissorMax.x));

                     if (rect == 0)
                     {
                            if (i & 1)
                            {
                                LD_REG(DrawLine01Tag, *((long *) &drawCmd));
                            }
                            else
                            {
                                LD_REG(DrawLine10Tag, *((long *) &drawCmd));
                            }
                     }
                     else
                     {
                            LD_REG(RepeatLineTag, 0);  // data field unused
                     }
                  }
           }
        }
}
```

```
// This function draws the mesh as a series of shaded triangles.

void TriangleMesh (gal &cx)
{
    __DeltaModeFmat      deltaMode;
    __DeltaRenderFmat    drawCmd;
    int                  vertexStore;
    int                  rect, v;

   // Assume the rendering device is already initialized.  Note we
   // expect the BiasCoords mode in the RasterizerMode register to be
   // set to add a bias of zero.

   // Set-up the DeltaMode register.

    deltaMode.pad                          = 0;
    deltamode.ColorOrder                   = 0;
    deltamode.BackfaceCallEnable           = 0;
    deltaMode.TextureParameterMode         = 2;  // Auto normalize.
    deltaMode.ClampEnable                  = 1;  // Clamp enabled.
    deltaMode.NoDraw                       = 0;  // Do drawing.
    deltaMode.DiamondExit                  = 1;  // Not needed for this
                                                 // example.
    deltaMode.SubPixelCorrectionEnable     = 1;  // Enable sub pixel
                                                 // correction.
    deltaMode.DiffuseTextureEnable         = 0;  // Disable.
    deltaMode.SpecularTextureEnable        = 0;  // Disable.
    deltaMode.DepthEnable                  = 1;  // Enable.
    deltaMode.SmoothShadingEnable          = 1;  // Enable.
    deltaMode.TextureEnable                = 0;  // Disabled.
    deltaMode.FogEnable                    = 1;  // Enabled, but
                                                 // controlled from
                                                 // the draw command.
    deltaMode.Reserved                     = 0;

    LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

    // Set-up the draw command data.
    drawCmd.pad                            = 0;
    drawCmd.ReuseBitMask                   = 0;
    drawCmd.SubPixelCorrectionEnable       = 1;  // Enable.
    drawCmd.RaservedA                      = 0;
    drawCmd.SyncOnBitMask                  = 0;  // Disable.
    drawCmd.FogEnable                      = 0;  // Disable.
    drawCmd.TextureEnable                  = 0;  // Disable.
    drawCmd.SyncOnHostData                 = 0;  // Disable.
    drawCmd.ReservedB                      = 0;
    drawCmd.AntialiasEnable                = 0;  // Disable.
    drawCmd.PrimitiveType                  = 1;  // Trapezoid.
    drawCmd.reserved                       = 0;
    drawCmd.FastFillEnable                 = 0;  // Disable.
    drawCmd.ReservedC                      = 0;
    drawCmd.AreaStippleEnable              = 0;  // Disable.
```

```
        for (v = 0; v < verticesInMesh; v++)
        {
           vertexStore = __Delta_V0FloatTag + 16 * (v % 3);

           LD_PARAM((vertexStore + paramR), mesh[v].r);
           LD_PARAM((vertexStore + paramG), mesh[v].g);
           LD_PARAM((vertexStore + paramB), mesh[v].b);
           LD_PARAM((vertexStore + paramA), mesh[v].a);
           LD_PARAM((vertexStore + paramX), mesh[v].x);
           LD_PARAM((vertexStore + paramY), mesh[v].y);
           LD_PARAM((vertexStore + paramZ), mesh[v].z);

           if (v >= 2)
           {
              // We now have enough vertices to draw a triangle.
              for (rect = 0; rect < numberClipRectangles; rect++)
              {
                     // Load in the scissor rectangle.
                     LD_REG(ScissorMinXYTag,
                            (clipRectangles[rect].scissorMin.y << 16 |
                            clipRectangles[rect].scissorMin.x));

                     LD_REG(ScissorMaxXYTag,
                            (clipRectangles[rect].scissorMax.y << 16
                            clipRectangles[rect].scissorMax.x));

                     if (rect == 0)
                     {
                            LD_REG(DrawTriangleTag,  *((long *) &drawCmd));
                     }
                     else
                     {
                            LD_REG(RepeatTriangleTag, 0); // data field not
                                                          // used.
                     }
              }
           }
        }
}
```

Appendix A. Graphics Register Reference

This chapter gives details of the format of each of the Graphics registers for PERMEDIA. The registers are listed alphabetically by name within their function, with the functions themselves listed alphabetically.

- **Tag** specifies the offset for this register from the base address of the region.
- **Read/write** indicates that the register can be both read and written.
- **Write** indicates that the register can only be written. The value of any read from this address is undefined.
- **Reset Value** specifies the value of the register following hardware reset. In general this is undefined for Graphics registers.

In the diagrams:

- **Reserved** indicates bits that may be used in future members of the PERMEDIA family. To ensure upwards compatibility, any software should not assume a value for these bits when read, and should always write them as zeros.
- **Not used** indicates bits that are adjacent to numeric fields. These may be used in future members of the PERMEDIA family, but only to extend the dynamic range of these fields. The data returned from a read of these bits is undefined. When a "Not used" field resides in the most significant position, a good convention to follow is to sign extend the numeric value, rather than masking the field to zero before writing the register. This will ensure compatibility if the dynamic range is increased in future members of the PERMEDIA family.
- For enumeration fields which do not specify the full range of possible values, only the specified values should be used. An example of an enumeration field is the comparison field in the **DepthMode** register. Future members of the PERMEDIA family may define a meaning for the unused values.

AlphaBlendMode

Name: Alpha Blend Mode

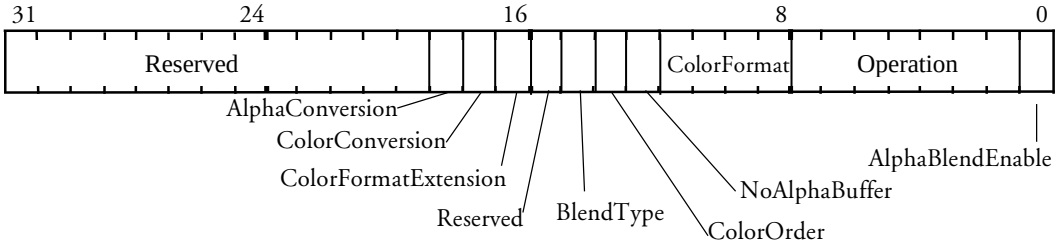
Unit: Texture/Fog/Blend

Region: 0 Offset: 0x0000.8810

Tag: 0x0102

Reset Value: Undefined

Read/write



Controls Alpha Blending.

Bit0 Enable:
0 = Disable
1 = Enable alpha blending or color formatting

Bit1-7 Operation:

Bit17 Color Conversion:
0 = Scale
1 = Shift

Bit18 Alpha Conversion:
0 = Scale
1 = Shift

Mode	Operation	R	G	B	A
16	Format	R_d	G_d	B_d	A_d
84	Blend	$R_s * A_s + R_d * (1-A_s)$	$G_s * A_s + G_d * (1-A_s)$	$B_s * A_s + B_d * (1-A_s)$	$A_s * A_s + A_d * (1-A_s)$
81	PreMult	$R_s + R_d * (1-A_s)$	$G_s + G_d * (1-A_s)$	$B_s + B_d * (1-A_s)$	$A_s + A_d * (1-A_s)$

For correct operation of Apple PreMult blending, the BlendType needs to be set to Ramp.

Result of different operations. C_s = source color component, C_d = destination color component.

(See overleaf for description of the remaining bits).

Bit8-11 Color Format:

			Internal Color Channel			
Format	Color Order	Name	R	G	B	A
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0
9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0
17	RGB	5:6:5 Back	5@27	6@21	5@16	0

Notes: The format column is also dependent on bit16. n@m means n bits starting at bit m. Front and Back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the 7 bit formatted value. If the format has no alpha bits, the alpha field defaults to 0xF8

Bit12 NoAlphaBuffer

0 = Alpha buffer present
1 = No alpha buffer present

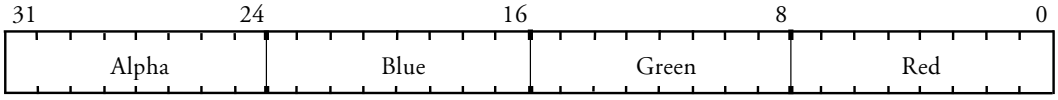
Bit13 ColorOrder:
0 = BGR
1 = RGB

Bit14 BlendType:
0 = RGB
1 = Ramp

Bit16 Color Format Extension. Most significant bit extension to Color Format held in bits8-11.

AlphaMapLowerBound

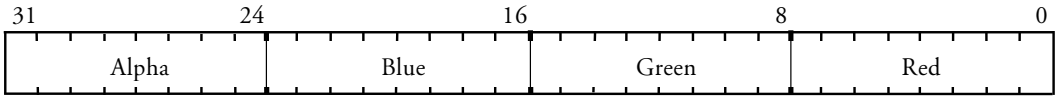
Name:	Alpha Map Color Test Lower and Upper Bounds	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8F20
	Tag:	0x01E4
Reset Value:	Undefined	
Read/write		



Specifies the lower and upper bounds for the alpha map test.

AlphaMapUpperBound

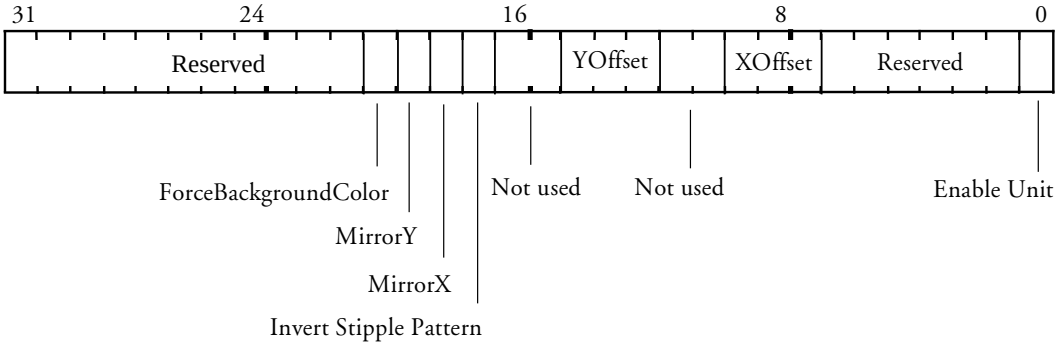
Name:	Alpha Map Color Test Lower and Upper Bounds	
Unit:	Texture Read	
Region: 0	Offset:	0x000.8F18
	Tag:	0x01E3
Reset Value:	Undefined	
Read/write		



Specifies the lower and upper bounds for the alpha map test.

AreaStippleMode

Name:	Area Stipple Mode	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.81A0
	Tag:	0x0034
Reset Value:	Undefined	
Read/write		

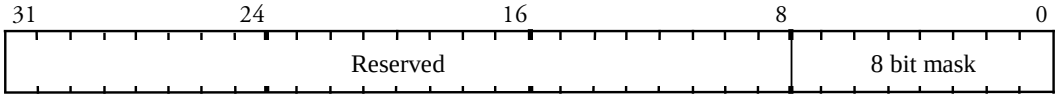


Controls Area Stippling. Both the AreaStippleEnable bit in the **Render** command and the enable in the **AreaStippleMode** register must be set to enable the area stipple test.

Bit0	Unit Enable 0 = Disable 1 = Enable
Bit7-9	XOffset
Bit12-14	YOffset
Bit17	Invert Stipple Pattern 0 = No Invert 1 = Invert
Bit18	Mirror X 0 = No Mirror in X 1 = Mirror stipple pattern in X direction
Bit19	Mirror Y 0 = No Mirror in Y 1 = Mirror stipple pattern in Y direction
Bit20	ForceBackgroundColor. Controls operation of the stipple test. If disabled any fragment failing the test is discarded. If enabled any fragment failing the test is drawn (other tests allowing) but the color is taken from the **Texel0** register. Used to support foreground and background colors. 0 = Disable 1 = Enable

AreaStipplePattern[0...7]

Name:	Area Stipple Pattern	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.8200, ...,0x0000.8238
	Tag:	0x0040, ...,0x0047
Reset Value:	Undefined	
Read/write		

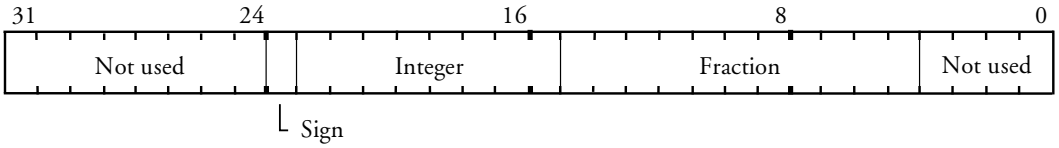


These 8 registers provide the bitmask which enables and disables corresponding fragments for drawing when rasterizing a primitive with area stippling.

Both the AreaStippleEnable in the **Render** command and enable in the **AreaStippleMode** register must be set, to enable the area stipple test.

AStart

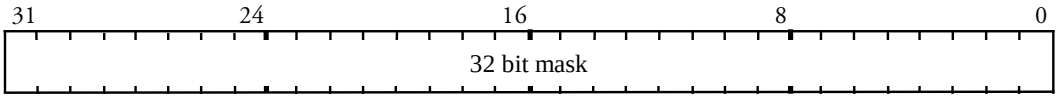
Name:	Initial Alpha Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87C8
	Tag:	0x00F9
Reset Value:	Undefined	
Read/write		



This register is used to set the initial value for the Alpha for a vertex when in Gouraud shading mode. The value is 2's complement 9.11 fixed point format.

BitMaskPattern

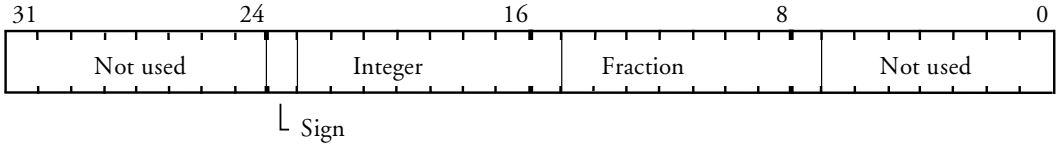
Name:	Bit Mask Pattern	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8068
	Tag:	0x000D
Reset Value:	Undefined	
Write only		



Value used to control the bit mask stipple operation (if enabled). Fragments are accepted or rejected based on the current BitMask test modes defined by the **RasterizerMode** register. Note that the SyncOnBitmask bit in the **Render** command must also be enabled.

BStart

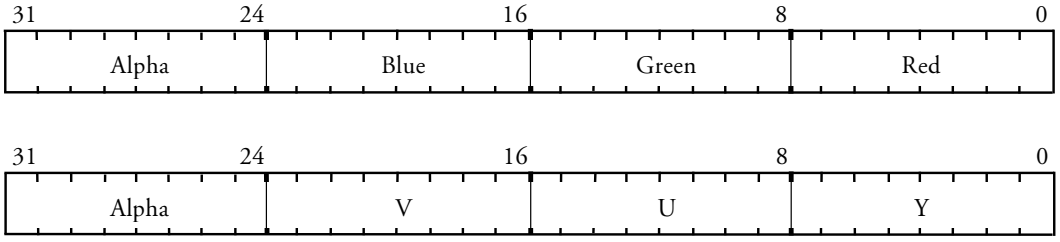
Name:	Initial Blue Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000. 87B0
	Tag:	0x00F6
Reset Value:	Undefined	
Read/write		



This register is used to set the initial value for the Blue for a vertex when in Gouraud shading mode. The value is 2's complement 9.11 fixed point format.

ChromaLowerBound,ChromaUpperBound

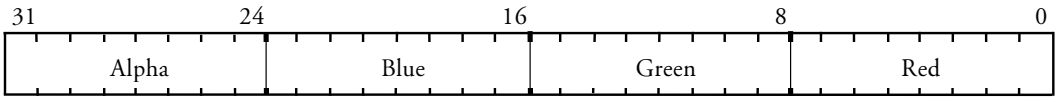
Name:	Chroma Lower Bound, Chroma Upper Bound	
Unit:	YUV	
Region: 0	Offset:	0x00008F10., 0x0000.8F08
	Tag:	0x01E2, 0x01E1
Reset Value:	Undefined	
Read/write		



Specifies the lower and upper bounds for the chroma test. The test is done against the contents of the **Texel0** register which holds data in the internal RGB format or the YUV format (before conversion) of 8 bits per component. The test is done on all 8 bits of each component. All components must be inside the bounds for the test to pass, if TestMode is set to 1 in the **YUVMode** register, or fail if TestMode is set to 2 in the **YUVMode** register.

Color

Name:	Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87F0
	Tag:	0x00FE
Reset Value:	Undefined	
Write		



Used for downloading image data to the framebuffer. The format is either the standard color format, or the raw framebuffer format if the Color Format unit is disabled.

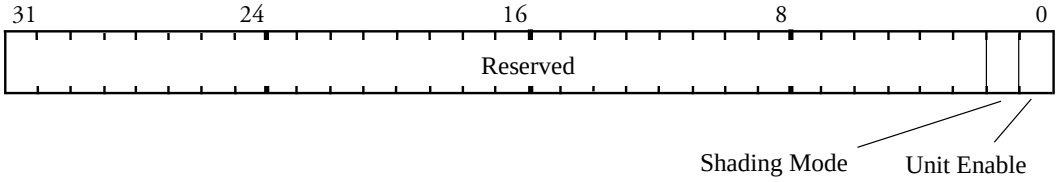
In CI mode the color index is placed in bits 0-7. If there are less than 8 bits in a component it should be left justified and the unused bits set to zero.

This register cannot be saved and restored as part of a task context switch.

When used this register should always be reloaded at start of every command, and the Color DDA unit must be disabled prior to loading it.

ColorDDAMode

Name:	Color DDA Mode	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87E0
	Tag:	0x00FC
Reset Value:	Undefined	
Read/write		



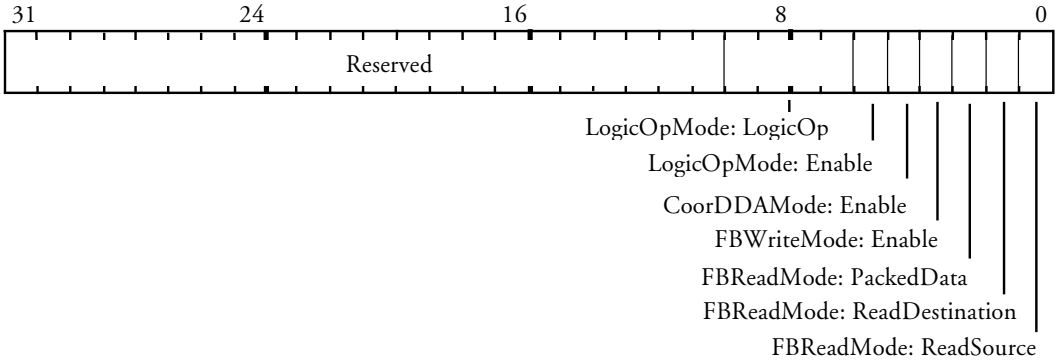
The bit fields control the mode of operation of the Color DDA unit:

Bit0 Unit Enable:
- 0 = Disable
- 1 = Enable

Bit1 Shading mode control:
- 0 = Flat
- 1 = Gouraud

Config

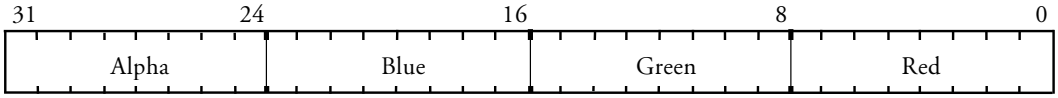
Name:	Configuration	
Unit:		
Region: 0	Offset:	0x0000.8D90
	Tag:	0x01B2
Reset Value:	Undefined	
Read/write		

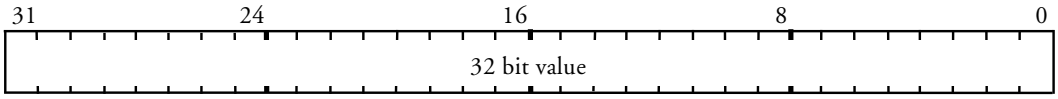


Sets the specified fields in various registers.

ConstantColor

Name:	Constant Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87E8
	Tag:	0x00FD
Reset Value:	Undefined	
Read/write		



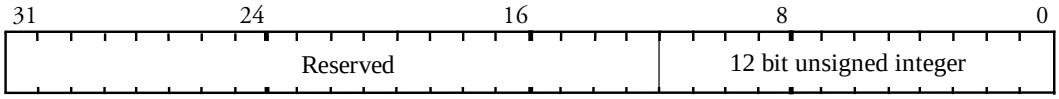


Holds the constant color in either RGBA or raw framebuffer format. This value is used when the **ColorDDAMode** register is set to flat shading mode.

The internal color format will interpret the 8 bit fields as either 5.3 fixed point for 3D operations or 8 bit integer for 2D operations. In CI mode the color index is placed in bits 0-7. If a component has less than 8 bits, it should be left justified and the unused bits set to zero.

Continue

Name:	Continue	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8058
	Tag:	0x000B
Reset Value:	Undefined	
Write		

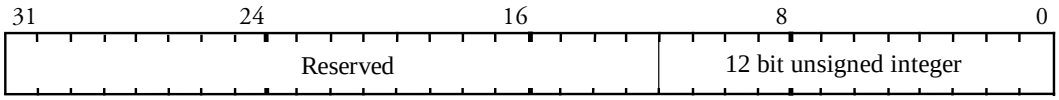


This command causes rasterization to continue after new delta value(s) have been loaded, but does not cause either of the trapezoid's edge DDAs to be reloaded.

The data field holds the number of scanlines to fill. Note this count does not get loaded into the **Count** register.

ContinueNewDom

Name:	Continue - New Dominant Edge	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8048
	Tag:	0x0009
Reset Value:	Undefined	
Write		



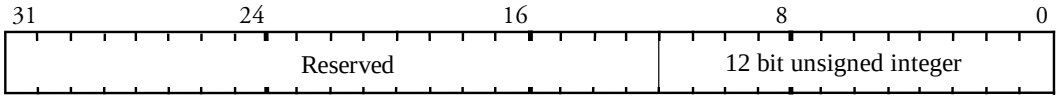
This command causes rasterization to continue with a new dominant edge. The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids and continuity maintained across boundaries.

Since this command only affects the Rasterizer DDA (and not of any other units), it is not suitable for 3D operations.

The data field holds the number of scanlines to fill. Note this count does not get loaded into the **Count** register.

ContinueNewLine

Name:	Continue - New Line Segment	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8040
	Tag:	0x0008
Reset Value:	Undefined	Write



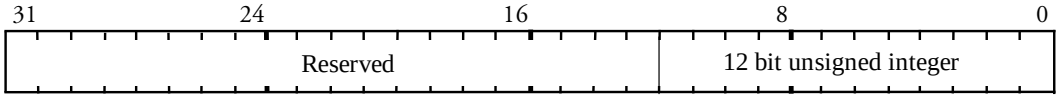
This command causes rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, however the fraction bits in the DDAs can be kept, set to zero, one half, or nearly one half, under control of the **RasterizerMode** register.

The data field holds the number of pixels in a line. Note this count does not get loaded into the **Count** register.

The use of **ContinueNewLine** is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments.

ContinueNewSub

Name:	Continue - New SubordinateEdge	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8050
	Tag:	0x000A
Reset Value:	Undefined	
Write		

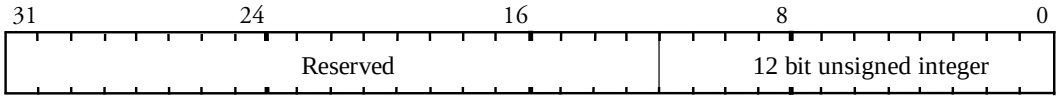


This command causes rasterization to continue with a new subordinate edge. The subordinate DDA is reloaded with the new parameters. The dominant edge is carried on from the previous trapezoid. This is very useful when scan converting triangles with a 'knee' (i.e. two subordinate edges).

The data field holds the number of scanlines to fill. Note this count does not get loaded into the **Count** register.

Count

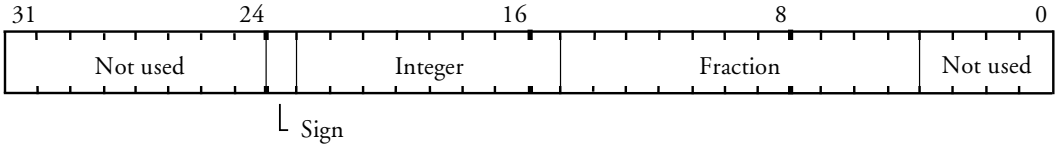
Name:	Count	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8030
	Tag:	0x0006
Reset Value:	Undefined	
Read/write		



Interpretation of contents is dependent on the mode set in the **Render** command i.e. it specifies the number of pixels in a line, or the number of scanlines in a trapezoid.

dBdx

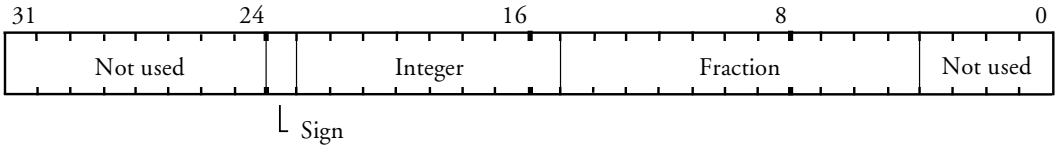
Name:	X Derivative - Blue	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87B8
	Tag:	0x00F7
Reset Value:	Undefined	
Read/write		



This register is used to set the X derivative for the Blue value for the interior of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

dBdyDom

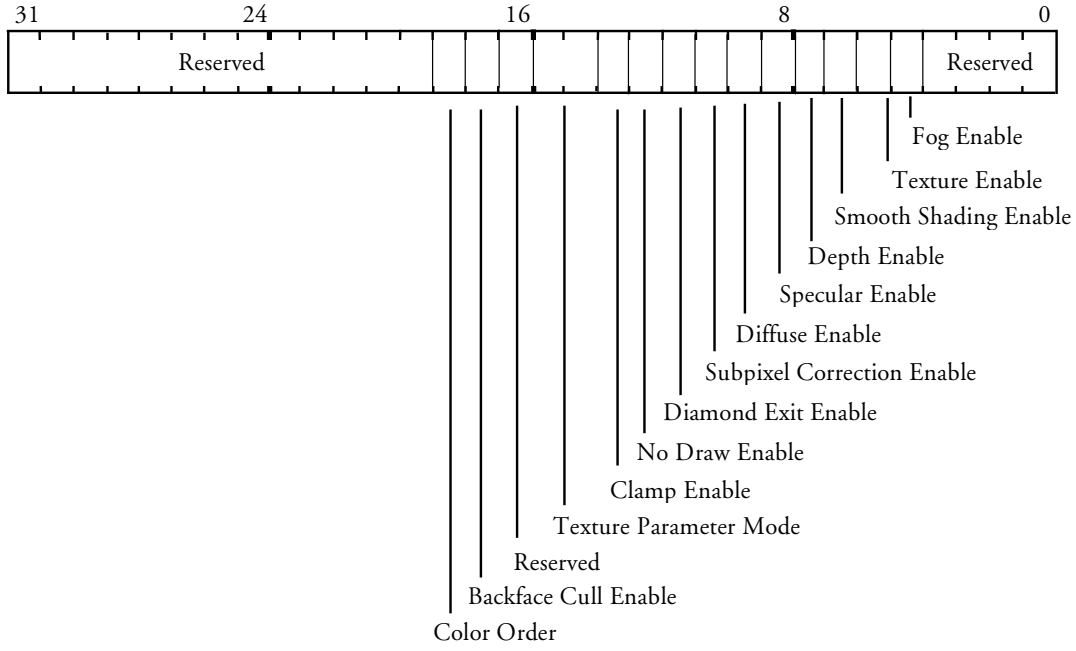
Name:	Y Derivative Dominant - Blue	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.87C0
	Tag:	0x00F8
Reset Value:	Undefined	
Read/write		



This register is used to set the Y derivative dominant for the Blue value along a line, or the dominant edge of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

DeltaMode

Name:	Delta Mode	
Unit:	Delta	
Region: 0	Offset:	0x0000.9300
	Tag:	0x00260
Reset Value:	Undefined	
Read/write		

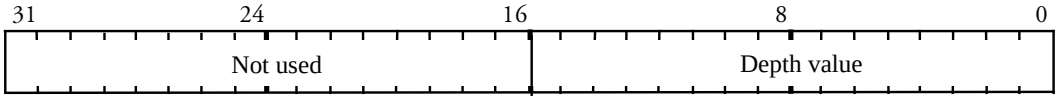


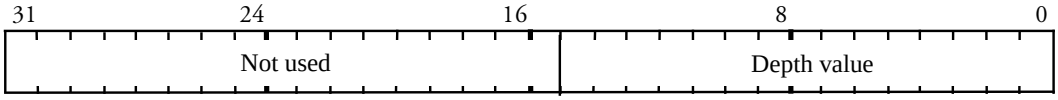
Bit4	FogEnable: This field is qualified by the FogEnable bit in the Draw command. 0 = Disable 1 = Enable
Bit5	TextureEnable: This field is qualified by the TextureEnable bit in the Draw command. 0 = Disable 1 = Enable
Bit 6	SmoothShadingEnable 0 = Disable 1 = Enable
Bit 7	DepthEnable 0 = Disable 1 = Enable

Bit 8	SpecularTextureEnable 0 = Disable 1 = Enable
Bit 9	DiffuseTextureEnable 0 = Disable 1 = Enable
Bit 10	SubPixelCorrectionEnable: This is qualified by the SubPixelCorrectionEnable in the Draw command. 0 = Disable 1 = Enable
Bit 11	DiamondExit 0 = Disable 1 = Enable
Bit 12	NoDraw: When set prevents a Render command from being sent to the rendering devices. This field only affects the Draw commands. This field allows the host to alter the set-up parameters before sending a Render command. 0 = Disable 1 = Enable
Bit 13	ClampEnable: When set causes the input values to be clamped to a parameter specific range. Note that the texture parameters are not affected by this field. 0 = Disable 1 = Enable
Bit 14, 15	TextureParameterMode: 0: Used as given 1: Clamped to lie in the range -1.0 to 1.0 2: Normalize to lie in the range -1.0 to 1.0
Bit 17	BackFaceCull 0 = Disable 1 = Enable
Bit 18	ColorOrder: Specifies order of colors in V*PackedColor messages. Bit 31 Bit 0 0 = Alpha, Blue, Green, Red 1 = Alpha, Red, Green, Blue Each color component is 8 bits.

Depth

Name:	Depth	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89A8
	Tag:	0x0135
Reset Value:	Undefined	
Read/write		





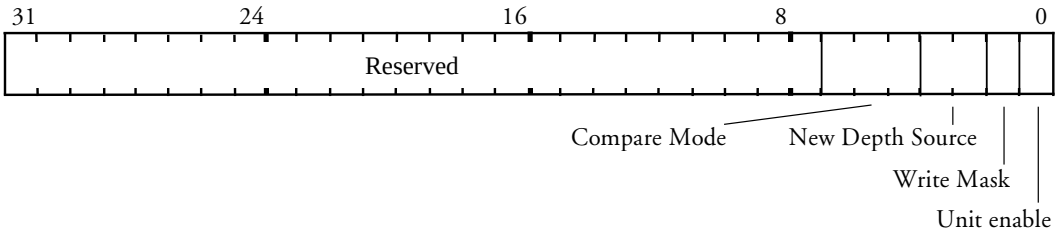
Holds an externally sourced 16 or 15 bit depth value. The unused most significant bits should be set to zero.

This is used in the draw pixels function where the host supplies the depth values through the **Depth** register.

Alternatively this is used when a constant depth value is needed, for example, when clearing the depth buffer, or for 2D rendering where the depth is held constant.

DepthMode

Name:	Depth Mode	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89A0
	Tag:	0x0134
Reset Value:	Undefined	
Read/write		



Controls the comparison of a fragment's depth value and updating of the depth buffer. If the compare function is LESS and the result is true then the fragment value is less than the source value.

Bit0 Unit Enable:
- 0 = Disable
- 1 = Enable

Bit1 Writemask:
- 0 = Disable write to depth buffer
- 1 = Enable write to depth buffer

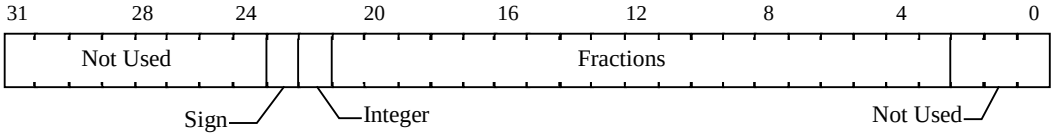
Bit2-3 Source of depth value for comparison:
- 0 = Fragment's depth value
- 1 = LBData -
  for copy pixels when destination depth planes are not updated.
- 2 = Depth register
- 3 = LBSourceData -
  for copy pixels when destination depth planes are updated.

Bit4-6 Comparison function:

- 0 = NEVER
- 1 = LESS
- 2 = EQUAL
- 3 = LESS OR EQUAL
- 4 = GREATER
- 5 = NOT EQUAL
- 6 = GREATER OR EQUAL
- 7 = ALWAYS

dFdx

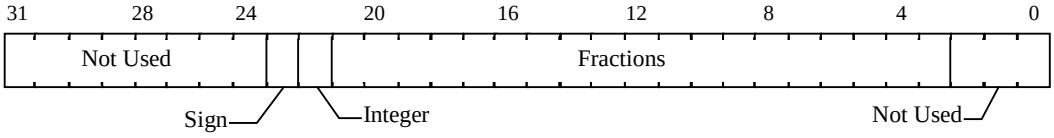
Name:	X Derivative - Fog	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86A8
	Tag:	0x00D5
Reset Value:	Undefined	
Read/write		



Fog coefficient derivative per unit X for use in rendering trapezoids. The value is in 2's complement 2.19 fixed point format.

dFdyDom

Name:	Y Derivative Dominant - Fog	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86B0
	Tag:	0x00D6
Reset Value:	Undefined	
Read/write		



Fog coefficient derivative per unit Y along a line, or the dominant edge of a trapezoid. The value is in 2's complement 2.19 fixed point format.

dGdx

Name: X Derivative - Green

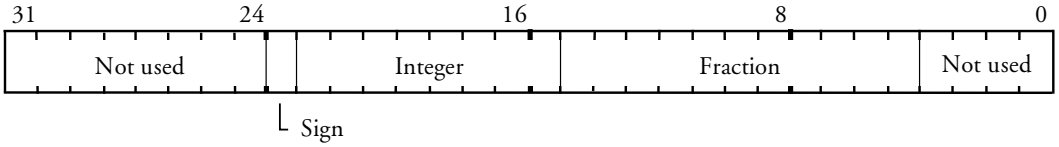
Unit: Color DDA

Region: 0 Offset: 0x0000.87A0

Tag: 0x00F4

Reset Value: Undefined

Read/write



This register is used to set the X derivative for the Green value for the interior of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

Name: Y Derivative Dominant - Green

Unit: Color DDA

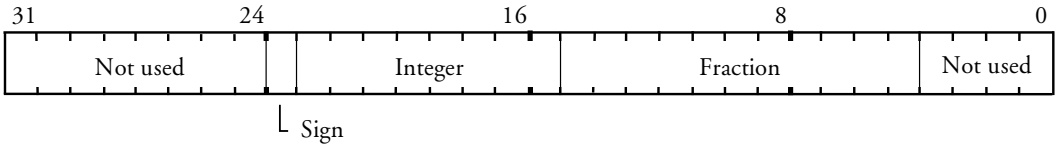
dGdyDom

Region: 0 Offset: 0x0000.87A8

Tag: 0x00F5

Reset Value: Undefined

Read/write

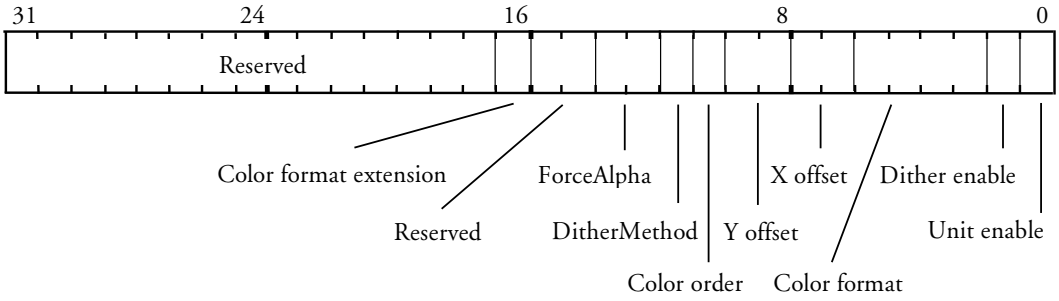


This register is used to set the Y derivative dominant for the Green value along a line, or the dominant edge of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

DitherMode

Name:	Dither Mode	
Unit:	Color Format	
Region: 0	Offset:	0x0000.8818
	Tag:	0x0103
Reset Value:	Undefined	

Read/write



Controls the Color Format unit.

Bit0 Unit Enable:
0 = Disable
1 = Enable

Bit1 Dither Enable:
0 = Disable
1 = Enable

(see overleaf for description of the remaining bits)

Bit2-5 Color Format:

			Internal Color Channel			
Format	Color Order	Name	R	G	B	A
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0
9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0
17	RGB	5:6:5 Back	5@27	6@21	5@16	0

Notes: The format column is also dependent on bit16. n@m means n bits starting at bit m. Front and Back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the 7 bit formatted value. If the format has no alpha bits, the alpha field defaults to 0xF8

Bit6-7 XOffset to enable window relative dithering.

Bit8-9 YOffset to enable window relative dithering.

Bit10 Color Order:
- 0 = BGR
- 1 = RGB

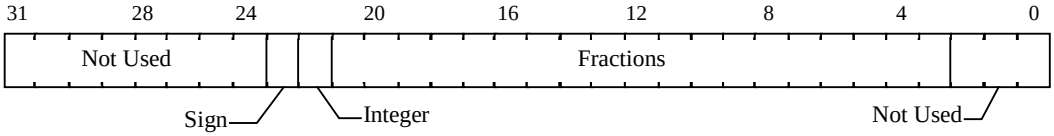
Bit11 Dither Method:
- 0 = Ordered
- 1 = Line

Bit12-13 ForceAlpha:
- 0 = Disable
- 1 = Force to 0
- 2 = Force to 0xF8

Bit16 Color Format Extension. Most significant bit extension to Color Format held in bits 2-5

dKddx

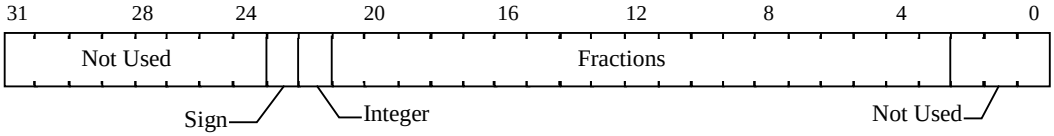
Name:	X Derivative - Kd	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86E8
	Tag:	0x00DD
Reset Value:	Undefined	
Read/write		



Diffuse light coefficient derivative per unit X for use in rendering texture mapped trapezoids using ramp application mode. The value is in 2's complement 2.19 fixed point format.

dKddyDom

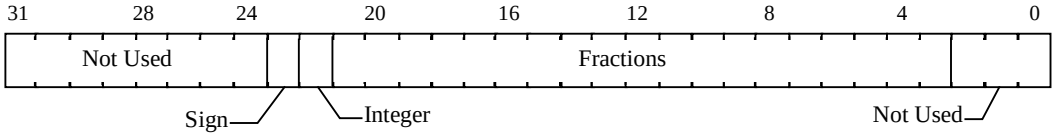
Name:	Y Derivative Dominant - Kd	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86F0
	Tag:	0x00DE
Reset Value:	Undefined	
Read/write		



Diffuse light coefficient derivative per unit Y along a line, or for the dominant edge of a trapezoid, for use with ramp texture application mode. The value is in 2's complement 2.19 fixed point format.

dKsdx

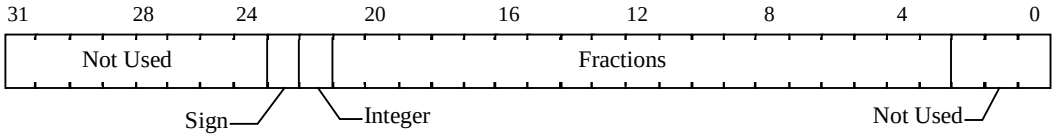
Name:	X Derivative - Ks	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86D0
	Tag:	0x00DA
Reset Value:	Undefined	
Read/write		



Specular light coefficient derivative per unit X for use in rendering texture mapped trapezoids using ramp application mode. The value is in 2's complement 2.19 fixed point format.

dKsdyDom

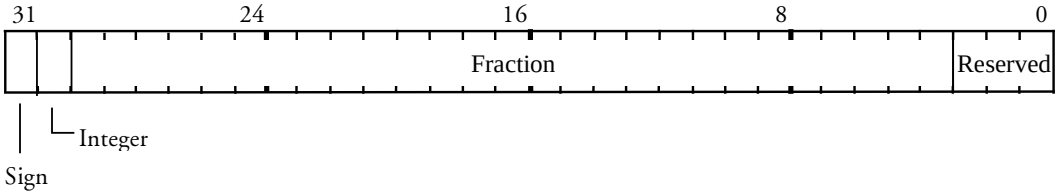
Name:	Y Derivative Dominant - Ks	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86D8
	Tag:	0x00DB
Reset Value:	Undefined	
Read/write		



Specular light coefficient derivative per unit Y along a line, or for the dominant edge of a trapezoid, for use with ramp texture application mode. The value is in 2's complement 2.19 fixed point format.

dQdx

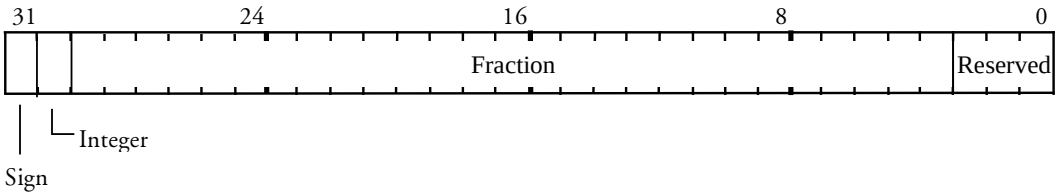
Name:	X Derivative - Homogeneous texture coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83C0
	Tag:	0x0078
Reset Value:	Undefined	
Read/write		



Used to set the X derivative for the Q coordinate when texture mapping. Format is 2's complement 2.27 fixed point.

dQdyDom

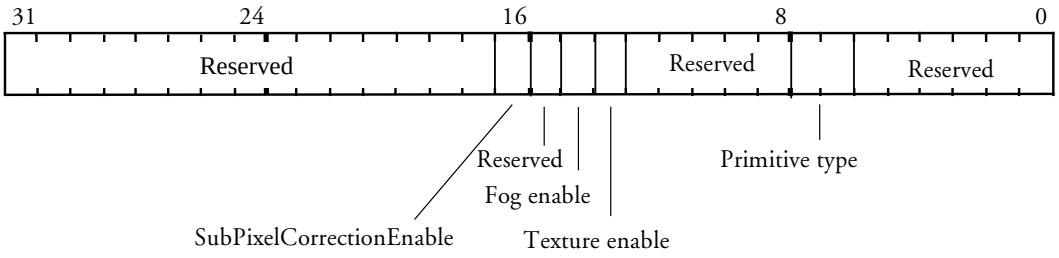
Name:	Y Derivative Dominant - Homogeneous texture coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83C8
	Tag:	0x0079
Reset Value:	Undefined	
Read/write		



Used to set the Y dominant derivative for the Q coordinate when texture mapping. Format is 2's complement 2.27 fixed point.

DrawLine01

Name:	Draw line	
Unit:	Delta	
Region: 0	Offset:	0x0000.9318
	Tag:	0x0263
Reset Value:	Undefined	
Write		



Bit6-7 PrimitiveType These bits indicate the type of PERMEDIA primitive to be drawn. The primitives supported and the corresponding codes are:

0 = lines,
1 = trapezoids,
2 = points,
3 = rectangles.

Bit13 TextureEnable. Note that the Texture Units must be suitably enabled as well for any texturing to occur.

0 = Disable
1 = Enable

Bit14 FogEnable. Note that the Fog Unit must be suitably enabled as well for any fogging to occur.

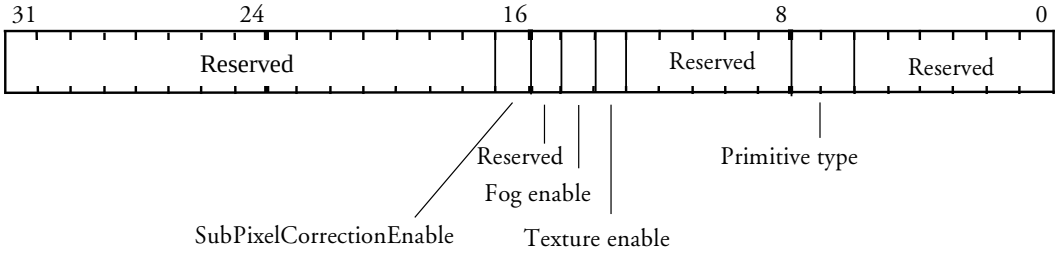
0 = Disable
1 = Enable

Bit16 SubPixelCorrectionEnable. Enables the sub pixel correction of color, depth, fog and texture values at the start of a scanline span.

0 = Disable
1 = Enable

DrawLine10

Name:	Draw line	
Unit:	Delta	
Region: 0	Offset:	0x0000.9320
	Tag:	0x0264
Reset Value:	Undefined	
Write		



Bit6-7 PrimitiveType These bits indicate the type of PERMEDIA primitive to be drawn. The primitives supported and the corresponding codes are:

0 = lines,
1 = trapezoids,
2 = points,
3 = rectangles.

Bit13 TextureEnable. Note that the Texture Units must be suitably enabled as well for any texturing to occur.

0 = Disable
1 = Enable

Bit14 FogEnable. Note that the Fog Unit must be suitably enabled as well for any fogging to occur.

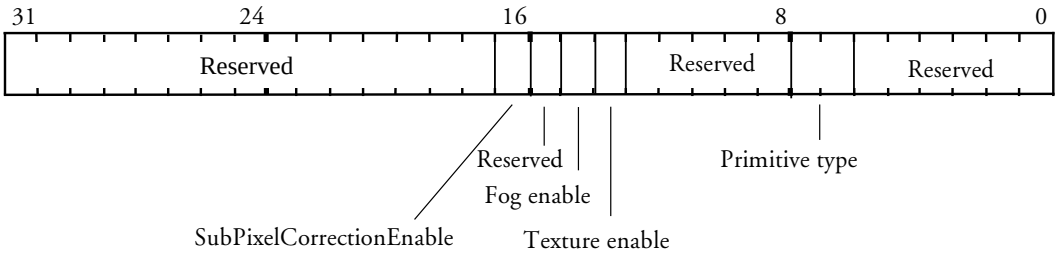
0 = Disable
1 = Enable

Bit16 SubPixelCorrectionEnable. Enables the sub pixel correction of color, depth, fog and texture values at the start of a scanline span.

0 = Disable
1 = Enable

DrawTriangle

Name:	Draw Triangle	
Unit:	Delta	
Region: 0	Offset:	0x0000.9308
	Tag:	0x0261
Reset Value:	Undefined	
Write		



Bit6-7 PrimitiveType These bits indicate the type of PERMEDIA primitive to be drawn. The primitives supported and the corresponding codes are:

0 = lines,
1 = trapezoids,
2 = points,
3 = rectangles.

Bit13 TextureEnable. Note that the Texture Units must be suitably enabled as well for any texturing to occur.

0 = Disable
1 = Enable

Bit14 FogEnable. Note that the Fog Unit must be suitably enabled as well for any fogging to occur.

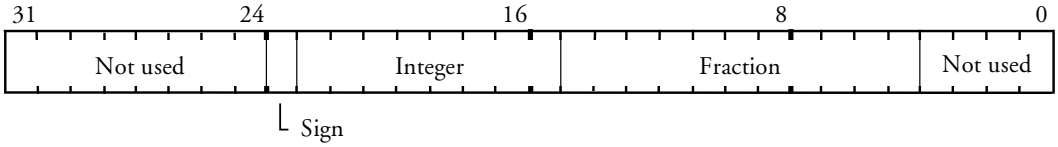
0 = Disable
1 = Enable

Bit16 SubPixelCorrectionEnable. Enables the sub pixel correction of color, depth, fog and texture values at the start of a scanline span.

0 = Disable
1 = Enable

dRdx

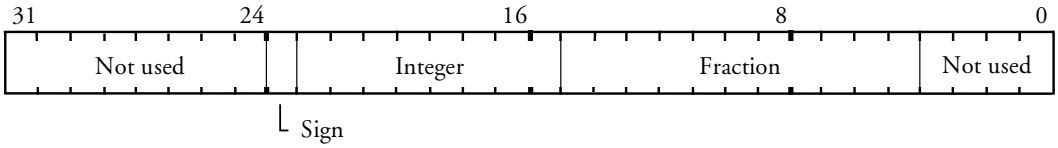
Name:	X Derivative - Red	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.8788
	Tag:	0x00F1
Reset Value:	Undefined	
Read/write		



This register is used to set the X derivative for the Red value for the interior of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

dRdyDom

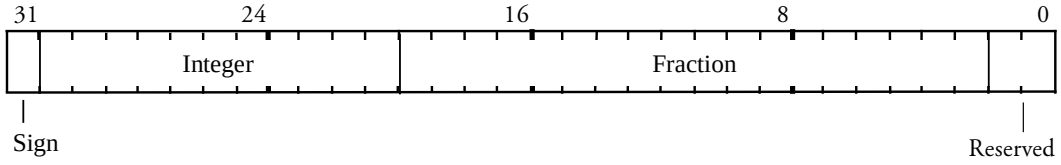
Name:	Y Derivative Dominant - Red	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.8790
	Tag:	0x00F2
Reset Value:	Undefined	
Read/write		



This register is used to set the Y derivative dominant for the Red value along a line, or the dominant edge of a trapezoid when Gouraud shading. The value is 2's complement 9.11 fixed point format.

dSdx

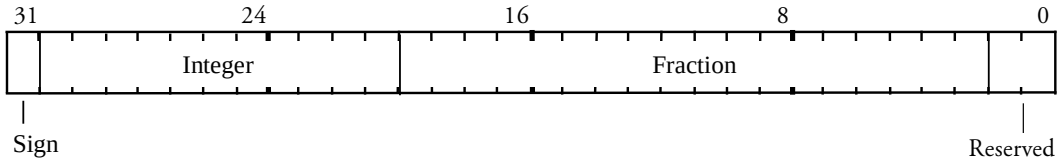
Name:	X Derivative - Texture S coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.8390
	Tag:	0x0072
Reset Value:	Undefined	
Read/write		



Used to set the X derivative for the S coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

dSdyDom

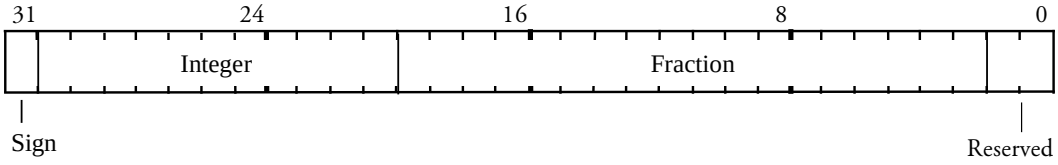
Name:	Y Derivative Dominant - Texture S coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.8398
	Tag:	0x0073
Reset Value:	Undefined	
Read/write		



Used to set the Y dominant derivative for the S coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

dTdx

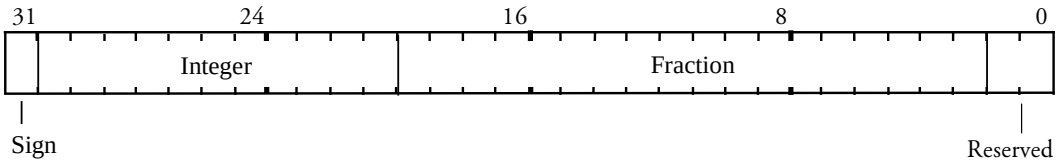
Name:	X Derivative - Texture T coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83A8
	Tag:	0x0075
Reset Value:	Undefined	
Read/write		



Used to set the X derivative for the T coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

dTdyDom

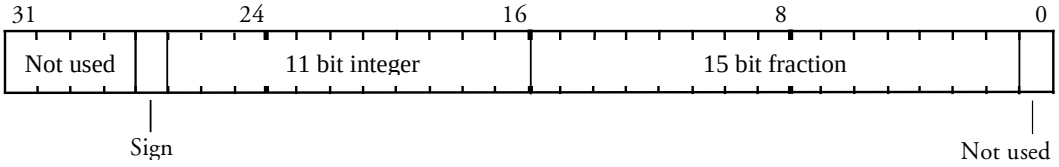
Name:	Y Derivative Dominant - Texture T coordinate	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83B0
	Tag:	0x0076
Reset Value:	Undefined	
Read/write		



Used to set the Y dominant derivative for the T coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

dXDom

Name:	Delta X Dominant	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8008
	Tag:	0x0001
Reset Value:	Undefined	
Read/write		

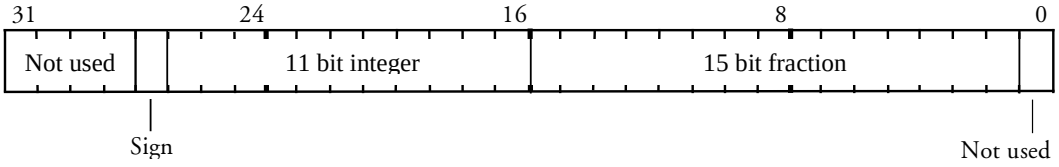


Value added when moving from one scanline to the next for the dominant edge in trapezoid filling. The value is in 2's complement 12.15 fixed point format.

Also holds the change in X when plotting lines. For Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0, depending on the required scanning direction.

dXSub

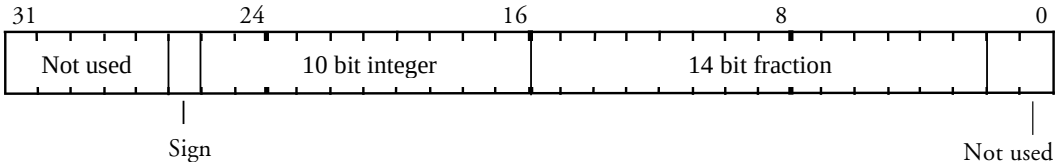
Name:	Delta X Subordinate	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8018
	Tag:	0x0003
Reset Value:	Undefined	
Read/write		



Value added when moving from one scanline to the next for the subordinate edge in trapezoid filling. The value is in 2's complement 12.15 fixed point format.

dY

Name:	Delta Y	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8028
	Tag:	0x0005
Reset Value:	Undefined	
Read/write		



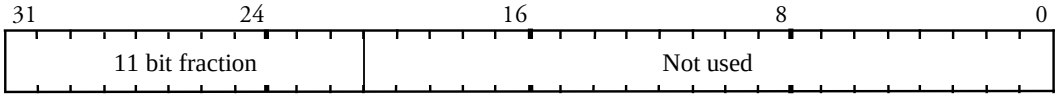
Value added to Y to move from one scanline to the next.

For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0, depending on the required scanning direction. The value is in 2's complement 11.14 fixed point format.

dZdxL

For trapezoids the value will be ±1.0 depending on the scanning direction.

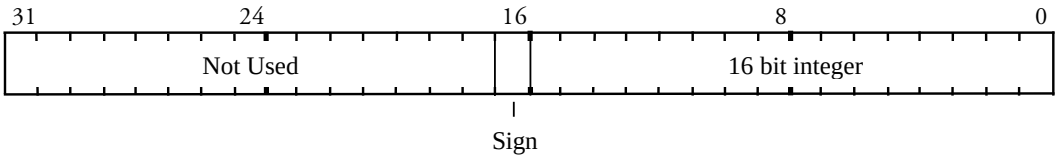
Name:	Depth Derivative X - Lower	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89C8
	Tag:	0x0139
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in X used in rendering trapezoids. **dZdxU** holds the most significant bits, and **dZdxL** the least significant bits. The combined value is in 2's complement 17.11 fixed point format.

dZdxU

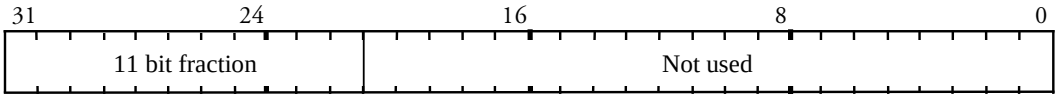
Name:	Depth Derivative X - Upper	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89C0
	Tag:	0x0138
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in X used in rendering trapezoids. **dZdxU** holds the most significant bits, and **dZdxL** the least significant bits. The value is in 2's complement 17.11 fixed point format.

dZdyDomL

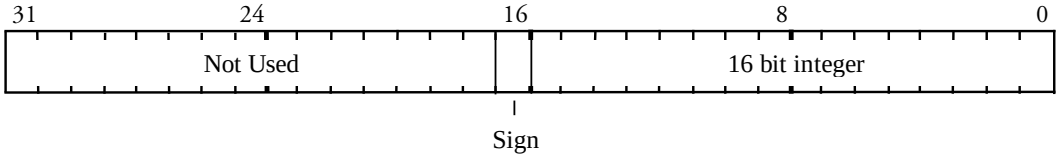
Name:	Depth Derivative Y Dominant - Lower	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89D8
	Tag:	0x013B
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in Y used for the dominant edge of a trapezoid, or along a line. **dZdyDomU** holds the most significant bits, and **dZdyDomL** the least significant bits. The value is in 2's complement 17.11 fixed point format.

dZdyDomU

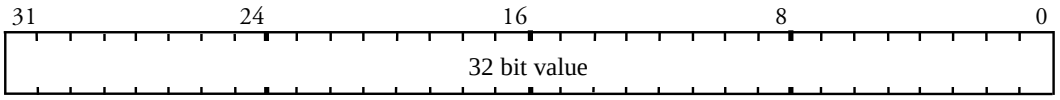
Name:	Depth Derivative Y Dominant - Upper	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89D0
	Tag:	0x013A
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in Y used for the dominant edge of a trapezoid, or along a line. **dZdyDomU** holds the most significant bits, and **dZdyDomL** the least significant bits. The value is in 2's complement 17.11 fixed point format.

FBBlockColor

Name:	Framebuffer Block Fill Color	
Unit:	FramebufferWrite	
Region: 0	Offset:	0x0000.8AC8
	Tag:	0x0159
Reset Value:	Undefined	
Read/write		



Note that this register should not be updated immediately after a **Render** command which performs a block write.

Contains the color (and optionally alpha value) to be written to the framebuffer during block writes. Note the format is the raw data format of the framebuffer.

If the framebuffer is used in 8 bit packed mode, then data should be repeated in all 4 bytes of the register.

If the framebuffer is in 16 bit packed mode then the data must be repeated in both halves of the register.

FBBlockColorL

Name: Framebuffer Block Fill Lower color

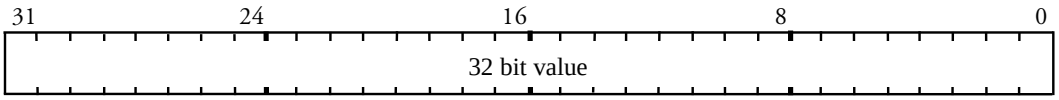
Unit: FramebufferWrite

Region: 0 Offset: 0x0000.8C70

Tag: 0x018E

Reset Value: Undefined

Read/write



Contains the color (and optionally alpha value) to be written to the framebuffer during block writes. Note the format is the raw data format of the framebuffer. Each block fill writes a pattern of 8 bytes defined by these registers, repeating the same data until 32 pixels have been filled.

If the framebuffer is used in 8 bit packed mode, then data should be repeated in all 4 bytes of the register.

If the framebuffer is in 16 bit packed mode then the data must be repeated in both halves of the register.

FBBlockColorU

Name: Framebuffer Block Fill Upper color

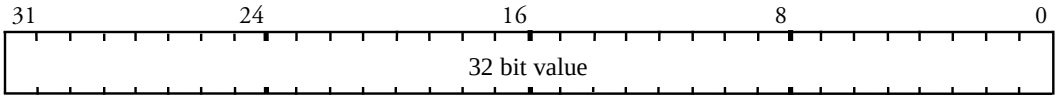
Unit: FramebufferWrite

Region: 0 Offset: 0x0000.8C68

Tag: 0x018D

Reset Value: Undefined

Read/write



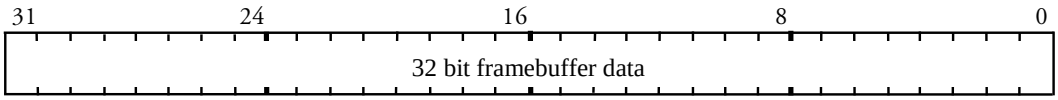
Contains the color (and optionally alpha value) to be written to the framebuffer during block writes. Note the format is the raw data format of the framebuffer. Each block fill writes a pattern of 8 bytes defined by these registers, repeating the same data until 32 pixels have been filled.

If the framebuffer is used in 8 bit packed mode, then data should be repeated in all 4 bytes of the register.

If the framebuffer is in 16 bit packed mode then the data must be repeated in both halves of the register.

FBColor

Name:	Framebuffer Color Upload	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8A98
	Tag:	0x0153
Reset Value:	Undefined	
Read/write		

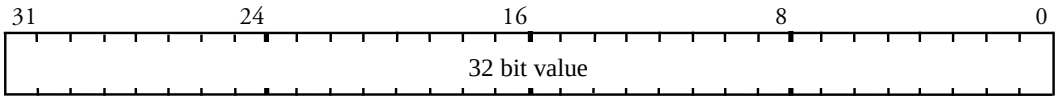


Internal register used in image upload. Note that this register should not be written to. It is documented here to give the format and tag value of the data returned through the Host Out FIFO.

The format is dependent on the raw framebuffer organization and any reformatting which takes place due to the format specified in the **DitherMode** register.

FBData

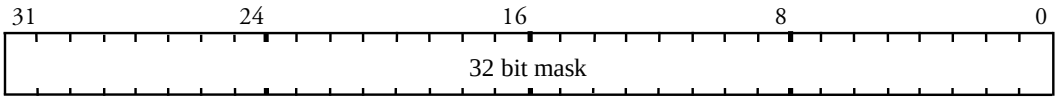
Name:	Framebuffer Data	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AA0
	Tag:	0x0154
Reset Value:	Undefined	
Write		



Supplies the data for image download, where subsequent formatting is required. The formatting can be achieved by means of the **AlphaBlendMode** register to convert to the internal PERMEDIA format, and then via the **DitherMode** register to convert to the required format.

FBHardwareWriteMask

Name:	Hardware Writemask	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AC0
	Tag:	0x0158
Reset Value:	Undefined	
Read/write		



Contains the hardware writemask for the framebuffer. If a bit is set to one then the corresponding bit in the framebuffer is enabled for writing, otherwise it is disabled. Only applicable to configurations where the framebuffer supports a hardware writemask. In cases where it is not supported, this register should NOT be written to.

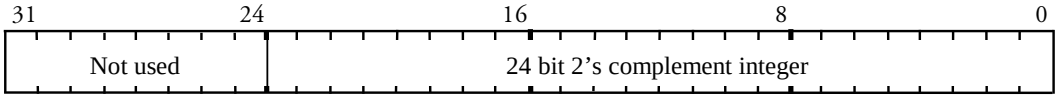
If hardware writemasks are used then all the bits in the **FBSoftwareWriteMask** register must be set to 1, so that software writemasking is disabled.

If the framebuffer is used in 8 bit packed mode, then an 8bit hardware writemask must be repeated in all 4 bytes of the **FBHardwareWriteMask** register.

If the framebuffer is in 16 bit packed mode then the 16 bit hardware writemask must be repeated in both halves of the **FBHardwareWriteMask** register.

FBPixelOffset

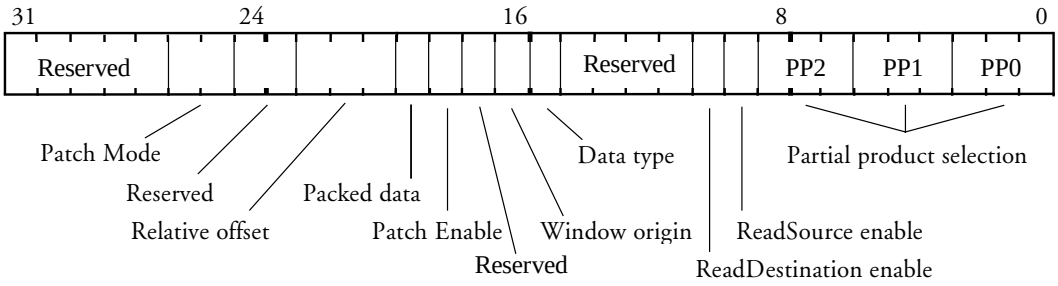
Name:	Framebuffer Pixel Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8A90
	Tag:	0x0152
Reset Value:	Undefined	
Read/write		



Offset between buffers when operating on multiple buffers in the framebuffer at the same time (e.g. left/right/top/bottom in some OpenGL implementations). The offset can be treated as signed or unsigned.

FBReadMode

Name:	Framebuffer Read Mode	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8A80
	Tag:	0x0150
Reset Value:	Undefined	
Read/write		



Controls reading from framebuffer memory.

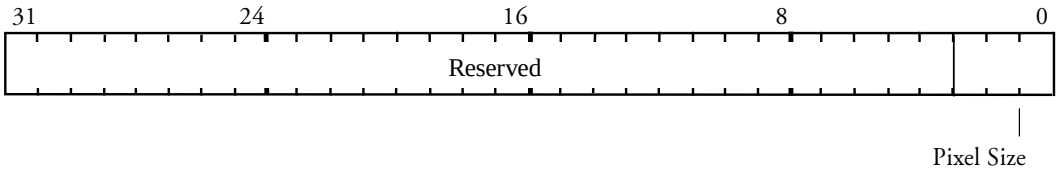
Incorrect data can be read if reads are enabled but the same data had just been written with reads disabled. To avoid this problem, a **WaitForCompletion** command should be sent after enabling reads, but prior to the next primitive.

Bit0-2	Partial Product 0 - See Appendix C for a table of values.
Bit3-5	Partial Product 1 - See Appendix C for a table of values.
Bit6-8	Partial Product 2 - See Appendix C for a table of values.
Bit9	Read Source Enable: 0 = no read 1 = do read
Bit10	Read Destination Enable: 0 = no read 1 = do read
Bit15	Data Type: 0 = FBDefault - for data that may be written back to the framebuffer 1 = FBColor - for image upload
Bit16	Window Origin: 0 = Top left 1 = Bottom left

Bit18	Patch Enable: 0 = Disable 1 = Enable patched addressing for framebuffer accesses
Bit19	PackedData: 0 = Disable. Force PERMEDIA to read one pixel at a time. 1 = Enable. Allow PERMEDIA to read multiple packed pixels when possible.
Bit20-22	RelativeOffset 3 bit 2's compliment value which specifies the number of pixels that the source data has to be adjusted to align with the destination data. The **PackedDataLimits** register also has this field and the last loaded of these two registers takes effect.
Bit25-26	Patch Mode 0 = Patch (suitable for depth buffer patching) 1 = Subpatch (suitable for texture buffer patching) 2 = SubpatchPack (suitable for packed texture patching)

FBReadPixel

Name:	Framebuffer Read Pixel	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AD0
	Tag:	0x015A
Reset Value:	Undefined	
Read/write		



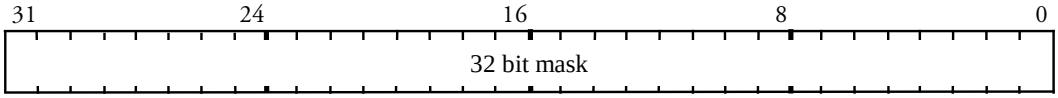
Sets the pixel size for reading from the framebuffer.

Bit0-1 Pixel Size:

0 = 8 bits
1 = 16 bits
2 = 32 bits
3 = reserved
4 = 24 bits

FBSoftwareWriteMask

Name:	Software Writemask	
Unit:	Logic Op	
Region: 0	Offset:	0x0000.8820
	Tag:	0x0104
Reset Value:	Undefined	
Read/write		

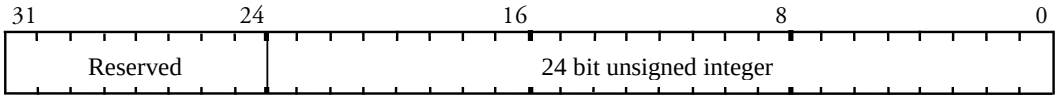


Contains the software writemask for the framebuffer. If a bit is set to one then the corresponding bit in the framebuffer is enabled for writing, otherwise it is disabled. In addition, whenever the writemask is other than all 1s, framebuffer reads must be enabled by setting the ReadSourceEnable bit in the **FBReadMode** register.

If hardware writemasks are used then all the bits in the software writemask must be set to 1, so that software writemasking is disabled.

FBSourceBase

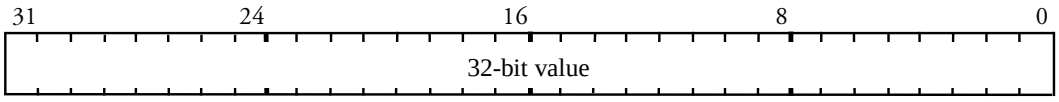
Name:	Base address of source framebuffer data	
Unit:	Framebuffer Read	
Region: 0	Offset:	0x0000.8D80
	Tag:	0x01B0
Reset Value:	Undefined	
Read/write		



The base address of source data for framebuffer copies. Tracks the value of FBWindowBase, so to modify this register it must be loaded after FBWindowBase.

FBSourceData

Name:	Framebuffer Source Data	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AA8
	Tag:	0x0155
Reset Value:	Undefined	
Write		

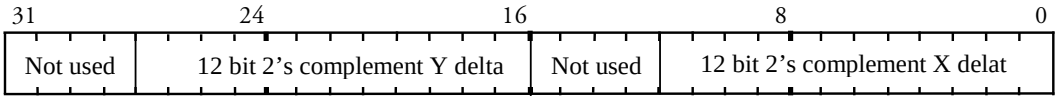


Supplies the data for image download with logic ops, where the data is treated as the source rather than the destination parameter.

The data supplied should be in raw framebuffer format.

FBSourceDelta

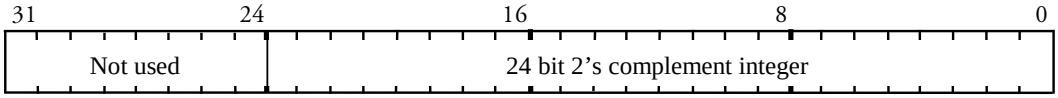
Name:	Difference between destination to source data	
Unit:	Framebuffer Read	
Region: 0	Offset:	0x0000.8D88
	Tag:	0x01B1
Reset Value:	Undefined	
Read/write		



The difference from destination to source data in the framebuffer. Loading this register causes an appropriate value to be calculated and loaded into FBSourceOffset.

FBSourceOffset

Name:	Framebuffer Source Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8A88
	Tag:	0x0151
Reset Value:	Undefined	
Read/write		

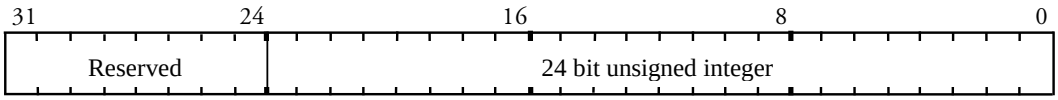


Sets the offset from destination to source for a copy operation in the framebuffer i.e.

source offset = destination address - source address

FBWindowBase

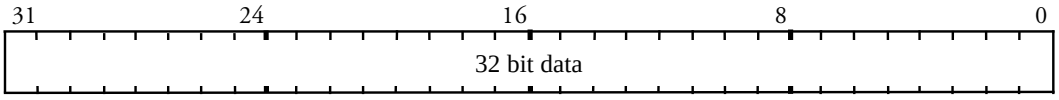
Name:	Framebuffer Window Base	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AB0
	Tag:	0x0156
Reset Value:	Undefined	
Read/write		



Contains the current base address of the window in the framebuffer.

FBWriteData

Name:	Framebuffer Write Data	
Unit:	Logic Op	
Register 0	Offset	0x0000. 8830
	Tag:	0x106
Reset Value:	Undefined	
Read/write		



It is not recommended that this register be used. It is included here for the benefit of understanding legacy PERMEDIA 1 software.

Contains the color value to be written to the framebuffer when the UseConstantFBWriteData bit of the **LogicalOpMode** register is set to one. Note that the following conditions must be met for this mode of rendering to be used:

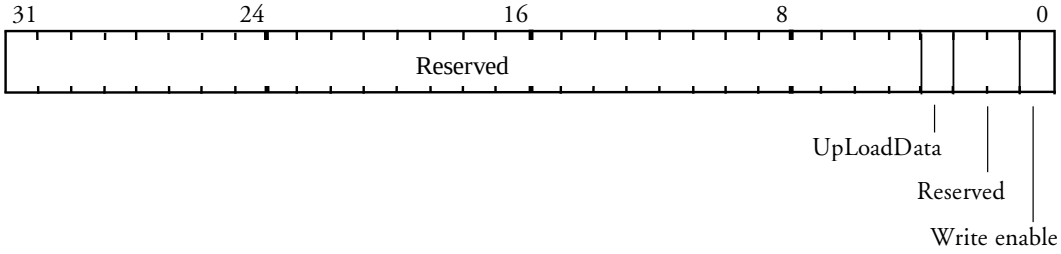
- Flat shaded aliased primitive
- No dithering required
- No logical operation involving a destination factor
- No stencil or depth test
- No texture, fog or alpha blending
- No software writemasking

The data is in the raw format of the framebuffer. If the pixel size is 8 bits then the data should be repeated in all four bytes. If the pixel size is 16 bits the data should be repeated in both halves of the word.

Hardware writemasks can be used if available.

FBWriteMode

Name:	Framebuffer Write Mode	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8AB8
	Tag:	0x0157
Reset Value:	Undefined	
Read/write		



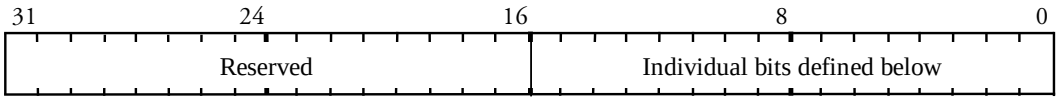
Controls writing to the framebuffer.

Bit0 Write Enable:
- 0 = Disable
- 1 = Enable

Bit3 UpLoadData:
- 0 = No upload
- 2 = Upload color to host

FilterMode

Name:	Filter Mode	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C00
	Tag:	0x0180
Reset Value:	Undefined	
Read/write		



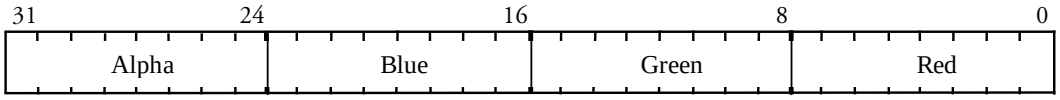
Controls culling of information from the output FIFO. If both tag and data are specified then the tag is always the first word in the FIFO.

Bit0-3	Reserved for future use - set to zero.
Bit4	Depth Tag Filter: Used in-depth buffer image upload. 0 = Cull Depth Tags from being passed to output FIFO 1 = Pass Depth Tags to output FIFO
Bit5	Depth Data Filter: Used in-depth buffer image upload 0 = Cull Depth data values from being passed to output FIFO 1 = Pass Depth data values to output FIFO
Bit6	Stencil Tag Filter: Used in Stencil buffer image upload 0 = Cull Stencil Tags from being passed to output FIFO 1 = Pass Stencil Tags to output FIFO
Bit7	Stencil Data Filter: Used in Stencil buffer image upload 0 = Cull Stencil data values from being passed to output FIFO 1 = Pass Stencil data values to output FIFO
Bit8	Color Tag Filter: Used in Framebuffer image upload 0 = Cull Color Tags from being passed to output FIFO 1 = Pass Color Tags to output FIFO
Bit9	Color Data Filter: Used in Framebuffer image upload 0 = Cull Color data values from being passed to output FIFO 1 = Pass Color data values to output FIFO
Bit10	Synchronization Tag Filter: 0 = Cull Synchronization Tags from being passed to output FIFO 1 = Pass Synchronization Tags to output FIFO
Bit11	Synchronization Data Filter: 0 = Cull Synchronization data values from being passed to output FIFO 1 = Pass Synchronization data values to output FIFO

Bit12	Statistics Tag Filter: Used in Picking and Extent read back 0 = Cull Statistics Tags from being passed to output FIFO 1 = Pass Statistics Tags to output FIFO
Bit13	Statistics Data Filter: Used in Picking and Extent read back 0 = Cull Statistics data values from being passed to output FIFO 1 = Pass Statistics data values to output FIFO
Bit14-15	Reserved for future use - set to zero.

FogColor

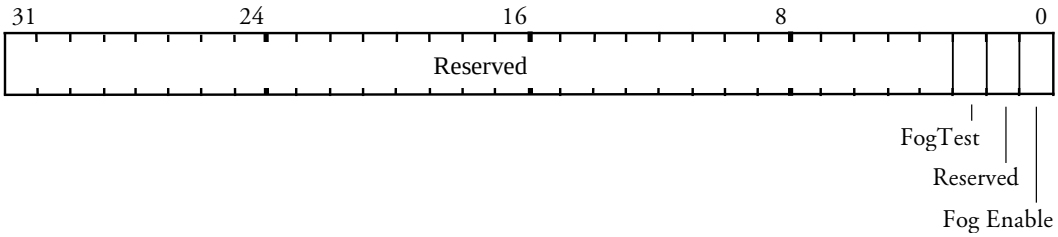
Name:	Fog Color	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.8698
	Tag:	0x00D3
Reset Value:	Undefined	
Read/write		



Provides the color to be blended with the fragment's color when fogging is enabled.

FogMode

Name:	Fog Mode	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.8690
	Tag:	0x00D2
Reset Value:	Undefined	
Read/write		



Controls operation of the Fog unit.

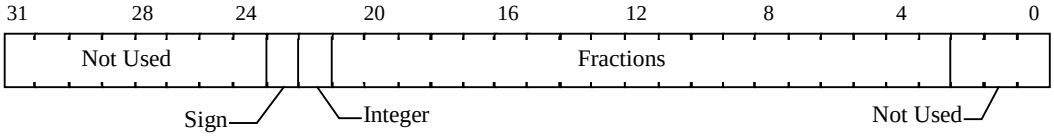
Enabling FogTest causes fragments with negative fog values to be rejected.

Note that the FogEnable bit in the **Render** command must be set for fogging to be applied to a primitive.

Bit0	Enable Fog: 0 = Disable 1 = Enable
Bit2	Fog Test: 0 = Disable 1 = Enable

FStart

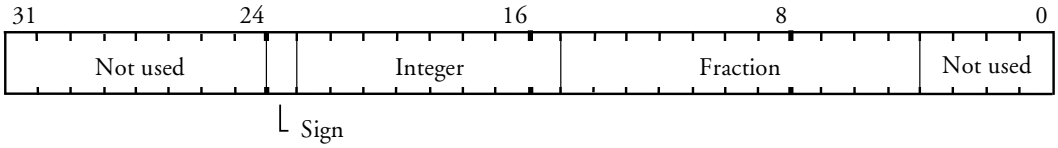
Name:	Initial Fog Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86A0
	Tag:	0x00D4
Reset Value:	Undefined	
Read/write		



Fog coefficient start value. Note the interpolation coefficient is used to blend the fragment's color with the color in the **FogColor** register. The value is in 2's complement 2.19 fixed point format.

GStart

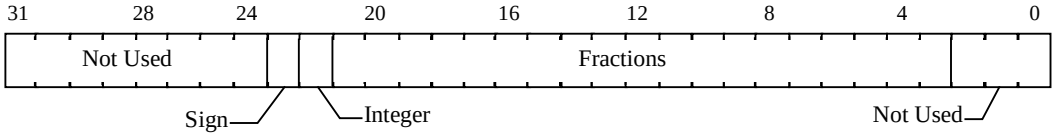
Name:	Initial Green Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.8798
	Tag:	0x00F3
Reset Value:	Undefined	
Read/write		



This register is used to set the initial value for the Green value for a vertex when in Gouraud shading mode. The value is 2's complement 9.11 fixed point format.

KdStart

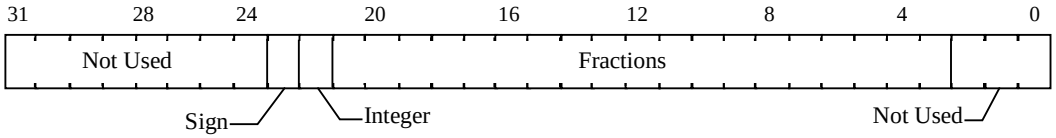
Name:	Initial Kd Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86E0
	Tag:	0x00DC
Reset Value:	Undefined	
Read/write		



Start value for diffuse light parameter when texture mapping using ramp application mode. The value is in 2's complement 2.19 fixed point format.

KsStart

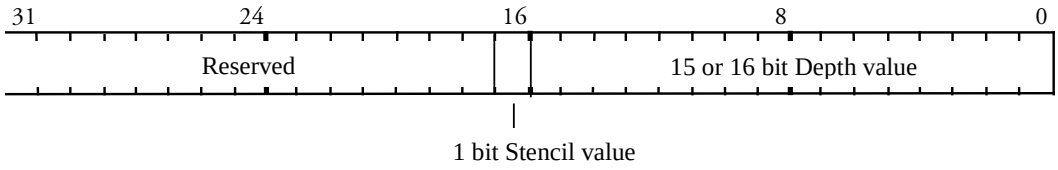
Name:	Initial Ks Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.86C8
	Tag:	0x00D9
Reset Value:	Undefined	
Read/write		



Start value for specular light parameter when texture mapping using ramp application mode. The value is in 2's complement 2.19 fixed point format.

LBData

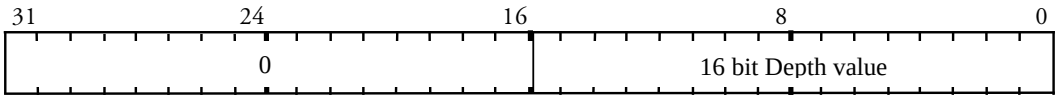
Name:	Localbuffer Data Download	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.8898
	Tag:	0x0113
Reset Value:	Undefined	
Write		



Used to download depth and/or stencil data to localbuffer memory. Data should be supplied in the raw localbuffer format.

LBDepth

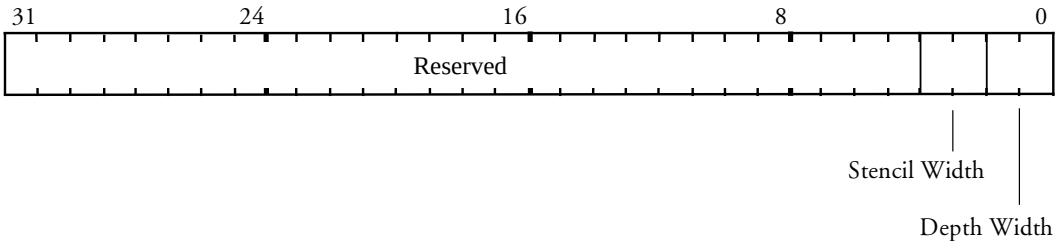
Name:	Localbuffer Depth Upload	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.88B0
	Tag:	0x0116
Reset Value:	Undefined	
Read/write		



Used to upload depth data from localbuffer memory. This register should not be written to. It is documented here to give the tag value and format of the data when read from the Host Out FIFO. If the depth buffer is less than 16 bits, the depth value is right justified and zero extended

LBReadFormat

Name:	Localbuffer Read Format	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.8888
	Tag:	0x0111
Reset Value:	Undefined	
Read/write		



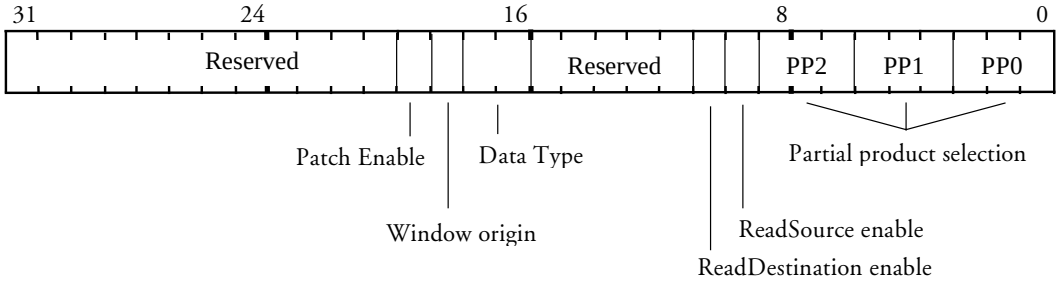
Specifies the format used when reading from localbuffer memory. The effect of creating a format with overlapping fields is undefined. There is no need to synchronize PERMEDIA before changing this register.

Bit0-1 Depth Width:
- 0 = 16
- 1 = reserved
- 2 = reserved
- 3 = 15

Bit2-3 Stencil Width:
- 0 = 0
- 1 = reserved
- 2 = reserved
- 3 = 1

LBReadMode

Name:	Localbuffer Read Mode	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.8880
	Tag:	0x0110
Reset Value:	Undefined	
Read/write		



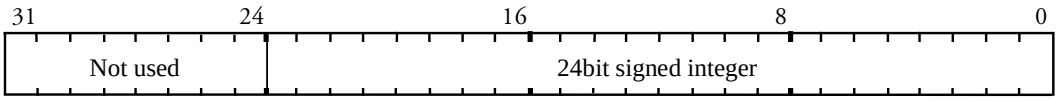
Controls reading from localbuffer memory.

Incorrect data can be read if reads are enabled but the same data had just been written with reads disabled. To avoid this problem, a **WaitForCompletion** command should be sent after enabling reads, but prior to the next primitive.

Bit0-2	Partial Product 0 - See Appendix C for a table of values
Bit3-5	Partial Product 1 - See Appendix C for a table of values
Bit6-8	Partial Product 2 - See Appendix C for a table of values
Bit9	Read Source Enable: 0 = no read 1 = do read
Bit10	Read Destination Enable: 0 = no read 1 = do read
Bit16-17	Data Type: 0 = Default 1 = Localbuffer Stencil 2 = Localbuffer Depth
Bit18	Window Origin: 0 = Top left 1 = Bottom left
Bit19	Patch Enable 0 = Disable 1 = Enable patched addressing of the localbuffer

LBSourceOffset

Name:	Localbuffer Source Offset	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.8890
	Tag:	0x0112
Reset Value:	Undefined	
Read/write		

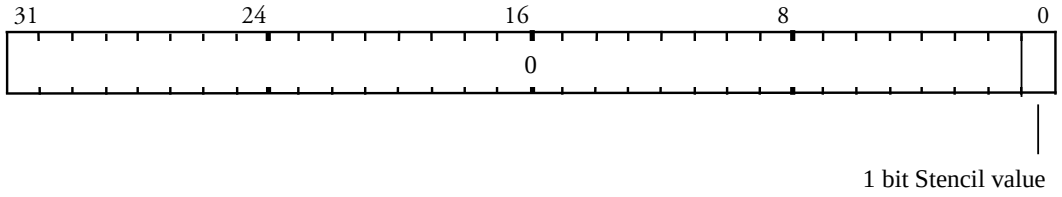


Sets the offset from destination to source for a copy operation in the localbuffer, i.e.:

source offset = destination address - source address

LBStencil

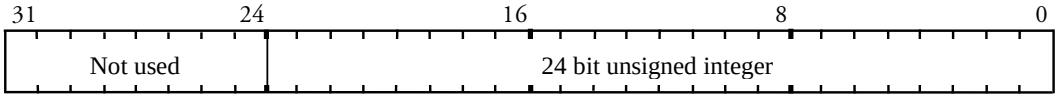
Name:	Localbuffer Stencil Upload	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.88A8
	Tag:	0x0115
Reset Value:	Undefined	
Read/Output		



Used to upload stencil data from localbuffer memory. This register should not be written to. It is documented here to give the tag value and format of the data when read from the Host Out FIFO.

LBWindowBase

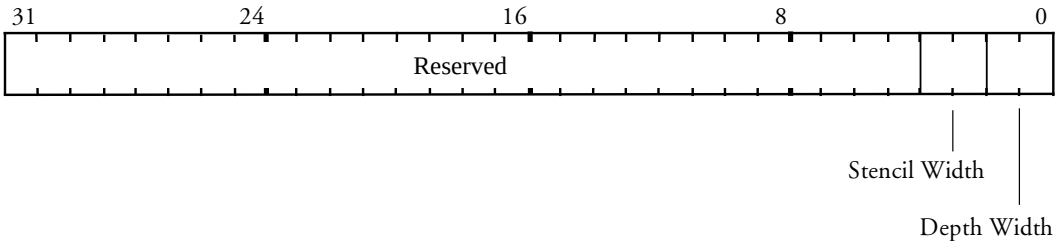
Name:	Localbuffer Window Base	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.88B8
	Tag:	0x0117
Reset Value:	Undefined	Read/write



Contains the current base address of the window in the localbuffer.

LBWriteFormat

Name:	Localbuffer Write Format	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.88C8
	Tag:	0x0119
Reset Value:	Undefined	
Read/write		



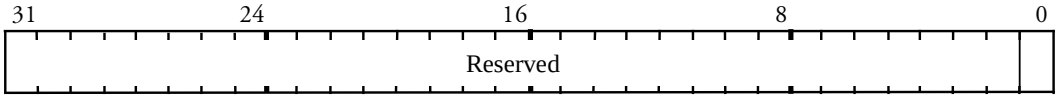
Specifies the format used when writing to localbuffer memory. The effect of setting a configuration with overlapping fields is undefined.

Bit0-1 Depth Width:
- 0 = 16
- 1 = reserved
- 2 = reserved
- 3 = 15

Bit2-3 Stencil Width:
- 0 = 0
- 1 = reserved
- 2 = reserved
- 3 = 1

LBWriteMode

Name:	Localbuffer Write Mode	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0x0000.88C0
	Tag:	0x0118
Reset Value:	Undefined	
Read/write		



Write Enable

Controls writing to the localbuffer.

Bit0	Write Enable: 0 = Disable 1 = Enable

LogicalOpMode

Name: Logic Op Mode

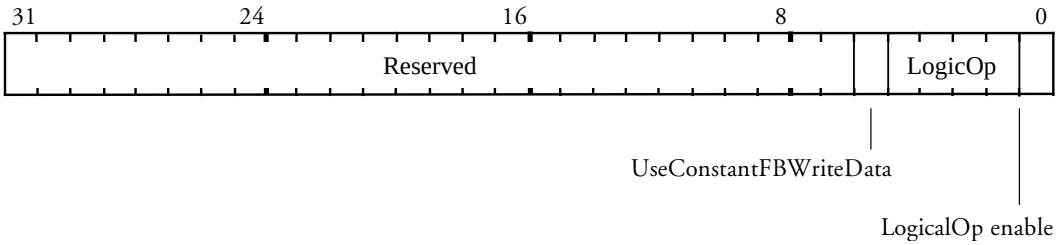
Unit: Logic Op

Region: 0 Offset: 0x0000.8828

Tag: 0x0105

Reset Value: Undefined

Read/write



Controls Logical Operations on the framebuffer.

The UseConstantFBWriteData bit when set to one, causes the color value in the **FBWriteData** register to be written to the framebuffer, rather than the fragment's color. This can achieve higher bandwidth into the framebuffer for flat shaded primitives, but may only be used when LogicalOps are disabled (bit 0 cleared to 0)

Bit0 Logic Op Enable:
0 = Disable
1 = Enable

Bit1-4 Logic Op:

Mode	Name	Operation
0	CLEAR	0
1	AND	S & D
2	AND REVERSE	S & ~D
3	COPY	S
4	AND INVERTED	~S & D
5	NO-OP	D
6	XOR	S ^ D
7	OR	S | D

Mode	Name	Operation
8	NOR	~(S | D)
9	EQUIV	~(S ^ D)
10	INVERT	~D
11	OR REVERSE	S | ~D
12	COPY INVERT	~S
13	OR INVERT	~S | D
14	NAND	~(S & D)
15	SET	1

Where: S = Source (fragment) color, D = Destination (framebuffer) color.

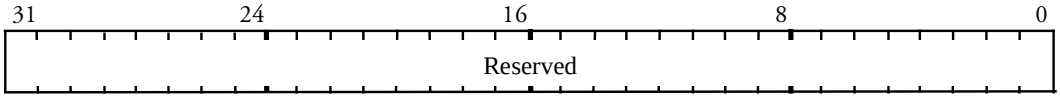
Bit5 UseConstantFBWriteData:
0 = Variable
1 = Constant

MaxHitRegion

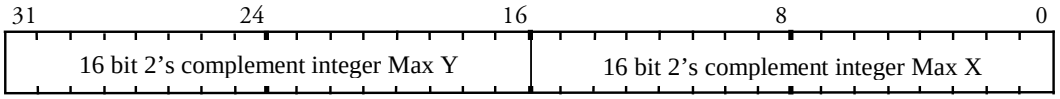
Name:	Max Hit Region	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C30
	Tag:	0x0186
Reset Value:	Undefined	

Write

The format of the data input is:



The format of the data output is:

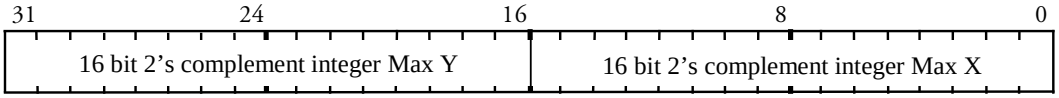


This command causes the maximum coordinates of the hit region to be passed to the Host Out FIFO, unless culled by the statistics bits in the FilterMode register.

The corresponding tag value output is: 0x186

MaxRegion

Name:	Max Region	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C18
	Tag:	0x0183
Reset Value:	Undefined	
Read/write		



This register has two uses:

1. During Picking it contains the maximum (X,Y) value for the pick region.

2. During Extent collection, it is set to the initial minimum (X,Y) extent, and thereafter will be updated whenever an eligible fragment is generated which has a higher X or Y value, with that higher value. Note eligible fragments can be either those that are written as pixels OR those that were rasterized, but were culled from being drawn, as controlled by the **StatisticMode** register.

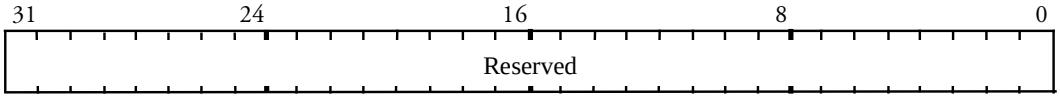
This register is unusual in that its contents are updated by PERMEDIA during rendering, and so if read back, will not necessarily be the same as when originally stored.

MinHitRegion

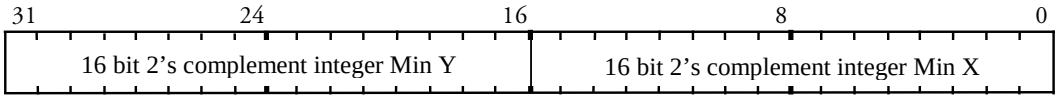
Name:	Min Hit Region	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C28
	Tag:	0x0185
Reset Value:	Undefined	

Write

The format of the data input is:



The format of the data output is:

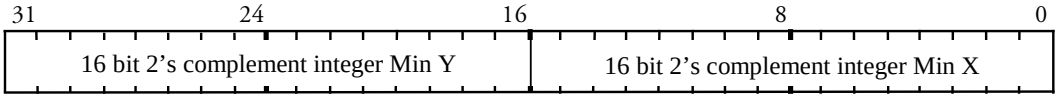


This command causes the minimum coordinates of the hit region to be passed to the Host Out FIFO, unless culled by the statistics bits in the **FilterMode** register.

The corresponding tag value output is: 0x185

MinRegion

Name:	Min Region	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C10
	Tag:	0x0182
Reset Value:	Undefined	
Read/write		



This register has two uses:

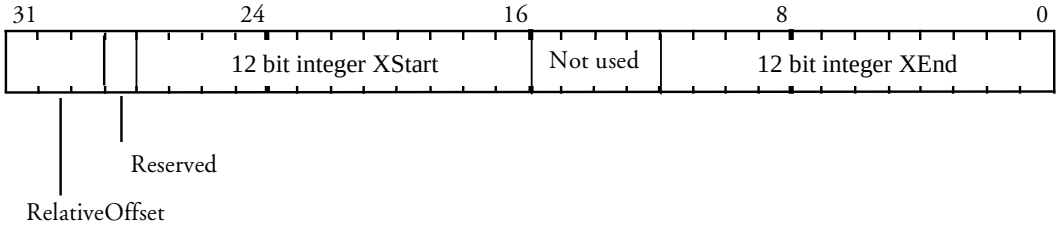
1. During Picking it contains the minimum (X,Y) value for the pick region.

2. During Extent collection, it is set to the initial maximum (X,Y) extent, and thereafter will be updated whenever an eligible fragment is generated which has a lower X or Y value, with that lower value. Note eligible fragments can be either those that are written as pixels OR those that were rasterized, but were culled from being drawn, as controlled by the **StatisticMode** register.

This register is unusual in that its contents are updated by PERMEDIA during rendering, and so if read back, will not necessarily be the same as when originally stored.

PackedDataLimits

Name:	Packed copy limits	
Units:	Framebuffer R/W	
Region: 0	Offset:	0x0000.8150
	Tag:	0x002A
Reset Value:	Undefined	
Read/write		



Sets the start and end limits in X for packed copies. Any pixels lying outside the specified range are not plotted. This test is only active when the PackedData bit in **FBReadMode** is enabled.

Bit0-11	XEnd: 12 bit 2's complement value
Bit16-27	XStart: 12 bit 2's complement value
Bit29-31	Relative Offset:

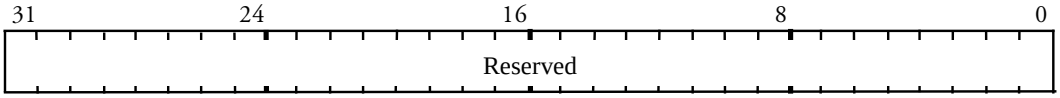
3 bit 2's compliment value which specifies the number of pixels that the source data has to be adjusted to align with the destination data. The **FBReadMode** register also has this field and the last loaded of these two registers takes effect.

PickResult

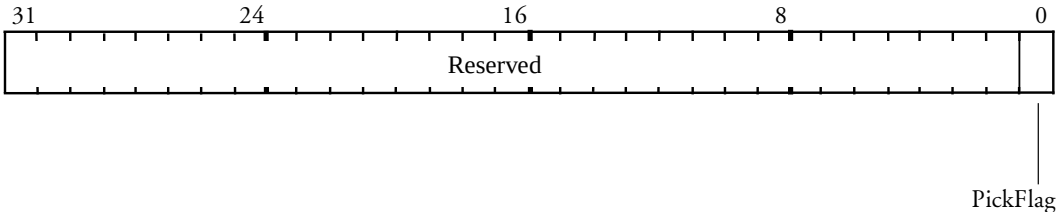
Name:	Pick Result	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C38
	Tag:	0x0187
Reset Value:	Undefined	

Write

The format of the data input is:



The format of the data output is:



This command causes the current status of the picking result to be passed to the Host Out FIFO, unless culled by the statistics bits in the **FilterMode** register.

The corresponding tag value output is: 0x187

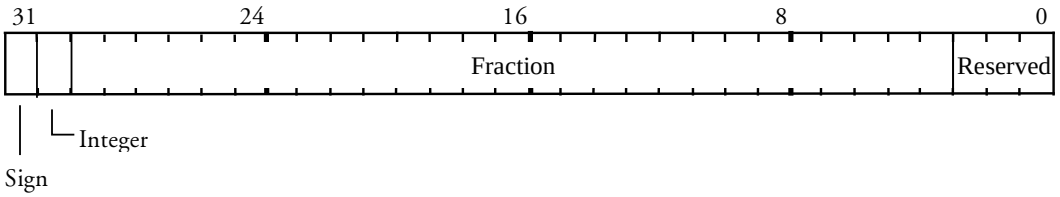
Bit0 PickFlag:
- 0 = Miss
- 1 = Hit has occurred

Bit1 BusyFlag:
- 0 = Idle
- 1 = Busy - used to validate the Pick Flag bit if this register is polled directly

QStart

Name:	Initial texture Q value	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83B8
	Tag:	0x0077
Reset Value:	Undefined	
Write		

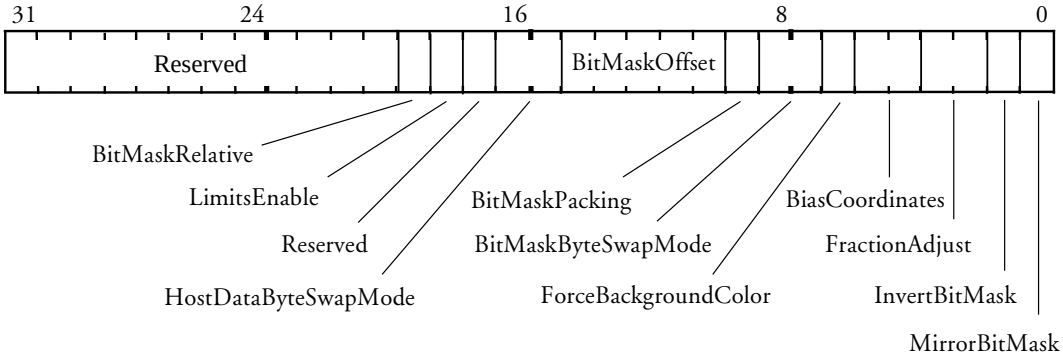
The format of the data input is:



Used to set the initial value for the Q coordinate when texture mapping. Format is 2's complement 2.27 fixed point.

RasterizerMode

Name:	Rasterizer Mode	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.80A0
	Tag:	0x0014
Reset Value:	Undefined	
Read/write		



Defines the long term mode of operation of the Rasterizer.

Bit0 MirrorBitMask

0 = use bit mask from least to most significant bit
1 = use bit mask from most to least significant bit

Bit1 InvertBitMask

0 = test against bitmask
1 = test against inverted bitmask

Bit2-3 FractionAdjust These bits are for the **ContinueNewLine** command and specify how the fraction bits in the Y and XDom DDAs are adjusted.
0 = No adjustment is done,
1 = Set the fraction bits to zero,
2 = Set the fraction bits to half.
3 = Set the fraction to nearly half, i.e. 0x7FFF

Bit4-5 BiasCoordinates These bits control how much is added onto the **StartXDom, StartXSub** and **StartY** values when they are loaded into the DDA units. The original registers are not effected.
0 = Zero is added,
1 = Half is added
2 = Nearly half is added, i.e. 0x7FFF

Bit6 ForceBackgroundColor Controls operation of bitmask test. If disabled any fragment failing the test is discarded. If enabled any fragment failing the test is drawn (other

tests allowing) but the color is taken from the **Texel0** register. Used to support foreground/background colors.

0 = disabled
1 = enabled

Bit7-8 BitMaskByteSwapMode. Controls byte swapping for bitmask. Input ABCD

0 = ABCD
1 = BADC
2 = CDAB
3 = DCBA

Bit9 BitMaskPacking.

0 = bitmask packed
1 = new data every scanline

Bit10-14 BitMaskOffset. Position of first bit to test in bitmask.

Bit15-16 HostDataByteSwapMode. Controls byte swapping for host data. Input ABCD

0 = ABCD
1 = BADC
2 = CDAB
3 = DCBA

Bit18 LimitsEnable. Enable X and Y limits checking

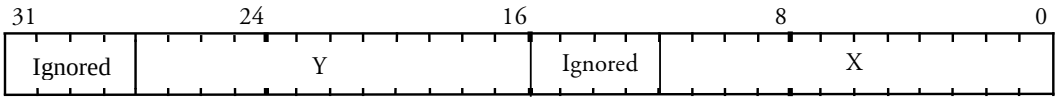
0 = disabled
1 = enabled

Bit19 BitMaskRelative

0 = bitmask indexed by counter
1 = bit mask indexed by X position

RectangleOrigin

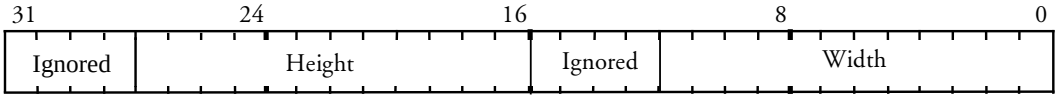
Name:	Rectangle Origin	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.80D0
	Tag:	0x001A
Reset Value:	Undefined	
Write		



Bits 0-15	X origin of the rectangle to be drawn.
Bits 16-31	Y origin of the rectangle to be drawn.
Name:	Rectangle Origin

RectangleSize

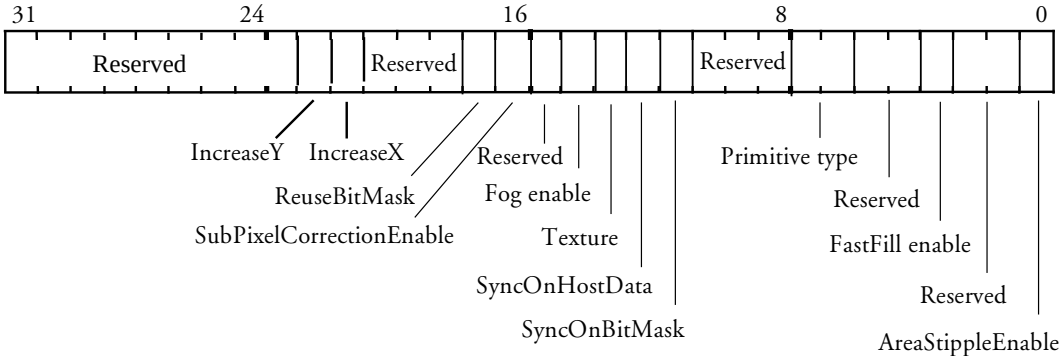
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.80D8
	Tag:	0x001B
Reset Value:	Undefined	
Write		



Bits 0-15	Width of the rectangle to be drawn.
Bits 16-31	Height of the rectangle to be drawn.

Render

Name:	Render	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8038
	Tag:	0x0007
Reset Value:	Undefined	
Write		



Command to start the rendering process.

The data field defines the short term modes required by this primitive. For details, see Table 5.4.

Bit0 AreaStippleEnable. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur.
- 0 = Disable
- 1 = Enable

Bit3 FastFillEnable
- 0 = Disable block filling
- 1 = Enable block filling

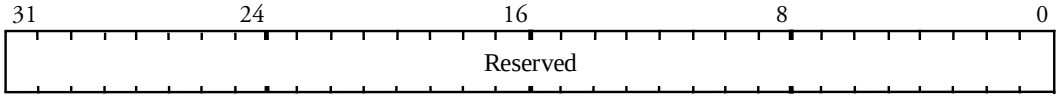
Bit6-7 PrimitiveType. These bits indicate the type of PERMEDIA primitive to be drawn. The primitives supported and the corresponding codes are:
- 0 = lines,
- 1 = trapezoids,
- 2 = points,
- 3 = rectangles.

Bit11 SyncOnBitMask. Enable bitmask test. Wait for new bitmask when current one expires unless SyncOnHostData or ReuseBitMask enabled.
- 0 = Disable
- 1 = Enable

Bit12	SyncOnHostData. When this bit is set, a fragment is produced only when one of the following registers has been written by the host: **Depth**, **FBData, FBSourceData**, **Stencil**, **Color** or **Texel0**. Also **BitMaskPattern** if SyncOnBitMask is set. 0 = Disable 1 = Enable
Bit13	TextureEnable. Note that the Texture Units must be suitably enabled as well for any texturing to occur. 0 = Disable 1 = Enable
Bit14	FogEnable. Note that the Fog Unit must be suitably enabled as well for any fogging to occur. 0 = Disable 1 = Enable
Bit16	SubPixelCorrectionEnable. Enables the sub pixel correction of color, depth, fog and texture values at the start of a scanline span. 0 = Disable 1 = Enable
Bit17	ReuseBitMask. Allows the bitmask to be reused when it has expired; if enabled the Rasterizer will not wait for a new mask when the current one has been used. 0 = Disable 1 = Enable
Bit18-20	Reserved.
Bit21	IncreaseX. Specifies that the rectangle primitive should be filled in the direction of increasing X. 0 = Disable 1 = Enable
Bit22	IncreaseY. Specifies that the rectangle primitive should be filled in the direction of increasing Y. 0 = Disable 1 = Enable

RepeatLine

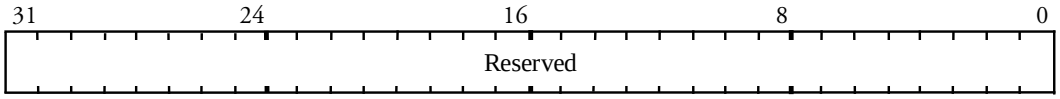
Name:	Repeat line	
Unit:	Delta	
Region: 0	Offset:	0x0000.9328
	Tag:	0x0265
Reset Value:	Undefined	
Write		



The data field is not used

RepeatTriangle

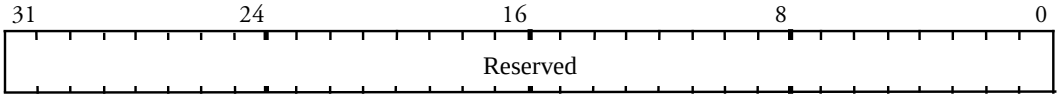
Name:	Repeat Triangle	
Unit:	Delta	
Region: 0	Offset:	0x0000.9310
	Tag:	0x0262
Reset Value:	Undefined	
Write		



The data field is not used.

ResetPickResult

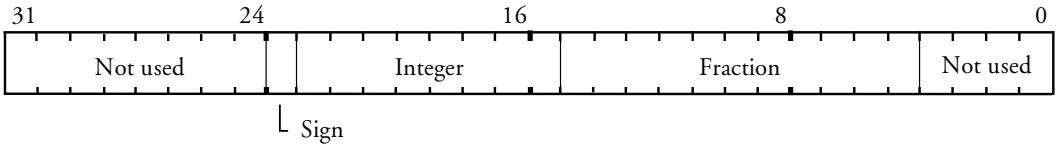
Name:	Reset Pick Result	
Units:	Host Out	
Region: 0	Offset:	0x0000.8C20
	Tag:	0x0184
Reset Value:	Undefined	
Write		



This command causes the current value of the picking result to be reset to zero. The data field is not used.

RStart

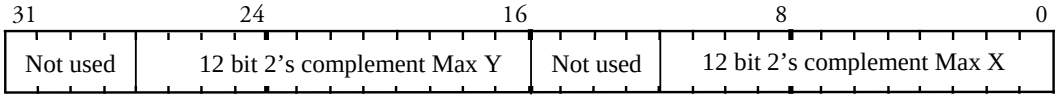
Name:	Initial Red Color	
Unit:	Color DDA	
Region: 0	Offset:	0x0000.8780
	Tag:	0x00F0
Reset Value:	Undefined	
Read/write		



This register is used to set the initial value for the Red value for a vertex when in Gouraud shading mode. The value is 2's complement 9.11 fixed point format.

ScissorMaxXY

Name:	Scissor Rectangle - Maximum XY	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.8190
	Tag:	0x0032
Reset Value:	Undefined	
Read/write		

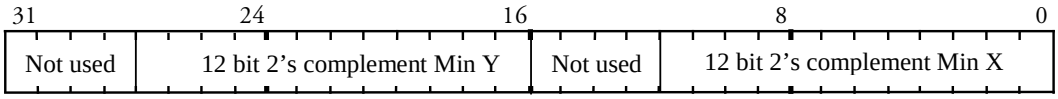


Specifies the user scissor rectangle corner farthest from the screen origin.

Name: Scissor Rectangle - Minimum XY

ScissorMinXY

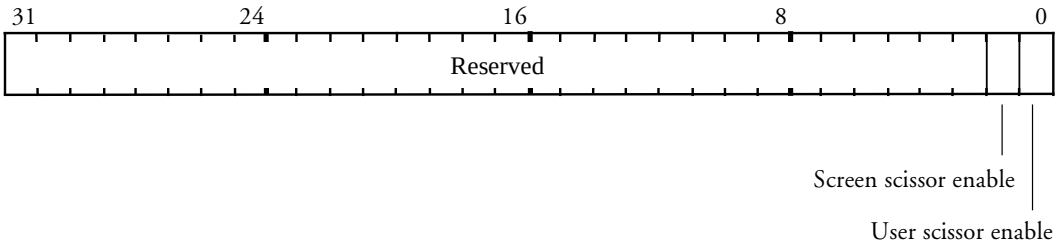
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.8188
	Tag:	0x0031
Reset Value:	Undefined	
Read/write		



Specifies the user scissor rectangle corner closest to the screen origin.

ScissorMode

Name:	Scissor Mode	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.8180
	Tag:	0x0030
Reset Value:	Undefined	
Read/write		

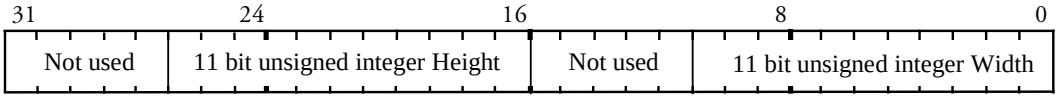


Controls enabling of the screen and user scissor tests.

Bit0	User Scissor Enable: 0 = Disable 1 = Enable
Bit1	Screen Scissor Enable: 0 = Disable 1 = Enable

ScreenSize

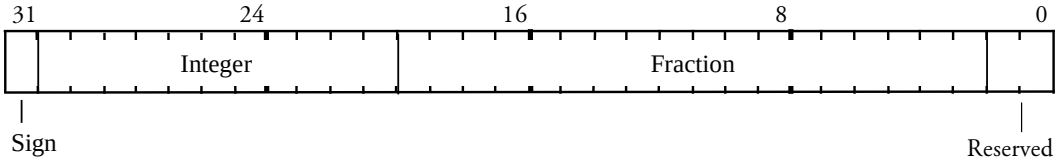
Name:	Screen Size	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.8198
	Tag:	0x0033
Reset Value:	Undefined	
Read/write		



Screen dimensions for screen scissor clip. The screen boundaries are (0, 0) to (width - 1, height - 1) inclusive.

SStart

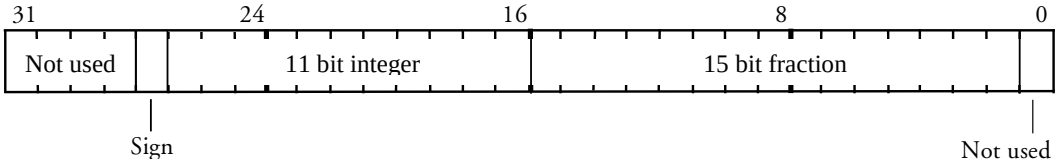
Name:	Initial texture S value	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.8388
	Tag:	0x0071
Reset Value:	Undefined	
Read/write		



Used to set the initial value for the S coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

StartXDom

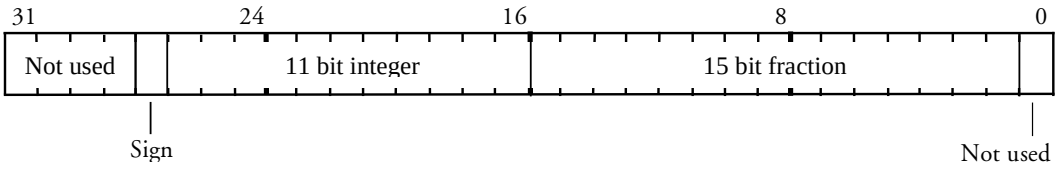
Name:	Start X Value - Dominant Edge	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8000
	Tag:	0x0000
Reset Value:	Undefined	
Read/write		



Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing. The value is in 2's complement 12.15 fixed point format.

StartXSub

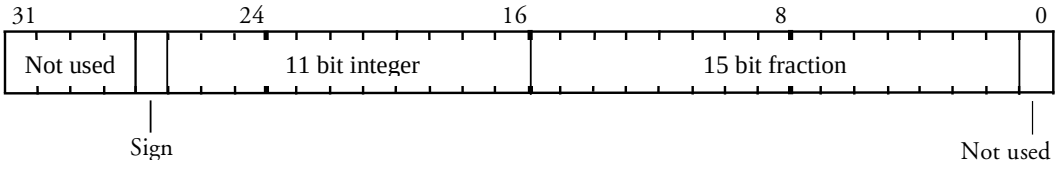
Name:	Start X Value - Subordinate Edge	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8010
	Tag:	0x0002
Reset Value:	Undefined	
Read/write		



Initial X value for the subordinate edge in trapezoid filling. The value is in 2's complement 12.15 fixed point format.

StartY

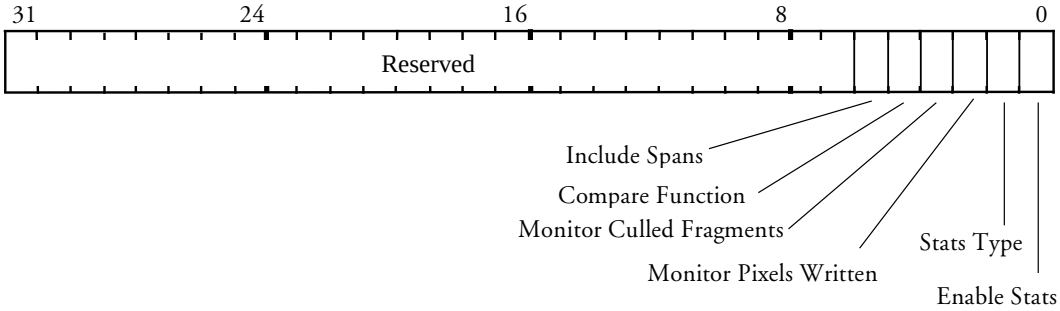
Name:	Start Y Value	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8020
	Tag:	0x0004
Reset Value:	Undefined	
Read/write		



Initial scanline in trapezoid filling, or initial Y position for line drawing. The value is in 2's complement 12.15 fixed point format.

StatisticMode

Name:	Statistic Mode	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C08
	Tag:	0x0181
Reset Value:	Undefined	
Read/write		



Controls the mode of statistics collection.

Bit0 EnableStats:
- 0 = Disable Statistics collection
- 1 = Enable Statistics collection

Bit1 StatsType:
- 0 = Picking mode
- 1 = Extent collection

Bit2 Active Steps:
- 0 = Excludes Pixels that were drawn
- 1 = Includes Pixels that were drawn

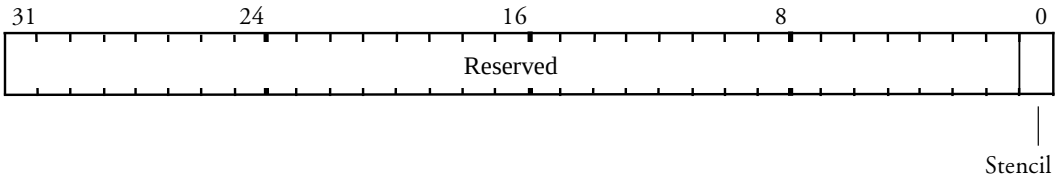
Bit3 Passive Steps:
- 0 = Excludes fragments that were culled from being drawn
- 1 = Includes fragments that were culled from being drawn

Bit4 CompareFunction:
- 0 = Inside region
- 1 = Outside region

Bit5 Spans:
- 0 = Exclude block filled spans
- 1 = Include block filled spans

Stencil

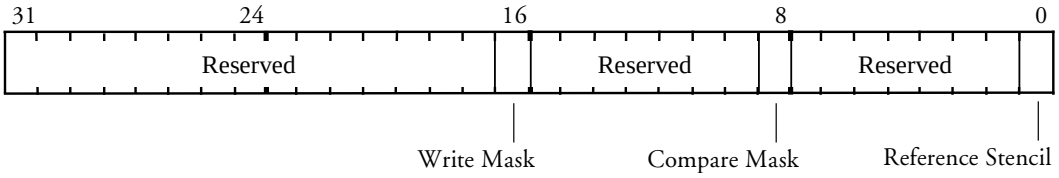
Name:	Stencil	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.8998
	Tag:	0x0133
Reset Value:	Undefined	
Read/write		



The stencil value to be used in clearing down the stencil buffer, or in drawing a primitive where the host supplies the stencil value.

StencilData

Name:	Stencil Data	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.8990
	Tag:	0x0132
Reset Value:	Undefined	
Read/write		



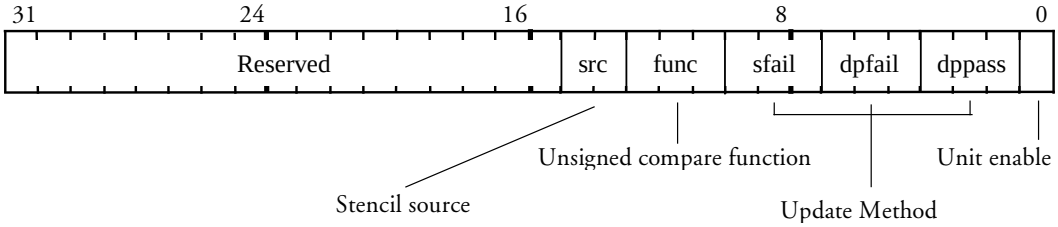
Holds data used in the stencil test.

The stencil writemask controls which stencil planes are updated as a result of the test.

Bit0	Reference Stencil is the reference value for the stencil test.
Bit8	Compare Mask is the mask used to determine which bits are significant in the comparison.
Bit16	Stencil Writemask is the mask used to determine which bits in the localbuffer are updated.

StencilMode

Name:	Stencil Mode	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.8988
	Tag:	0x0131
Reset Value:	Undefined	
Read/write		



Controls the stencil test, which conditionally rejects fragments based on the outcome of a comparison between the value in the stencil buffer and a reference value in the **StencilData** register. If the test is LESS and the result is true then the fragment value is less than the source value.

Bit0 Unit Enable:
0 = Disable
1 = Enable

Bit1-3 Update Method if Depth test passes and Stencil test passes:
(see table below)

Bit4-6 Update Method if Depth test fails and Stencil test passes:
(see table below)

Bit7-9 Update Method if Stencil test fails:

Mode	Method	Result
0	Keep	Source stencil
1	Zero	0
2	Replace	Reference stencil
3	Increment	Clamp (Source stencil + 1) to $2^{\text{stencil width}} - 1$
4	Decrement	Clamp (Source stencil -1) to 0
5	Invert	~Source stencil

Bit10-12 Unsigned Comparison Function:

Mode = Comparison Function
0 = NEVER
1 = LESS
2 = EQUAL
3 = LESS OR EQUAL
4 = GREATER

5 = NOT EQUAL
6 = GREATER OR EQUAL
7 = ALWAYS

Bit13-14 Stencil Source:
0 = Test Logic
1 = Stencil Register
2 = LBData
3 = LBSourceData

SuspendUntilFrameblank

Name: Suspend until frameblank

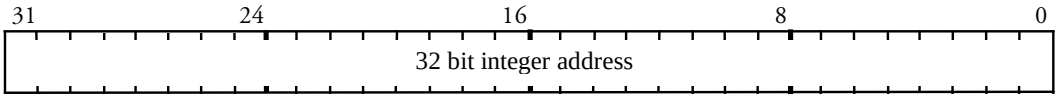
Unit: Framebuffer R/W

Region: 0 Offset: 0x0000.8C78

Tag: 0x018F

Reset Value: Undefined

Write

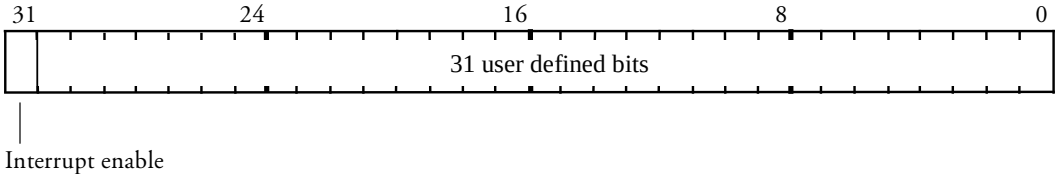


This command causes all outstanding framebuffer writes to be flushed and then suspension of framebuffer accesses until the next frameblank period. The data field is the start address of the next frame to be displayed. This address will be used from the next frameblank until a new address is supplied.

Bit0-31 Address

Sync

Name:	Synchronization	
Unit:	Host Out	
Region: 0	Offset:	0x0000.8C40
	Tag:	0x0188
Reset Value:	Undefined	
Write		



This command can be used to synchronize PERMEDIA with the host. It is also used to flush outstanding PERMEDIA operations such as pending memory accesses. It also causes the current status of the picking result to be passed to the Host Out FIFO, unless culled by the statistics bits in the **FilterMode** register.

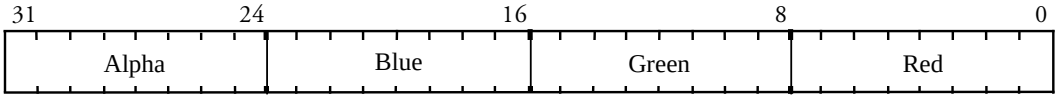
Bit0-30	User Defined
Bit31	InterruptEnable: 0 = Disable Interrupt for this command 1 = Enable Interrupt for this command

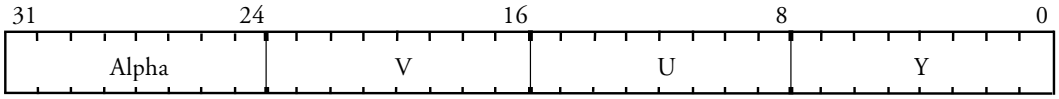
The data output is the value written to the register by this command. If interrupts are enabled, then the interrupt does not occur until the tag and/or data have been written to the output FIFO.

The corresponding tag value output is: 0x188

Texel0

Name:	Texel Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.8600
	Tag:	0x00C0
Reset Value:	Undefined	
Read/write		



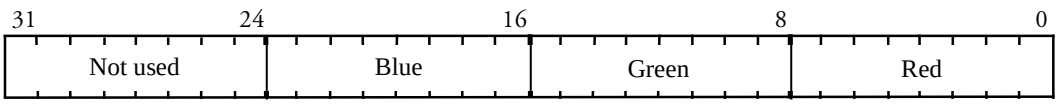


The texel value can be loaded using the Rasterizer SyncOnHostData mode. This is useful for direct application of procedural textures. It is also used when downloading YUV data which needs to be converted to RGB; the YUV conversion is done on the contents of this register.

This register is also used to supply the background color if ForceBackgroundColor has been enabled in either the **RasterizerMode** or the **AreaStippleMode** registers.

TexelLUT[0..15]

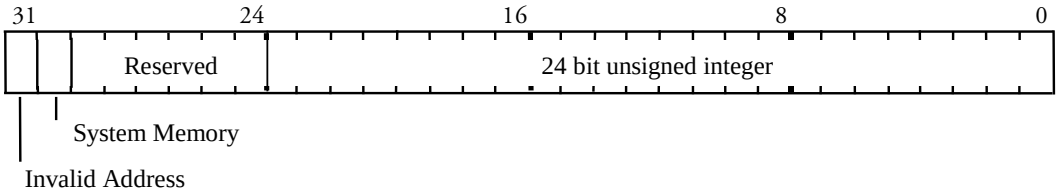
Name:	Texel LUT entries 0 to 15	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8E80 ,…, 0x0000.8EF8
	Tag:	0x01D0,…,0x1DF
Reset Value:	Undefined	
Read/write		



The value to be loaded into the specified texel look-up-table entry.

TexelLUTAddress

Name:	Address of LUT in memory	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.84D0
	Tag:	0x009A
Reset Value:	Undefined	
Read/write		



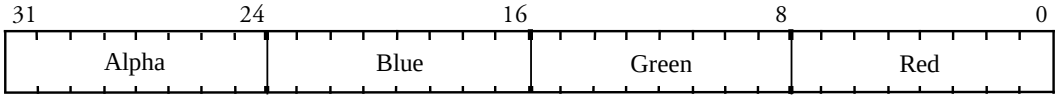
The address in memory in 32 bit units of data to be loaded into the texture look-up table. If bit 30 is set the LUT resides in system memory rather than local buffer and should be loaded across the PCI bus. Bit 31 is ignored if this register is loaded directly. If it is loaded indirectly by the TexelLUTID

TexelLUTData

register, bit 31 indicates that the address is invalid and should not be used.

.

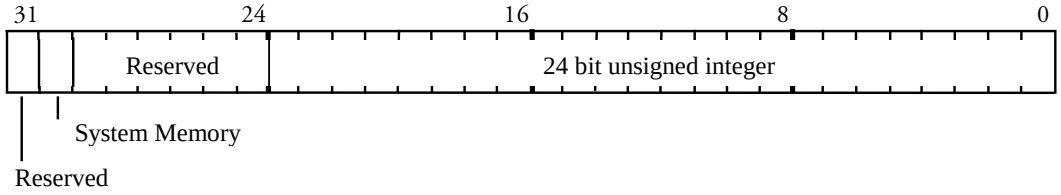
Name:	Data for texture LUT	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.84C8
	Tag:	0x0099
Reset Value:	Undefined	
Read/write		



Data to be loaded into the texture look-up table.

TexelLUTID

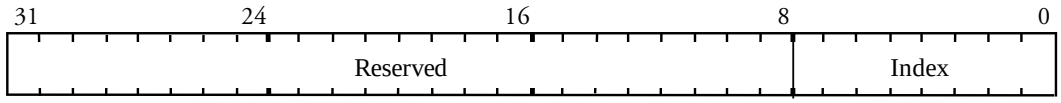
Name:	Indirect handle for texture LUT	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8F78
	Tag:	0x001EF
Reset Value:	Undefined	
Read/write		



The 24 bit field holds the address of the data that should be loaded into the TexelLUTAddress register. If bit 30 is set this data is in system memory and should be fetched across the PCI bus.

TexelLUTIndex

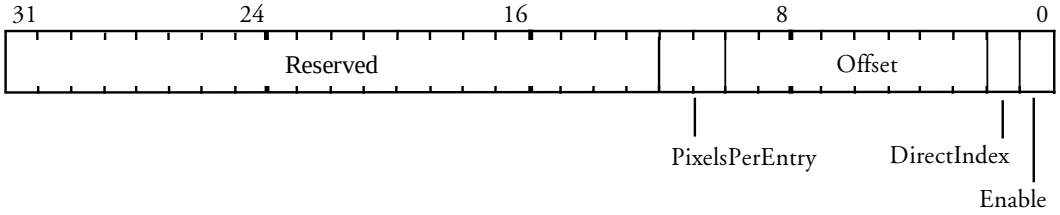
Name:	Index data for LUT	
Unit:	Texture Read	
Region: 0	Offset:	0x0000. 84C0
	Tag:	0x00098
Reset Value:	Undefined	
Read/write		



The TexelLUTIndex register holds the index into the texel LUT where the write of subsequent TexelLUTData will be written. The index is held in the lower 8 bits of the TexelLUTIndex register and this is auto incremented after every write to TexelLUTData. Reading back from TexelLUTIndex returns the auto incremented value, if any writes to TexelLUTData have occured. A side effect of reading the TexelLUTIndex register is to reset an internal counter used to generate the LUT index when reading TexelLUTData. This internal counter will autoincrement after every read of TexelLUTData.

TexelLUTMode

Name:	Texel LUT Mode	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8678
	Tag:	0x00CF
Reset Value:	Undefined	
Read/write		

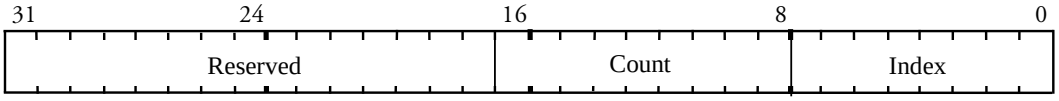


Controls the operation of the texture look-up table.

Bit0	Enable: 0 = No 1 = Lookup
Bit1	DirectIndex: 0 = Index from texture data 1 = Index from fragment XY values
Bit2-9	Offset: 0x0000. Offset to add index in DirectIndex mode
Bit10-11	PixelsPerEntry: number of pixels per entry in LUT 0 = 1 pixel 1 = 2 pixels 2 = 4 pixels

TexelLUTTransfer

Name:	Initiates loading of LUT data from memory	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.84D8
	Tag:	0x0009B
Reset Value:	Undefined	
Read/write		



The index field specifies the first entry in the LUT to load, and the Count field specifies the number of entries to load.

Name: Texture Address Mode

TextureAddressMode

Unit:	Texture Address	
Region: 0	Offset:	0x0000.8380
	Tag:	0x0070
Reset Value:	Undefined	
Read/write		

Controls the calculation of texture addresses.

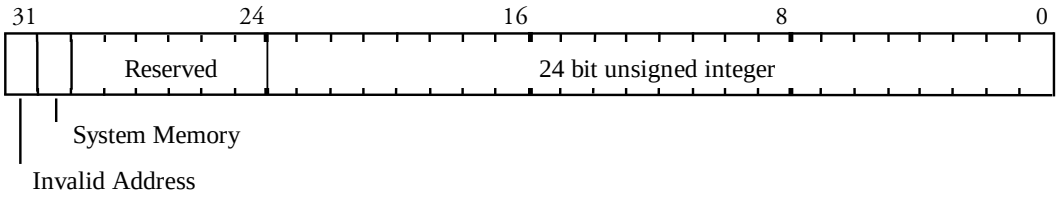
If bit 1 is set, PERMEDIA performs fast, accurate perspective correction.

Bit0 Texture Address Enable:
- 0 = Disable
- 1 = Enable

Bit1 Perspective Correction:
- 0 = Disable
- 1 = Enable

TextureBaseAddress

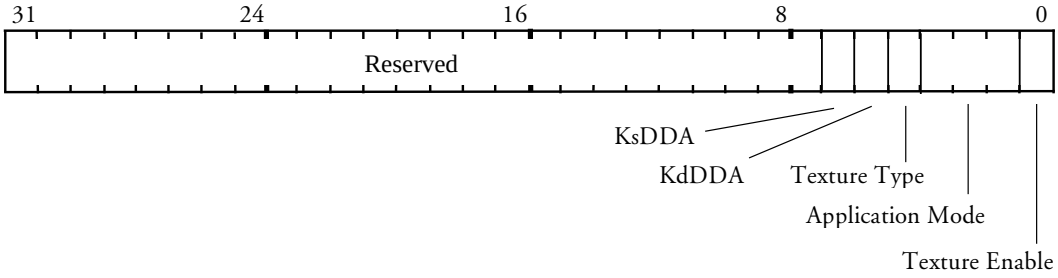
Name:	Address of texture in memory	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8580
	Tag:	0x00B0
Reset Value:	Undefined	
Read/write		



Base address of texture map. Specified in texels from the base of the memory. If bit 3o is set the texture resides in system memory rather than local buffer and should be fetched across the PCI bus. Bit 31 is ignored if this register is loaded directly. If it is loaded indirectly by the TextureID register, bit 31 indicates that the address is invalid and should not be used.

TextureColorMode

Name:	Texture Color Mode	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0x0000.8680
	Tag:	0x00D0
Reset Value:	Undefined	Read/write



Controls the application of texture. The KsDDA and KdDDA bits enable the internal DDAs and should be set for modulate or highlight Ramp texture application modes. The Texture Type field differentiates between RGB (OpenGL) and Ramp (Apple) application modes. With Ramp Application Mode, various modes can be simultaneously applied e.g. decal with highlight.

*Note: The TextureEnable bit in the **Render** command must also be set for a primitive to be texture mapped.*

Bit0 Texture Enable:
- 0 = Disable
- 1 = Enable texture application

Bit1-3 Application Mode:

RGB	Ramp
0 = Modulate	Bit 1 = Decal
1 = Decal	Bit 2 = Modulate
2 = Reserved	Bit 3 = Highlight
3 = Copy	
4 = Modulate + Highlight	
5 = Decal + Highlight	
6 = Reserved	
7 = Copy + Highlight	

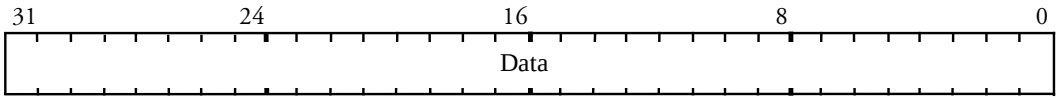
Bit4 Texture Type:
- 0 = RGB
- 1 = Ramp

Bit5 KdDDA:
- 0 = Disable
- 1 = Enable

Bit6 KsDDA:
- 0 = Disable
- 1 = Enable

TextureData

Name:	Texture Data	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.88E8
	Tag:	0x011D
Reset Value:	Undefined	
Write		



Used with **TextureDownloadOffset** to load raw texture data into memory. This may include multiple texels depending on the texel size.

Bit0-31	Data

TextureDataFormat

Name: Texture Data Format

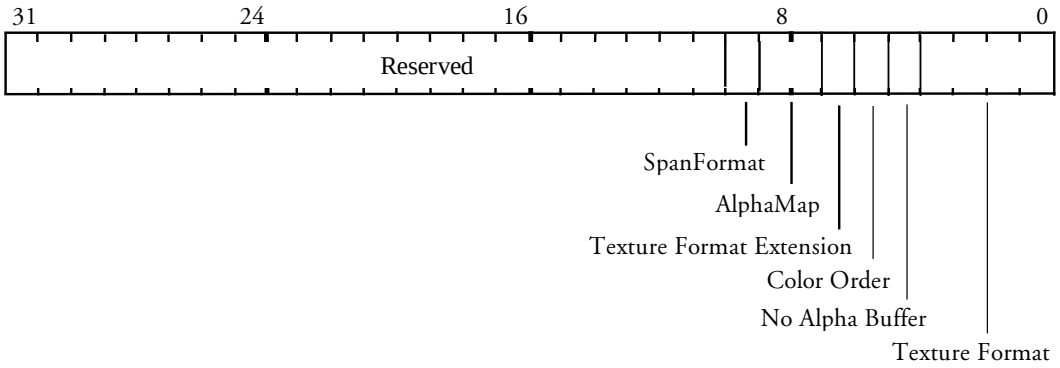
Unit: Texture Read

Region: 0 Offset: 0x0000.8590

Tag: 0x00B2

Reset Value: Undefined

Read/write



Specifies the color format of the texture map in memory. (see overleaf for description of the bit fields)

Bit0-3 Texture Format:

			Internal Color Channel			
Format	Color Order	Name	R/Y	G/U	B/V	A
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
15	BGR	CI4	4@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
18	BGR	YUV444	8@0	8@8	8@16	8@24
19	BGR	YUV422	8@0	8@8	8@8	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0

9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
15	RGB	CI4	4@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0
17	RGB	5:6:5 Back	5@27	6@21	5@16	0
18	RGB	YUV444	8@16	8@8	8@0	8@24
19	RGB	YUV422	8@8	8@0	8@0	0

Notes: The format column is also dependent on bit6. n@m means n bits starting at bit m. Front and Back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the 7 bit formatted value. If the format has no alpha bits, the alpha field defaults to 0xF8

Bit4 No Alpha Buffer:
- 0 = Alpha buffer present
- 1 = Alpha buffer not present

Bit5 Color Order:
- 0 = BGR
- 1 = RGB

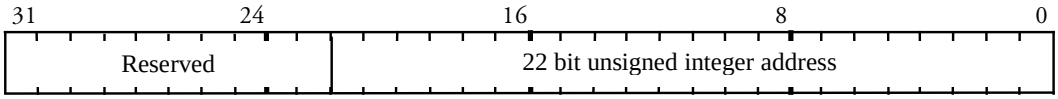
Bit6 Texture Format Extension. Most significant bit extension to Texture Format held in bits0-3

Bit7-8 AlphaMap:
- 0 = None
- 1 = Include: pass texels that lie within the AlphaMap bounds
- 2 = Exclude: fail texels that lie within the AlphaMap bounds

Bit9 SpanFormat: used to control the data format of a texture map holding block fill masks.
- 0 = Normal
- 1 = Flip: mirror the bits within each byte

TextureDownloadOffset

Name:	Texture Download Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0x0000.88F0
Tag:		0x011E
Reset Value:	Undefined	
Write/Read		

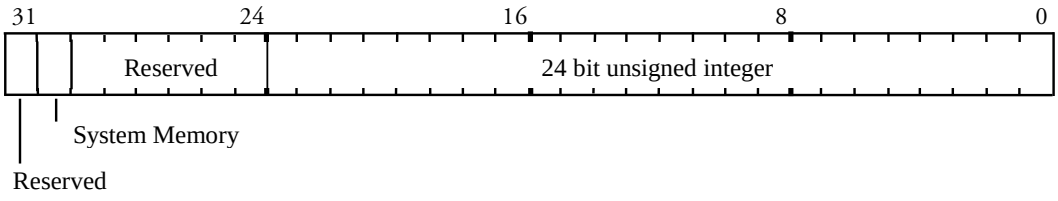


32 bit aligned address at which the texture load will start. Each write to **TextureData** increments this value by one after the store has taken place. Note, if this register is read back it will not necessarily

TextureID

contain the same value as the written value.

Bit0-21	Address	
Name:	Indirect handle for texture map	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8F70
	Tag:	0x001EE
Reset Value:	Undefined	
Read/write		

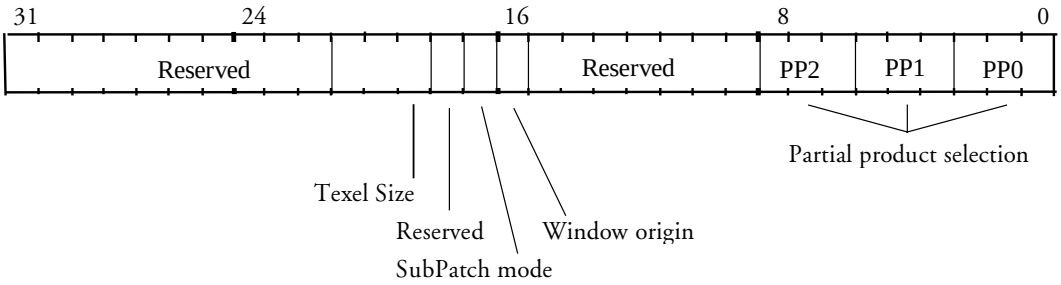


The 24 bit field holds the address of the data that should be loaded into the TextureBaseAddress register. If bit 30 is set this data is in system memory and should be fetched across the PCI bus.

TextureMapFormat

Name:	Texture Map Format	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8588
	Tag:	0x00B1
Reset Value:	Undefined	

Read/write



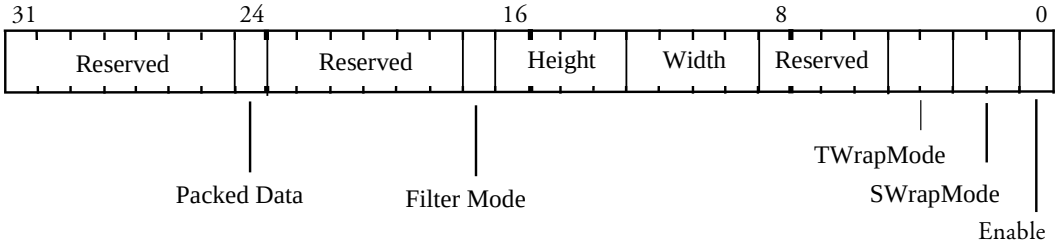
Specifies the organization of the texture map in memory.

Enabling subpatch addressing improves the performance of texture mapping in typical situations.

Bit0-2	Partial Product 0 - See Appendix C for a table of values
Bit3-5	Partial Product 1 - See Appendix C for a table of values
Bit6-8	Partial Product 2 - See Appendix C for a table of values
Bit16	Window Origin: 0 = Top 1 = Bottom Left
Bit17	Subpatch Mode: 0 = Disable 1 = Enable
Bit19-20	Texel Size: 0 = 8 bits 1 = 16 bits 2 = 32 bits 3 = 4 bits 4 = 24 bits

TextureReadMode

Name:	Texture Read Mode	
Unit:	Texture Read	
Region: 0	Offset:	0x0000.8670
	Tag:	0x00CE
Reset Value:	Undefined	Read/write



Controls texture read operations. When **FilterMode** is set, bilinear texture mapping is performed otherwise nearest neighbor texture mapping occurs. The S and TWrapModes specify the action to be taken when the S and T coordinates fall outside the required range. Clamp is useful when texture mapping a single image onto an object, Repeat cause the texture pattern to be repeated, whilst mirror causes the texture pattern to be alternately reversed. The Packed Data bit is used to define how texels are read from memory. If this bit is cleared, each texel is read one at a time; if set several texels can be read simultaneously improving efficiency. The actual number of texels read in this case is dependent on the texel size.

Bit0 Enable
- 0 = Disable texture reads
- 1 = Enable texture reads

Bit1-2 SWrapMode
- 0 = Clamp
- 1 = Repeat
- 2 = Mirror

Bit3-4 TWrapMode
- 0 = Clamp
- 1 = Repeat
- 2 = Mirror

Bit9-12 Width - log2 texture map width

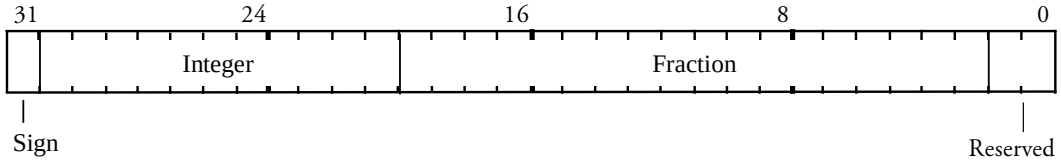
Bit13-16 Height - log2 texture map height

Bit17 FilterMode
- 0 = Disable bilinear texture filtering
- 1 = Enable bilinear texture filtering

Bit24 PackedData
- 0 = off
- 1 = on

TStart

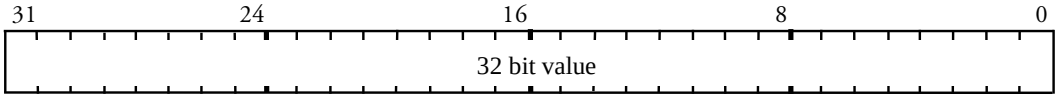
Name:	Initial texture T value	
Unit:	Texture Address	
Region: 0	Offset:	0x0000.83A0
	Tag:	0x0074
Reset Value:	Undefined	
Read/write		



Used to set the initial value for the T coordinate when texture mapping. Format is 2's complement 12.18 fixed point.

V0Fixed[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9000,.. 0x0000.9078
	Tag:	0x00200,..0x0020F
Reset Value:	Undefined	
Read/write		



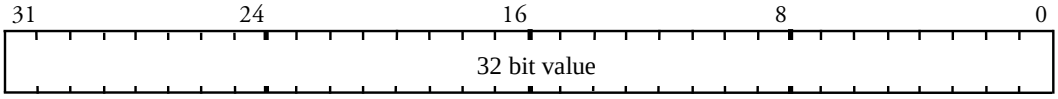
Data for vertex 0. The following table shows the valid entries:

Offset	Category	Parameter	Fixed Point Format
0	Texture	s	2.30 s[1]
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	1.30 us
7		blue	1.30 us
8		alpha	1.30 us
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]This is the range when Normalise is not used. When Normalise is enabled the fixed point format can be anything, providing it is the same for the s, t and q parameters. The numbers will be interpreted as if they had 2.30 format for the purpose of conversion to floating point. If the fixed point format (2.30) is different from what the user had in mind then the input values are just pre-scaled by a fixed amount (i.e. the difference in binary point positions) prior to conversion.

V1Fixed[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9080, 0x0000.90F8
	Tag:	0x00210, 0x0021F
Reset Value:	Undefined	
Read/write		



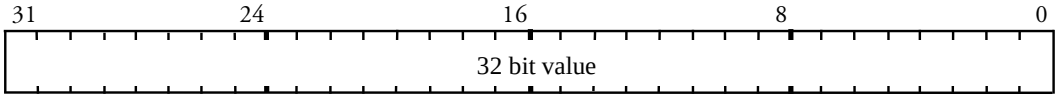
Data for vertex 1. The following table shows the valid entries:

Offset	Category	Parameter	Fixed Point Format
0	Texture	s	2.30 s[1]
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	1.30 us
7		blue	1.30 us
8		alpha	1.30 us
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]This is the range when Normalise is not used. When Normalise is enabled the fixed point format can be anything, providing it is the same for the s, t and q parameters. The numbers will be interpreted as if they had 2.30 format for the purpose of conversion to floating point. If the fixed point format (2.30) is different from what the user had in mind then the input values are just pre-scaled by a fixed amount (i.e. the difference in binary point positions) prior to conversion.

V2Fixed[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9100, .. 0x0000.9178
	Tag:	0x00220 .. 0x0022F
Reset Value:	Undefined	
Read/write		



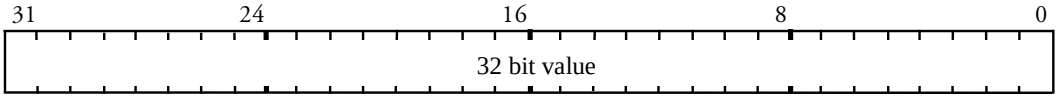
Data for vertex 2. The following table shows the valid entries:

Offset	Category	Parameter	Fixed Point Format
0	Texture	s	2.30 s[1]
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	1.30 us
7		blue	1.30 us
8		alpha	1.30 us
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]This is the range when Normalise is not used. When Normalise is enabled the fixed point format can be anything, providing it is the same for the s, t and q parameters. The numbers will be interpreted as if they had 2.30 format for the purpose of conversion to floating point. If the fixed point format (2.30) is different from what the user had in mind then the input values are just pre-scaled by a fixed amount (i.e. the difference in binary point positions) prior to conversion.

V0Float[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9180, 0x0000.91F8
	Tag:	0x00230, 0x0023F
Reset Value:	Undefined	
Read/write		



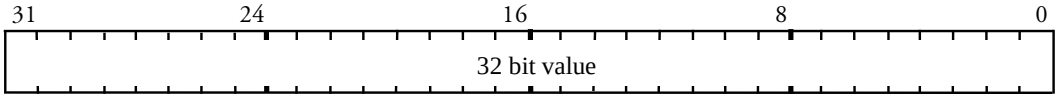
Data for vertex 0. The following table shows the valid entries:

Offset	Category	Parameter	IEEE Single Precision Floating Point Range
0	Texture	s	-1.0…1.0
1		t	-1.0…1.0
2		q	-1.0…1.0
3		Ks	0.0…2.0
4		Kd	0.0…1.0
5	Color	red	0.0…1.0
6		green	0.0…1.0
7		blue	0.0…1.0
8		alpha	0.0…1.0
9	Fog	f	-1.0…1.0
10	Co-ordinate	x	-32K…+32K [footnotes 1,2]
11		y	-32K…+32K [footnotes 1,2]
12		z	0.0…1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]The normal range here is limited by the size of the screen.
[2]K = 1024.

V1Float[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9200, 0x0000.8278
	Tag:	0x00240, 0x0024F
Reset Value:	Undefined	
Read/write		



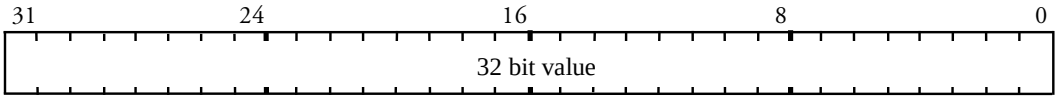
Data for vertex 1. The following table shows the valid entries:

Offset	Category	Parameter	IEEE Single Precision Floating Point Range
0	Texture	s	-1.0…1.0
1		t	-1.0…1.0
2		q	-1.0…1.0
3		Ks	0.0…2.0
4		Kd	0.0…1.0
5	Color	red	0.0…1.0
6		green	0.0…1.0
7		blue	0.0…1.0
8		alpha	0.0…1.0
9	Fog	f	-1.0…1.0
10	Co-ordinate	x	-32K…+32K [footnotes 1,2]
11		y	-32K…+32K [footnotes 1,2]
12		z	0.0…1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]The normal range here is limited by the size of the screen.
[2]K = 1024.

V2Float[0..14]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset:	0x0000.9280, 0x0000.92F8
	Tag:	0x00250, 0x0025F
Reset Value:	Undefined	
Read/write		



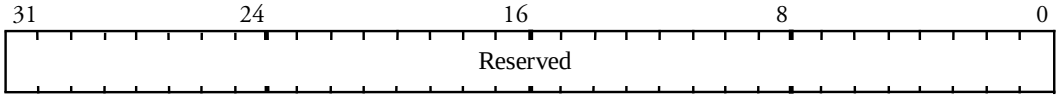
Data for vertex 2. The following table shows the valid entries:

Offset	Category	Parameter	IEEE Single Precision Floating Point Range
0	Texture	s	-1.0…1.0
1		t	-1.0…1.0
2		q	-1.0…1.0
3		Ks	0.0…2.0
4		Kd	0.0…1.0
5	Color	red	0.0…1.0
6		green	0.0…1.0
7		blue	0.0…1.0
8		alpha	0.0…1.0
9	Fog	f	-1.0…1.0
10	Co-ordinate	x	-32K…+32K [footnotes 1,2]
11		y	-32K…+32K [footnotes 1,2]
12		z	0.0…1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

[1]The normal range here is limited by the size of the screen.
[2]K = 1024.

WaitForCompletion

Name:	Wait for completion	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.8088
	Tag:	0x0017
Reset Value:	Undefined	
Write		

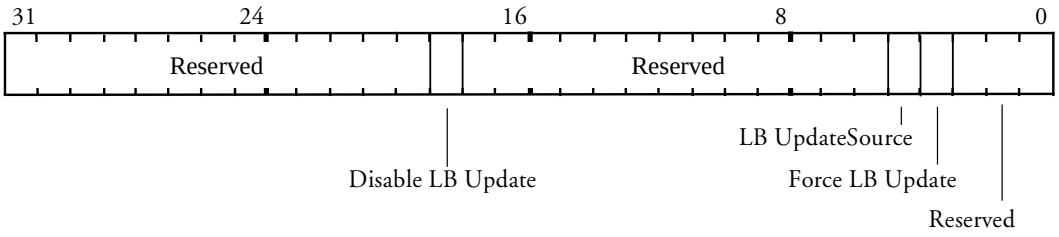


This command register causes PERMEDIA to suspend operation until all framebuffer writes have completed. Useful to separate, say, a texture download from subsequent primitives.

Bit0-31	Reserved

Window

Name:	Window	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.8980
	Tag:	0x0130
Reset Value:	Undefined	
Read/write		



If the Force LB Update bit is set, this overrides the stencil and depth tests, and the per unit enables, to force the localbuffer to be updated. Writes must still be enabled in the **LBWriteMode** register. When this bit is clear any update is conditional on the outcome of the stencil and depth tests.

If the Disable LB Update bit is set the results of the stencil and depth tests are overridden and the localbuffer not updated, even if localbuffer writes are enabled. When writes are disabled in **LBWriteMode** there may be a performance advantage in also setting Disable LB Update.

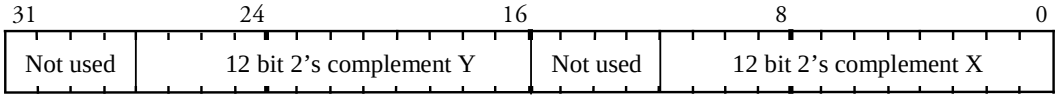
Bit3 Force LB Update:
- 0 = Not Forced
- 1 = Forced

Bit4 LB Update Source:
- 0 = LBSourceData
- 1 = Registers

Bit18 Disable LB Update
- 0 = Update
- 1 = No Update

WindowOrigin

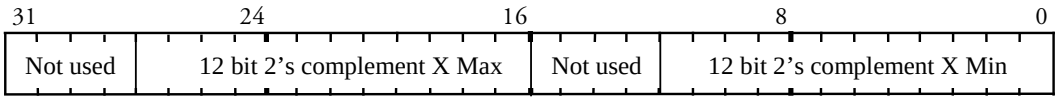
Name:	Window Origin	
Unit:	Scissor/Stipple	
Region: 0	Offset:	0x0000.81C8
	Tag:	0x0039
Reset Value:	Undefined	
Read/write		



As the Rasterizer unit generates each fragment, the fragment's coordinates are adjusted by the amount of the origin to generate the fragment's screen coordinates. This occurs prior to doing the screen scissor test.

XLimits

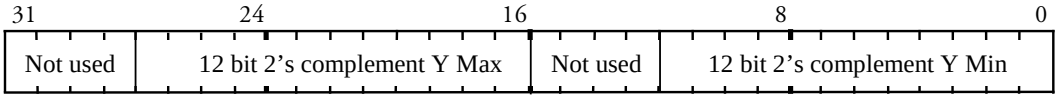
Name:	X extent for rasterizing	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.80C8
	Tag:	0x0019
Reset Value:	Undefined	
Read/write		



Defines the X extent the Rasterizer should fill between.

YLimits

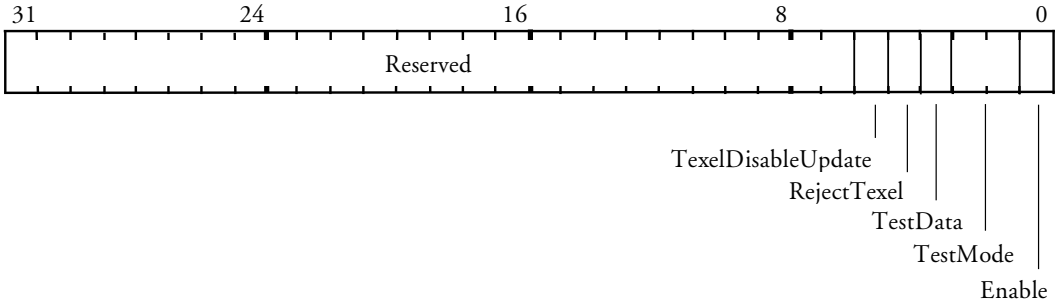
Name:	Y extent for rasterizing	
Unit:	Rasterizer	
Region: 0	Offset:	0x0000.80A8
	Tag:	0x0015
Reset Value:	Undefined	
Read/write		



Defines the Y extent the Rasterizer should fill between.

YUVMode

Name:	YUV Mode	
Unit:	YUV	
Region: 0	Offset:	0x0000.8F00
	Tag:	0x01E0
Reset Value:	Undefined	
Read/write		



Control YUV to RGB conversion and/or chroma test.

Bit0 Enable
- 0 = YUV to RGB color space conversion disabled
- 1 = YUV to RGB color space conversion enabled

Bit1-2 TestMode
- 0 = No chroma test
- 1 = Pass if within chroma bounds
- 2 = Fail if within chroma bounds

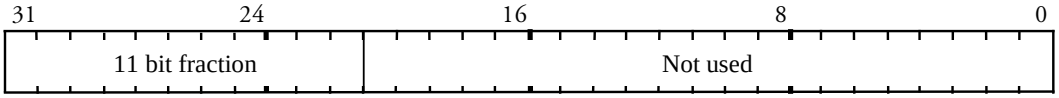
Bit3 TestData
- 0 = Apply chroma test on input data (before color space conversion if enabled)
- 1 = Apply chroma test on output data (after color space conversion if enabled)

Bit4 RejectTexel
- 0 = Do not plot pixel if chroma test fails
- 1 = Do not texture pixel if chroma test fails

Bit5 TexelDisableUpdate
- 0 = Pass on texel data
- 1 = Reject texel data immediately after chroma test

ZStartL

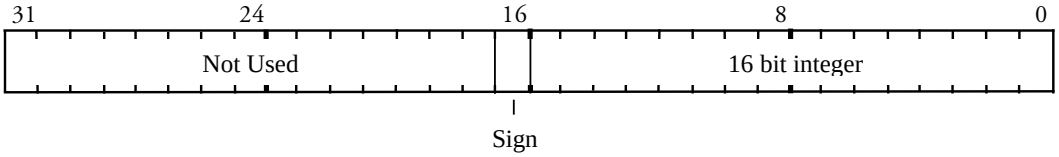
Name:	Depth Start Value - Lower	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89B8
	Tag:	0x0137
Reset Value:	Undefined	
Read/write		



This register holds part of the start value for depth interpolation. **ZStartU** holds the most significant bits, and **ZStartL** the least significant bits. The combined value is in 2's complement 17.11 fixed point format.

ZStartU

Name:	Depth Start Value - Upper	
Unit:	Stencil/Depth	
Region: 0	Offset:	0x0000.89B0
	Tag:	0x0136
Reset Value:	Undefined	
Read/write		



This register holds part of the start value for depth interpolation. **ZStartU** holds the most significant bits, and **ZStartL** the least significant bits. The combined value is in 2's complement 17.11 fixed point format.

Appendix B. Pseudocode Definitions

In many areas of the document fragments of pseudocode are given, to describe the loading of registers. These are based on a C interface to PERMEDIA in which each 32 bit register is represented as a C structure, potentially split into a series of bit fields. In an example where only a subset of the bit fields in a register are set, it is assumed either that a software copy of the register is being modified, or that the current contents of the register has first been read back to the host. This style has been chosen for clarity; there are often more efficient strategies.

The constant definitions and register bit field definitions are based upon those used in the 3Dlabs driver software. Sources including header files for this are available under source license agreement.

Warning: the order of loading control registers into the HyperPipeline has also been chosen for clarity, rather than efficiency. The optimal order is documented in section §7.2.3.

Loading of a PERMEDIA register is expressed as:

```
register-name(value)
```

When writing directly to the register file (i.e. to a FIFO) this would be implemented by writing "value" to the mapped-in address of the register called "register-name".

Fragmentary examples are not in strict C syntax, a typical example is:

```
// Sample code to rasterize a 10x10 rectangle at the
// framebuffer origin.

StartXDom (0)                  // Start dominant edge
StartXSub (1<<16)              // Start of subordinate
dXDom (0x0)
dXSub (0x0)
Count (0xA)
YStart(0)
dY (1<<16)

// Set-up to render a trapezoid.

render.AreaStippleEnable = PERMEDIA_DISABLE
render.PrimitiveType = PERMEDIA_TRAPEZOID
render.FastFillEnable = PERMEDIA_DISABLE
render.FogEnable = PERMEDIA_DISABLE
render.TextureEnable = PERMEDIA_DISABLE
render.ReuseBitMask = PERMEDIA_DISABLE
render.SyncOnBitMask = PERMEDIA_FALSE
render.SyncOnHostData = PERMEDIA_FALSE

Render (render)                // Render the rectangle
```

Code is shown in courier and comments are C++ style '//' indicating that the rest of the line is a comment. Any statement which ends in parenthesis is a register update, other statements will generally be assignments. A variable, say render, is of a type associated with the register being modified. This will usually be clear by the

context and will not usually be declared as such. All the type definitions are in the header files. The values assigned to a register will be either a variable as described above, a macro i.e. PERMEDIA_TRUE, as found in the headers, or an immediate constant in C style format i.e. 0x45. In registers which have several fields, some of which are not relevant to a particular example, the field can be ignored completely or set to *don't care*. In some registers, values for fields which need to be set but are not readily available will typically be set *as appropriate*.

In some fragments, simply a list of commands is given e.g.:

```
// Sample code to rasterize a rectangle

StartXDom ()// Start dominant edge
StartXSub ()// Start of subordinate
dXDom ()
dXSub ()
Count ()
YStart()
dY ()

// Set-up to render an aliased trapezoid.

Render ()         // Render the rectangle
```

This technique is used to simply give a feel for the registers involved in a particular operation and where a detailed treatment is not warranted.

To take the address of a register, the name is used, thus this example stores the address of the **StartXDom** register in the buffer pointed to by the variable `buf` and increments the pointer:

```
*buf++ = StartXDom
```

To test the value of a register the register name is dereferenced using the C '*' operator as for instance in this example which tests for the completion of a DMA operation:

```
while( *DMACount != 0 ) ;
```

Appendix C. Screen Widths Table

The screen width is specified as the sum of selected partial products so a full multiply operation is not needed. The partial products are selected by the fields PP0, PP1 and PP2 in the **FBReadMode** register, **LBReadMode** register and **TextureMapFormat** register. The range of widths supported by this technique are tabulated below, together with the values for each of the PP fields.

Screen Width	PP0	PP1	PP2
0	0	0	0
32	1	0	0
64	1	1	0
96	1	1	1
128	2	1	1
160	2	2	1
192	2	2	2
224	3	2	1
256	3	2	2
288	3	3	1
320	3	3	2
384	3	3	3
416	4	3	1
448	4	3	2
512	4	3	3
544	4	4	1
576	4	4	2
640	4	4	3
768	4	4	4
800	5	4	1
832	5	4	2
896	5	4	3
1024	5	4	4
1056	5	5	1
1088	5	5	2
1152	5	5	3
1280	5	5	4
1536	5	5	5
1568	6	5	1
1600	6	5	2
1664	6	5	3
1792	6	5	4
2048	6	5	5

Table C.1 Partial Products

Note that PERMEDIA supports a maximum screen resolution of 2048 x 2048.

Appendix D. A Gouraud Shaded Triangle without using the Delta Unit

For best performance, the Delta unit in PERMEDIA should be used to calculate the edge deltas used by the Graphics Processor. For backward compatibility, or special situations, the edge delta registers may be programmed directly, and this appendix describes the calculations that are needed to do this correctly.

In this section we show how to render a typical 3D graphics primitive without using the Delta Unit. The primitive is a Gouraud shaded, depth buffered triangle. This appendix is included to understand any legacy PERMEDIA 1 software to allow alternative rasterization techniques to be used. For this example, assume the coordinate origin is bottom left of the window and drawing will be from top to bottom. PERMEDIA can draw from top to bottom or bottom to top.

D1 A Gouraud Shaded Triangle

Consider a triangle with vertices, v_1, v_2 and v_3 where each vertex comprises X, Y and Z coordinates, shown below. Each vertex has a different color made up of red, green and blue (R, G and B) components.

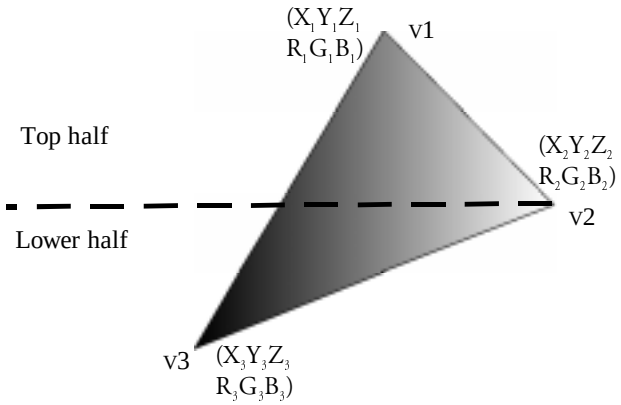


Figure D1 Example Triangle

The diagram makes a distinction between top and bottom halves because PERMEDIA is designed to rasterize screen aligned trapezoids and flat topped or bottomed triangles as shown below:

Figure D2 Screen aligned trapezoid and flat topped triangle

D2 Initialization

PERMEDIA requires many of its registers to be initialized in a particular way, regardless of what is to be drawn; for instance, the screen size and appropriate clipping must be set-up. Normally this only needs to be done once and for clarity this example assumes that all initialization has already been done. More details may be found in the chapter on initialization, chapter §6.

Other state will change occasionally, though not usually on a per primitive basis, for instance enabling Gouraud shading and depth buffering. A detailed treatment will be found in later sections of this chapter, and details are not included here.

D3 Dominant and Subordinate Sides of a Triangle

The dominant side of a triangle is that with the greatest range of Y values. The choice of dominant side is optional when the triangle is either flat bottomed or flat topped.

PERMEDIA always draws triangles starting from the dominant edge towards the subordinate edges. This simplifies the calculation of set-up parameters as will be seen below.

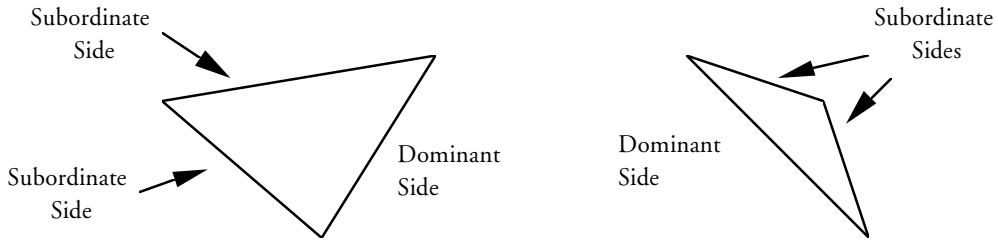


Figure D3 Dominant and Subordinate Sides of a Triangle

D4 Calculating Color values for Interpolation

To draw from left to right and top to bottom, the color gradients (or deltas) required are:

$$dRdy_{13} = \frac{R_3 - R_1}{Y_3 - Y_1} \quad dGdy_{13} = \frac{G_3 - G_1}{Y_3 - Y_1} \qquad dBdy_{13} = \frac{B_3 - B_1}{Y_3 - Y_1}$$

And from the plane equation:

$$dRdx = \{(R_1 - R_3) \times \frac{(Y_2 - Y_3)}{a}\} - \{(R_2 - R_3) \times \frac{(Y_1 - Y_3)}{a}\}$$

$$dGdx = \{(G_1 - G_3) \times \frac{(Y_2 - Y_3)}{a}\} - \{(G_2 - G_3) \times \frac{(Y_1 - Y_3)}{a}\}$$

$$dBdx = \{(B_1 - B_3) \times \frac{(Y_2 - Y_3)}{a}\} - \{(B_2 - B_3) \times \frac{(Y_1 - Y_3)}{a}\}$$

where:

$$a = ABS(\{(X_1 - X_3) \times (Y_2 - Y_3)\} - \{(X_2 - X_3) \times (Y_1 - Y_3)\})$$

These values allow the color of each fragment in the triangle to be determined by linear interpolation. For example, the red component color value of a fragment at X_n,Y_m could be calculated by:

- adding $dRdy_{13}$, for each scanline between Y_1 and Y_n, to R_1.
- then adding dRdx for each fragment along scanline Y_n from the left edge to X_n.

The example chosen has the 'knee' i.e. vertex 2, on the right hand side, and drawing is from left to right. If the knee were on the left side (or drawing was from right to left), then the Y deltas for both the subordinate sides would be needed to interpolate the start values for each color component (and the depth value) on each scanline. For this reason PERMEDIA always draws triangles starting from the dominant edge and towards the subordinate edges. For the example triangle, this means left to right.

D5 Register Set-up for Color Interpolation

For the example triangle the PERMEDIA registers must be set as follows. Details of register formats are given later.

```
// Load the color start and delta values to draw
// a triangle

RStart (R1)
GStart (G1)
BStart (B1)
dRdyDom (dRdy13) // To walk up the dominant edge
dGdyDom (dGdy13)
dBdyDom (dBdy13)
dRdx (dRdx)            // To walk along the scanline
dGdx (dGdx)
dBdx (dBdx)
```

D6 Calculating Depth Gradient Values

To draw from left to right and top to bottom, the depth gradients (or deltas) required for interpolation are:

$$dZdy_{13} = \frac{Z_3 - Z_1}{Y_3 - Y_1}$$

And from the plane equation:

$$dZdx = \{(Z_1 - Z_3) \times \frac{(Y_2 - Y_3)}{a}\} - \{(Z_2 - Z_3) \times \frac{(Y_1 - Y_3)}{a}\}$$

where

$$a = ABS(\{(X_1 - X_3) \times (Y_2 - Y_3)\} - \{(X_2 - X_3) \times (Y_1 - Y_3)\})$$

The divisor, shown here as a, is the same as for color gradient values. The two deltas, $dZdy_{13}$ and dZdx allow the Z value of each fragment in the triangle to be determined by linear interpolation as was described for the color interpolation above.

D7 Register Set-up for Depth Testing

Internally PERMEDIA uses fixed point arithmetic. The formats for each register are described later. Each depth value must be converted into a 2's complement fixed point number and then loaded into the appropriate pair of registers. The 'Upper' or 'U' registers store the integer portion, whilst the 'Lower' or 'L' registers store the fractional bits, left justified and zero filled.

For the example triangle, PERMEDIA would need its registers set-up as follows:

```
// Load the depth start and delta values
// to draw a triangle

ZStartU (Z1_MS)
ZStartL (Z1_LS)
dZdyDomU (dZdy13_MS)
dZdyDomL (dZdy13_LS)
dZdxU (dZdx_MS)
dZdxL (dZdx_LS)
```

D8 Calculating the Slopes for each Side

PERMEDIA draws filled shapes such as triangles as a series of spans with one span per scanline. Therefore it needs to know the start and end X coordinate of each span. These are determined by 'edge walking'. This process involves adding one delta value to the previous span's start X coordinate and another delta value to the previous span's end X coordinate to determine the X coordinates of the new span. These delta values are in effect the slopes of the triangle sides. To draw from left to right and top to bottom, the slopes of the three sides are calculated as:

$$dX_{13} = \frac{X_3 - X_1}{Y_3 - Y_1} \qquad dX_{12} = \frac{X_2 - X_1}{Y_2 - Y_1} \qquad dX_{23} = \frac{X_3 - X_2}{Y_3 - Y_2}$$

This triangle will be drawn in two parts, top down to the 'knee' i.e. vertex 2 and then from there to the bottom. The dominant side is the left side so for the top half:

$$dXDom = dX_{13} \qquad dXSub = dX_{12}$$

The start X,Y, the number of scanlines, and the above deltas give PERMEDIA enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat topped triangle (PERMEDIA is designed to rasterize screen aligned trapezoids and flat topped triangles), the same start position in terms of X must be given twice as **StartXDom** and **StartXSub**.

To edge walk the lower half of the triangle, selected additional information is required. The slope of the dominant edge remains unchanged, but the subordinate edge slope needs to be set to:

$$dXSub = dX_{23}$$

Also the number of scanlines to be covered from Y_2 to Y_3 needs to be given. Finally to avoid any rounding errors accumulated in edge walking to X_2 (which can lead to pixel errors), **StartXSub** must be set to X_2.

D9 Rasterizer Mode

The PERMEDIA Rasterizer has a number of modes which remain effective from the time they are set until they are modified and can thus affect many primitives. In the case of the Gouraud shaded triangle, the default values for these modes are suitable.

```
RasterizerMode (0)     // Default Rasterizer mode
```

D10 Subpixel Correction

PERMEDIA can perform subpixel correction of all interpolated values when rendering aliased trapezoids. This correction ensures that any parameter (color/depth/texture/fog) is correctly sampled at the center of a fragment. In general, subpixel correction will always be enabled when rendering any trapezoid which has interpolated parameters. Control of subpixel correction is in the **Render** command register described in the next section, and is selectable on a per primitive basis. It does not need to be enabled for any primitive that does not use interpolation, including copy operations. If it is disabled and interpolators are used,

the values calculated for the primitive may not be exactly correct; enabling sub-pixel correction may reduce the performance of the chip, particularly for small primitives.

D11 Rasterization

PERMEDIA is almost ready to draw the triangle. Setting up the registers as described here and sending the **Render** command will cause the top half of the example triangle to be drawn.

For drawing the example triangle, all the bit fields within the **Render** command should be set to 0 except the PrimitiveType which should be set to trapezoid and the SubPixelCorrectionEnable bit which should be set to TRUE.

```
// Draw triangle with knee

// Set deltas

StartXDom (X1<<16)     // Converted to 16.16 fixed point
dXDom (((X3 - X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2 - X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<<16)
Count (Y1 - Y2)

// Set the render command mode
render.PrimitiveType = PERMEDIA_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE

// Draw the top half of the triangle

Render (render)
```

After the **Render** command has been issued, the registers in PERMEDIA can immediately be altered to draw the lower half of the triangle. Note that only two registers need be loaded and the command **ContinueNewSub** sent. Once PERMEDIA has received **ContinueNewSub**, drawing of this sub-triangle will begin.

```
// Set-up the delta and start for the new edge

StartXSub (X2<<16)
dXSub (((X3 - X2)<<16)/(Y3 - Y2))

// Draw sub-triangle

ContinueNewSub (Y2 - Y3)      // Draw lower half
```

Appendix E. Register Tables

The following tables list registers by: unit, name and register address, giving their tag values and indicating their type. The register groups may be used to improve data transfer rates to PERMEDIA when using DMA.

The following types of register are distinguished:

- **Control:** Set state and control bits ready to draw a primitive. This is the default and is indicated by a blank entry in the "Type" column.
- **Command:** Initiates some operation e.g. drawing of a primitive.
- **Mixed** A control register which may also be used to supply successive data values during download.
- **Output:** An internal register that cannot be read or written, but whose contents is passed to the Host Out FIFO under the control of certain commands.

In addition the table indicates whether the register can be read back. A blank entry in this column indicates that the register's contents cannot be read back.

The following table is a list of registers in unit order:

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
Delta	V0Fixed[15]	20	0…D		•
	V1Fixed[15]	21	0…D		•
	V2Fixed[15]	22	0…D		•
	V0Float[15]	23	0…D		•
	V1Float[15]	24	0…D		•
	V2Float[15]	25	0…D		•
	DeltaMode	26	0		•
	DrawTriangle	26	1		
	RepeatTriangle	26	2		
	DrawLine01	26	3		
	DrawLine10	26	4		
	RepeatLine	26	5		
Rasterizer	StartXDom	00	0		•
	dXDom	00	1		•
	StartXSub	00	2		•
	dXSub	00	3		•
	StartY	00	4		•
	dY	00	5		•
	Count	00	6		•
	Render	00	7	Command	
	ContinueNewLine	00	8	Command	
	ContinueNewDom	00	9	Command	
	ContinueNewSub	00	A	Command	
	Continue	00	B	Command	
	BitMaskPattern	00	D	Mixed	
	RectangleOrigin	01	A		
	RectangleSize	01	B		
	RasterizerMode	01	4		•
	Ylimits	01	5		•
	WaitForCompletion	01	7	Command	

	Xlimits	01	9		•
	PackedDataLimits	02	A		•
Scissor/Stipple	ScissorMode	03	0		•
	ScissorMinXY	03	1		•
	ScissorMaxXY	03	2		•
	ScreenSize	03	3		•
	AreaStippleMode	03	4		•
	WindowOrigin	03	9		•
	AreaStipplePattern[0..7]	04	0..7		•
LBRead/Write	LBReadMode	11	0		•
	LBReadFormat	11	1		•
	LBSourceOffset	11	2		•
	LBData	11	3		
	LBStencil	11	5	Output	
	LBDepth	11	6	Output	
	LBWindowBase	11	7		•
	LBWriteMode	11	8		•
	LBWriteFormat	11	9		•
Stencil/Depth	Window	13	0		•
	StencilMode	13	1		•
	StencilData	13	2		•
	Stencil	13	3	Mixed	•
	DepthMode	13	4		•
	Depth	13	5	Mixed	•
	ZstartU	13	6		•
	ZStartL	13	7		•
	dZdxU	13	8		•
	dZdxL	13	9		•
	dZdyDomU	13	A		•
	dZdyDomL	13	B		•
Texture Address	TextureAddressMode	07	0		•
	Sstart	07	1		•
	dSdx	07	2		•
	dSdyDom	07	3		•
	Tstart	07	4		•
	dTdx	07	5		•
	dTdyDom	07	6		•
	Qstart	07	7		•
	dQdx	07	8		•
	dQdyDom	07	9		•
Texture Read	TextureBaseAddress	0B	0		•
	TextureMapFormat	0B	1		•
	TextureDataFormat	0B	2		•
	Texel0	0C	0		•
	TextureReadMode	0C	E		•
	TexelLUTMode	0C	F		•
	TexelLUT[0..15]	1D	0..F		•
	AlphaMapUpperBound	1E	3		•
	AlphaMapLowerBound	1E	4		•
	TexelLUTIndex	09	8		•
	TexelLUTData	09	9		•
	TexelLUTAddress	09	A		•
	TexelLUTTransfer	09	B		•

	TextureID	1E	E	Command	•
	TexelLUTID	1E	F	Command	•
YUV	YUVMode	1E	0		•
	ChromaUpperBound	1E	1		•
	ChromaLowerBound	1E	2		•
FBRead/Write	FBReadMode	15	0		•
	FBSourceOffset	15	1		•
	FBPixelOffset	15	2		•
	FBColor	15	3	Output	
	FBData	15	4	Mixed	
	FBSourceData	15	5	Mixed	
	FBWindowBase	15	6		•
	FBWriteMode	15	7		•
	FBHardwareWriteMask	15	8		•
	FBBlockColor	15	9		•
	FBReadPixel	15	A		•
	TextureData	11	D		
	TextureDownloadOffset	11	E		•
	SuspendUntilFrameBlank	18	F	Command	
	FBBlockColorU	18	D		•
	FBBlockColorL	18	E		•
	FBSourceBase	1B	0		•
	FBSourceDelta	1B	1	Command	•
Color DDA	Rstart	0F	0		•
	dRdx	0F	1		•
	dRdyDom	0F	2		•
	Gstart	0F	3		•
	dGdx	0F	4		•
	dGdyDom	0F	5		•
	Bstart	0F	6		•
	dBdx	0F	7		•
	dBdyDom	0F	8		•
	Astart	0F	9		•
	ColorDDAMode	0F	C		•
	ConstantColor	0F	D		•
	Color	0F	E	Mixed	
Texture/Fog/Blend	TextureColorMode	0D	0		•
	FogMode	0D	2		•
	FogColor	0D	3		•
	Fstart	0D	4		•
	dFdx	0D	5		•
	dFdyDom	0D	6		•
	KsStart	0D	9		•
	dKsdx	0D	A		•
	dKsdyDom	0D	B		•
	KdStart	0D	C		•
	dKddx	0D	D		•
	dKddyDom	0D	E		•
	AlphaBlendMode	10	2		•
Color Format	DitherMode	10	3		•
Logical Ops	FBSoftwareWriteMask	10	4		•
	LogicalOpMode	10	5		•
Host Out	FilterMode	18	0		•

	StatisticMode	18	1		•
	MinRegion	18	2		•
	MaxRegion	18	3		•
	ResetPickResult	18	4	Command	
	MinHitRegion	18	5	Command	
	MaxHitRegion	18	6	Command	
	PickResult	18	7	Command	•
	Sync	18	8	Command	
Multiple	Config	1B	2		

Table E1 Registers by Unit

The following table is a list of registers in register order.

Register	Major Group (hex)	Offset (hex)	Type	Readable
AlphaBlendMode	10	2		•
AlphaMapLowerBound	1E	4		•
AlphaMapUpperBound	1E	3		•
AreaStippleMode	03	4		•
AreaStipplePattern[0..7]	04	0..7		•
AStart	0F	9		•
BitMaskPattern	00	D	Mixed	
BStart	0F	6		•
ChromaLowerBound	1E	2		•
ChromaUpperBound	1E	1		•
Color	0F	E	Mixed	
ColorDDAMode	0F	C		•
Config	1B	2		
ConstantColor	0F	D		•
Continue	00	B	Command	
ContinueNewDom	00	9	Command	
ContinueNewLine	00	8	Command	
ContinueNewSub	00	A	Command	
Count	00	6		•
dBdx	0F	7		•
dBdyDom	0F	8		•
DeltaMode	26	0		•
Depth	13	5	Mixed	•
DepthMode	13	4		•
dFdx	0D	5		•
dFdyDom	0D	6		•
dGdx	0F	4		•
dGdyDom	0F	5		•
DitherMode	10	3		•
dKddx	0D	D		•
dKddyDom	0D	E		•
dKsdx	0D	A		•
dKsdyDom	0D	B		•
dQdx	07	8		•
dQdyDom	07	9		•
DrawLine01	26	3		
DrawLine10	26	4		
DrawTriangle	26	1		
dRdx	0F	1		•
dRdyDom	0F	2		•
dSdx	07	2		•
dSdyDom	07	3		•
dTdx	07	5		•
dTdyDom	07	6		•
dXDom	00	1		•
dXSub	00	3		•
dY	00	5		•
dZdxL	13	9		•
dZdxU	13	8		•
dZdyDomL	13	B		•

dZdyDomU	13	A		•
FBBlockColor	15	9		•
FBBlockColorL	18	E		•
FBBlockColorU	18	D		•
FBColor	15	3	Output	
FBData	15	4	Mixed	
FBHardwareWriteMask	15	8		•
FBPixelOffset	15	2		•
FBReadMode	15	0		•
FBReadPixel	15	A		•
FBSoftwareWriteMask	10	4		•
FBSourceBase	1B	0		•
FBSourceData	15	5	Mixed	
FBSourceDelta	1B	1	Command	•
FBSourceOffset	15	1		•
FBWindowBase	15	6		•
FBWriteMode	15	7		•
FilterMode	18	0		•
FogColor	0D	3		•
FogMode	0D	2		•
FStart	0D	4		•
GStart	0F	3		•
KdStart	0D	C		•
KsStart	0D	9		•
LBData	11	3		
LBDepth	11	6	Output	
LBReadFormat	11	1		•
LBReadMode	11	0		•
LBSourceOffset	11	2		•
LBStencil	11	5	Output	
LBWindowBase	11	7		•
LBWriteFormat	11	9		•
LBWriteMode	11	8		•
LogicalOpMode	10	5		•
MaxHitRegion	18	6	Command	
MaxRegion	18	3		•
MinHitRegion	18	5	Command	
MinRegion	18	2		•
PackedDataLimits	02	A		•
PickResult	18	7	Command	•
QStart	07	7		•
RasterizerMode	01	4		•
RectangleOrigin	01	A		
RectangleSize	01	B		
Render	00	7	Command	
RepeatLine	26	5		
RepeatTriangle	26	2		
ResetPickResult	18	4	Command	
RStart	0F	0		•
ScissorMaxXY	03	2		•
ScissorMinXY	03	1		•
ScissorMode	03	0		•
ScreenSize	03	3		•

SStart	07	1		•
StartXDom	00	0		•
StartXSub	00	2		•
StartY	00	4		•
StatisticMode	18	1		•
Stencil	13	3	Mixed	•
StencilData	13	2		•
StencilMode	13	1		•
SuspendUntilFrameBlank	18	F	Command	
Sync	18	8	Command	
Texel0	0C	0		•
TexelLUT[0..15]	1D	0..F		•
TexelLUTAddress	09	A		•
TexelLUTData	09	9		•
TexelLUTID	1E	F	Command	•
TexelLUTIndex	09	8		•
TexelLUTMode	0C	F		•
TexelLUTTransfer	09	B		•
TextureAddressMode	07	0		•
TextureBaseAddress	0B	0		•
TextureColorMode	0D	0		•
TextureData	11	D		
TextureDataFormat	0B	2		•
TextureDownloadOffset	11	E		•
TextureID	1E	E	Command	•
TextureMapFormat	0B	1		•
TextureReadMode	0C	E		•
TStart	07	4		•
V0Fixed[14]	20	0…D		•
V0Float[14]	23	0…D		•
V1Fixed[14]	21	0…D		•
V1Float[14]	24	0…D		•
V2Fixed[14]	22	0…D		•
V2Float[14]	25	0…D		•
WaitForCompletion	01	7	Command	
Window	13	0		•
WindowOrigin	03	9		•
XLimits	01	9		•
YLimits	01	5		•
YUVMode	1E	0		•
ZStartL	13	7		•
ZStartU	13	6		•

Table E2 Registers by Name

The following table is a list of registers in address order.

Major Group (hex)	Offset (hex)	Register	Type	Readable
00	0	StartXDom		•
00	1	dXDom		•
00	2	StartXSub		•
00	3	dXSub		•
00	4	StartY		•
00	5	dY		•
00	6	Count		•
00	7	Render	Command	
00	8	ContinueNewLine	Command	
00	9	ContinueNewDom	Command	
00	A	ContinueNewSub	Command	
00	B	Continue	Command	
00	D	BitMaskPattern	Mixed	
01	4	RasterizerMode		•
01	5	Ylimits		•
01	7	WaitForCompletion	Command	
01	9	XLimits		•
01	A	RectangleOrigin		
01	B	RectangleSize		
02	A	PackedDataLimits		•
03	0	ScissorMode		•
03	1	ScissorMinXY		•
03	2	ScissorMaxXY		•
03	3	ScreenSize		•
03	4	AreaStippleMode		•
03	9	WindowOrigin		•
04	0..7	AreaStipplePattern[0..7]		•
07	0	TextureAddressMode		•
07	1	SStart		•
07	2	dSdx		•
07	3	dSdyDom		•
07	4	TStart		•
07	5	dTdx		•
07	6	dTdyDom		•
07	7	QStart		•
07	8	dQdx		•
07	9	dQdyDom		•
09	8	TexelLUTIndex		•
09	9	TexelLUTData		•
09	A	TexelLUTAddress		•
09	B	TexelLUTTransfer		•
0B	0	TextureBaseAddress		•
0B	1	TextureMapFormat		•
0B	2	TextureDataFormat		•
0C	0	Texel0		•
0C	E	TextureReadMode		•
0C	F	TexelLUTMode		•
0D	0	TextureColorMode		•
0D	2	FogMode		•
0D	3	FogColor		•

0D	4	FStart		•
0D	5	dFdx		•
0D	6	dFdyDom		•
0D	9	KsStart		•
0D	A	dKsdx		•
0D	B	dKsdyDom		•
0D	C	KdStart		•
0D	D	dKddx		•
0D	E	dKddyDom		•
0F	0	RStart		•
0F	1	dRdx		•
0F	2	dRdyDom		•
0F	3	GStart		•
0F	4	dGdx		•
0F	5	dGdyDom		•
0F	6	BStart		•
0F	7	dBdx		•
0F	8	dBdyDom		•
0F	9	AStart		•
0F	C	ColorDDAMode		•
0F	D	ConstantColor		•
0F	E	Color	Mixed	
10	2	AlphaBlendMode		•
10	3	DitherMode		•
10	4	FBSoftwareWriteMask		•
10	5	LogicalOpMode		•
11	0	LBReadMode		•
11	1	LBReadFormat		•
11	2	LBSourceOffset		•
11	3	LBData		
11	5	LBStencil	Output	
11	6	LBDepth	Output	
11	7	LBWindowBase		•
11	8	LBWriteMode		•
11	9	LBWriteFormat		•
11	D	TextureData		
11	E	TextureDownloadOffset		•
13	0	Window		•
13	1	StencilMode		•
13	2	StencilData		•
13	3	Stencil	Mixed	•
13	4	DepthMode		•
13	5	Depth	Mixed	•
13	6	ZStartU		•
13	7	ZStartL		•
13	8	dZdxU		•
13	9	dZdxL		•
13	A	dZdyDomU		•
13	B	dZdyDomL		•
15	0	FBReadMode		•
15	1	FBSourceOffset		•
15	2	FBPixelOffset		•
15	3	FBColor	Output	

15	4	FBData	Mixed	
15	5	FBSourceData	Mixed	
15	6	FBWindowBase		•
15	7	FBWriteMode		•
15	8	FBHardwareWriteMask		•
15	9	FBBlockColor		•
15	A	FBReadPixel		•
18	0	FilterMode		•
18	1	StatisticMode		•
18	2	MinRegion		•
18	3	MaxRegion		•
18	4	ResetPickResult	Command	
18	5	MinHitRegion	Command	
18	6	MaxHitRegion	Command	
18	7	PickResult	Command	•
18	8	Sync	Command	
18	D	FBBlockColorU		•
18	E	FBBlockColorL		•
18	F	SuspendUntilFrameBlank	Command	
1B	0	FBSourceBase		•
1B	1	FBSourceDelta	Command	•
1B	2	Config		
1D	0..F	TexelLUT[0..15]		•
1E	3	AlphaMapUpperBound		•
1E	4	AlphaMapLowerBound		•
1E	E	TextureID	Command	•
1E	F	TexelLUTID	Command	•
1E	0	YUVMode		•
1E	1	ChromaUpperBound		•
1E	2	ChromaLowerBound		•
20	0…D	V0Fixed[14]		•
21	0…D	V1Fixed[14]		•
22	0…D	V2Fixed[14]		•
23	0…D	V0Float[14]		•
24	0…D	V1Float[14]		•
25	0…D	V2Float[14]		•
26	0	DeltaMode		•
26	1	DrawTriangle		
26	2	RepeatTriangle		
26	3	DrawLine01		
26	4	DrawLine10		
26	5	RepeatLine		

Table E3 Registers by Address

Appendix F. PERMEDIA 1 and PERMEDIA 2 Differences

F1 Introduction

This document describes the differences between the original PERMEDIA referred to as PERMEDIA 1 and PERMEDIA 2. Most of these differences are due to additional functionality provided in PERMEDIA 2.

F2 New Units

F2.1 Video Streams

PERMEDIA 2 supports independent input and output of digital video. The input stream complies to the VESA VMI specification. Input data may be scaled and filtered before being written to local memory. The output stream is based on the VMI specification and is designed to work with common PAL/NTSC encoders. Both streams are independent of the video output to the monitor.

The interface may be configured to meet different needs. The table shows the modes supported:

Input width	Output width	Notes
8	8	Simultaneous input and output
16	0	Input only Zoom Video port
0	16	Output only Zoom Video port
8	0	Input data with random access parallel bus

Input data may be scaled and filtered to reduce memory requirements. The output stream may be gamma corrected and converted from RGB to YUV. The output video is a slave and supplies data on demand from the external encoder chip. Both streams support automatic hardware triple buffering.

Separate control is provided for Vertical Blank Interval (VBI) data such as closed caption, Teletext, or Intercast. VBI data may be inserted into the output stream or extracted from the input stream as required.

The interface supports two separate buses for programming devices connected to the video streams. The I2C bus is a two wire serial bus that is commonly used to control chips supplying or receiving data on the video ports. The general purpose bus is a parallel bus that supports a higher bandwidth and uses an eight bit data path with a four bit address. If the parallel bus is used, only input video is available.

In PERMEDIA 2, the external ROM is used to store the Video BIOS and is also used to store the power up configuration information (removing most of the configuration resistors needed for a PERMEDIA 1 design). Access to the ROM is by the general purpose bus during which both video streams are disabled.

F2.2 RAMDAC

PERMEDIA 2 incorporates a high performance 230MHz RAMDAC. Resolutions of up to 1600x1280 @ 85Hz are supported, with a wide variety of pixel formats and a hardware cursor of 64x64x2. There are also integrated phase locked loops for generating all clocks required by PERMEDIA 2.

PERMEDIA 2 directly supports DDC1 and DDC2 monitor configuration, and Apple Macintosh monitor sensing. The DDC2 serial bus is independent of the serial bus in the VMI interface.

F2.3 Delta

The 100MFLOP geometry pipeline processor used in the Delta chip is integrated into PERMEDIA 2. The integrated Delta has been enhanced to support backface culling; this is enabled by in the DeltaMode register, and rejection of positive or negative area triangles i.e. front or back faces, is controlled by the Render command.

A packed color format has been added to the Delta vertex interface allowing all four color components to be loaded in a single 32 bit word. The data should be written to offset 14 of the vertex store as packed 8888 format; the order of the color components within the word can be controlled by the DeltaMode register.

F3 PCI Differences

F3.1 AGP Support

The Advanced Graphics Port extensions to the PCI protocol are supported by PERMEDIA 2. When in an AGP slot, PERMEDIA 2 will function as a 66MHz PCI device, and also perform single edge AGP read master transfers, optionally with sideband addressing.

F3.2 Bypass DMA Engine

A DMA engine has been added to allow high speed transfers from system memory to local memory through the bypass. As the transfer is done the data can be formatted to match the patching organization used by the graphics core texture units. It can also do conversion from YUV420 to YUV422 formats.

F3.3 Host Out DMA engine

A DMA engine has been added to the PCI interface to allow high speed transfers from the graphics core output FIFO to system memory.

F3.4 Extra Interrupts

The following interrupts have been added to PERMEDIA 2.

- Invalid texture
- Bypass DMA complete
- Video stream A interrupt
- Video stream B interrupt
- Video streams external interrupt
- DDC interrupt

F4 Video Unit Differences

F4.1 FIFO Threshold Control

Programmable high and low watermarks have been added to the video FIFO to allow optimum bursting of video data for different screen resolutions.

F4.2 Stereo Control

Support has been added for left and right eye screens that are displayed alternately. An external pin signals which eye is being displayed and may be used to drive LCD shutter glasses.

F4.3 Frameblank Control

PERMEDIA 2 has additional control over behavior at frameblank. It continues to support automatic synchronization to frameblank where the new base address for the screen is only accepted during the vertical blank interval. In addition, it supports a free running mode where the base address is updated immediately without waiting for the blanking period.

PERMEDIA 2 also supports a sync to frame rate mode which only allows the base address to be updated in frameblank while the frame rate keeps up with the monitor refresh rate. If the frame rate drops below the refresh rate the base address is updated immediately.

F5 Core Differences

F5.1 Maximum Screen Size

The maximum screen size has been increased to 2048x2048.

F5.2 Rectangle Primitive

A new primitive is supported for drawing rectangles. It is restricted to integer pixel positions only; rectangles requiring sub-pixel positioning should continue to use the trapezoid primitive. The rectangle is defined with new registers, RectangleOrigin which defines the X and Y start point, and RectangleSize which defines the width and height. The direction in which the rectangle is filled can be controlled by the Render command, with separate control of fill direction in X and Y making the primitive suitable for copy operations.

F5.3 Texture Mapping

The TextureAddressMode option for accurate or fast perspective divide has been removed in PERMEDIA 2. All divides are done faster than the PERMEDIA 1 fast speed, while the accuracy has been improved beyond that of the PERMEDIA 1 accurate mode.

Textures may be stored in system memory or local memory but an individual texture map must not be split across system and local memory.

The base address of the texture map may be accessed indirectly through a table instead of by the TextureBaseAddress register. If the TexureID register is loaded, PERMEDIA 2 will access memory to fetch the actual base address of the texture. Bit 31 of this address is a validity flag, and if set to invalid the graphics pipeline halts and an interrupt is generated. The host can then load the texture through the bypass and restart the graphics core. This mechanism allows efficient texture caching by decoupling the memory management of textures from their use.

F5.4 LUTs

PERMEDIA 2 has a 256 entry texture LUT, each entry is 32 bits wide. It can also be used as 16 smaller LUTs of 16 entries each. The contents of the LUT can be loaded through the graphics pipeline or from memory (local or system). If the LUT is held in memory its address can be loaded indirectly using the same mechanism as the texture caching.

The LUT can be used to index 4 or 8 bit textures, in which case the single index is used to generate all 4 color components. If the texture type has separate color components (i.e. it is not an index) each component is indexed independently through the LUT. This allows color remapping operations such as gamma correction.

The LUT can also be accessed directly from the XY position of the pixel being drawn, and it can hold block fill colors.

F5.5 Block Fills

The block fill color register has been extended to 64 bits to allow greater flexibility. Two new registers have been added, BlockColorUpper and BlockColorLower,

which set the upper and lower 32 bits of the color respectively. If the PERMEDIA 1 BlockColor register is used, its contents are used for both upper and lower halves of the block color giving full backward compatibility.

Texture mapping has been extended to hold block fill masks. Designed specifically for font caching, a byte packed font may be stored in local memory and used to control which pixels are drawn by a block fill.

Any block fill pattern may be stippled using the normal stipple pattern table.

The texture LUT can hold data that is used to update the block fill color on each scanline. This is designed for pattern filling.

F5.6 Sprite Control

The chroma key testing has been extended to improve the quality of cut-outs which have been bilinear filtered, and to smooth the edges of sprites. Two additional registers, AlphaMapUpperBound and AlphaMapLowerBound, have been added to define the range of colors that should have their alpha value mapped to zero. The PERMEDIA 1 chroma key registers are used to reject the pixels with an alpha value not equal to one. Texels that have failed the alpha map test are not included in filtering, so edge effects often seen with filtered cut-outs are removed.

The alpha values of the edge pixels are filtered so that they form a range from one within the area to be drawn to zero within the area not to be drawn. In the region close to the edge of what is to be drawn, the alpha values are filtered to lie between zero and one. The range of alpha values rejected by the chroma key test can be adjusted to allow fine control over the exact size of the cut-out. If blending is enabled then the varying alpha values smooth the transition of the edge of the sprite to the background.

F5.7 Alpha Blending

An optimization has been added to PERMEDIA 2 which reduces the memory bandwidth if blending is enabled. If the alpha value used for blending is derived exclusively from a texture map, the FBReadMode register can be set to disable reading of the framebuffer for any pixels for which the corresponding texel has an alpha value of one. If the alpha value is one, the final color will not include any of the previous framebuffer color so it does not need to be read.

PERMEDIA 2 adds extra control over formatting of the framebuffer color when blending. The AlphaBlendMode register allows control over the way that framebuffer data is mapped to the internal color format. This can prevent visual artifacts when blending with a dithered framebuffer.

F5.8 Color Formats

PERMEDIA 2 performs all internal color calculations at true color accuracy, where PERMEDIA 1 performed 3D calculations at 5 bits per color component.

An additional pixel and texel size has been added to PERMEDIA 2, 24 bits, which has 8 bits of red, green , and blue, but no alpha channel. All pixel operations are available at this size except block fills which are restricted to colors which have all bytes the same value (i.e. shades of grey). This restriction is due to the operation of the memory devices.

F5.9 Miscellaneous

PERMEDIA 2 will calculate the value of FBSourceOffset from an XY delta value, removing a multiply from the set-up needed for a copy operation.

The relative offset field in the FBReadMode register is also in the PackedDataLimits register, the one used will be the last one set before a primitive is drawn. This reduces the number of registers that have to be written for a copy operation.

A register has been added to the PERMEDIA 2 PCI interface which can be used to determine when the chip is idle. If the idle status is set then PERMEDIA 2 will process the next command without delay. It does not mean that all previous operations have completed and data written to memory.

An additional PERMEDIA 2 register can be used to configure a number of controls that are normally set by separate registers. The Config register controls parts of the FBReadMode, FBWriteMode, LogicalOpMode, and ColorDDAMode, registers.

Glossary

accumulation buffer A color buffer of higher resolution than the displayed buffer (typically 16bits per component for an 8bit per component display). Typically used to sum the result of rendering several frames from slightly different viewpoints to achieve motion blur effects or eliminate aliasing effects.

active fragment A fragment which passes all the various culling tests, such as scissor, depth(Z), alpha, etc., is written to/combined with the corresponding pixel in the framebuffer. See also "fragment" and "passive fragment".

aliasing A phenomena resulting from a rendering style which ignores the fact that a pixel may not be wholly covered by a primitive, leading to jagged edges on primitives.

alpha blending The ability to combine supplied Red, Green and Blue color values with those that exist in the framebuffer according to the supplied alpha value. Alpha blending forms the basis for techniques such as transparency and painting.

alpha buffer A memory buffer containing the fourth component of a pixel's color in addition to Red, Green and Blue. This component is not displayed, but may be used for instance to control color blending.

area stipple A two dimensional binary pattern which is used to cull fragments from being drawn.

bitblt Bit aligned block transfer. Copy of a rectangular array of pixels in a bitmap from one location to another.

bitblt double buffering A technique to provide independent windowed double buffering by blting an area from one buffer to the other.

bitplane double buffering A technique whereby fast independent windowed double buffering can be achieved by using a single bitplane bit.

block write A feature provided in some memory devices such as VRAM and SGRAM which allows multiple pixels to be set to a given value by a single write. Fast fill is an alternative name for this feature.

chroma keying Also known as bluescreening, this is the practice of excluding color from an image allowing an underlying image to show through.

chroma test The means by which chroma keying can be achieved.

color index The mode in which the color information is stored for each pixel as a single number, the color index rather than as separate Red, Green, Blue and optionally Alpha values (RGBA mode). Each color index references an entry in a color look up table that contains a particular set of Red, Green and Blue values.

command register A register which when loaded triggers activity in PERMEDIA. For instance the **Render** command register when loaded will cause PERMEDIA to start rendering the specified primitive with the parameters currently set-up in the control registers.

context The state information associated with a particular task. Typically in a system more than one task will be using PERMEDIA to render primitives. Software on the host must save away the current contents of the PERMEDIA control registers when suspending one task to allow another to run, and must restore the state when that task is next scheduled to run.

control register A register which contains state that dictates how PERMEDIA will execute a command.

culling The process of eliminating a fragment, object face, or primitive, so that it is not drawn.

DDA Digital Differential Analyzer. An algorithm for determining the pixels to draw along a line or polygon edge. Also used to interpolate linearly varying values such as color and depth.

delta A gradient of color, fog, depth etc. in the X or Y directions for a primitive.

depth (Z) buffer A memory buffer containing the depth component of a pixel. Used to, for example, eliminate hidden surfaces.

depth-cueing A technique which determines the color of a pixel based on its depth. Used, for instance, to fade far away objects into the background. Also known as fogging.

dithering A rendering style which increases the perceived range of displayed colors at the cost of spatial resolution. The technique is similar to the use of stippled patterns of black and white pixels, to achieve shades of grey on a black and white display.

dominant edge The side of a primitive such as a triangle, which has the greatest range of Y values.

double-buffering A technique for achieving smooth animation, by rendering only to an undisplayed back buffer, and then swapping the back buffer to the front once drawing is complete.

extent checking A technique which determines the rectangular bounds of the area which has been rendered to.

fast fill A feature provided in some memory devices such as VRAM and SGRAM which allows multiple pixels to be set to a given value by a single write. Block write is an alternative name for this feature.

flat shading The constant color shading or area filling of a primitive.

fogging A technique which determines the color of a pixel based on its depth. Used, for instance, to fade far away objects into the background. Also known as depth-cueing.

fragment A fragment is an object generated as a result of the rasterization of a primitive. It corresponds to and contains all the components of a single pixel. If a fragment passes all the various culling tests, such as scissor, depth(Z), stencil, etc., it will be written to/combined with the corresponding pixel in the framebuffer.

framebuffer An area of memory containing the displayable color buffers (front, back, left, right, overlay, underlay), their (optional) associated alpha components, and any associated (optional) window control information. This memory is typically separate from the localbuffer.

Gouraud shading The technique of variable color shading or area filling of a primitive using interpolation to gradually vary the color between vertices. Often known as smooth shading.

hardware writemask A bitmask implemented in memory devices such as VRAM and SGRAM to enable or inhibit the writing of the corresponding bits of a fragment's color into the framebuffer.

host The processor which controls PERMEDIA.

localbuffer An area of memory which may be used to store textures and/or non-displayable depth(Z) and/or stencil pixel information. This memory is typically separate from the framebuffer.

logic ops The technique of applying logical operations such as OR, XOR or AND to the fragment color values and/or those in the framebuffer.

LUT A look-up-table. This normally contains color values to allow mapping from an index value to the desired Red, Green and Blue value.

overlays The technique of ensuring certain drawn objects always remain foremost in view and not obscured by others. Historically this was one method of providing a cursor and was usually achieved by providing extra bit planes.

packed data The arrangement of data in a buffer which allows multiple pixels to be read or written in a single access.

passive fragment A fragment which fails one or more of the various culling tests, such as scissor, depth(Z), stencil, etc., is nor written to/combined with the corresponding pixel in the framebuffer. See also "fragment" and "active fragment".

patched addressing A technique whereby data is organized in memory such that there is improved performance for accesses to adjacent scanlines in a buffer. For PERMEDIA, this is available for depth and/or stencil buffer accesses. For textures a special form, subpatch addressing is provided.

picking A means of selecting drawn objects or primitives .

preMult A method of alpha blending, also known as Ramp blend mode, used by QuickDraw3D.

pixel Picture element. A pixel comprises the bits in all the buffers (whether stored in the localbuffer or framebuffer), corresponding to a particular location in the framebuffer.

primitive A geometric object to be rendered. The PERMEDIA primitives are points, lines, trapezoids (including triangles as a subset), and bitmaps.

Ramp blend mode A method of alpha blending, also known as preMult, used by QuickDraw3D.

rasterization The act of converting a point, line, polygon, or bitmap, in device coordinates, into fragments.

rendering Conversion of primitives in object coordinates into an image.

scissor test A means of culling fragments which lie outside the defined scissor rectangle. The scissor rectangle is defined in device coordinates.

software writemasking A means of simulating hardware writemasking by performing a read-modify-write operation on framebuffer data.

stencil buffer A buffer used to store information about a pixel which controls how subsequent stenciled fragments at the same location may be combined with its current value. Typically used to mask complex two-dimensional shapes.

stipple A one or two dimensional binary pattern which is used to cull fragments from being drawn.

subordinate edge The sides of a primitive such as a triangle, which do not have the greatest range of Y values.

subpatch addressing A technique whereby data is organized in memory such that there is improved performance for accesses to adjacent scanlines in a buffer. For PERMEDIA, this particular form of patched addressing is available for accessing texture maps. See also Patch Addressing.

subpixel correction A means of ensuring that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment's center. This is required, for example, to ensure correct color shading of objects comprised of many primitives.

tag The data item that uniquely identifies a Graphics Core register.

task A process, or thread on the host which uses the PERMEDIA co-processor. Typically tasks assume that they have sole use of PERMEDIA and rely on a device driver to save and restore their PERMEDIA context, when they are swapped out.

texel Texture element. An element of an image stored in texture memory which represents the color of the texture to be applied (fully or in part) to a corresponding fragment.

texture An image used to modify the color of fragments during processing. Often used for instance to achieve high realism in a scene, with relatively few primitives.

texture mapping The process of applying a two dimensional image to a primitive. For instance to apply a wood grain effect to a table.

writemask A bit pattern used to enable or inhibit the writing of the corresponding bits of a fragment's color into the framebuffer. See also Software Writemask and Hardware Writemask.

YUV An alternative color format to RGB, also known as YCbCr. Color format used by MPEG.

Z buffer An alternative name for the depth buffer.

Index

accumulation buffer, 49, 282
active fragment, 282
Address of texture in memory, 236
aliasing, 80, 282
Alpha blend, 99
Alpha blend unit, 28, 104, 123
Alpha Blend unit, 23
Alpha blending, 28, 30, 37, 48, 89, 91, 93, 103, 104, 106, 112, 124, 125, 147, 192, 282
alpha buffer, 80, 103, 148, 240, 282
alpha color, 152, 160, 166, 175, 197, 220
Alpha Map Color Test Lower and Upper Bounds, 149
AlphaBlendMode, 23, 29, 103, 104, 106, 123, 126, 147, 183, 268, 270, 274
Application Initialization, 125
area stipple, 30, 61, 62, 64, 150, 151, 217, 282
Area Stippling, 61
AreaStippleMode, 58, 61, 62, 64, 125, 150, 151, 230, 267, 270, 273
AreaStipplePattern, 15, 64, 151, 267, 270, 273
AStart, 98, 152, 160, 166, 175, 197, 220, 268, 270, 274
Base address of source framebuffer data, 189
bilinear texture mapping, 79, 243
bitblt, 27, 34, 282
bitblt double buffering, 34, 282
bitblt Double Buffering, 31
bitmap, 50, 51
bitmaps, 24, 27
Bitmaps, 49, 51
bitmask, 50, 51, 53, 54, 57, 59, 215, 218
bitmask packing, 54, 59, 215
bitmask pattern, 30
bitmask test, 69, 70, 152, 217
BitMaskPattern, 50, 51, 59, 152, 218, 266, 270, 273
bitmasks, 23
bitplane double buffering, 34, 282
block fills, 51
block write, 30, 52
Block write, 49
block writes, 30, 49, 112, 130, 131, 181, 182, 282
Block Writes, 51
BlockWrites, 30
`BStart`, 98, 99, 107, 153, 262, 268, 270, 274
bypass, 4, 20, 22, 33, 34, 126
Bypass Initialization, 126
byte swap, 7
byte swapped, 54
byte swapping, 22, 54, 59, 93, 215
Byte Swapping, 23
chroma keying, 282
chroma test, 37, 86, 87, 88, 134, 149, 153, 255, 282
Chroma Test, 134
ChromaLowerBound, 88, 153, 268, 270, 275
ChromaUpperBound, 88, 153, 268, 270, 275
CI, 28, 29, 81, 96, 108, 111, 148, 154, 157, 168, 239, 240
CI4, 28
Clears, 130
Color, 14, 19, 28, 92, 104, 114, 122, 154, 194, 218, 268, 270, 274
Color DDA, 53, 197
Color DDA unit, 37, 87, 96, 97, 98, 101, 105, 107, 152, 153, 154, 155, 156, 157, 160, 161, 166, 175, 182, 189, 190, 197, 220, 231, 232, 234, 236, 241, 245, 246, 247, 248, 249, 250, 268
color format, 28, 53, 80, 82, 86, 92, 106, 109, 119, 123, 132, 133, 147, 154, 157, 239
Color Format, 97, 123, 148, 168
Color Format Examples, 109
Color Format unit, 23, 37, 92, 104, 108, 123, 134, 154, 167, 268
Color Format Unit, 108
Color Formats, 29, 108
color formatting, 92, 93
color index, 36
Color Index, 28, 81, 96, 108, 154, 157, 282
color interpolation, 30, 97, 98, 261, 263
Color Interpolation, 262
color order, 29, 82, 148, 168, 239, 240
ColorDDAMode, 98, 99, 107, 125, 155, 157, 268, 270, 274
Command, 266
Command Register, 4, 8, 9, 22, 33, 46, 48, 49, 57, 58, 61, 62, 93, 104, 105, 116, 129, 251, 264, 283
Command Registers, 8, 9
Common Blend Mode, 103
Computer Graphics Principles and Practice, 2
Configuration, 156
ConstantColor, 97, 98, 131, 157, 268, 270, 274
context, 5, 119, 130, 258
context switch, 23, 154
context switching, 22, 112, 130
Continue, 46, 48, 56, 157, 266, 270, 273
Continue commands, 9, 22, 130
ContinueNewDom, 46, 56, 130, 158, 266, 270, 273

ContinueNewLine, 9, 47, 48, 54, 56, 59, 130, 158, 214, 266, 270, 273
ContinueNewSub, 45, 48, 56, 159, 265, 266, 270, 273
Control, 266
Control register, 4, 7, 9, 11, 98, 120, 129, 257, 266, 283
Control registers, 8
Control Registers, 8
Copies, 131
Copy, 52, 65, 66, 89, 91, 95
Count, 56, 57, 157, 158, 159, 266, 270, 273
culling, 60, 194, 207, 208, 209, 210, 212, 225, 229, 283
Data for texture LUT, 231
dBdx, 98, 99, 107, 262, 268, 270, 274
dBdyDom, 98, 99, 156, 161, 182, 189, 190, 231, 232, 234, 236, 241, 245, 246, 247, 248, 249, 250, 262, 268, 270, 274
DDA, 46, 47, 48, 54, 56, 59, 71, 73, 76, 98, 99, 101, 105, 157, 158, 159, 214, 237, 283
decal, 100, 237
delta, 45, 46, 47, 48, 51, 56, 75, 76, 78, 98, 99, 102, 106, 107, 129, 157, 261, 262, 263, 264, 265, 283
Delta, 219
Delta Mode, 161
depth, 192, 199
Depth, 18, 24, 65, 66, 71, 73, 114, 122, 163, 164, 194, 218, 267, 270, 274
depth buffer, 18, 69, 71, 72, 73, 92, 122, 131, 163, 164, 187, 194, 199
depth buffered, 71, 75, 260
depth buffering, 68, 124, 125, 261
Depth Example, 75
Depth gradients, 262
depth interpolation, 263
depth test, 25, 30, 37, 69, 74, 75, 119, 131, 227, 252
Depth test, 70
depth testing, 112
Depth Testing, 263
depth writemask, 73, 75
Depth(Z) buffer, 37, 114, 283
depth-cueing, 101, 283
DepthMode, 71, 73, 75, 125, 146, 164, 267, 270, 274
device ID, 119
device revision, 119
dFdx, 101, 102, 106, 107, 165, 268, 270, 274
dFdyDom, 101, 102, 106, 107, 165, 268, 270, 274
dGdx, 98, 99, 107, 262, 268, 270, 274
dGdyDom, 98, 99, 107, 262, 268, 270, 274
Difference between destination and source data, 190
Dither Example, 109
dithering, 27, 30, 108, 109, 112, 123, 168, 192, 283
Dithering, 37, 108, 109
DitherMode, 23, 29, 35, 85, 108, 109, 110, 113, 123, 167, 183, 268, 270, 274
dKddx, 105, 169, 268, 270, 274
dKddyDom, 105, 169, 268, 270, 274
dKsdx, 105, 170, 268, 270, 274
dKsdyDom, 105, 170, 268, 270, 274
DMA, 10, 12, 13, 14, 18, 20, 22, 127, 128, 258, 266
DMA buffer, 12, 13, 14, 15, 16, 17, 18, 56, 128
DMA Buffer Address, 17
DMA buffers, 20, 128
DMA controller, 10, 12, 16, 17, 21
DMA Example, 16
DMA Interface, 12
DMA interrupts, 17, 18, 128
DMA Tag Format, 14
DMAAddress, 12, 17
DMACount, 12, 13, 17, 18, 21, 258
Dominant, 156, 161, 165, 169, 170, 171, 176, 177, 178, 180, 181, 182, 189, 190, 231, 232, 234, 236, 241, 245, 246, 247, 248, 249, 250
dominant edge, 45, 71, 129, 264
Dominant edge, 158, 261
double buffered, 13, 26, 31, 34
double buffering, 20, 28, 32, 93, 133, 148, 168, 240, 283
Double Buffering, 31
Double Buffering - fast, 133
download, 10, 12, 13, 14, 51, 52, 53, 65, 66, 79, 85, 89, 91, 92, 96, 104, 127, 128, 132, 133, 154, 183, 190, 199, 230, 266
Download, 52, 131
dQdx, 77, 78, 86, 171, 267, 270, 273
dQdyDom, 77, 78, 86, 171, 267, 270, 273
Draw line, 172, 173
Draw Triangle, 174
dRdx, 98, 99, 107, 262, 268, 270, 274
dRdyDom, 98, 99, 107, 262, 268, 270, 274
dSdx, 76, 77, 78, 86, 176, 267, 270, 273
dSdyDom, 76, 77, 78, 86, 176, 267, 270, 273
dTdx, 77, 78, 86, 177, 267, 270, 273
dTdyDom, 77, 78, 86, 177, 267, 270, 273
dX, 47
dXDom, 11, 49, 57, 178, 266, 270, 273
dXSub, 11, 49, 57, 178, 265, 266, 270, 273
dY, 11, 47, 49, 53, 57, 129, 179, 266, 270, 273
dZdxL, 72, 74, 75, 179, 180, 263, 267, 270, 274
dZdxU, 74, 75, 179, 180, 263, 267, 270, 274

dZdyDomL, 72, 74, 75, 180, 181, 263, 267, 270, 274
dZdyDomU, 74, 75, 180, 181, 263, 267, 271, 274
extent checking, 30, 113, 114, 130, 195, 283
Extent Checking, 115
extent collection, 208, 210, 225
extent regions, 117
Fast block fill lower and upper colors, 182
fast fill, 91, 283
FBBlockColor, 30, 181, 268, 271, 275
FBColor, 91, 95, 114, 183, 186, 268, 271, 274
FBData, 92, 104, 122, 132, 183, 218, 268, 271, 275
FBHardwareWriteMask, 30, 93, 95, 124, 184, 268, 271, 275
FBPixelOffset, 26, 32, 33, 89, 90, 94, 122, 185, 268, 271, 274
FBRead, 189, 190
FBReadMode, 26, 27, 30, 85, 89, 90, 92, 94, 111, 112, 120, 122, 123, 125, 132, 186, 189, 211, 259, 268, 271, 274
FBReadPixel, 94, 125, 188, 268, 271, 275
FBSoftwareWriteMask, 30, 93, 111, 113, 124, 184, 189, 268, 271, 274
FBSourceData, 92, 190, 218, 268, 271, 275
FBSourceOffset, 26, 27, 89, 90, 94, 122, 191, 268, 271, 274
FBWindowBase, 26, 90, 94, 121, 124, 191, 268, 271, 275
FBWriteData, 112, 131, 192, 206
FBWriteMode, 85, 94, 104, 124, 193, 268, 271, 275
FIFO Control, 11
Filter Mode Example, 117
FilterMode, 18, 19, 21, 114, 115, 116, 117, 118, 122, 194, 207, 209, 212, 229, 243, 268, 271, 275
flat shaded, 98, 192, 206
flat shading, 37, 111, 157, 284
Flat shading, 97
Flat shading - high speed, 112
Flat Shading example, 98
fog, 49, 58, 99, 103, 106, 165, 172, 173, 174, 192, 195, 196, 197, 218
Fog, 37
Fog Application, 101
Fog DDA, 101
Fog Example, 106
fog interpolation, 102
FogColor, 106, 107, 197, 268, 271, 273
fogging, 105, 172, 173, 174, 196, 218, 284
FogMode, 58, 101, 105, 107, 125, 196, 268, 271, 273
fonts, 52
ForceAlpha, 108
ForceBackgroundColor, 54, 59, 61, 214, 230
fragment, 37, 38, 44, 49, 262, 264, 284
Frame Blank Synchronization, 93
framebuffer, 4, 20, 24, 33, 38, 53, 79, 89, 92, 116, 284
Framebuffer, 26, 121
framebuffer base address, 124, 191
framebuffer clears, 131
Framebuffer Color Formats, 27
Framebuffer coordinates, 26
framebuffer depth, 120
framebuffer format, 36, 92, 93, 99, 104, 108, 157, 183, 192
Framebuffer Read, 112
Framebuffer Read unit, 37, 89, 122, 124, 131
Framebuffer Read/Write units, 89, 181, 183, 184, 185, 186, 188, 190, 191, 193, 211, 228, 238, 241, 268
framebuffer reads, 30, 48, 54, 89, 91, 93, 104, 132, 189
framebuffer units, 87
Framebuffer Write unit, 35, 37, 49, 79, 91, 114, 131, 132
framebuffer writes, 54, 89, 93, 94, 95, 104, 133, 228, 251
FStart, 101, 102, 106, 107, 197, 268, 271, 274
Full Screen Double Buffering, 31
Glossary, 282, 287
glyph, 52
Gouraud shading, 37, 96, 97, 98, 99, 260, 261, 264, 284
Gouraud Shading examples, 98
GP FIFO Interface, 18
Graphics HyperPipeline, 37
GStart, 98, 99, 107, 262, 268, 271, 274
hardware writemask, 34, 49, 91, 129, 184, 189, 284
hardware writemasking, 30, 111
hardware writemasks, 93, 192
Hardware Writemasks, 30, 93
highlight, 100, 105, 237
Hold Format, 14
host, 284
Host Interface, 4
Host Memory Bypass, 21
Host Out FIFO, 21, 53, 56, 122, 183, 199, 202, 207, 209, 212, 229, 266
Host Out Filtering, 113
Host Out unit, 37, 53, 55, 113, 122, 129, 130, 194, 207, 208, 209, 210, 212, 220, 225, 229, 268
Host Out Unit, 122
I/O Interface, 10

Image Formatting, 104
Increment Format, 15
Indexed Format, 15
Indirect handle for texture LUT, 232
Indirect handle for texture map, 241
InFIFOSpace, 10, 11, 20, 128
Iniates loading of LUT data from memory, 234
Initial Blue Color, 153
Initial Green Color, 197
Initial Red Color, 220
Initializing PERMEDIA, 119
input FIFO, 11, 18, 32
Internal Registers, 9
Internal Video Timing, 120
interpolation, 27, 44, 49
Interrupts, 20
invert bitmask, 51, 52, 54, 59
invert bitmasks, 214
invert stencil, 70, 227
invert stipple, 61, 64, 150
KdStart, 105, 198, 268, 271, 274
KsStart, 105, 198, 268, 271, 274
LBData, 132, 164, 199, 228, 267, 271, 274
LBDepth, 66, 199, 267, 271, 274
LBReadFormat, 25, 66, 67, 68, 122, 200, 267, 271, 274
LBReadMode, 25, 65, 66, 67, 68, 120, 122, 123, 124, 201, 259, 267, 271, 274
LBSourceOffset, 25, 65, 66, 68, 70, 122, 202, 267, 271, 274
LBStencil, 66, 202, 267, 271, 274
LBWindowBase, 25, 65, 68, 121, 124, 203, 267, 271, 274
LBWriteFormat, 25, 66, 67, 68, 122, 204, 267, 271, 274
LBWriteMode, 66, 67, 68, 122, 124, 205, 252, 267, 271, 274
LineCount, 33
Lines, 47, 49
localbuffer, 4, 24, 53, 65, 72, 116, 124, 134, 284
Localbuffer, 24, 121
localbuffer clears, 130
Localbuffer Coordinates, 25
Localbuffer example, 68
Localbuffer Read, 65, 67, 68, 132
Localbuffer Read unit, 37, 122, 124, 129
Localbuffer Read/Write units, 65, 199, 200, 201, 202, 203, 204, 205
Localbuffer Reads, 54
Localbuffer Write, 54, 68, 129, 132, 252
Localbuffer Write unit, 37
Logic Op unit, 37, 131, 189, 192, 206
logical op, 51, 192
Logical Op Unit, 111
Logical Operations, 37
logical ops, 9, 27, 30, 37, 48, 89, 91, 93, 111, 112, 125, 206, 284
LogicalOpMode, 112, 113, 126, 192, 206, 268, 271, 274
LUT, 27, 82, 133, 284
  Data for texture, 231
  Indirect handle for texture, 232
  Iniates loading of a LUT from memory, 234
  Texel Mode, 233
MaxHitRegion, 114, 116, 117, 207, 269, 271, 275
MaxRegion, 115, 116, 117, 118, 208, 269, 271, 275
Memory Configuration, 120
Memory I/O and Organization, 24
Memory Organization, 134
Memory Subsystem, 4
MinHitRegion, 114, 116, 117, 209, 269, 271, 275
MinRegion, 115, 116, 117, 118, 210, 269, 271, 275
mirror bitmask, 54, 59, 214
mirror stipple pattern, 150
modulate, 100, 105, 237
Multi-Buffering, 133
nearest neighbour, 81, 243
OpenGL Programming Guide, 2
Origin, 123
Output FIFO, 18
OutputFIFOWords, 19
overlays, 108, 284
Overlays, 133
packed copies, 94, 131, 211
Packed Copies, 92
Packed copy limits, 211
packed data, 284
packed framebuffer, 34
packed mode, 91, 93, 181, 182, 184
packed texture patching, 187
PackedDataLimits, 92, 94, 95, 131, 132, 211, 267, 271, 273
Panning, 34
passive fragment, 284
patch, 24, 67
patched addressing, 92, 187, 201, 285
Patched Data, 24
patched textures, 35
patches, 132
patching, 92, 93, 187
Patching, 92
PCI, 7, 23, 116, 119, 127
PCI burst transfers, 127
PCI bus bandwidth, 127
PCI bus mastership, 128
PCI Disconnect, 10, 128

Picking Example, 117
PickResult, 114, 115, 117, 118, 212, 269, 271, 275
pixel, 285
Pixel Size - setting, 125
Points, 48, 172, 173, 174, 217
preMult, 104, 106, 147, 285
primitive, 285
primitives, 37
procedural texture, 105
procedural textures, 230
Programmed I/O, 127
QStart, 77, 78, 86, 213, 267, 271, 273
Ramp blend mode, 103, 104, 148, 285
ramp texture application, 99, 105, 169, 170, 198, 237
Ramp Texture Application, 100
rasterization, 56, 265, 285
Rasterizer, 19, 35, 46, 49, 50, 51, 52, 53, 112, 131, 132, 133, 158, 230
Rasterizer unit, 37, 44, 49, 57, 104, 129, 152, 157, 158, 159, 172, 173, 174, 178, 179, 214, 216, 217, 219, 223, 224, 251, 253, 254, 264, 266
RasterizerMode, 50, 51, 54, 55, 56, 57, 59, 152, 158, 214, 230, 264, 266, 271, 273
Red and Blue Swapping, 23
Register file, 8
Register load order, 129
Register Read back, 22
Register Tables, 266
Register Types, 8
Register updates - avoiding, 129
Render, 9, 14, 30, 48, 54, 56, 57, 91, 104, 129, 130, 172, 173, 174, 216, 217
rendering, 285
Repeat line, 219
Repeat Triangle, 219
reserved, 146
reset, 112, 119
reset value, 146
ResetPickResult, 8, 115, 117, 118, 220, 269, 271, 275
reuse bitmask, 218, 257
RGB Texture Application, 100
RStart, 98, 99, 107, 262, 268, 271, 274
scissor, 52
Scissor, 37
scissor clip, 8, 9, 30, 52, 55
Scissor example, 63
scissor rectangle, 37
scissor test, 30, 60, 62, 70, 112, 115, 116, 222, 253, 285
scissor tests, 69
Scissor/Stipple tests, 37
Scissor/Stipple unit, 38, 55, 60, 124, 129, 150, 151, 217, 221, 222, 253, 267
ScissorMaxXY, 62, 63, 221, 267, 271, 273
ScissorMinXY, 62, 63, 221, 267, 271, 273
ScissorMode, 62, 63, 121, 222, 267, 271, 273
Screen Clipping Region, 121
screen scissor, 53
screen scissor clip, 121
Screen Scissor Tests, 60
Screen Width, 120
Screen Widths Table, 259
ScreenBase, 32, 33, 34
ScreenSize, 60, 62, 63, 121, 222, 267, 271, 273
ScreenStride, 34
software writemask, 89, 91
software writemask example, 113
software writemasking, 89, 93, 111, 124, 125, 184, 189, 192, 285
Software writemasking, 30
Software Writemasks, 111
Specialized Modes - disabling, 122
SStart, 76, 77, 78, 86, 223, 267, 272, 273
StartX, 47
StartXDom, 9, 53, 57, 59, 214, 223, 264, 266, 272, 273
StartXSub, 53, 57, 59, 214, 224, 264, 265, 266, 272, 273
StartY, 9, 44, 47, 53, 57, 59, 214, 224, 266, 272, 273
Statistic Operations, 114
StatisticMode, 115, 117, 126, 208, 210, 225, 269, 272, 275
stencil, 4, 53, 68, 74, 112, 122, 132, 192, 199, 202, 226, 228
Stencil, 18, 24, 65, 66, 70, 73, 74, 114, 218, 226, 228, 267, 272, 274
stencil buffer, 18, 114, 122, 129, 194, 226, 227, 285
Stencil Example, 74
stencil test, 37
Stencil Test, 69
stencil testing, 30, 65, 68, 72, 124, 226, 227, 252
stencil writemask, 73, 226
Stencil/Depth, 37
Stencil/Depth unit, 37, 68, 163, 164, 179, 180, 181, 226, 227, 252, 256, 267
StencilData, 70, 73, 75, 226, 227, 267, 272, 274
StencilMode, 69, 70, 72, 75, 125, 227, 267, 272, 274
stipple, 285
Stipple, 37
stipple pattern, 37, 50, 64
stipple test, 38, 53, 60, 69, 70, 105, 131, 150, 151
Sub Pixel Precision, 49

subordinate edge, 45, 56, 57, 71, 76, 96, 101, 158, 159, 178, 224, 257, 258, 264, 285
Subordinate edge, 261
subordinate side, 262
subpatch, 187
subpatch addressing, 242, 285
subpatch mode, 24, 82, 85, 86, 92, 242
subpatch pack mode, 92, 187
subpixel correction, 46, 49, 57, 77, 97, 172, 173, 174, 218, 264, 265, 285
SuspendUntilFrameBlank, 20, 33, 93, 133, 228, 268, 272, 275
SVGA, 5, 7, 120
Sync, 8, 48, 55, 116, 122, 130, 229
Sync interrupt, 21
Sync Interrupt Example, 118
Synchronization, 19, 20, 32, 54, 114, 116
System Initialization, 119
tag, 8, 11, 12, 114, 146, 285
task, 286
Task Switching, 5
texel, 286
Texel LUT Mode, 233
Texel0, 50, 54, 59, 61, 86, 87, 105, 150, 153, 215, 218, 230, 267, 272, 273
TexelLUT, 232
TexelLUT[0..15], 82, 230, 267, 272, 275
TexelLUTData, 232
TexelLUTIndex, 232
TexelLUTMode, 82, 125, 267, 272, 273
texture, 24, 49, 52, 80, 81, 87, 99, 103, 132, 134, 172, 173, 174, 192, 218, 238, 286
texture address, 86, 234
Texture Address unit, 35, 37, 76, 79, 171, 176, 177, 213, 223, 234, 244, 267
texture allocation, 121
texture application, 99, 237
Texture Application Example, 106
texture buffer, 4, 24, 28, 187
Texture Buffer, 35
Texture Buffer Coordinates, 35
Texture Color Formats, 36
texture coordinates, 76
texture download, 55, 56, 86, 132, 133, 241, 251
Texture Download, 92
texture download example, 85
Texture Filtering, 80
texture format, 35, 82
Texture Formatting, 80
texture interpolation, 77
Texture Interpolation, 76
Texture Interpolation Example, 78
Texture Loading, 132
texture map, 79, 82, 242
texture mapped, 11
texture mapped trapezoid, 106
texture mapping, 81, 87, 92, 134, 244, 286
Texture Mapping, 37
texture mapping example, 85
texture maps, 79
texture read, 243
Texture Read, 232, 241
Texture Read unit, 23, 27, 35, 37, 50, 55, 77, 79, 105, 132, 230, 233, 239, 242, 243, 267
Texture/Fog/Blend unit, 23, 37, 52, 86, 87, 92, 99, 106, 147, 165, 169, 170, 195, 196, 197, 198, 230, 237, 268
Texture| Read, 149
TextureAddress, 38
TextureAddressMode, 78, 86, 125, 234, 267, 272, 273
TextureBaseAddress, 35, 79, 121, 267, 272, 273
TextureColorMode, 99, 104, 106, 125, 237, 268, 272, 273
TextureData, 35, 93, 132, 238, 241, 268, 272, 274
TextureDataFormat, 23, 29, 80, 82, 86, 239, 267, 272, 273
TextureDownloadOffset, 35, 93, 132, 238, 241, 268, 272, 274
TextureMapFormat, 79, 82, 86, 120, 123, 242, 259, 267, 272, 273
TextureReadMode, 36, 79, 81, 86, 125, 243, 267, 272, 273
textures, 28, 35, 81
Trapezoid Fills, 131
Trapezoids, 44
Triple Buffering, 133
TStart, 77, 78, 86, 244, 267, 272, 273
Unused units - disabling, 129
upload, 18, 19, 53, 62, 65, 66, 89, 91, 92, 93, 95, 96, 104, 183, 186, 193, 194, 199, 202
Upload, 52
uploading, 114
UseConstantFBWriteData, 112
user scissor, 51, 53
User Scissor Test, 60
VBLANK, 33
Vertex 0 data, 245, 246, 247, 248, 249, 250
VGA, 5
VGAControlReg, 120
Video Output, 31
Video Timing, 120
VTG, 120
WaitForCompletion, 55, 56, 91, 132, 186, 201, 251, 266, 272, 273
Window, 72, 73, 74, 123, 124, 129, 252, 267, 272, 274
Window Address and Origin, 123

Window Initialization, 123
WindowOrigin, 60, 62, 253, 267, 272, 273
Windows NT 3.1.Graphics Programming, 2
writemask, 28, 286
writemasking, 27, 30
writemasks, 108
Writemasks, 124
Writing - enabling, 124
X and Y limits, 55
X Derivative - Blue, 160
X Derivative - Green, 166
X Derivative - Red, 175
XLimits, 55, 57, 253, 267, 272, 273
XOR example, 113
Y Derivative Dominant - Blue, 160
Y Derivative Dominant - Green, 166
Y Derivative Dominant - Red, 175
YCbCr, 86
YLimits, 55, 57, 254, 266, 272, 273
YUV, 36, 230, 286
YUV color format, 80, 86, 105, 153
YUV textures, 28
YUV to RGB conversion, 255
YUV unit, 37, 86, 153, 255, 268
YUVMode, 87, 125, 134, 153, 255, 268, 272, 275
Z buffer, 286
ZStartL, 72, 74, 75, 256, 263, 267, 272, 274
ZStartU, 74, 75, 256, 263, 267, 272, 274