3Dlabs®

GLINT® 500TX™

Programmer's Reference Manual

Issue 2

3D*labs Inc.*
181 Metro Drive, Suite 520,
San Jose, CA 95110
United States
Tel: (408) 436 3455
Fax: (408) 436 3458
Email: info@3Dlabs.com
WWW: http://www.3Dlabs.com

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

Change History

Document	Issue	Date	Change
109.6.2	1	29 April 96	First Draft
109.6.3	1	4 June 1997	General Update

Contents

Figures

Tables

1. Introduction

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

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

This document has been written as the reference for programmers and system designers who wish to develop software to drive the GLINT 500TX. For convenience, the GLINT 500TX is referred to throughout simply as GLINT. There are separate manuals for the GLINT 300SX and GLINT Delta processors.

Familiarity with the OpenGL Specification will be useful when reading this document.

1.1 How to use this manual

Chapter 2 gives an overview of GLINT, its capabilities and architecture, and highlights the key differences between the GLINT 300SX and GLINT 500TX.

Chapter 3 details the programming model for the chip, including the DMA interface, and the host framebuffer and localbuffer bypass route.

Chapter 4 describes the hardware data structures that GLINT supports in the framebuffer and the localbuffer.

Chapter 5 describes how to use GLINT for graphics rendering.

Chapter 6 describes the initialization of GLINT.

Chapter 7 discusses programming systems with multiple GLINT chips.

Chapter 8 provides some programming performance tips.

Appendix A details the GLINT graphics registers, their format and use.

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 tabulates the GLINT registers showing the groupings which may be used to improve performance when using DMA.

Appendix E details software compatibility issues between the GLINT 300SX and 500TX.

Appendix F gives example code for rendering a triangle accurately.

A Glossary of technical terms follows the Appendices.

1.2 Further Reading

- *GLINT 500TX Data Sheet,* 3Dlabs
- *GLINT 500TX Hardware Reference Manual ,* 3Dlabs
- *GLINT 500TX Architecture Overview,* 3Dlabs
- *OpenGL Programming Guide*, Jackie Neider et al, Reading MA: Addison-Wesley
- *OpenGL Reference Manual*, Jackie Neider et al, Reading MA: Addison-Wesley
- *The OpenGL Graphics System: A Specification (Version 1.0)* , Mark Segal and Kurt Akeley, SGI (see below)
- *PCI Local Bus Specification Rev2.1* , 1Jun95, PCI Special Interest Group, PO Box 14070, Hillsboro, Oregon 97214 (503-797-4207)
- *Multiprocessor Methods For Computer Graphics Rendering* , Scott Whitman, ISBN 0-86720-229-7
- *Microsoft WIN32 Software Development Kit 3.1* , Microsoft
- *Windows NT 3.1 Graphics Programming*, Emeryville CA, Ziff-Davis Press
- *The X Window System*, Sebastopol CA, O'Reilly & Associates Inc.
- *The X Window System Server*, Elias Israel and Erik Fortune, Digital Press
- *Computer Graphics: Principles and Practice*, James D. Foley et al, Reading MA: Addison-Wesley

1.

2. Architecture Overview

2.1 Functional Overview

GLINT is a single chip 3D graphics processor. It fully implements the functionality of "The OpenGL Machine" from edge walk and span interpolation downwards through fragment level processing including:

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

Systems using GLINT can easily be configured to address a wide range of price, performance and functionality points by simply tuning the external memory design. GLINT supports 4, 8, 16, 20 or 32-bit RGBA and 4 and 8-bit color index framebuffers. The framebuffer can be a maximum of 32Mbytes in size.

2.1.1 Block Diagram

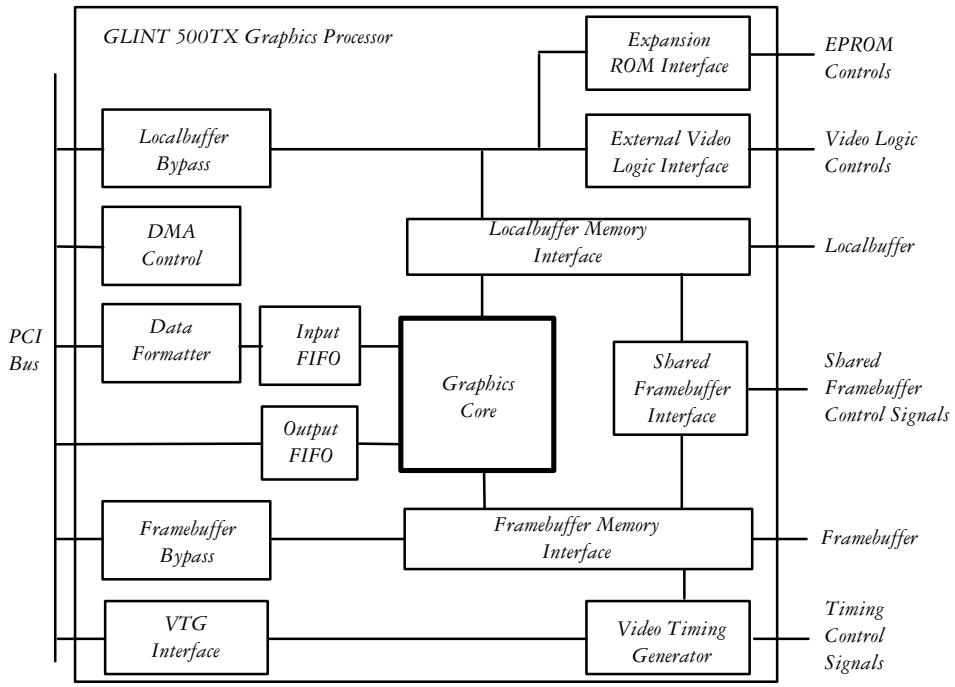


Figure 0.1 High level blocks in the GLINT architecture

The GLINT architecture consists of a Graphics Core augmented by I/O and memory interfaces as shown in Figure 0.1. There are three external interfaces to GLINT: the Host Bus Interface (PCI Local Bus), the Localbuffer Interface and the Framebuffer Interface.

The framebuffer incorporates:

- Color buffer (optionally including back, left and right buffers in addition to the front buffer) up to 32 bit RGBA
- Overlay (optional)
- Underlay (optional)
- Window control buffer (optional)

The localbuffer (any or all of which can be duplicated for the overlays) incorporates:

- Depth (Z) buffer (optional) up to 32 bits
- Stencil buffer (optional) up to 8 bits
- Fast Clear Planes (optional) up to 8 planes
- Pixel Ownership buffer (for optional Graphic ID s) up to 4 bits, to support per pixel clipping
- Texture Map Storage

2.1.2 Host Interface

Conceptually GLINT can be viewed as a register file. Control register s are primed with

the information required for a primitive, and then to start the chip drawing a write is made to a Command register.

Data can be provided to GLINT either using programmed I/O through the FIFO, or using the internal DMA channel. In addition to being able to set any of the standard graphics registers, the GLINT DMA controller accepts data for some common groups of registers in an auto-increment mode to maximize bandwidth. The DMA mode also allows a sequence of data entries to be written to the same register.

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

2.1.3 Task Switching

Where multiple applications wish to make simultaneous access to GLINT, it is the responsibility of the software driving the chip to handle the loading of the correct state. GLINT has been designed to support a number of different software architectures. For instance some of the facilities available are:

- 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
- Sync command to flush all rendering which can be polled or return an interrupt

2.1.4 The GLINT Family

The GLINT 500TX is pin and software compatible with the GLINT 300SX. The major enhancements in the GLINT 500TX are:

- Greatly enhanced hardware support for texture mapping ;
- 50% increase in Gouraud shaded, depth buffered triangle rendering rate;
- 100% increase in peak Gouraud shaded, depth buffered pixel rendering rate;
- Extended Data Out (EDO) DRAMs may be used for the localbuffer;
- Support for multiple GLINT configurations;
- Increased GUI acceleration;
- Support for Apple's QuickDraw 3D texture and alpha blend modes.

The GLINT 500TX registers are a superset of those in the GLINT 300SX.

2.1.5 GLINT 500TX Texture Mapping Enhancements

The GLINT 500TX enhances the texture mapping capabilities of the GLINT 300SX with additional hardware functionality:

- Storage of texture maps in localbuffer;
- Interpolation of texture coordinates;

- Calculation of perceptively correct texture map addresses;
- Retrieval of texture data from the localbuffer memory;
- Formatting of the data from th e wide variety of texture map formats into a uniform internal format;
- Nearest Neighbor, Bilinear or trilinear interpolation of texture values;
- Assistance to the host in doing Mip Mapping by reading the texel data and filtering. The host supplies the two mip map addresses and the inter-map interpolation coefficient.

This extra functionality means that the GLINT 500TX performs all the rendering operations required for nearest neighbor or bilinear texture mapping and greatly assists in performing trilinear mip mapping.

3. Programming Model

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

3.1 GLINT as a Register file

The simplest way to view the interface to GLINT is as a flat block of memory-mapped registers (i.e. a register file). This register file appears as part of Region 0 of the PCI address map for GLINT. See the GLINT Hardware Reference Manual for details of this address map.

When a GLINT 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 32 entry deep FIFO, and each write to a register causes the written value and the register's address tag to be written as a new entry in the FIFO.

Programming GLINT to draw a primitive consists of writing initial values to the appropriate registers followed by a write to a command register. The last write triggers the start of rendering.

GLINT has a few hundred registers. All registers are 32 bits wide and should be 32-bit addressed. Many registers are split into bit fields

Note: *bit 0 is the least significant bit.*

This document describes in detail the graphics registers shown in the text as bold font (for example: **AlphaBlendMode**). In addition there are registers related to initialization and I/O, which are documented in the GLINT Hardware Reference Manual. Where these registers are referred to in the text of this manual, they are shown in italic font, for example: *InFIFOSpace*.

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 sign extended values to be written. These fields are marked as "not used" in Appendix A.

Register Types

GLINT 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 unit min and max registers. Once initialized by the host, the chip only reads these registers to determine the scissor clip extents.

Command Registers are those which, when written to, typically cause the chip to start rendering (some command registers such as ResetPickResult or Sync do not initiate rendering). Normally, the host will initialize the appropriate control registers and then write to a command register to initiate drawing. 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 GLINT when drawing multiple connected objects such as polylines. Examples of command registers include the **Render** and **ContinueNewLine** registers.

Note: For convenience in this document we often refer to "sending a ***Render*** *command to GLINT" 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 GLINT should become familiar with the different types of registers. Some control registers such as the **StartX** and **StartY** registers have to be updated for almost every primitive whereas other control registers such as the **ScissorMaxXY** or the **LogicalOpMode** can be updated much less frequently. Pre-loading of the appropriate control registers can reduce the amount of data that has to be loaded into the chip for a given primitive thus improving efficiency. In addition, as described above, the final values in internal registers can sometimes be

used for subsequent drawing operations.

The table in Appendix D lists the graphics registers according to their type.

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

3.2 GLINT I/O Interface

There are a number of ways of loading GLINT registers for a given context:

- The host writes a value to the mapped address of the register
- The host writes address-tag/data pairs into a host memory buffer and uses the on-chip DMA to transfer this data to the FIFO.
- The host can perform a Block Command Transfer by writing address and data values to the FIFO interface registers.

In cases where the host writes data values directly to the chip (via the register file) it has to worry about 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, 32bit 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 GLINT. PCI Disconnect is enabled by writing to bit zero of the *DisconnectControl* register which is at offset 0x68 in PCI Region0. Once the GLINT is in this mode, if the host processor attempts to write to the full FIFO then instead of the write being lost, the GLINT 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 GLINT, 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 GLINT frees up an entry in its FIFO. In general this mode should only be used either for operations where it is known that the GLINT can consume data faster than the host can generate it, or where there are no time critical peripherals sharing the PCI bus.

Note *If a GLINT Delta geometry processor is in front of the GLINT 500TX, then the PCI Disconnect must always be set on the GLINT 500TX for the secondary PCI bus, and the host PCI bus Disconnect is then controlled by the GLINT Delta, whose DisconnectControl register is at 0x868 in PCI Region0 of the GLINT Delta.*

3.2.2 FIFO control

The description above considered the GLINT 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 GLINT processes this entry. In the case where GLINT is busy performing a time consuming operation (e.g. drawing a large polygon), and not draining the FIFO very quickly, it is possible for the FIFO to become full. If a write to a register is performed when the FIFO is full no entry is put into the FIFO and that write is effectively lost (unless PCI Disconnect is enabled as described above).

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

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

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

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

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

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

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

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

3.2.3 The DMA Interface

Loading registers directly via the FIFO is often an inefficient way to download data to GLINT. Given that the FIFO can accommodate only a small number of entries, GLINT 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 GLINT. If the FIFO is written directly then a return from this function is not possible until almost all the data has been consumed by GLINT. This may take some

time depending on the types of primitives being drawn.

To avoid these problems GLINT 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. At chip reset the MasterEnable bit in the *CFGCommand* register must be set to allow DMA to operate (see the GLINT 500TX Hardware Reference Manual for further details). Then, for the simplest form of DMA, the host software has to prepare a host buffer containing register address tag descriptions and data values. The host then writes the base address of this buffer to the *DMAAddress* register and the count of the number of words to transfer to the *DMACount* register. Writing to the *DMACount* register starts the DMA transfer and the host can now perform other work. In general, if the complete set of rendering commands required by a given call to a driver function can be loaded into a single DMA buffer, then the driver function can return. Meanwhile, in parallel, GLINT 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 (*DMACount != 0)
     ;    // wait for DMA to complete
          // note use a backoff algorithm here
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, GLINT 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 GLINT 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 GLINT processing.

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

```
mark the old buffer as free
return
```

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.

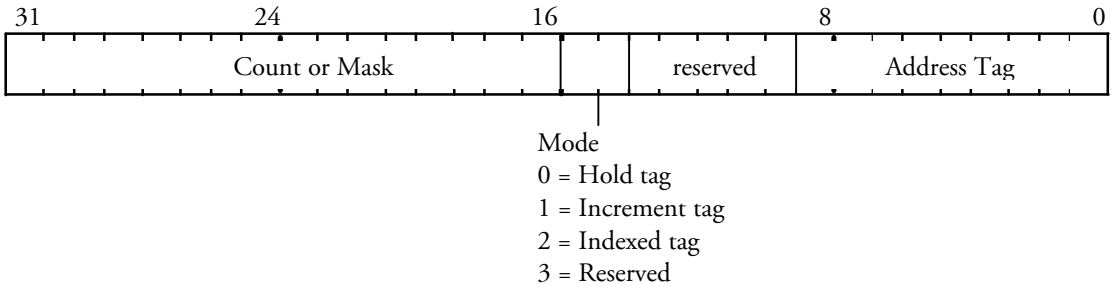


Figure 0.1

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 A.

Hold Format

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

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

A special case of this format is where the top 16 bits are zero indicating that a single data

value follows the tag (i.e the 32-bit tag description is simply the address tag value itself). This allows simple DMA buffers to be constructed which consist of tag/data pairs. For example to render a horizontal span 10 pixels long starting from (2,5) the DMA buffer could look like this:

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

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; GLINT updates an internal copy). Thus, this mode allows contiguous GLINT 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 GLINT register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 32 **AreaStipplePattern** registers can be loaded as follows:

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

Indexed Format

GLINT 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.

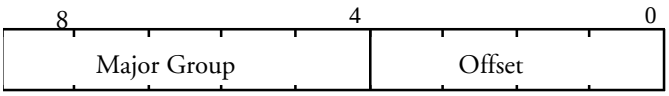


Figure 0.2

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++ = GlintTagdXDom;
*pbuf++ = 0;
*pbuf++ = GlintTagdXSub;
*pbuf++ = 0;
*pbuf++ = GlintTagdY;
*pbuf++ = 1 << 16;
for (i = 0; i < nrects; ++i) {
     *pbuf++ = GlintTagStartXDom;
     *pbuf++ = rect->x1 << 16; // Start dominant edge
     *pbuf++ = GlintTagStartXSub
     *pbuf++ = rect->x2 << 16; // Start of subordinate
     *pbuf++ = GlintTagCount;
     *pbuf++ = rect->y2 - rect->y1;
     *pbuf++ = GlintTagYStart;
     *pbuf++ = rect->y1 << 16;
     *pbuf++ = GlintTagRender;
     *pbuf++ = GLINT_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".

DMA Buffer Addresses

Host software must generate the correct DMA buffer address for the GLINT DMA controller. Normally, this means that the address passed to GLINT must be the physical address of the DMA buffer in host memory. The buffer must also reside at contiguous physical addresses as accessed by GLINT. 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

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

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.

3.2.4 Output FIFO and Graphics Processor FIFO Interfa ce

To read data back from GLINT 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 GLINT 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 § 0.

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 GLINT 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. GLINT will block if the FIFO becomes full. Programmers must be careful to avoid the deadlock condition that will result if the host is waiting for space to become free in the input FIFO while GLINT is waiting for the host to read data from the output FIFO.

Graphics Processor FIFO Interface

GLINT 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 GLINT 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 GLINT DMA it is possible to transfer data to GLINT 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 GLINT 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: *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 GLINT Hardware Reference Manual for more details of the Graphics Processor FIFO Interface address range.

3.3 Other Interrupts

GLINT also provides interrupt facilities for the following:

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

Each of these are enabled and cleared in a similar way to the DMA interrupt. See the GLINT Hardware Reference Manual for more details.

3.4 Synchronization

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

- before reading back from registers
- before directly accessing the framebuffer or the localbuffer via the bypass mechanism
- framebuffer management tasks such as double buffering (though this may be better handled using the **SuspendUntilFrameBlank** command)

Synchronizing with GLINT 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:

```
GlintData 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 GLINT 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 GLINT will generally empty the FIFO more quickly than it takes to set up and handle the interrupt.

3.5 Host Framebuffer Bypass

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

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

A driver making use of the framebuffer bypass mechanism should synchronize framebuffer accesses made through the FIFO, with those made directly through the memory map. If data is written to the FIFO and then an access is made to the framebuffer, it is possible that the framebuffer access will occur before the commands in the FIFO have been fully processed. This lack of temporal ordering is generally not desirable.

Once mapped in, the framebuffer can be read or written with 8, 16 or 32-bit accesses. GLINT does not use bank switching since it is a PCI device and the PCI bus provides a 32 bit address space[1]. With GLINT the complete framebuffer is mapped in as a linear 32-bit addressable memory region.

The framebuffer is accessible via Regions 2 and 4 of the PCI address map for GLINT.

3.5.1 Framebuffer Dimensions and Depth

At reset time the hardware stores the size of the framebuffer in the *FBMemoryControl* register. This register can be read by software to determine the amount of VRAM on the display adapter. For a given amount of VRAM, software can configure different screen resolutions and off-screen memory regions.

The framebuffer width must be set up in the **FBReadMode** register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. Typically, these values will be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register.

Once the offset from one scanline to the next has been established, determining the visible screen width and height becomes a clipping issue. The visible screen width and height are set up in the **ScreenSize** register and enabled by setting the ScreenScissorEnable bit in the **ScissorMode** register.

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

The pixel depth must be set at initialization time. On the GLINT 300SX it was useful to change the pixel depth temporarily to optimize certain 2D rendering operations. This is no longer necessary on the GLINT 500TX due to the introduction of the span

[1]On address limited buses such as ISA, devices limit the amount of address space that they occupy by using bank switching hardware. This typically provides a 64K byte window through which part of the framebuffer is visible. Hardware registers control which part of the framebuffer is visible through this window.

operations discussed later. However code written to use this technique will still work as long as the pixel size is set using the **PixelSize** register. See Appendix E for further details.

3.6 Host Localbuffer Bypass

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

At reset time the hardware saves the size of the localbuffer in the *LBMemoryControl* register (localbuffer visible region size). In bypass mode the number of bits per pixel is either 32 or 64. This information is also set in the *LBMemoryControl* register (localbuffer bypass packing). This pixel packing defines the memory offset between one pixel and the next. A further set of 3 bits (localbuffer width) in the *LBMemoryControl* register defines the number of valid bits per pixel. A typical localbuffer configuration might be 48 bits per pixel but in bypass mode the data for each pixel starts on a 64-bit boundary. In this case valid pixel data will be contained in bits 0 to 47. Software must set the **LBReadFormat**, and **LBWriteFormat** registers to tell GLINT how to interpret these valid bits.

Host software must set the width in pixels of each scanline of the localbuffer in the **LBReadMode** register. The first 9 bits of this register define 3 partial products which determine the offset in pixels from one scanline to the next. As with the framebuffer partial products, these values will usually be worked out at initialization time and a copy kept in software. When this register needs to be modified the software copy is retrieved and any other bits modified before writing to the register. If the system is set up so that each pixel in the framebuffer has a corresponding pixel in the localbuffer then this width will be the same as that set for the framebuffer.

The localbuffer is accessible via Regions 1 and 3 of the PCI address map for GLINT. The localbuffer bypass supports big-endian and little-endian formats. These are described in a later section.

3.7 Register Read back

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

To perform a context switch the host must first synchronize with GLINT. This means waiting for outstanding DMA to complete, 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 GLINT 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: *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 when the actual register is updated.

3.8 Byte Swapping

Internally GLINT operates in little-endian mode. However, GLINT 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, GLINT provides facilities to handle byte swapping. Each of the Configuration Space, Control Space, Framebuffer Bypass and Localbuffer Bypass memory areas have both big and little endian mappings available. The mapping to use typically depends on the endian ordering of the host processor.

The Configuration Space may be set by a resistor in the board design to be either little endian or big endian.

The Control Space in PCI address region 0, is 128K bytes in size, and consists of two 64K sized spaces. The first 64K provides little endian access to the control space registers; the second 64K provides big endian access to the same registers.

The framebuffer bypass consists of two PCI address regions: Region 2 and Region 4. Each is independently configurable by the *Aperture0* and *Aperture1* control registers respectively, to one of three modes: no byte swap, 16-bit swap, full byte swap.

The 16 bit mode is needed for the following reason. If the framebuffer is configured for 16-bit pixels and the host is big-endian then simply byte swapping is not enough when a 32-bit access is made (to write two pixels). In this case, the required effect is that the bytes are swapped within each 16-bit word, but the two 16-bit halves of the 32-bit word are not swapped. This preserves the order of the pixels that are written as well as the byte ordering within each pixel. The 16 bit mode is referred to as GIB-endian in the PCI Multimedia Design Guide, version 1.0.

The localbuffer bypass consists of two PCI address regions: Region 1 and Region 3. Each is independently configurable by the *Aperture0* and *Aperture1* control registers respectively, to one of two modes: no byte swap, full byte swap.

To save on the size of the address space required for GLINT, board vendors may choose to turn off access to the big endian regions (3 and 4) by the use of resistors on the board.

There is a bit available in the *DMAControl* control register to enable byte swapping of DMA data. Thus for big-endian hosts, this control bit would normally be enabled.

See the GLINT Hardware Reference Manual for more details of these control registers.

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 ARGB (alpha at byte 3, red at byte 2, green at byte 1 and blue at byte 0) or ABGR (blue at byte 2 and red at byte 0). The byte position for red and blue may be important for software which has been written to expect one byte order or the other, in particular when handling image data stored in a file.

GLINT provides two registers to specify the byte positions of blue and red internally. In the Alpha Blend Unit the **AlphaBlendMode** register contains a 1-bit field called ColorOrder. If this bit is set to zero then the byte ordering is ABGR; if the bit is set to one then the ordering is ARGB. As well as setting this bit in the Alpha Blend unit, it must also normally be set in the Color Formatting unit, though in some cases it may be useful to set them differently. In this unit the **DitherMode** register contains a Color Order bit with the same interpretation. The order applies to all of the true color pixel formats, regardless of the pixel depth. See § 0 and § 0 for more details of the Alpha Blend and Color Formatting units.

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

1.

4. Hardware Data Structures

4.1 Localbuffer

The localbuffer holds per pixel information corresponding to each displayed pixel. The per pixel information held in the localbuffer are Graphic ID (GID), Depth, Stencil and Frame Count Planes (FCP). The possible formats for each of these fields, and their use are covered individually in the following sections.

In addition spare localbuffer memory may be used to store texture maps. This is discussed further in section 4.1.6 below.

The maximum width of the localbuffer is 48 bits, however this can be reduced by changing the external memory configuration, albeit at the expense of reducing the functionality or dynamic range of one or more of the fields.

The localbuffer memory can be from 16 bits (assuming a depth buffer is always needed) to 48 bits wide in steps of 4 bits. The four fields supported in the localbuffer, their allowed lengths and positions are shown in the following table:

Field	Lengths	Start bit positions
Depth	16, 24, 32	0
Stencil	0, 4, 8	16, 20, 24, 28, 32
FrameCount	0, 4, 8	16, 20, 24, 28, 32, 36, 40
GID	0, 4	16, 20, 24, 28, 32, 36, 40, 44, 48

Table 0.1

In addition there is a compact mode for a 32bit wide localbuffer with depth(24bit), stencil(1bit), FrameCount(4bits) and GID(3bits).

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

The localbuffer data is always formatted into a consistent internal format which is:

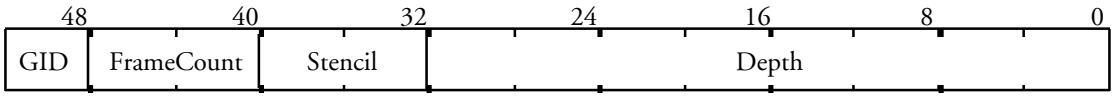


Figure 0.1

The GID, FrameCount, Stencil and Depth fields in the localbuffer are converted into the internal format by right justification if they are less than their internal widths, i.e. the

unused bits are the most significant bits and they are set to 0.

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

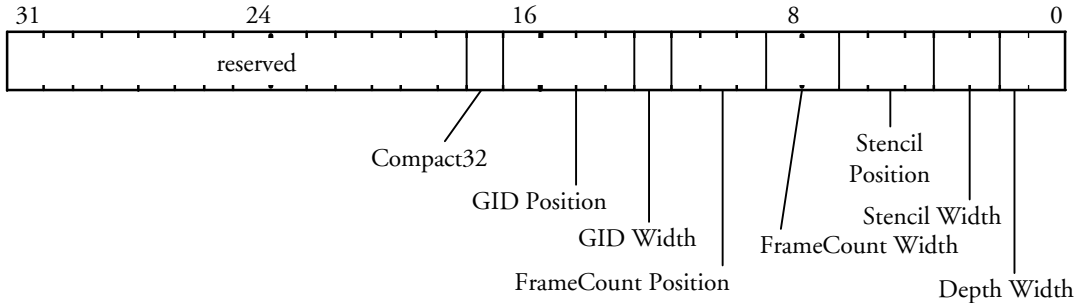


Figure 0.2

It is still possible to part populate the localbuffer so other combinations of the field widths are possible (i.e. depth field width of 0).

Any non-bypass read or write to the localbuffer always reads or writes all 48 bits simultaneously.

4.1.1 GID field

The 4 bit GID field is used for pixel ownership tests to allow per pixel window clipping. Each window using this facility is assigned one of the GID values, and the visible pixels in the window have their GID field set to this value. If the test is enabled the current GID (set to correspond with the current window) is compared with the GID in the localbuffer for each fragment. If they are equal this pixel belongs to the window so the localbuffer and framebuffer at this coordinate may be updated.

Using the GID field for pixel ownership tests is optional and other methods of achieving the same result are:

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

The GID field can be 0 or 4 bits wide. More details on the GID field and these registers may be found in the Graphics Programming chapter.

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

4.1.2 Depth Field

The depth field holds the depth (Z) value associated with a pixel and can be 16, 24 or 32 bits wide.

4.1.3 Stencil Field

The stencil field holds the stencil value associated with a pixel and can be 0, 4 or 8 bits wide, or 1bit wide in the Compact32 mode.

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

4.1.4 FrameCount Field

The Frame Count Field holds the frame count value associated with a pixel and can be 0, 4 or 8 bits wide. It is used during animation to support a fast clear mechanism to aid the rapid clearing of the depth and/or stencil fields needed at the start of each frame.

The fast clear mechanism provides a method where the cost of clearing the depth and stencil buffers can be amortized over a number of clear operations issued by the application. This works as follows:

The system must be configured with 4 or 8 FrameCount planes, such that each pixel has storage for its own corresponding FrameCount value.

The Clear

The area that the application is rendering to comprising say S pixels, is divided up into n regions, where n is the range of the frame counter (for a system with 4 FrameCount planes $n=2^4=16$, with 8 FrameCount planes n=256). Every time the application issues a clear command the reference FrameCount is incremented (and allowed to roll over if it exceeds the maximum value n) and only the i^{th} region is cleared.

The clear of the i^{th} region updates the depth(Z) and/or stencil buffers to their corresponding new values - typically this might be infinity for the depth(Z) and zero for the stencil buffer. At the same time the FrameCount buffer for every pixel in the i^{th} region is updated with the latest reference FrameCount value. The region is smaller than the full region the application specifies to be cleared, so only S/n pixels need to be written. This takes of order $1/n^{th}$ as long as clearing the full S pixels.

Lastly the latest reference FrameCount is stored in the **Window** register, and the depth(Z) and/or stencil value(s) used in the clear , are stored **FastClearDepth** register and the **StencilData** register for later use as detailed below.

Drawing the Next Frame

Now the application starts to render the i^{th} frame. When the localbuffer is read for a depth(Z) comparison, or stencil operation, the FrameCount value for the pixel is also read by the chip and tested against the reference FrameCount in the **Window** register. If the FrameCount values are found to be the same, then the localbuffer data is used directly.

However, whenever the FrameCount is found to be different from the reference FrameCount, the data which would have been written if all S pixels in the localbuffer had been cleared (contained in the **FastClearDepth** and/or **StencilData** registers), is substituted by the chip for the stale data returned from the read.

In any new writes to the localbuffer, the chip will set the FrameCount to the reference value held in the **Window** register, thus the next read on this pixel will not return stale data and will not result in a substitution.

Other Considerations

The fast clear mechanism does not present a total solution as the application can elect to clear just the stencil planes or just the depth planes, or both. The situation where the stencil planes only are 'cleared' using the fast clear method, then some rendering is done and then the depth planes are 'cleared' using the fast clear will leave ambiguous pixels in the localbuffer. The driver software will need to catch this situation, and fall back to using a per pixel write to do the second clear. Which field(s) the frame count plane refers to is recorded in the **Window** register.

When clear data is substituted for real memory data (during normal rendering operations) the depth writemask and stencil writemasks are ignored to mimic the operation of clearing under OpenGL.

In addition to the fast clear mechanism the extent of all updates to the localbuffer and framebuffer can be recorded (**MinRegion** and **MaxRegion** registers) and read back (**MinHitRegion** and **MaxHitRegion** commands) to give the bounding box of the smallest area to clear. For some applications this will be significantly smaller than the whole window or screen, and hence faster.

4.1.5 Texture Map Storage

To achieve high texture mapping performance, GLINT stores the texture maps in a texture store within the localbuffer. The texture store occupies the spare localbuffer entries after each pixel has allocated an entry for depth, stencil etc., i.e. the texture store and the per pixel buffers occupy distinct address spaces.

Each entry in the texture store contains 32 bits. If localbuffer entries contain more than 32 bits, then the extra bits will not be used for texture storage. If a texture map is less than 32 bits deep, then the entries will be packed into the 32 bit words, e.g. if a texture map is 8 bits deep, then each 32 bit word in the texture store will contain 4 texture map entries.

Increasing the size of the texture store will increase both image quality (larger textures may be used) and rendering performance (textures will not have to be swapped in and out of the texture store). Hence GLINT based designs should include as much texture store as possible within the given design and price constraints. Typical high end graphics workstations contain 4 Mbytes of texture store.

The localbuffer should optimally be organized as two separate physical banks, with the Depth(Z) buffer configured by software to be in one bank, and texture storage allocated in the other bank. This is because the GLINT supports two pagemode detectors, allowing Depth(Z) buffered texture mapped rendering to be carried out without forcing a page break on every localbuffer memory access.

4.1.6 Calculating The Required Localbuffer Size

The required localbuffer size can be calculated using the following steps:

1. Choose the number of bits in the depth field. The typical options are 16 or 24 bits.

2. Choose the number of bits in the stencil field. If the design should support OpenGL, then the typical choice is 4 bits.
3. Choose the number of bits in the fast clear control field. This is optional but is typically 4 bits.
4. Choose the number of bits in the Graphics ID field. For designs which support X Windows this may be 4 and otherwise it's typically zero.
5. Add the answers from steps 1 to 4 together. This gives a minimum number of bits for each localbuffer entry.
6. If the value from step 5 is greater than 32, then use this value as the number of bits for each localbuffer entry, otherwise use 32.
7. Determine the maximum screen resolution. The localbuffer should contain an entry for each pixel on the screen. So the maximum screen resolution and the number of bits for each localbuffer entry give a minimum size for the localbuffer.

   For example, if the maximum screen resolution is 1024 x 768 and the number of bits for each localbuffer entry is 32, then the minimum local buffer size is 1024 x 768 x 4 bytes = 3 Mbytes.
8. Space should now be allocated for the texture store. This space should typically be 4 Mbytes. The texture storage space plus the value from step 7 give the required localbuffer size.

Here is an example configuration:

Max screen resolution	1152 x 900
Depth field	24 bits
Stencil field	4 bits
Fast Clear Control	4 bits
Graphics ID	None
Texture Store Entry	32 bits
Texture Store Size	4 Mbytes†
Localbuffer Size (e.g. four 2M x 8 bit devices)	8 Mbytes

Note†: *The available texture store size will increase if a lower screen resolution is used with the same size localbuffer.*

In a multi-GLINT design the localbuffer storage may be duplicated.

4.1.7 Localbuffer Coordinates

The coordinates generated by the rasterizer [1] are 16 bit 2's complement numbers, and so have the range +32767 to -32768. The rasterizer will produce values in this range, however any which have a negative coordinate, or exceed the screen width or height (as programmed into the **ScreenSize** register) are discarded.

[1]The input co-ordinates to the rasterizer are in 16.16 fixed point format.

Coordinates can be defined window relative or screen relative and this is only relevant when the coordinate gets converted to an actual physical address in the localbuffer. In general it is expected that the windowing system will use absolute coordinates and the graphics system will use relative coordinates (to be independent of where the window really is).

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

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

Bottom left origin

```
Destination address = LBWindowBase - Y/S * W + X
Source address = LBWindowBase - Y/S * W + X + LBSourceOffset
```

Top left origin

```
Destination address = LBWindowBase + Y/S * W + X
Source address = LBWindowBase + Y/S * 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.
`S`	is the Scanline interval for multi-GLINT systems
`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.

These address calculations translate a 2D address into a linear address.

Note: *Turning on Patch addressing introduces additional complexity into the address calculation which is beyond the scope of this manual. Localbuffer bypass accesses are not recommended when Patch mode addressing is enabled.*

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 **LBReadMode** register. The range of widths supported by this technique 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 width screens, for instance bitmaps in 'off screen' memory, 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. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.*

4.2 Framebuffer

The framebuffer is a region of memory where the information produced during rasterization is written prior to being displayed. This information is not restricted to color but can include window control data for LUT management and double buffering [1].

The framebuffer region can hold up to 32MBytes and there are very few restrictions on the format and size of the individual buffers which make up the video stream. Typical buffers include:

- True color or color index main pl anes,
- Overlay planes,
- Underlay planes,
- Window ID planes for LUT and double buffer management,
- Cursor planes.

Any combination of these planes can be supported up to a maximum of 32MBytes, but usually it is the video level processing which is the limiting factor. The remainder of this section examines the options and choices available from GLINT for rendering, copying, etc. data to these buffers. The necessary video hardware, and how it is controlled is outside the scope of this document.

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. This is described in more detail below.

4.2.1 Buffer Organization

Each buffer resides at an address in the framebuffer memory map. For rendering and copying operations the actual buffer addresses can be on any pixel boundary. Display hardware will place some restrictions on this as it will need to access the multiple buffers in parallel to mix the buffers together depending on their relative priority, opacity and double buffer selection. For instance, visible buffers (rather than offscreen bitmaps) will typically need to be on a page boundary.

Consider the following highly configured example with a 1280x1024 double buffered system with 32 bit main planes (RGBA), 8 bit overlay and 4 bits of window control information (WID). The framebuffer memory map for this example is shown below:

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

Combining the WID and overlay planes in the same 32 bit pixel has the advantage of reducing the amount of data to copy when a window moves, as only two copies are required - one for the main planes and one for the overlay and WID planes.

Note the position of the overlay and WID planes. This was not an arbitrary choice but one imposed by the (presumed) desire to use the color processing capabilities of GLINT (dither and interpolation) in the overlay planes. The conversion of the internal color format to the external one stored in the framebuffer depends on the size and position of the component. The possible formats are given in Error! Reference source not found. . Note that GLINT does not support all possible configurations. For example; if the overlay and WID bits were swapped, then eight bit color index starting at bit 4 would be required to render to the overlay, but this is not supported.

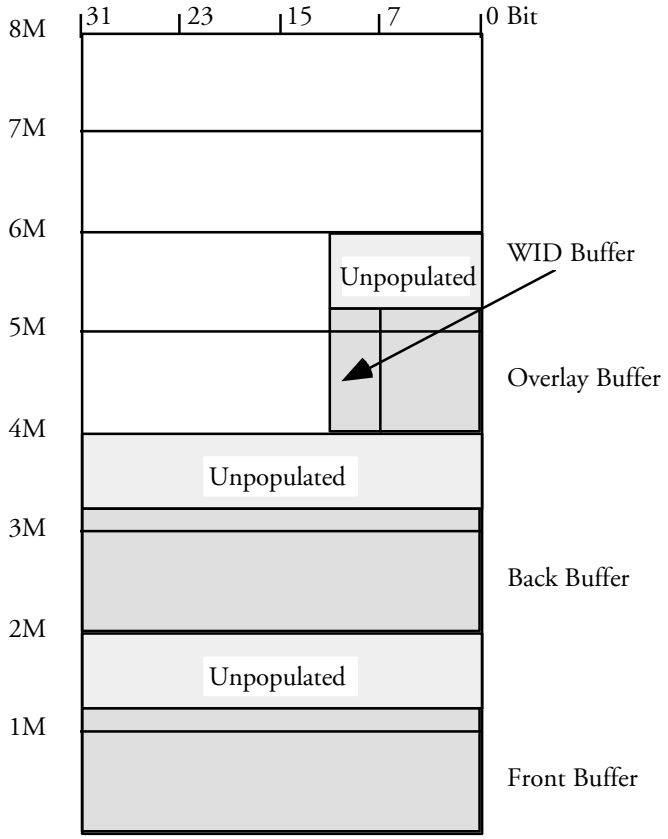


Figure 0.3 Example memory organization

4.2.2 Framebuffer Coordinates

Coordinate generation for the framebuffer is similar to that for the localbuffer, however, there are some key differences.

As was mentioned before, the coordinates generated by the rasterizer [1] are 16 bit 2's complement numbers. Coordinates can be defined as window relative or screen relative, though this is only relevant when the coordinate gets converted to an actual physical

[1]The input co-ordinates to the rasterizer are in 16.16 format.

address in the framebuffer. The WindowOrigin bit in the **FBReadMode** register selects top left (0) or bottom left (1) 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/S * W + X +
                                                FBPixelOffset

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

Top left origin

```
Destination address = FBWindowBase + Y/S * W + X +
                                                FBPixelOffset

Source address = FBWindowBase + Y/S * W + X + FBPixelOffset +
                                                FBSourceOffset
```

where:

`X`	is the pixel's X coordinate,
`Y`	is the pixel's Y coordinate,
`S`	is the scanline interval for multi-GLINT systems
`FBWindowBase`	holds the base address in the framebuffer of the current window.
`FBPixelOffset`	is normally zero except when multi-buffer writes are needed[1] 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 (i.e. 1280) are economical in memory.

The 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. This is the same mechanism as is used to set the width of the

[1]OpenGL, for example, allows any combination of the Front, Back, Left and Right color buffers to be updated 'simultaneously'.

localbuffer, however the widths may be set independently. The range of widths supported by this technique 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. However, often windowing systems store offscreen bitmaps in rectangular regions which use the same stride as the screen. In this case normal bitblts can be used.

4.2.3 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 GLINT is concerned. Bit planes may be allocated to control cursor, LUT, multi-buffer visibility or priority functions. In this case GLINT 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. Obviously this technique can also be used for overlay buffers, etc. providing color processing is not required.
- 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 of 8 bits per red, green, blue and alpha color channels. 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 Error! Reference source not found. . 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. The ColorOrder is specified by a bit in the **DitherMode** register.

Some important points to note:

- The alpha channel 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. If the framebuffer is not configured with an alpha channel (e.g. 24 bit framebuffer width with 8:8:8:8 RGB format) then some of the rendering modes which use the retained alpha buffer cannot be used. In these cases the NoAlphaBuffer bit in the **AlphaBlendMode** register should be set so that an alpha value of 255 is substituted. For the RGB modes where no alpha channel is present (e.g. 3:3:2) then this substitution is done

automatically.

- For the Front and Back modes the data value is replicated into both buffers. Note though the Front and Back modes are identical, the redundant modes are included for symmetry with the Color format field of the **AlphaBlendMode** register.
- All writes to the framebuffer try to update all 32 bits irrespective of the color format. This may not matter if the memory planes don't exist, but if they are being used (as overlay planes, for example) then the writemasks (**FBSoftwareWriteMask** or **FBHardwareWriteMask**) must be set up to protect the alternative planes.

			Internal Color Channel			
	Format	Name	R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4Front	4@0	4@8	4@16	4@24
	4	4:4:4:4Back	4@4	4@12	4@20	4@28
	5	3:3:2Front	3@0	3@3	2@6	255
	6	3:3:2Back	3@8	3@11	2@14	255
	7	1:2:1Front	1@0	2@1	1@3	255
	8	1:2:1Back	1@4	2@5	1@7	255
	13	5:5:5Back	5@16	5@21	5@26	255
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4Front	4@16	4@8	4@0	4@24
	4	4:4:4:4Back	4@20	4@12	4@4	4@28
	5	3:3:2Front	3@5	3@2	2@0	255
	6	3:3:2Back	3@13	3@10	2@8	255
	7	1:2:1Front	1@3	2@1	1@0	255
	8	1:2:1Back	1@7	2@5	1@4	255
	13	5:5:5Back	5@26	5@21	5@16	255
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Table 0.2

- When reading the framebuffer RGBA components are scaled to their internal width of 8 bits, if needed for alpha blending.

  CI values are left justified with the unused bits (if any) set to zero and are subsequently processed as the red component. The result is replicated into each of the streams G,B and A giving four copies for CI8 and eight copies for CI4.

- The 5:5:5 Back format is designed to support multiple independent

15bpp double buffered windows, on systems which have a RAMDAC that can select the front and back buffer on a per pixel basis based on the top bit of the 32bit pixel stream. The front or back buffer may be selected for writing using writemasking.

- The 4:4:4:4 Front and Back formats are designed to support 12 bit double buffering with 4bit Alpha, in a 32 bit system.
- The 3:3:2 Front and Back form ats are designed to support 8 bit double buffering in a 16 bit system.
- The 1:2:1 Front and Back formats are designed to support 4 bit double buffering in an 8 bit system.
- It is possible to have a color index buffer at other positions as long as reduced functionality is acceptable. For example a 4 bit CI buffer at bit position 16 can be achieved using writemasking and 4:4:4:4 Front format with color interpolation, however dithering is lost.

The format information needs to be stored in two places: the **DitherMode** register and the **AlphaBlendMode** register.

4.2.4 Overlays and Underlays

In a GUI system there are two possible relationships between the overlay planes (or underlay) and the main planes.

- The overlay planes are fixed to the main planes, so that if the window is moved then both the data in the main planes and overlay planes move together.
- The overlay planes are not fixed to the main planes but floating, so that moving a window only moves the associated main or overlay planes.

In the fixed case both planes can share the same GID. The pixel offset is used to redirect the reads and writes between the main planes and the overlay (underlay) buffer. The pixel ownership tests using the GID field in the localbuffer work as expected.

In the floating case different GIDs are the best choice, because the same GID planes in the localbuffer can not be used for pixel ownership tests. The alternatives are not to use the GID based pixel ownership tests for one of the buffers but rely on the scissor clipping, or to install a second set of GID planes so each buffer has it's own set. GLINT allows either approach.

If rendering operations to the main and overlay planes both need the depth or stencil buffers, and the windows in each overlap then each buffer will need its own exclusive depth and/or stencil buffers. This is easily achieved with GLINT by assigning different regions in the localbuffer to each of the buffers. Typically this would double the localbuffer memory requirements.

One scenario where the above two considerations do not cause problems, is when the overlay planes are used exclusively by the GUI system, and the main planes are used for the 3D graphics.

4.2.5 VRAM Modes

High performance systems will typically use VRAM for the framebuffer and the extended functionality of VRAM over DRAM can be used to enhance performance for many rendering tasks.

Hardware Writemasks.

These allow writemasking in the framebuffer without incurring a performance penalty. If hardware writemasks are not available, GLINT 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, 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, stencil or GID testing can be done
- All the pixels must be written with the same value so no color interpolation, blending, dithering or logical ops can be done

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

The following registers need to be set up before block fills can be used:

FBBlockColor register with the value to write to each pixel

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 of a trapezoid any inappropriate state is ignored so even if color interpolation, depth testing and logical ops, for example, are enabled they have no effect.

See the discussion on span operations later in this manual for further details.

4.3 Double Buffering

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

Which techniques are available will depend on the board design, however, this section discusses how GLINT may be used to provide support for four common types of double buffering, assuming that the framebuffer memory and LUT-DAC have the necessary capabilities.

- BitBLT
- Full Screen
- Bitplane
- Colorspace

4.3.1 BitBlt Double Buffering

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

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

4.3.2 Full Screen Double Buffering

This section describes how to implement full-screen double buffering with GLINT when using the internal timing generator. To perform full-screen double buffering the available VRAM 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 VRAM at which a given buffer starts. This offset is used to program various GLINT registers. For a given resolution and pixel depth there must be enough VRAM configured on the display adapter for this to be possible. For example, with 32 bit deep pixels and 4MB of VRAM 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 GLINT 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.

Some sample code to work out the location in VRAM of a second buffer for the purposes of full-screen double buffering is given below.

Video Output

To display a given buffer, the video output hardware must be programmed with the

offset of that buffer in VRAM. In the GLINT internal timing generator this is controlled by the VTGFrameRowAddr register located in the GLINT control space at offset 0x3068. It is updated immediately it is written, but is not used by the video hardware until the start of the next frame. This register contains a count measured in RAS (row address) length units. A RAS length unit is the number of bytes that make up a VRAM row address line. This value will be board rather than GLINT specific, and may be calculated as follows:

RLP = 1024 * (4/BYP) * IL

where:

RLP = RAS length in pixels

BYP = bytes per pixel

IL = 1 for non-interleaved VRAMs
2 for 2 way interleaved VRAMs
4 for 4 way interleaved VRAMs

For example on a board which uses 2 way interleaved VRAMS the length in pixels would be:

8bpp: 1024 * (4/1) * 2 = 8K pixels

16bpp: 1024 * (4/2) * 2 = 4K pixels

32bpp: 1024 * (4/ 4) * 2 = 2K pixels

The interleave value can be worked out by reading the FBModeSel control space register. This is described in the GLINT 500TX Hardware Reference Manual. The value 1024 is related to the width of the video shift register and never changes.

Note that for a given board design RLP will depend on the pixel depth only and not the resolution.

The value loaded into the VTGFrameRowAddress register is multiplied by the RAS line length to give an offset into VRAM at which to start scanning pixels for the currently displayed buffer. This means that a given buffer must start on a RAS line boundary.

One common configuration for a double buffered system is to position buffer 0 at RAS line 0, and buffer 1 at the first RAS boundary after the end of buffer 0. Note that in this case the pixel coordinates of the start of buffer 1, may have an X coordinate which is not zero. It depends on whether the pixel coordinate at the start of the first scanline past the end of the screen lies on the correct boundary.

Here are some examples for 32 bit pixels on a 2 way interleaved board:

640x480: Buffer 0 at RAS 0, coordinates (0, 0).
Buffer 1 at RAS 150, coordinates (0, 480).

800x600: Buffer 0 at RAS 0, coordinates (0, 0).
Buffer 1 at RAS 235, coordinates (480, 601).

1024x768: Buffer 0 at RAS 0, coordinates (0, 0).
Buffer 1 at RAS 384, coordinates (0, 768).

For most standard resolutions, except 800x600, the start of the first scanline after the visible screen coincides with a RAS boundary. Hence, in the examples, the pixel coordinates of the start of buffer 1 have an X value of 0 and a Y value equal to the screen height.

The 800x600 resolution is different, since 800*600 = 480000 is not divisible by 2K. In this case the first RAS boundary after the end of buffer 0 lies at a pixel coordinate with X = 480 and Y = 601. In other words, from the end of buffer 0, slightly more than one and a half scanlines must be skipped to get to the next boundary. It does not matter that, conceptually, this position is not aligned with the left edge of the screen when buffer 0 is being displayed.

To swapbuffers the *VTGFrameRowAddress* register is loaded with the RAS line value for the buffer to be displayed.

GLINT Rendering

The video output hardware (when using the internal timing generator) restricts the position of each buffer to be on a RAS boundary. When determining the VRAM location of a pixel being rendered GLINT works in screen coordinates. Thus we need to translate the RAS address of the start of a buffer into a pixel position in screen coordinates. We do this as follows:

Y = (RA * RLP) / WP

X = (RA * RLP) % WP

where:

Y =	Y position in screen coordinates
X =	X position in screen coordinates
RA =	RAS line value for the given buffer
RLP =	RAS length in pixels
WP =	width of the screen in pixels

For example, at a pixel depth of 32 and a screen resolution of 800x600, as noted above RA = 235 and RWP = 2048 pixels. So the (X, Y) coordinates of buffer 1 are:

Y = (235 * 2048) / 800 = 601

X = (235 * 2048) % 800 = 480

Hence buffer 1 starts at (480, 601).

To simplify the calculation of pixel coordinates that are loaded into GLINT, this value may be loaded into the **FBPixelOffset** register. The last thing GLINT 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 VRAM. When swapping buffers it is normal to move this position to be the pixel at which a given buffer starts. Thus, in the example just given, to start

rendering into buffer 1, we would load (800 * 601) + 480 = 481280 into the **FBPixelOffset** register.

To summarize, generally buffer 0 will be at RAS value 0 and screen coordinates (0, 0). So to display buffer 0 we load 0 into VTGFrameRowAddress and to render into buffer 0 we load 0 into the **FBPixelOffset** register. Buffer 1 will normally live at some offset into VRAM. As an example, for 32 bpp at 800x600 as worked out above, we load 235 into VTGFrameRowAddress to display buffer 1 and we load 481280 into **FBPixelOffset** to render into buffer 1.

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

Synchronization

The commonest use of double buffering is to display one buffer (the front buffer) while rendering into the other (the back buffer). When the rendering has been completed to this 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 breakup.

GLINT reads the VTGFrameRowAddr 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:

```
VTGFrameRowAddr = 0              // display buffer 0
FBPixelOffset = Buf1_Offset      // draw to buffer 1 now
Render Commands                  // draw next frame
VTGFrameRowAddr = Buf1_RAS       // display buffer 1
FBPixelOffset = 0                // draw to buffer 1 now
Render Commands                  // draw next frame
```

There are two problems here. Firstly, even though the write to the VTGFrameRowAddr register happens immediately, GLINT 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 GLINT executes the command. Accesses to the VTGFrameRowAddr register bypass the FIFO so it is possible for the host to update it, and for the buffer swap to happen, before GLINT has completed rendering the last frame.

The GLINT 500TX includes the **SuspendUntilFrameBlank** command to solve these problems without the need for the host synchronizing with GLINT. Here is the correct 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 GLINT to swap the buffers automatically when the VBLANK occurs either by loading a new buffer offset into the *VTGFrameRowAddr* register as discussed above, or by updating one or more registers in the RAMDAC where colorspace double buffering is being used. 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 GLINT without risk of them affecting the displayed buffer.

As a general performance note, it is best to send non-framebuffer related commands to GLINT following the **SuspendUntilFrameBlank** command. For example, any commands to clear the Z buffer between frames should be sent as these will not affect the framebuffer and will be executed while GLINT waits for the VBLANK. This allows better overlap between the host and GLINT. In general any commands that will not cause rendering to the framebuffer to occur can be queued in the GLINT FIFO before waiting on VBLANK.

Eventually more framebuffer rendering commands will be sent by the host, and the GLINT will then stall its hyperpipeline until the buffer swap completes. Ideally the host should use this time to prepare additional DMA buffers, assuming that an interrupt driven DMA driver is being used.

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 we can poll the VTGVLineNumber register (there is also a VBLANK interrupt available). This register is reset to 1 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 1 indicates the start of VBLANK. Since this register always has a value ≥ 1, it is better to wait for its value to be less than some small positive integer such as 3 or 4. The vertical blanking period typically lasts for 10 – 30 scanlines so this improves our hit rate but still leaves plenty of blanking time for us to complete any work we have to do.

4.3.3 Bitplane Double Buffering

Bitplane double buffering is of use at 32bpp framebuffer depth using 32768 5:5:5:1 true color mode. It relies on the board being designed with a RAMDAC which will select

between the high and low 16bits of its input stream based on whether bit31 is set or clear. Effectively the front and back buffer for each pixel, become interleaved within the same 32bit word in the framebuffer, i.e. buffer 0 becomes the lower 16bits and buffer 1 becomes the upper 16bits.

The buffer swap is thus implemented as a block fill of bit31 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 VTGFrameRowAddr 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.) bit31 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) nibbles 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.

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 punted back to the Microsoft implemented DIB engine, rather than being implemented in the display driver.

4.3.4 Color Space Double Buffering

Colorspace double buffering is primarily of use at 32bpp framebuffer depth using 4096 colors, though in principal the technique can be used at other depths.

It relies on the board being designed with a LUT-DAC which can toggle between displaying the high nibbles of each of the R, G and B 8bit streams, to displaying the low nibbles. This effectively interleaves the pixels of a 12bit true color RGB front buffer with a 12bit true color RGB back buffer at the same 32bit memory location in VRAM.

The implementation of the toggling will depend on the particular RAMDAC. Some have a readmask on the input which can be updated with a single memory access. Others can accept writes sufficiently quickly that their complete LUT can be reloaded by the host during the VBLANK.

GLINT can be set into a mode where it replicates the pixel color information into the high and low nibbles. This is useful when it is rendering GUI data (such as window borders, titles etc.). The hardware writemask can then be used to write to only the high (or only the low) nibbles when rendering the animating contents of the window.

The features are:

- "instantaneous" buffer swap

- no offscreen buffer required (e.g. 1152x900 would be the maximum resolution on a 4MB framebuffer at 32bpp depth)
- ONE window can be double buffered. GUI can double write with no performance penalty.
- in practice most useful at 12bpp RGB

For further details see section § 0 and §0 of this manual, and refer to the IBM RGB525 data sheet

5. Graphics Programming

GLINT provides a rich variety of operations for 2D and 3D graphics supported by its HyperPipelined architecture. Section § 0 shows the basic units in the HyperPipeline, section §0 shows how to use GLINT to render a simple graphic primitive, the Gouraud shaded triangle, and sections § 0 to §0 describe each of the units in detail.

5.1 The Graphics HyperPipeline

This section describes each of the units in the graphics HyperPipeline. Figure 0.1 shows a schematic of the pipeline. In this diagram, the localbuffer contains the pixel ownership values (known as Graphic IDs), the FrameCount Planes (FCP), Depth (Z) and Stencil buffer. The framebuffer contains the Red, Green, Blue and Alpha bitplanes. The units in the HyperPipeline are:

- **Rasterizer** scan converts the given primitive into a series of fragments for processing by the rest of the pipeline.
- **Scissor Test** clips out fragments that lie outside the bounds of a user defined scissor rectangle and also performs screen clipping to stop illegal accesses outside the screen memory.
- **Stipple Test** masks out certain fragments according to a specified pattern. Line and area stipples are available.
- **Color DDA** is responsible for generating the color information (True Color RGBA or Color Index(CI)) associated with a fragment.
- **Texture** is concerned with mapping a portion of a specified image (texture) onto a fragment. The process involves interpolating to determine the texel coordinates including perspective division, reading the texels, filtering to calculate the texture color, and application which applies the texture color to the fragment color.
- **Fog** blends a fog color with a fragment's color according to a given fog factor. Fogging is used for depth cueing images and to simulate atmospheric fogging.
- **Antialias Application** combines the incoming fragment's alpha value with its coverage value when antialiasing is enabled.
- **Alpha Test** conditionally discards a fragment based on the outcome of a comparison between the fragments alpha value and a reference alpha value.

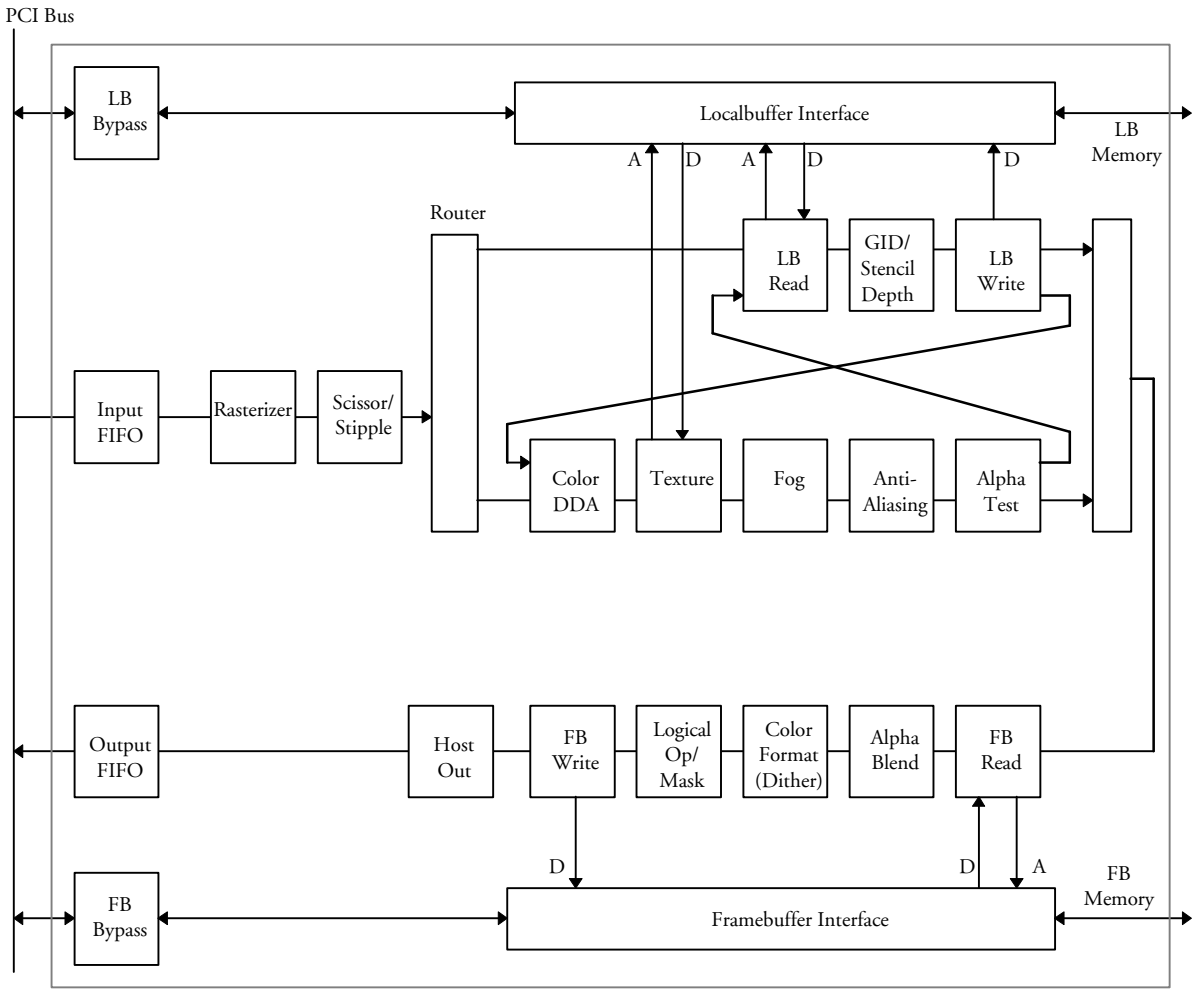


Figure 0.1 HyperPipeline

- **GID (Pixel Ownership)** is concerned with ensuring that the location in the framebuffer for the current fragment is owned by the current visual. Comparison occurs between the given fragment and the Graphic ID value in the localbuffer, at the corresponding location, to determine whether the fragment should be discarded.
- **Stencil Test** conditionally discards a fragment based on the outcome of a test between the given fragment and the value in the stencil buffer at the corresponding location. The stencil buffer is updated dependent on the result of the stencil test and the depth test.
- **Depth Test** conditionally discards a fragment based on the outcome of a test between the depth value for the given fragment and the value in the depth buffer at the corresponding location. The result of the depth test can be used to control the updating of

the stencil buffer.

- **Alpha Blending** combines the incoming fragment's color with the color in the framebuffer at the corresponding location.
- **Color Formatting** converts the fragment's color into the format in which the color information is stored in the framebuffer. This may optionally involve dithering.
- **Logical Op/Framebuffer Mask** performs Logical Operations between the fragment and destination, and optionally applies a writemask.
- **Host Out** optionally gathers statistics for picking and extent checking, and returns data to the host for image uploads.

The HyperPipeline structure of GLINT is very efficient at processing fragments, for example, texture mapping calculations are not actually performed on fragments that get clipped out by scissor testing. This approach saves substantial computational effort. The pipelined nature does however mean that when programming GLINT you should be aware of what all the pipeline stages are doing at any time, for example, many operations require both a read and/or write to the localbuffer and framebuffer, it is not sufficient to set a logical operation to XOR and enable logical operations, you must also enable the reading/writing of data from/to the framebuffer.

5.1.1 The Router

One important performance feature of the hyperpipeline is the Router. This is essentially a switch which allows the order of some of the units to be swapped, by setting or clearing the Order bit of the **RouterMode** register.

Textured primitives are typically more costly than non-textured primitives. When the Order bit is set fragments are tested against the GID (Pixel Ownership), Stencil and Depth(Z), prior to the texture value being calculated. If the fragment fails any of these tests, then nothing will be drawn, and so there is normally no need to calculate the texture value, leading to higher performance.

OpenGL defines the order of operations on a fragment to be texture, alpha test, stencil and then depth(Z), which is the sequence used when the Order bit in the **Router** register is cleared. However, if the alpha test is disabled (or cannot reject fragments) then OpenGL compatible semantices are maintained even if the operation order is changed to the more optimal stencil, depth(Z), texture, alpha test.

The order can be dynamically reconfigured at any time without any need to synchronize simply by writing to the Order bit.

5.2

A Gouraud Shaded Triangle

In this section we show how to render a typical 3D graphics primitive, the Gouraud shaded, depth buffered triangle using GLINT. For this example assume that the triangle is to be drawn into a window which has its colormap set for RGB as opposed to color index operation. This means that all three color components; red, green and blue, must be handled. Also, assume the coordinate origin is bottom left of the window and drawing will be from top to bottom. GLINT can draw from top to bottom or bottom to top.

For clarity the equations below are shown in full, though in practice there are many common terms and factors which need only be computed once. A full C code example is given in Appendix F.

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. The alpha component will be omitted for this example.

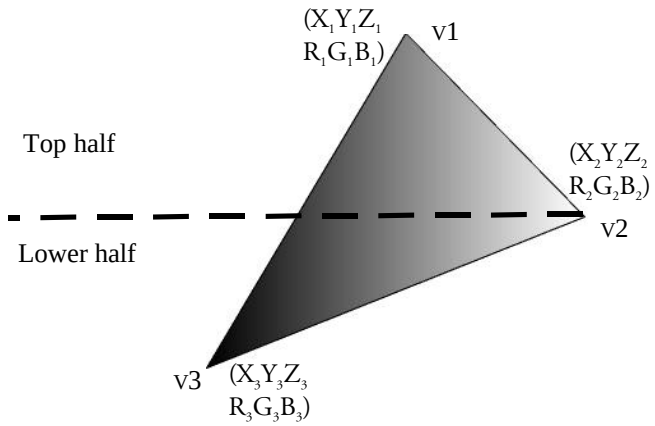


Figure 0.2 Example Triangle

The diagram makes a distinction between top and bottom halves, this is because GLINT is designed to rasterize screen aligned trapezoids and flat topped or bottomed triangles as shown below:

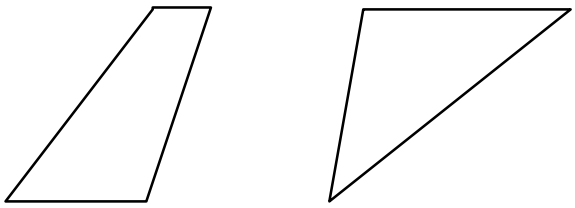


Figure 0.3 Screen aligned trapezoid and flat topped triangle

5.2.1 Initialization

GLINT 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 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.

5.2.2 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.

GLINT always draws triangles from the dominant edge towards the subordinate edges. This simplifies the calculation of set up parameters as will be seen below.

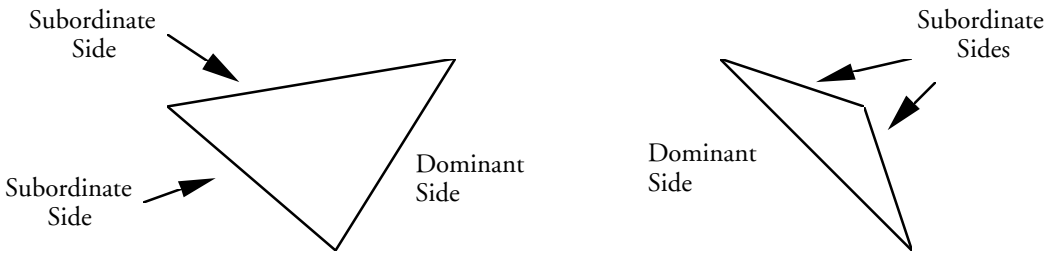


Figure 0.4 Dominant and Subordinate Sides of a Triangle

5.2.3 Calculating Color values for Interpolation

To draw from left to right, 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:

$$dRdx = \{(R_1 - R_3) \times \frac{(Y_2 - Y_3)}{c}\} - \{(R_2 - R_3) \times \frac{(Y_1 - Y_3)}{c}\}$$

$$dGdx = \{(G_1 - G_3) \times \frac{(Y_2 - Y_3)}{c}\} - \{(G_2 - G_3) \times \frac{(Y_1 - Y_3)}{c}\}$$

$$dBdx = \{(B_1 - B_3) \times \frac{(Y_2 - Y_3)}{c}\} - \{(B_2 - B_3) \times \frac{(Y_1 - Y_3)}{c}\}$$

where, to be independent of the order the vertices are provided:

$$c = abs\{(X_1 - X_3) \times (Y_2 - Y_3) - (X_2 - X_3) \times (Y_1 - Y_3)\}$$

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 GLINT always draws triangles starting from the dominant edge and towards the subordinate edges. For the example triangle, this means left to right.

5.2.4 Register Set Up for Color Interpolation

For the example triangle, the GLINT registers must be set as follows, for color interpolation. Note that the format for color values is 24bit, fixed point 2's complement.

```
// 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)
```

5.2.5 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:

$$dZdx = \{(Z_1 - Z_3) \times \frac{(Y_2 - Y_3)}{c}\} - \{(Z_2 - Z_3) \times \frac{(Y_1 - Y_3)}{c}\}$$

where, as before:

$$c = abs\{(X_1 - X_3) \times (Y_2 - Y_3) - (X_2 - X_3) \times (Y_1 - Y_3)\}$$

The divisor, shown here as c, 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.

5.2.6 Register Set Up for Depth Testing

Internally GLINT uses fixed point arithmetic. The formats for each register are described later. Each depth value must be converted into a 2's complement 16.32 bit fixed point number and then loaded into the appropriate pair of 32 bit registers. The 'Upper' or 'U' registers store the integer portion, whilst the 'Lower' or 'L' registers store the 16 fractional bits, left justified and zero filled.

For the example triangle, GLINT 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)
```

5.2.7 Calculating the Slopes for each Side

GLINT 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 GLINT enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat topped triangle (GLINT 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.

5.2.8 Rasterizer Mode

The GLINT rasterizer has a number of modes which can be set which have effect 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 value for these modes are suitable.

```
RasterizerMode(0)   // Default rasterizer mode
```

5.2.9 Subpixel Correction

GLINT 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. Subpixel correction will generally always be enabled when rendering any trapezoid which is smooth shaded, textured, fogged or depth buffered. Control of subpixel correction is in the **Render** command register described in the next section, and is selectable on a per primitive basis. A full code example is given in Appendix F.

5.2.10 Rasterization

GLINT 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 = GLINT_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE

// Draw the top half of the triangle

Render(render)
```

After the **Render** command has been issued, the registers in GLINT 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 GLINT 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
```

5.3 Rasterizer Unit

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

GLINT can directly rasterize:

- aliased screen aligned trapezoids
- aliased single pixel wide lines
- aliased single pixel points
- antialiased screen aligned trapezoids
- antialiased circular points

All other primitives are treated as one or more of the above, for example an antialiased line is drawn as a series of antialiased trapezoids.

5.3.1 Trapezoids

GLINT'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 Figure 0.5 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. GLINT has adopted the convention of omitting the pixels down the right hand edge. Control of 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.

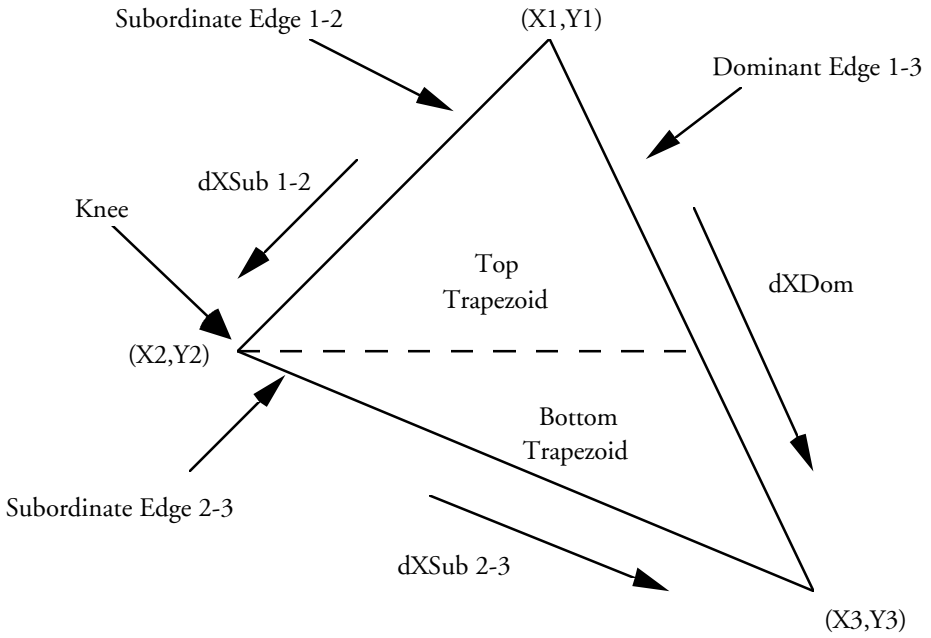


Figure 0.5 Rasterizing a triangle.

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
// §0

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.

render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SubpixelCorrectionEnable = GLINT_TRUE
render.AntialiasEnable = GLINT_DISABLE

// 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 GLINT 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 GLINT 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.

5.3.2 Lines

Single pixel wide aliased lines are drawn using a DDA algorithm, so all GLINT 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)

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.

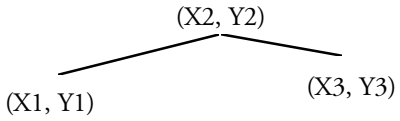


Figure 0.6 Polyline

```
// 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 = GLINT_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.

When a **Continue** command is issued some error will be propagated along the line, to minimize this, a choice of actions are available as to how the DDA units are restarted on the receipt of a **Continue** command. It is recommended that for OpenGL rendering the **ContinueNewLine** command is not used and individual segments are rendered.

Antialiased lines, of any width, are rendered as antialiased screen-aligned trapezoids.

5.3.3 Points

GLINT 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 GLINT_POINT_PRIMITIVE. The pseudocode portion to render an aliased unity sized point is:

```
// Set the rasterizer mode to the default, see
// §0

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 = GLINT_POINT_PRIMITIVE

// Render the point

Render (render)
```

5.3.4 Antialiasing

GLINT uses a subpixel point sampling algorithm to antialias primitives. GLINT can directly rasterize antialiased trapezoids and points. Other primitives are composed from these base primitives.

The rasterizer associates a coverage value with each fragment produced when antialiasing. This value represents the percentage coverage of the pixel by the fragment. GLINT supports two levels of antialiasing quality:

- normal, which represents 4x4 pixel subsampling
- high, which represents 8x8 pixel subsampling

Selection between these two is made by the AntialiasingQuality bit within the **Render** command register.

When rendering antialiased primitives with GLINT the **FlushSpan** command is used to terminate rendering of a primitive. This is due to the nature of GLINT antialiasing. When a primitive is rendered which does not happen to complete on a scanline boundary, GLINT retains antialiasing information about the last sub-scanline(s) it has processed, but does not generate fragments for them unless a **FlushSpan** command is received. The commands **ContinueNewSub**, **ContinueNewDom** or **Continue** can then be used, as appropriate, to maintain

continuity between adjacent trapezoids. This allows complex antialiased primitives to be built up from simple trapezoids or points.

To illustrate this consider using screen aligned trapezoids to render an antialiased line. The line will in general consist of three screen aligned trapezoids as shown in the diagram below.

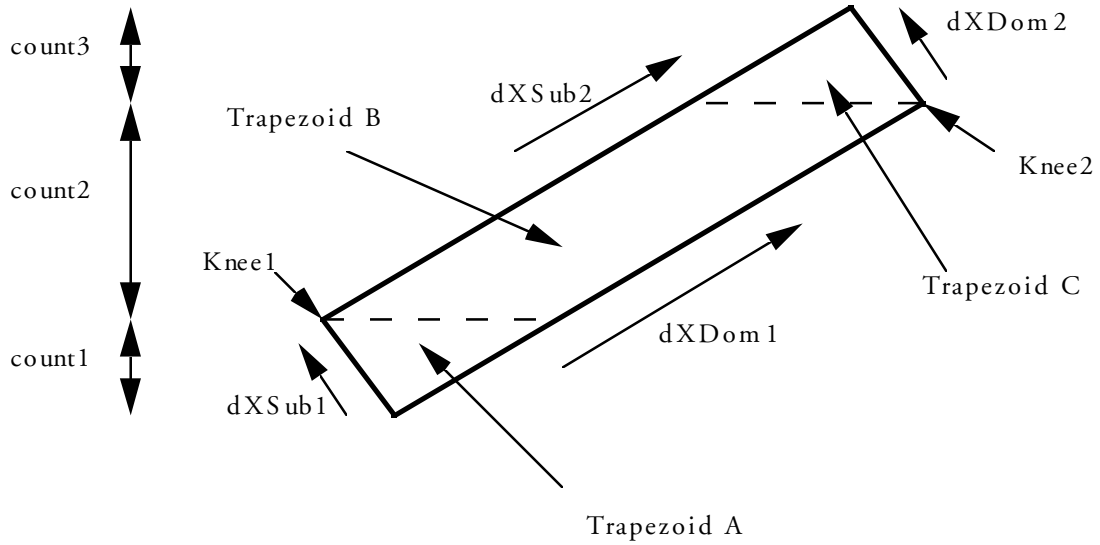


Figure 0.7 Antialiased Line

The procedure to render the line is as follows:

```
// Set-up the blend and coverage application units
// as appropriate - not shown
// In this example only the edge deltas are shown
// loaded into registers for clarity.  In reality
// start X and Y values are required. This example
uses
// 4x4 antialiasing.

// Render Trapezoid A

dY(1<<14)
dXDom(dXDom1<<14)
dXSub(dXSub1<<14)
Count(count1<<2)
render.PrimitiveType = GLINT_TRAPEZOID
render.AntialiasEnable = GLINT_TRUE
render.AntialiasQuality = GLINT_MIN_ANTIALIAS
render.CoverageEnable = GLINT_TRUE
Render(render)

// Render Trapezoid B
```

```
dXSub(dXSub2<<14)
ContinueNewSub(count2<<2)

// Render Trapezoid C

dXDom(dXDom2<<14)
ContinueNewDom(count3<<2)

// Now we have finished the primitive flush out
// the last scanline
FlushSpan()
```

Note that when rendering antialiased primitives, any count values should be given in subscanlines. For example if the quality is 4x4 then the count will be 4 times the number of scanlines completely covered by the primitive plus the number of subscanlines contained in the remaining partially covered scanlines. Also, if using 4x4 quality then any delta value must be divided by 4. If using 8x8 quality then the multiply/divide factor is 8.

When rendering, AntialiasEnable must be set in the **AntialiasMode** register to scale the fragments color by the coverage value. An appropriate blending function should also be enabled. See the Antialias Application and Alpha Blend sections for more details.

Note, when rendering antialiased bow-ties, the coverage value on the cross-over scanline may be incorrect.

Section §0 describes in more detail how to render scenes with antialiased polygons.

GLINT can render small antialiased points. Antialiased points are treated as circles, with the coverage of the boundary fragments ranging from 0% to 100%. GLINT supports:

- point diameter of 0.5 to 16.0 in steps of 0.25 for 4x4 antialiasing
- point diameter of 0.25 to 8.0 in steps of 0.125 for 8x8 antialiasing

To scan convert an antialiased point as a circle, GLINT traverses the boundary in sub scanline steps to calculate the coverage value. For this, the sub scanline intersections are calculated incrementally using a small table. The table holds the change in X for a step in Y. Symmetry is used so the table only holds the delta values for one quadrant.

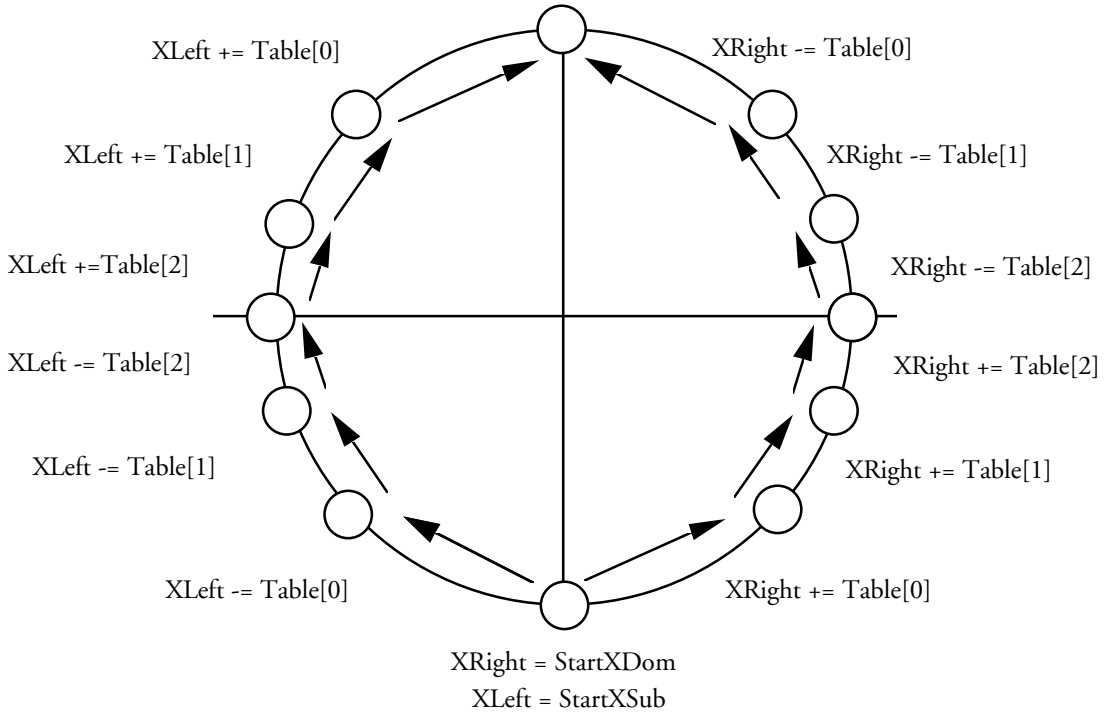


Figure 0.8 Antialiased Point

The pattern of table accesses, additions and subtractions are shown in Figure 0.8 for an odd diameter point. On the diagram the symbol +/-= Table[n] by an arrow indicates the contents of the table at address n are added/subtracted to move along the arrow.

StartXDom, **StartXSub** and **StartY** are set to the top or bottom of the circle and **dY** set to the subscanline step. In this example the point table will have three entries. Note in the case of an even diameter, the last of the required entries in the table is set to zero. Appendix A Register Reference, gives full details of how the point table is laid out.

Note, as the table is configurable, point shapes other than circles can be rendered. Also if the **StartXDom** and **StartXSub** values are not coincident then horizontal thick lines with rounded ends, can be rendered.

5.3.5 Span Operations

The GLINT 500TX has greatly increased the speed of many 2D operations through the addition of 3Dlabs proprietary Span Filling technology.

The span mechanism may be used for various operations such as image upload, image download, filling with constant color, filling with a pattern, character glyphs, monochrome bitmaps, copies and copies with logical ops. Any trapezoid may be used and the scanning direction may be left-to-right or right-to-left.

2D performance is greatly improved by:

- Better utilization of the VRAM block fill capability for solid fills,

stippled fills, characters and pattern fills.

- The span mechanism is independent of pixel size. Hence maximum use is made of the framebuffer bandwidth for 8, 16 and 32 bit wide pixels.
- Multiple pixels are processed in parallel.
- No alignment restrictions. Any span operation may be performed to any pixel alignment for all pixel sizes.
- Page break overheads are amortized over many more read and write operations during a BitBlt operation. Hence performance of BitBlt operations is much closer to the peak bandwidth of the memory.
- Window or screen relative operations are supported.
- Scissor clipping can also be used in conjunction with span operations.

The span mechanism does have some restrictions:

- No accesses to the localbuffer are made. Hence GID, Stencil and Depth tests are not available.
- 3D operations including gouraud shading, alpha tests, alpha blend, dither operations, fogging and anti-aliasing are not available.

When the span operation is enabled, the rasterizer divides the pixels between the left and right hand edges of the polygon or rectangle into a succession of spans, each 32 pixels wide. Each span is described by a 32 bit wide span mask and each pixel in the span has a corresponding bit in the span mask. If a bit in the span mask is set, then the corresponding pixel will be read and/or written. The least significant bit in the span mask (bit 0) corresponds to the left most pixel on the screen for the span. The span mask does not have any fixed alignment with the pixels stored in the framebuffer, i.e. the first pixel in the span may correspond to any pixel in the framebuffer. Any masking or shifting to align the span data being read or written to the 64 bit framebuffer architecture is performed automatically.

Span filling may be performed left-to-right or right-to-left. However the pixels within an individual span are always read and/or written in a left to right order. Hence if a bitmask or image download data is provided, then the data in each individual span must be ordered left to right. Normally if any data is provided, then span filling should be performed left-to-right.

5.3.6

Span Mask Processing

The span mask undergoes several processing steps before it is used by the Framebuffer Interface Unit to determine which pixel to read and/or write:

- The Rasterizer generates the mask using the left and right hand edge information. Note that the edges may be vertical or sloped.
- If SyncOnBitMask is enabled in the Render command, then the span mask is ANDed with the bit mask data provided by the host. If no bit mask data is present, then the Rasterizer will wait for it to arrive before proceeding. The bit mask data may be optionally inverted, byte swapped, word swapped or mirrored (in any combination) before the ANDing is performed. The inversion may be used to enable drawing of the background bits. The byte and word swapping allows bit mask data from different endian hosts to be accommodated. The mirror operation swaps bits 0 and 31, bits 1 and 30, etc. which changes the left most pixel in a span from being controlled by the least significant bit to the most significant bit in the bit mask.
- If the Screen Scissor is enabled, then pixels falling outside the left and right edges of the screen scissor region have their corresponding bits in the span mask cleared.
- If the User Scissor is enabled, then pixels falling outside the left and right edges of the user scissor region have their corresponding bits in the span mask cleared.
- If Area Stippling is enabled, then the stipple mask is extracted from the area stipple table for the appropriate scan line and expanded, if necessary, to 32 bits by replication. The normal offset, select and mirror controls in X and in Y may be used as for non span rendering. The stipple mask is ANDed with the span mask.
- If texture mapping is enabled, then a texel is read from the localbuffer (under control of the TextureAddressMode, TextureReadMode and the S, T and Q DDA parameters). If the texel is to be used as a bit mask, then any specified texel formatting is performed and the final 32 bit texel value is optionally inverted, byte swapped and mirrored before being ANDed with the span mask. This procedure allows character bit masks to be held in the localbuffer.
- The span mask is now used (in conjunction with some mode bits) to read and/or write pixel data in the framebufffer.
- Finally the span mask may be optionally used to grow the extent region, or perform picking as part of the statistics operation in the Host Out Unit.

5.3.7 Block Write Operation

The span operation of the GLINT 500TX includes the block write functionality of the GLINT 300SX. The same algorithms that used block write to fill trapezoids on the GLINT 300SX will also work on the GLINT 500TX. However, the block write size is now fixed at 32 pixels for all depths. This means that it is no longer necessary to specify the block write size in the FastFillIncrement field of the **Render** command, nor in the BlockWidth field of the **FBWriteMode** register. In the GLINT 500TX these bits are now unused and their values are ignored.

5.3.8 Pixel Sizes

The GLINT 300SX core operated independently of the pixel depth. With the introduction of span operations, and in order to maximize the number of pixels per 32 bits processed, the GLINT 500TX core now takes account of the depth of the pixels. The Rasterizer unit includes a new register called the **PixelSize** register. This register replaces the relevant bits in the PCI *FBModeSel* register. The bits in *FBModeSel* are now read-only. To change pixel depth on the GLINT 500TX, the core register must be used instead.

The **PixelSize** register can have the following values:

0 = 32 bit pixels
1 = 16 bit pixels
2 = 8 bit pixels

Since the **PixelSize** register is a core register, it can be modified at any time without affecting in-progress rendering. Thus, unlike the GLINT 300SX it is not necessary to synchronize with the chip before changing pixel depth.

5.3.9 Sub Pixel Precision and Correction

As the rasterizer has 16 bits of fraction precision, and the screen width used is typically less than 2^{16} wide a number of bits called subpixel precision bits, are available. Consider a screen width of 4096 pixels. This figure gives a subpixel precision of 4 bits ($4096=2^{12}$). The extra bits are required for a number of reasons:

- antialiasing (where vertex start positions can be supplied to subpixel precision)
- 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

GLINT supports subpixel correction of interpolated values when rendering aliased 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 aliased rendering which uses interpolated parameters.

Subpixel correction is not applied to antialiased primitives.

5.3.10 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 by default, 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 Figure 0.9.

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.

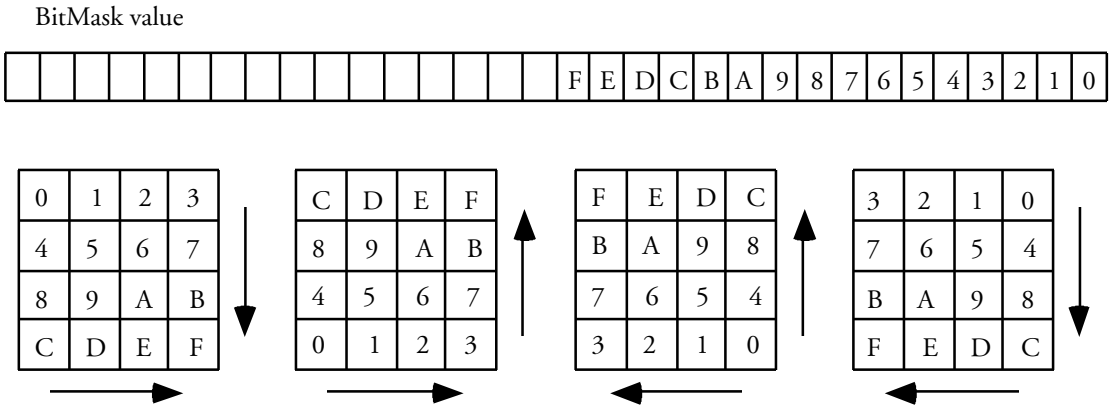


Figure 0.9 Relationship between Bitmask and Scanning Directions

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. The 500TX allows the pattern to be byte swapped on download. This is useful for downloading Windows bitmaps in their native format. 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 § 0.

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. Any unused bits in the last Bitmap word are discarded.

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
```

```
// §0

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 = GLINT_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = GLINT_TRUE

// 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.

The GLINT 500TX provides the ability to start a scanline at an arbitrary offset into the first bitmask that is downloaded for each scanline, and to discard unused bits at the end of a scanline. This is useful for allowing the host to download data directly from a host bitmap without having to shift and pack the bits. This functionality is controlled by the BitMask Packing and BitMask Offset bits in the **RasterizerMode** register.

5.3.11 Span Operations and Bitmaps

The fastest way to render downloaded bitmap data, not requiring logical op processing, is to use a span operation. The rasterizer is set up as normal setting the FastFillEnable bit.

When the bitmap data is downloaded, it is now ANDed with the span mask generated by the rasterizer. This resulting mask is passed through the core to be used as the VRAM block fill mask. Thus a single VRAM access can be used to process up to 32 pixels. Since, only the foreground color can be set with a block fill. 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 GLINT 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 span 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.

The following gives the outline for this algorithm:

```
leftalign  = bitmapxleft & 31
width = Xright - Xleft + leftalign

StartXDom ((Xleft - leftalign)<<16)
dXDom (0)
StartXSub (Xright<<16)

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

// protect the edge pixels with the scissor
```

```
minXY.X = Xleft
minXY.Y = Y
maxXY.X = Xright
maxXY.Y = Y + height
ScissorMinXY(minXY)      // Load the registers
ScissorMaxXY(maxXY)

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

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

render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = GLINT_TRUE
render.FastFillEnable = GLINT_TRUE

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

Render (render)

// download the bits from the source bitmap 32 bits
at
// a time aligning the bitmap pointer at the start
of
// each scanline

BitmapBase += bitmapyorg * bitmapwidth
bitmapxleft &= ~31
for (h = 0; h < height; ++h) {
     pulBitmap = BitmapBase + bitmapxleft/8;
     for (c = 0; c < width; c += 32) {
          BitMaskPattern(pulBitmap)
          pulBitmap += sizeof(ULONG)
     }
     BitmapBase += bitmapwidth
}
```

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.

5.3.12 Image Copy/Upload/Download

GLINT supports three "pixel rectangle" operations: copy, upload and download. These can apply to the Depth or Stencil Buffers (held within the localbuffer) or the framebuffer as will be seen in section § 0.

GLINT copy moves raw blocks of data around buffers. To zoom or re-format data external software must upload the data, process it and then download it again.

To copy a rectangular area, the rasterizer would be configured to render the destination rectangle, thus generating fragments for the area to be copied. GLINT 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.

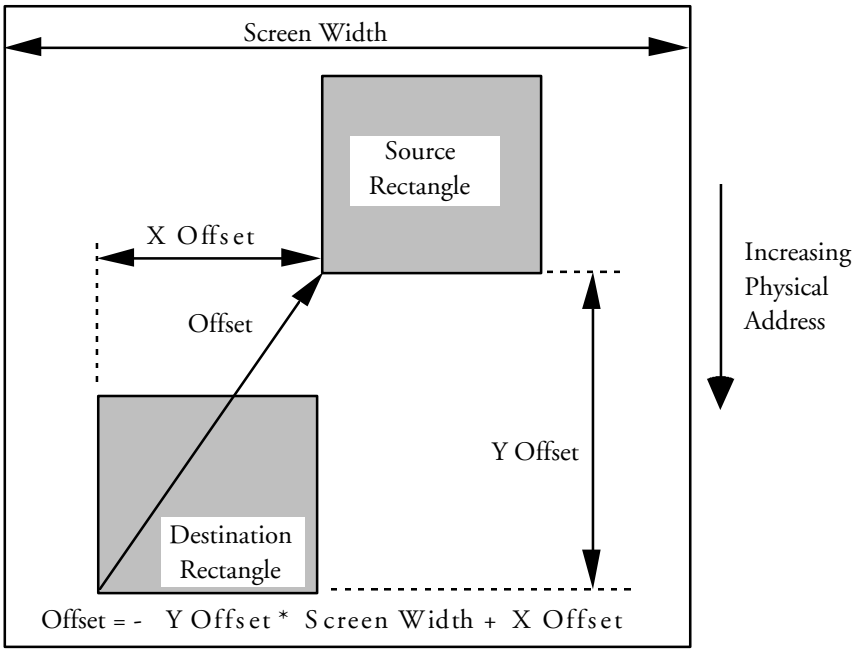


Figure 0.10 GLINT Copy Operation

Localbuffer copy operations are correctly tested for pixel ownership. Note that this implies two reads of the localbuffer, one to collect the source data, and one to get the destination GID for the pixel ownership test.

GLINT 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 exception is that an image load may use pixel ownership tests, in which case the localbuffer destination read must be enabled.

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 GLINT pipeline will stall. In addition it is strongly recommended that any units which can discard fragments (for instance the following tests: bitmask, alpha, user scissor, screen scissor, stipple, pixel ownership, 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 GLINT color formats, section § 0 fully describes this operation.

An example of a rectangular copy may be found in section § 0.

5.3.13 Span Operations and Image Copy/Upload/Download

2D image operations to and from the framebuffer can be optimized by using a span operation. The benefits are greatest at lower pixel depths since packed pixel data is transferred through the core.

To use span operations when copying pixel data within the framebuffer is straightforward. The FastFillEnable and SpanOperation bits in the Render command must be set as follows. This will work both with and without logical op processing.

```
render.FastFillEnable = GLINT_TRUE
render.SpanOperation  = 1
```

For image download operations, the GLINT 500TX supports multiple pixel download using span operations. This is not supported where logical op processing is required. For a straightforward packed pixel image download, the algorithm is very similar to that for monochrome bitmap download using spans.

The source data should be downloaded in 32 bit quantities, starting on a 32 bit boundary. The host should download as many 32 bit quantities per scanline as are necessary to include all pixels to be downloaded. The rasterizer should be set to rasterize the appropriate rectangle but adjusting the left edge backwards to allow for extra pixels required in aligning the source. Finally, the scissor unit should be enabled to clip out the extra left hand pixels now being rasterized.

For image upload, a similar algorithm applies. In this case the image data can be delivered to the output FIFO as packed 32 bit data. For pixels at the start and end of each scanline the GLINT 500TX will zero out any pixels which are outside the rectangle being rasterized. In this case the scissor unit is not required, but using it will mean that unwanted left-hand pixels are also returned as zero.

5.3.14

Rasterizer Mode

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

- ByteSwapBitMask: This is a two-bit flag which specifies that any bits which are downloaded as part of a SyncOnBitMask operation will be byte re-ordered before being used. There are four re-ordering possibilities. Assuming that the bytes are downloaded in the order ABCD then we get the following re-ordering depending on the value of this two bit field:
  0: ABCD (no swap)
  1: BADC (swap within halfwords)
  2: CDAB (halfword swap)
  3: DCBA (full byte swap)
  Using a value of 3 is most useful when used in conjunction with the MirrorBitMask bit for handling Microsoft Windows bitmaps since this causes a complete byte swap of the downloaded data.
- MirrorBitMask: 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).
- InvertBitMask: 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.
- BitMaskPacking: This is a single bit which controls the packing of bits which are downloaded as part of a SyncOnBitMask operation. If this bit is reset then any spare bits at the end of a scanline are used to start the next scanline. If this bit is set then extra bits at the end of a scanline are discarded. This is not available for use with span fills.
- BitMaskOffset: This is a 5 bit field which specifies the first bit to be used in the first bitmask word of every scanline downloaded as part of a SyncOnBitMask operation. This is not available for use with span fills.
- Fraction Adjust: These 2 bits control the action taken by the rasterizer on receiving a **ContinueNewLine** command. As GLINT uses a DDA algorithm to render lines, an error accumulates in the DDA value. GLINT provides for greater control of the error by:
  a) leaving the DDA running, which means errors will be propagated along a line.
  OR
  b) setting the fraction bits to either zero, a half or almost a half

(0x7FFF).

- Bias Coordinates is a 2-bit field with the following actions:-
  0 - Add 0 to the coordinates (Effectively do nothing)
  1 - Add exactly one half to the coordinates
  2 - Add nearly one half (0x7FFF) to the coordinates
- Host Data Byte Swapping: The data downloaded by the host when using SyncOnHostData can have its bytes re-ordered. If the downloaded data has a byte ordering of ABCD then, this 2 bit field specifies re-ordering as follows:
  0: ABCD (no swap)
  1: BADC (swap within halfwords)
  2: CDAB (halfword swap)
  3: DCBA (full byte swap)
- Y Limits Clipping: When set, this bit enables Y Limits clipping. When reset Y Limits clipping is disabled. This is described in the next section.
- Multi GLINT: If set this bit causes the rasterizer to work in multi-GLINT mode. If reset the rasterizer works in single GLINT mode.

5.3.15 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, one of the scissor units, 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 is effectively a Y scissor clip in the Rasterizer.

If Y limits testing has been enabled in the **RaserizerMode** 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.

Y Limits clipping is automatically disabled when SyncOnHostData or SyncOnBitMask is used.

5.3.16 Rasterizer Unit Registers

Real coordinates with fractional parts are provided to the rasterizer in 2's complement 16 bit integer, 16 bit fraction format, as illustrated below:

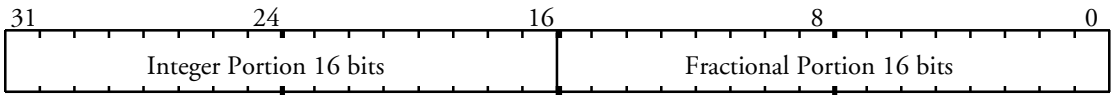


Figure 0.11 Real Coordinate Representation

Table 0.1 Command Register Descriptions lists the command registers which control the rasterizer unit. The control registers are shown separately in Table 0.2

Rasterizer Registers.

Register Name	Data Field	Description
Render	See below	Starts the rasterization process
ContinueNewDom	16 bit integer	Allows the rasterization to continue with a new dominant edge. The dominant edge DDA in the rasterizer 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. Note however, that other DDAs are not reloaded with new start values until the next Render command. Thus it is not possible to use this command, for example, to Gouraud shade a triangle from left to right which has a knee on the left hand side. To avoid this, 3D rendering should always start from the side without the knee. The data field holds the number of scanlines (or sub scanlines) to fill. This count is not loaded into the **Count** register.
ContinueNewSub	16 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 (or sub scanlines) to fill. This count is not loaded into the **Count** register.
Continue	16 bit integer	Allows the 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 (or sub scanlines) to fill. This count is not loaded into the **Count** register.

ContinueNewLine	16 bit integer	Allows rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, but 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 or subpixels in a line. This count is not loaded into the **Count** register. The use of **ContinueNewLine** is not recommended in OpenGL as for the second and subsequent segments the DDA units will start with a slight error compared with the value they would have been loaded with.
FlushSpan	Not used	Used when antialiasing to force the last span out when not all sub spans may be defined.
PixelSize	0 = 32 bits 1 = 16 bits 2 = 8 bits	Configures the Rasterizer (and other core units) with the size of pixel to process when spans are used. It also informs the framebuffer interface Unit, but in this case all reads and writes are affected and not just spans. This replaces the pixel size field in the PCI *FBModeSel* register and works the same way for single pixel reads and writes (i.e. the framebuffer can be set to 32 bit pixels even though it is displaying 8 bit pixels to process 4 pixels at a time).
WaitForCompletion	Not used	This is used to suspend the GLINT 500TX core until all outstanding reads and writes in both the localbuffer and 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 message and waiting for it in the Host Out FIFO; however, this method doesn't involve the host and can be inserted into a DMA buffer.

Table 0.1 Command Register Descriptions

RasterizerMode	See below	Defines the long term mode of operation of the rasterizer.
StartXDom	Fixed point 16.16 format	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing.
dXDom	Fixed point 16.16 format	Value added when moving from one scanline (or sub 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	Fixed point 16.16 format	Initial X value for the subordinate edge.
dXSub	Fixed point 16.16 format	Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling.
StartY	Fixed point 16.16 format	Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing.
dY	Fixed point 16.16 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	16 bit integer	Number of pixels in a line. Number of scanlines in a trapezoid. Number of sub scanlines in an antialiased trapezoid. Diameter of a point in sub scanlines.
BitMaskPattern	32 bits defined earlier	Value used to control the BitMask stipple operation (if enabled).
PointTable0 PointTable1 PointTable2 PointTable3	Packed dx point data.	Antialias point data table. There are 4 words in the table and the register tag is decoded to select a word.
ScanLine Ownership	See Multi-GLINT chapter	Defines which scanlines are owned when in multi-GLINT mode.
Ylimits	Ymax: 2's complement 16 bit value in the upper word. Ymin: 2's complement 16 bit value in the lower word.	Defines the Y extents the rasterizer should fill between. A scanline is filled if its Y value satisfies: Ymin ≤ Y < Ymax

Table 0.2 Rasterizer 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 GLINT. These bits are AreaStippleEnable, LineStippleEnable, ResetLineStipple, TextureEnable FogEnable, CoverageEnable 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. Initially the **FogMode, AreaStippleMode** and **LineStippleMode** registers are enabled through the UnitEnable bits. Now bits need only be set or cleared within the **Render** command to achieve the required result, removing the need for the **FogMode, AreaStippleMode** and **LineStippleMode** registers to be loaded for every render operation.

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

Bit No.	Name	Description
0	AreaStippleEnable	This bit, when set, enables area stippling of the fragments produced during rasterization. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no area stippling occurs irrespective of the setting of the area stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no area stippling for this primitive.
1	LineStippleEnable	This bit, when set, enables line stippling of the fragments produced during rasterization in the Stipple Unit. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no line stippling occurs irrespective of the setting of the line stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no line stippling for this primitive.
2	ResetLineStipple	This bit, when set, causes the line stipple counters in the Stipple Unit to be reset to zero, and would typically be used for the first segment in a polyline. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit. When this bit is reset the stipple counters carry on from where they left off (if line stippling is enabled)

3	FastFillEnable	This bit, when set, causes the span fill mechanisms to be used for the rasterization process. The type of span filling is specified in the SpanOperation field. When this bit is reset the normal rasterization process occurs. This bit only has an effect if the PrimitiveType is Trapezoid.
4, 5	Unused	The block fill size is always 32 pixels on a GLINT 500TX.
6, 7	PrimitiveType	This two bit field selects the primitive type to rasterize. The primitives are: 0 = Line 1 = Trapezoid 2 = Point
8	AntialiasEnable	This bit, when set, causes the generation of sub scanline data and the coverage value to be calculated for each fragment. The number of sub pixel samples to use is controlled by the AntialiasingQuality bit. When this bit is reset normal rasterization occurs. This bit only has an effect if the PrimitiveType is Trapezoid.
9	AntialiasingQuality	This bit, when set, sets the sub pixel resolution to be 8x8. When this bit is reset the sub pixel resolution is 4x4.
10	UsePointTable	When this bit and the AntialiasingEnable are set, the dx values used to move from one scanline to the next are derived from the Point Table. This bit only has an effect if the PrimitiveType is Trapezoid.

11	SyncOnBitMask	This bit, when set, causes a number of actions: The least significant bit or most significant bit (depending on the MirrorBitMask bit) in the Bit Mask register is extracted and optionally inverted (controlled by the InvertBitMask bit). If this bit is 0 then the corresponding fragment is culled from being drawn. After every fragment the **BitMaskPattern** register is rotated by one bit. If all the bits in the **BitMaskPattern** register have been used then rasterization is suspended until a new **BitMaskPattern** is received. If any other register is written while the rasterization is suspended then the rasterization is aborted. The register write which caused the abort is then processed as normal **.** Note the behavior is slightly different when the SyncOnHostData bit is set to prevent a deadlock from occurring. In this case the rasterization doesn't suspend when all the bits have been used and if new **BitMaskPattern** data words are not received in a timely manner then the subsequent fragments will just reuse the bitmask.
12	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**, **Stencil, Color or FBSourceData**. If SyncOnBitMask is reset, then if any register other than one of these four is written to, the rasterization is aborted. If SyncOnBitMask is set, then if any register other than one of these four, or **BitMaskPattern**, is written to, the rasterization is aborted. The register write which caused the abort is then processed as normal **.** Writing to the **BitMaskPattern** register doesn't cause any fragments to be generated.
13	TextureEnable	This bit, when set, enables texturing of the fragments produced during rasterization. Note that the Texture Units must be suitably enabled as well for any texturing to occur. When this bit is reset no texturing occurs irrespective of the setting of the Texture Unit controls. This bit is useful to temporarily force no texturing for this primitive.

14	FogEnable	This bit, when set, enables fogging of the fragments produced during rasterization. Note that the Fog Unit must be suitably enabled as well for any fogging to occur. When this bit is reset no fogging occurs irrespective of the setting of the Fog Unit controls. This bit is useful to temporarily force no fogging for this primitive.
15	CoverageEnable	This bit, when set, enables the coverage value produced as part of the antialiasing to weight the alpha value in the alpha test unit. Note that this unit must be suitably enabled as well. When this bit is reset no coverage application occurs irrespective of the setting of the AntialiasMode in the Alpha Test unit.
16	SubPixelCorrection Enable	This bit, when set enables the sub pixel correction of the color, depth, fog and texture values at the start of a scanline. When this bit is reset no correction is done at the start of a scanline. Sub pixel corrections are only applied to aliased trapezoids.
17	Reserved	
18	SpanOperation	This bit, when clear, indicates that writes are to use the constant color found in the **FBBlockColor** register. When this bit is set write data is variable and is either provided by the host (i.e. SyncOnHostData is set) or is read from the framebuffer, or the Pattern RAM.

Table 0.3 Render Command Register Fields

A number of 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 control the action of a 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	Reserved	
7,8	ByteSwapBitMask	This bit controls the byte swapping of the BitMask data before it is used. If the bytes are labeled ABCD on input, then the bytes are swapped as follows:. 0: ABCD 1: BADC 2: CDAB 3: DCBA
9	BitMaskPacking	This bit controls whether the BitMask data is packed or if new BitMask data is required on every scanline. 0: BitMask data is packed 1: BitMask data is provided for each scanline
10 .. 14	BitMaskOffset	These 5 bits hold the position in the 32 bit BitMask data where the first bit is taken from for the BitMask test for the first BitMask data on a new scanline. Subsequent BitMask data starts from bit 0 until the next scanline. Successive bits are taken from increasing bit positions until the bit mask is consumed (i.e. bit 31 is reached). The least significant bit is bit zero.
15,16	HostDataByteSwapMode	These bits control the byte swapping of data associated with the SyncOnHostData operation. If the bytes are labeled ABCD on input, then the bytes are swapped as follows:. 0: ABCD 1: BADC 2: CDAB 3: DCBA
17	MultiGLINT	This bit selects whether the rasterizer is to work in single GLINT mode, or in multi-GLINT mode and consequently only process the scanlines allocated to it. 0: Single GLINT mode 1: Multi-GLINT mode
18	YLimitsEnable	This bit, when set, enables the Y limits testing to be done between the minimum and maximum Y values given by the Ylimits register.

Table 0.4 Rasterizer Mode Register

5.3.17 Examples

Many examples of the use of the rasterizer are found throughout the manual.

5.4 Scissor Unit

Two scissor tests are provided in GLINT, the User Scissor test and the Screen Scissor test. The user scissor checks each fragment or span against a user supplied scissor region; the screen scissor checks that the fragment or span lies within the screen.

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.

5.4.2 Screen Scissor Tests

This test ensures that a fragment 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.

Note that the **WindowOrigin** register only affects the origin for clipping, it does not affect the base address for rendering. Section § 0 Window Initialization gives further details on how to set the base address of a window for rendering.

The following test is made:

$0 \leq (X + WX) < SW$

$0 \leq (Y + WY) < SH$

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.

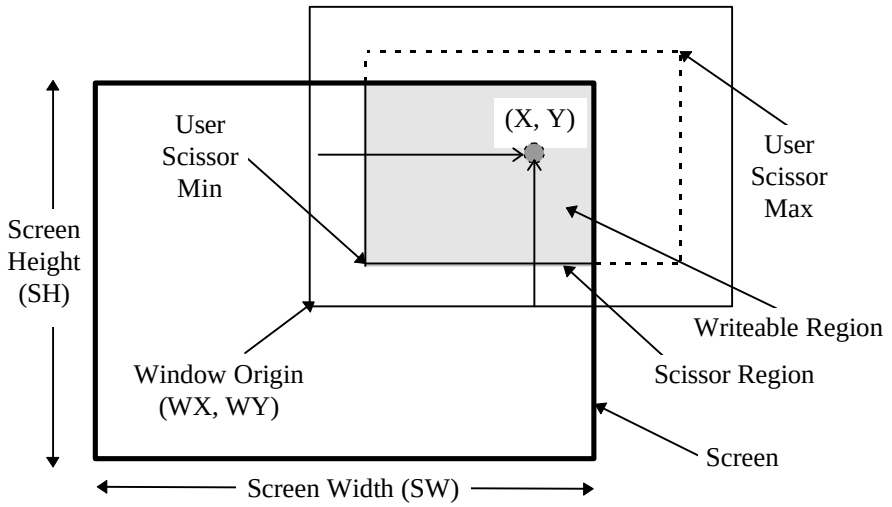


Figure 0.12 Screen Scissor and User Scissor Tests

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 (GID testing can be used to detect this, see section § 0).

5.4.3 Registers

The unit is controlled by the **ScissorMode** register:

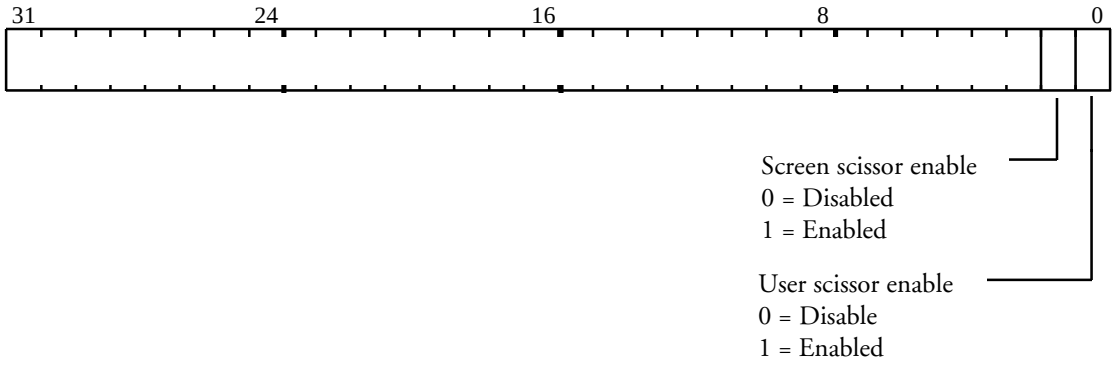


Figure 0.13 Scissor Register

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 rasterization 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.

5.4.4 Span Operations and the Scissor Unit

If a span mask is presented to the scissor unit, then the mask is modified to zero out bits corresponding to pixels which lie outside the scissor region. This is true for both the user scissor and the screen scissor.

An example of how to use this was given above.

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 = GLINT_ENABLE
scissorMode.ScreenScissorEnable = GLINT_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 (0xAAAAAAAA)
AreaStipplePatternl (0x55555555)
```

```
AreaStipplePattern2 (0xAAAAAAAA)
AreaStipplePattern3 (0x55555555)
AreaStipplePattern4 (0xAAAAAAAA)
AreaStipplePatfern5 (0x55555555)
RTeaStipplePattern6 (0xAAAAAAAA)
AreaStipplePattern7 (0x55555555)
AreaStipplePattern31(0x55555555)

// Set-up mode register
areaStippleMode.UnitEnable = GLINT_ENABLE
areaStippleMode.Xselect = 0
areaStippleMode.Yselect = 0
areaStippleMode.Xoffset = 0
areaStippleMode.Yoffset = 0
areaStippleMode.lnvert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0

// Load mode register
AreaStippleMode(areaStippleMode)

// When issuing a Render command, the
// AreaStippleEnable bit should be set to
enabled:
// Arender.AreaStippleEnable = GLINT_TRUE
```

5.5 Stipple Unit

Stippling is a process whereby each fragment is checked against a bit in a defined pattern, the fragment can either be rejected or accepted depending on the result of the stipple test. If it is rejected, then it undergoes no further processing, otherwise it proceeds down the pipeline. GLINT supports two types of stippling, line and area.

5.5.1 Area Stippling

A 32 x 32 bit area stipple pattern can be applied to fragments. The least significant n 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. The number of address bits used, allow regions of 1,2,4,8,16 and 32 pixels to be stippled. The address selection can be controlled independently in the X and Y directions. In addition the bitpattern 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.

Area stippling is enabled using the **AreaStippleMode** register and must be qualified by the AreaStippleEnable bit in the **Render** command register.

5.5.2 Line Stippling

In this test, fragments are conditionally rejected on the outcome of testing a linear stipple mask. If the bit is zero then the test fails, otherwise it passes. The line stipple pattern is 16 bits in length and is scaled by a repeat factor, r, (in the range 1 to 512). The stipple mask bit, b, which controls the acceptance or rejection of a fragment is determined using:

$$b = (\text{floor}\ (s / r)) \bmod 16$$

where s is the stipple counter which is incremented for every fragment (normally along the line). This counter may be reset at the start of a polyline, but between segments it continues as if there were no break.

The stipple pattern can be optionally mirrored, that is the bit pattern is traversed from most significant to least significant bits, rather than the default, from least significant to most significant.

The **UpdateLineStippleCounters** register controls initialization of the line stipple counters, which can be reset or loaded from a previously saved value. The **SaveLineStippleCounters** register is used to save the current line stipple counters. The combination of **UpdateLineStippleCounters** and **SaveLineStippleCounters** is useful to implement stippling of wide polylines.

Line stippling is enabled using the **LineStippleMode** register and must be qualified by the LineStippleEnable bit in the **Render** command register.

5.5.3 Span Operations and Stippling

A span mask generated by the rasterizer will be modified by the area stipple if it is enabled. The line stipple has no effect on the span mask.

If area stippling is enabled, then the current stipple mask is replicated to form a full 32 bit mask. This is ANDed with the span mask. The stipple mask replication happens after all mirroring, inverting and shifting have been performed.

This is very useful for enabling use of the VRAM block write capability for doing monochrome pattern expansions. For example, Microsoft Windows monochrome brushes are normally 8x8. If this brush pattern is stored in the area stipple and solid fill with span operation is enabled, then the foreground color for the pattern will be rendered using block writes. The background color can be filled on a second pass by turning on the InvertStipplePattern bit in the **AreaStippleMode** register.

5.5.4 Registers

The **LineStippleMode** register controls line stipple:

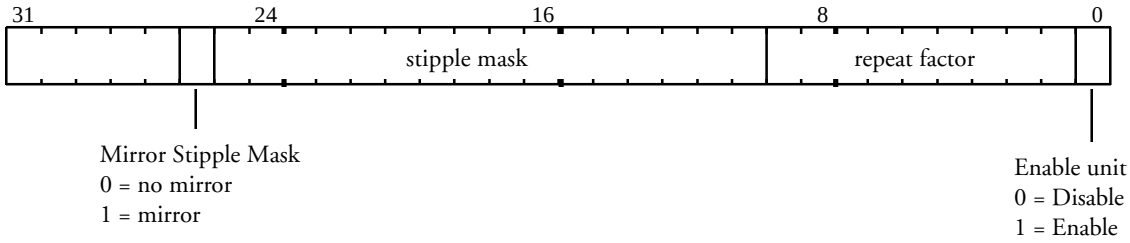


Figure 0.14 LineStippleMode Register

The repeat factor is set to one less than the required value.

The least significant bit of the **UpdateLineStippleCounters** register, controls loading the line stipple counters. If set the line stipple counters are loaded with the previously saved values. If reset, the counters are cleared to zero. The counters can also be reset by means of the ResetLineStipple bit in the **Render** command.

The **AreaStippleMode** register controls area stipple operation:

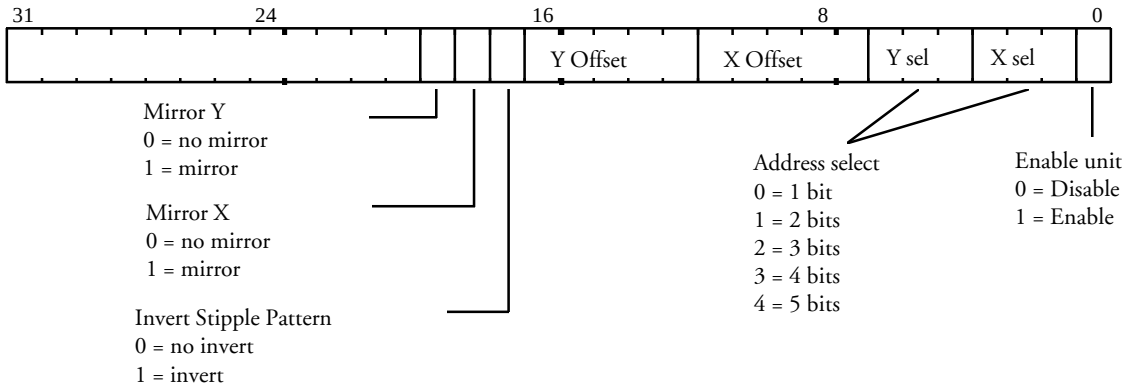


Figure 0.15 AreaStippleMode Register

The EnableUnit bit in the **LineStippleMode** and **AreaStippleMode** registers are qualified by the LineStippleEnable and AreaStippleEnable bits in the **Render** command register.

SaveLineStippleCounters register (which has no data field) saves the line stipple counters internally.

The area stipple is set up in the **AreaStipplePattern*n*** register, where n represents an integer between 0 and 31.

The **LoadLineStippleCounters** register is shown below:

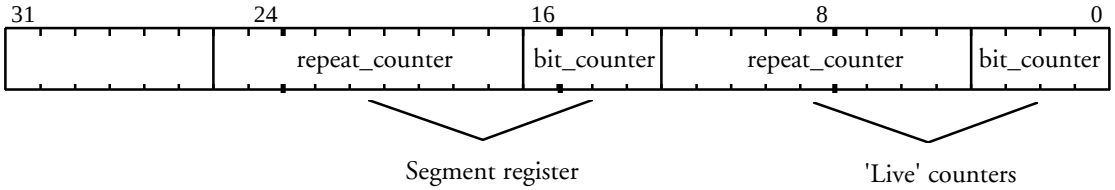


Figure 0.16 LoadLineStippleCounters register

5.5.5 Examples

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

```
// Use only the first two table entries
AreaStipplePattern0(0x1)
AreaStipplePattern1(0x2)

// Set-up mode register
areaStippleMode.UnitEnable = GLINT_ENABLE
areaStippleMode.XSel = 0 // Address index based on
areaStippleMode.YSel = 0 // LSB of address, repeats
                         // every 2nd pixel in X &
Y
areaStippleMode.XOffset = 0
areaStippleMode.YOffset = 0
```

```
areaStippleMode.Invert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0

// Load mode register
AreaStippleMode(areaStippleMode)

// When the Render command is sent the
AreaStippleEnable
// bit should be set in addition to the area
stipple
// test being enabled:
// render.AreaStippleEnable = GLINT_TRUE
```

A line stipple which rejects alternate fragments:

```
// Set counters to zero
UpdateLineStippleCounters(0x0)

// Set the stipple mode
lineStippleMode.UnitEnable = GLINT_ENABLE
lineStippleMode.RepeatFactor = 0  // Repeat factor
1
lineStippleMode.StippleMask = 0xAAAA

LineStippleMode(lineStippleMode)
// When issuing a Render command the
LineStippleEnable // bit should be set in addition
to the line stipple
// test being enabled:
// render.LineStippleEnable = GLINT_TRUE
```

5.6

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).

5.6.1 RGBA and Color-Index(CI) Modes

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

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

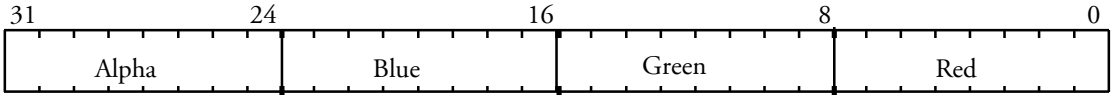


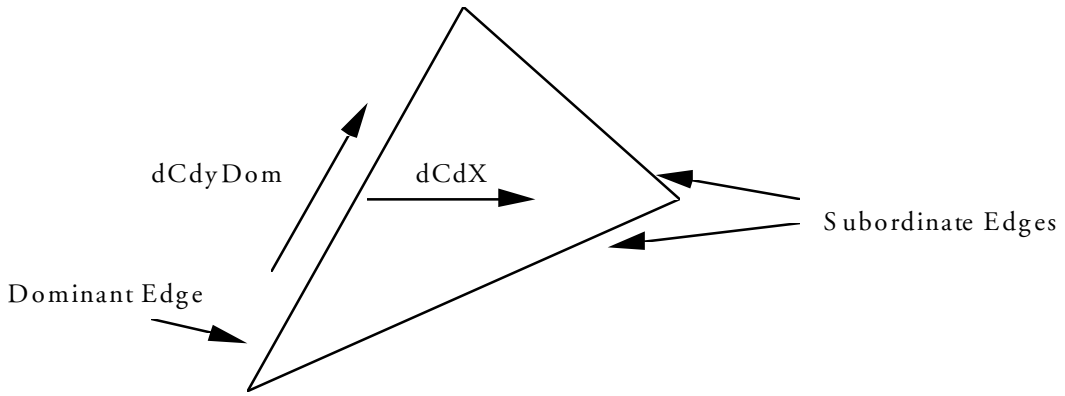
Figure 0.17 GLINT Color Representation

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). If less than 8 bits are used the index is left justified to be in the most significant end of the red component. The unused least significant bits should be set to zero.

5.6.2 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.



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

Figure 0.18 Color Interpolation

For a Gouraud shaded trapezoid, GLINT 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 Figure 0.18, where C represents a color component (red, green, blue, alpha or color index).

See section 5.2 A Gouraud Shaded Triangle for details of how to calculate the required increment values.

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 24bit fixed point format. The format is 2's complement with 9 bits integer and 15 bits fraction:

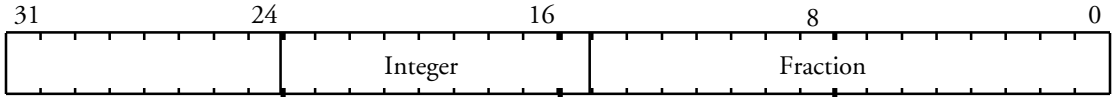


Figure 0.19 Fixed Point Color Format

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.

5.6.3 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 Figure 0.17.

5.6.4 Registers

The control register for the color DDA unit is the **ColorDDAMode** register:

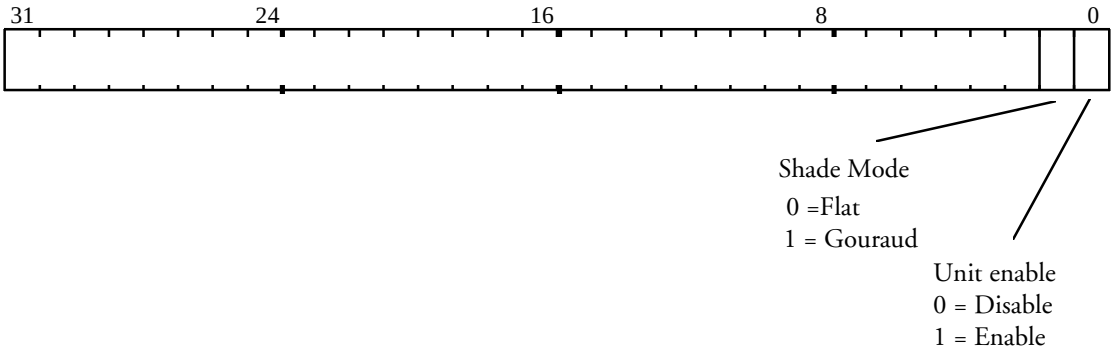


Figure 0.20 ColorDDAMode Register

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

Register	Data Field	Description
RStart	Fixed point 9.15 format	Red start value
dRdx	Fixed point 9.15 format	Red derivative per unit X
dRdyDom	Fixed point 9.15 format	Red derivative per unit Y, dominant edge
GStart	Fixed point 9.15 format	Green start value
dGdx	Fixed point 9.15 format	Green derivative per unit X
dGdyDom	Fixed point 9.15 format	Green derivative per unit Y, dominant edge
BStart	Fixed point 9.15 format	Blue start value
dBdx	Fixed point 9.15 format	Blue derivative per unit X
dBdyDom	Fixed point 9.15 format	Blue derivative per unit Y, dominant edge
AStart	Fixed point 9.15 format	Alpha start value
dAdx	Fixed point 9.15 format	Alpha derivative per unit X
dAdyDom	Fixed point 9.15 format	Alpha derivative per unit Y, dominant edge

Table 0.5 Color Interpolation Registers

5.6.5 Flat Shading Example

A flat shaded primitive:

```
// Set DDA to flat shade mode
colorDDAMode.UnitEnable = GLINT_ENABLE
colorDDAMode.Shade = GLINT_FLAT_SHADE_MODE
ColorDDAMode(colorDDAMode)
ConstantColor(0xFFFFFFFF)     // Load the flat
```

```
color
```

5.6.6 Gouraud Shaded Trapezoid Example

See section §0 for details of how to calculate delta values.

```
// Enable unit in Gouraud shading mode
colorDDAMode.UnitEnable = GLINT_ENABLE
colorDDAMode.Shade = GLINT_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 ()
```

5.6.7 Gouraud Shaded Line Example

See section §0 for details of how to calculate delta values.

```
// Set DDA for Gouraud shaded mode
colorDDAMode.UnitEnable = GLINT_ENABLE
colorDDAMode.Shade = GLINT_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
```

5.7

Texture Mapping

For each fragment within a primitive, texture mapping involves the following stages:

- calculation of the texture coordinates for each fragment.
- fetching the appropriate texel data from the localbuffer.
- derivation of the texture color from t he texel(s) (a filtering process).
- application of the texture color to the fragment's color, which is dependent on the texture application mode.

See the OpenGL Specification, and OpenGL Programming Guide for details of the theory and practice of texture mapping .

In addition to the texel filtering and application operations supported in the GLINT 300SX, the GLINT 500TX also:

- Interpolates the texture coordinates.
- Calculates the perspectively correct texture map address(es).
- Reads the texel data from the localbuffer memory.
- Formats the data from the wide variety of texture map formats into a uniform internal format.
- Assists the host in doing mip mapping by reading the texel data and filtering. The host supplies the two mip map addresses and the inter-map interpolation coefficient on a pixel by pixel basis.

The texture operation is carried out in three phases:

- Texture Address Generation
- Texture Read
- Texture Color Generation and application

5.7.1 Texture Address Generation

To generate the texture addresses, DDAs are used to interpolate the texture coordinates over a trapezoid or line primitive.

There are two general modes of operation: 2D and 3D. In 3D mode, the task divides into the following steps:

- interpolate the texture coordinates (S, T, Q) using the DDA units
- perspective correction of the coordinates by calculating S/Q and T/Q
- wrap the corrected coordinates (s, t) using mirror, repeat or clamp operations to map the coordinates into the range 0.0 to 1.0 (u, v)
- pass the resulting coordinates (u, v) to the texture read unit

For the 2D mode, the perspective correction stage is omitted and the wrap operation is always a repeat operation.

Texture Coordinate Nomenclature

A vertex has a homogeneous coordinate and texture coordinate donated by:

$$\begin{bmatrix} x_e & y_e & z_e & w_e \end{bmatrix} \text{ and } \begin{bmatrix} s_e & t_e & r_e & q_e \end{bmatrix}$$

In OpenGL the texture coordinate is transformed using a 4x4 matrix (frequently this is a unit matrix) and the default values for r_e, q_e are 0.0 and 1.0 respectively. r_e is a place holder in anticipation of 3D textures, and q_e can be used to apply perspective projections to the texture map. The values of S, T and Q are given by:

$$S = \frac{s_e}{w_e} \qquad T = \frac{t_e}{w_e} \qquad Q = \frac{q_e}{w_e}$$

The S, T and Q parameters are interpolated in DDA units in the same way all other interpolants in GLINT are. The 9 registers: **SStart**, **dSdx**, **dSdyDom**, **TStart**, **dTdx**, **dTdyDom**, **QStart**, **dQdx** and **dQdyDom** hold the start, dx and dyDom parameters for S, T and Q. The values of S, T and Q at each vertex are used to calculate the gradient values in much the same way as the color gradients when Gouraud shading.

The fixed point format of these registers can be defined as you wish, but they must be the same - the divide operation yields consistent internal results. One method of ensuring that the full range of accuracy available in the DDAs is used but not exceeded (the DDAs will clamp if the range is exceeded) is to normalize the S, T, Q values before calculating the gradient values. For example, for a triangle primitive this involves finding the maximum absolute value of the 9 values defined at the vertices and scaling the other 8 values appropriately.

At each pixel there is a division operation to achieve perspective correction of the texture coordinates and derive the s, t coordinates used to index the texture map through the equations:

$$s = \frac{S}{Q} \qquad t = \frac{T}{Q}$$

After the division, the s, t coordinates are wrapped to lie in the range 0.0 to 1.0 inclusive (and therefore within the range of the defined texture map). The wrapped coordinates are denoted as u, v. It is the u, v coordinates that are passed on to the Texture Read Unit which uses them to calculate the physical address in the localbuffer where the texture is stored.

Texture Coordinate Wrapping Modes

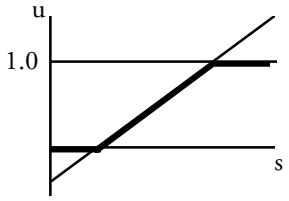
Three wrapping modes are available, and s and t can be wrapped differently. The selected mode is held in the SWrap and TWrap fields in the **TextureAddressMode**

register, and in the UWrap and VWrap fields in the **TextureReadMode** register.

Clamp

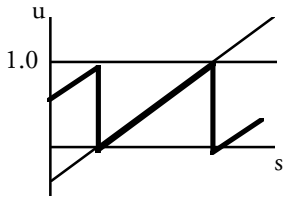
This tests the coordinate against 1.0 and if the coordinate is larger sets the coordinate to 1.0. Similarly if the coordinate is less that 0.0 it is set to 0.0.

This causes texels outside of the texture map to be set to the edge values.



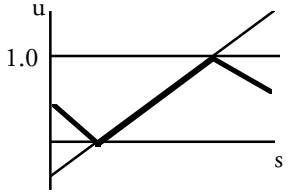
Repeat

The integer part of the coordinate is discarded just to leave the fractional part. The **Repeat** mode creates a saw-tooth transfer function, which as the name suggests, causes the texture pattern to be repeated (i.e. tiled) over the polygon. Abutting edges are from opposite sides of the texture map so unless care is taken a discontinuity may be seen.



Mirror

This is similar to **Repeat**, but when the integer part is odd the value (1.0 - fraction) is used instead of just the fraction. This creates a triangle transfer function, which has the advantage that butting edges always match.



The **Repeat** and **Clamp** modes are identical to those defined by OpenGL.

Texture Address Registers

The **TextureAddressMode** register contains four control bits:

- An enable bit, which when clear stops this unit generating texture coordinates. If this bit is set and the texture enable bit in the **Render** command is set then texture coordinates are generated.
- S Wrap. Reduces the texel s coordinate into the narrow u range as outlined above.

- T Wrap. Reduces the texel t coordinate into the narrow v range as outlined above.
- Operation bit. When this is clear the addresses are calculated in '2D mode' so no perspective correction is done. This will typically run twice as fast as when in '3D mode' where perspective correction is done. In the 2D case the wrap operation is always "repeat" as the DDA units are allowed to wrap around and have the fixed 0.32 fixed point format.

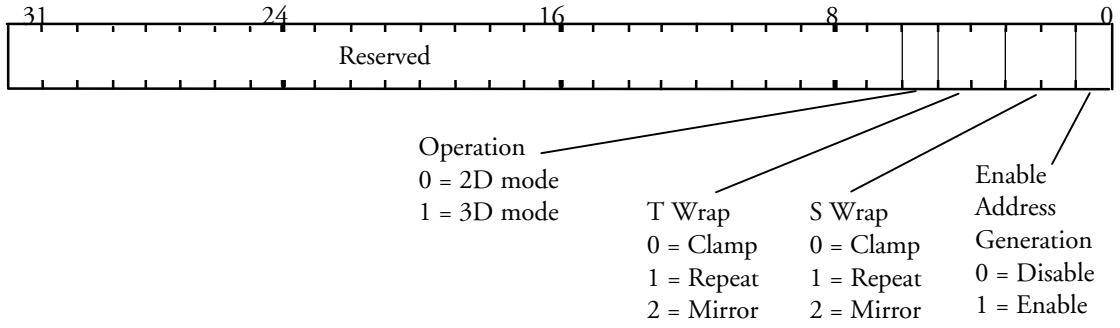


Figure 0.21 TextureAddressMode Register

The following registers set up the texture interpolation deltas :

Register	Data Field	Description
SStart		S start value
dSdx		S derivative per unit X
dSdyDom		S derivative per unit Y, dominant edge
TStart		T start value
dTdx		T derivative per unit X
dTdyDom		T derivative per unit Y, dominant edge
QStart		Q start value
dQdx		Q derivative per unit X
dQdyDom		Q derivative per unit Y, dominant edge

Table 0.6 Texture Interpolation Registers

5.7.2 Texture Read Phase

The texture read phase fetches and formats texel data from the localbuffer. This involves taking the u, v coordinates generated by the texture address unit and calculating the physical address in the localbuffer where the texture is stored. The texture information (texels) are read and converted into the internal format (8 bits per component) before being passed onto the Texture Color Generation. The interpolation coefficients (if any are needed) are derived from the u, v coordinates and passed on as well.

Filter Modes

All the filter modes of OpenGL are supported, that is:

Minification	Nearest
	Linear
	NearestMipMapNearest
	NearestMipMapLinear
	LinearMipMapNearest
	LinearMipMapLinear
Magnification	Nearest
	Linear

Table 0.7 OpenGL Filter Modes

Minification is the name given to the filtering situation where multiple texels map to a single fragment, while **magnification** is the name given to the filtering situation where only a portion of a single texel maps to a single fragment.

Nearest is the simplest form of filtering where the nearest texel to the texture coordinate location is selected.

Linear is a more sophisticated filtering algorithm which is dependent on the type of primitive. For lines (which are 1D), it involves linear interpolation between the two nearest texels. For polygons and points which are considered to have finite area, linear is in fact bi-linear interpolation which interpolates between the nearest 4 texels.

Calculating the Texel Address(es)

The address generation is controlled by the **TextureReadMode** register. It has the following fields (which are explained in more detail later on):

Field	Width	Function
Enable	1	Enables texel reads.
Width	4	1…2048 encoded as a power of two.
Height	4	1…2048 encoded as a power of two.
Depth	3	1…32 encoded as a power of two.
Border	1	No border (0) or border present (1)
Patch	1	No (0), or Yes (1)
MagFilter	1	Nearest (0), Linear (1)
MinFilter	3	Nearest (0), Linear (1), Nearest Mipmap Nearest (2), Nearest Mipmap Linear (3), Linear Mipmap Nearest (4), Linear Mipmap Linear (5)
UWrap	2	Clamp (0), Repeat (1) or Mirror (2)
VWrap	2	Clamp (0), Repeat (1) or Mirror (2)
MapType	1	1D (0) or 2D (1)
MipmapAssist	1	Disabled (0) or Enabled (1).

Table 0.8 TextureReadMode Register

The texel address(es) is calculated from the following parameters:

- *Dimensions*. A texture map is a two dimensional image, possibly with differing width and height. The width and height are given by (2^n+2b) and (2^m+2b) respectively where *b* is one when a border is present, otherwise it is zero. The values of n, m and b are stored in the **TextureReadMode** register in the Width, Height and Border fields respectively. The width or height can be one (more normally height) so the texture map is reduced to be one dimensional as required for 1D maps in OpenGL. The largest texture map supported is 2K by 2K without a border, or 2050x2050 with a border. When a texture map doesn't fit in with the above width and height equations it must be padded out to the nearest acceptable size. This is likely to occur when a font is held as a texture map but will not cause any problems as the texture coordinate DDAs can be adjusted.
- *Borders*. In OpenGL any texture map can have an extra row on the top and bottom, and an extra column on the left and right of the map so the size of a texture map may not be a power of 2. These extra border texels are only ever accessed during linear filtering, but may need to be skipped over when not needed. If a border has not been provided in the texture map, but a border texel is needed, they are taken from the **BorderColor** register.
- *Texel Size*. A texel may be 1, 2, 4, 8, 16 or 32 bits in size. The interpretation of

these bits is covered later and is of no concern for the address calculation. Texel ordering within a word is always sequential and can start from either end. The texel size, d, is encoded as a power of 2 so it can have the values 0…5 inclusive and is held in the Depth field in the **TextureReadMode** and **TextureFormat** registers.

- *Base Address*. The base address is given in units of the smallest texel size (i.e. 1 bit). The address is 29 bits in size to accommodate bit level addressing in a maximal memory system of 16M words (i.e. 24 + 5). A texture map must always start on the natural boundary for the size of texels it contains. For example a 32 bit texture will always have the bottom 5 bits set to zero. This fine level of addressing allows sub images in the texture map to be used. It is not intended to allow more efficient packing of texture maps in memory (i.e. it is not possible to store two 4x4 one bit maps in one localbuffer word) as texture download only replaces the contents of a whole localbuffer word and will not do a merge. The base address is held in the **TextureBaseAddress** register with the pixel address in the high order 24-bits.
- *Origin*. The origin is always at the base address (i.e. 0, 0) and all texels in the texture map are at higher addresses.
- *Texture map patch*. Storing the texture map in memory with one row following the next can gives poor access times when scanning along a column due to the page breaks. If the texture map is smaller than the page size then this will not occur, but frequently the texture map will be much larger than the page size so it is a concern. To make the access time less dependent on the scanning direction the texture map can be optionally stored in patches such that a 2D region of the map is stored in the same DRAM page. All the texels within a word are always sequential along a row and a patch is 16x16 words, hence the patch size in texels varies from 16x16 (for 32 bit texels) to 512x16 (for 1 bit texels). If packed texture maps are required then the packing can be done automatically during texture download, or must be done by the host if the localbuffer bypass is used. Note that some wastage of the memory space will occur if the texture map dimensions are not an integer multiple of the patch size.

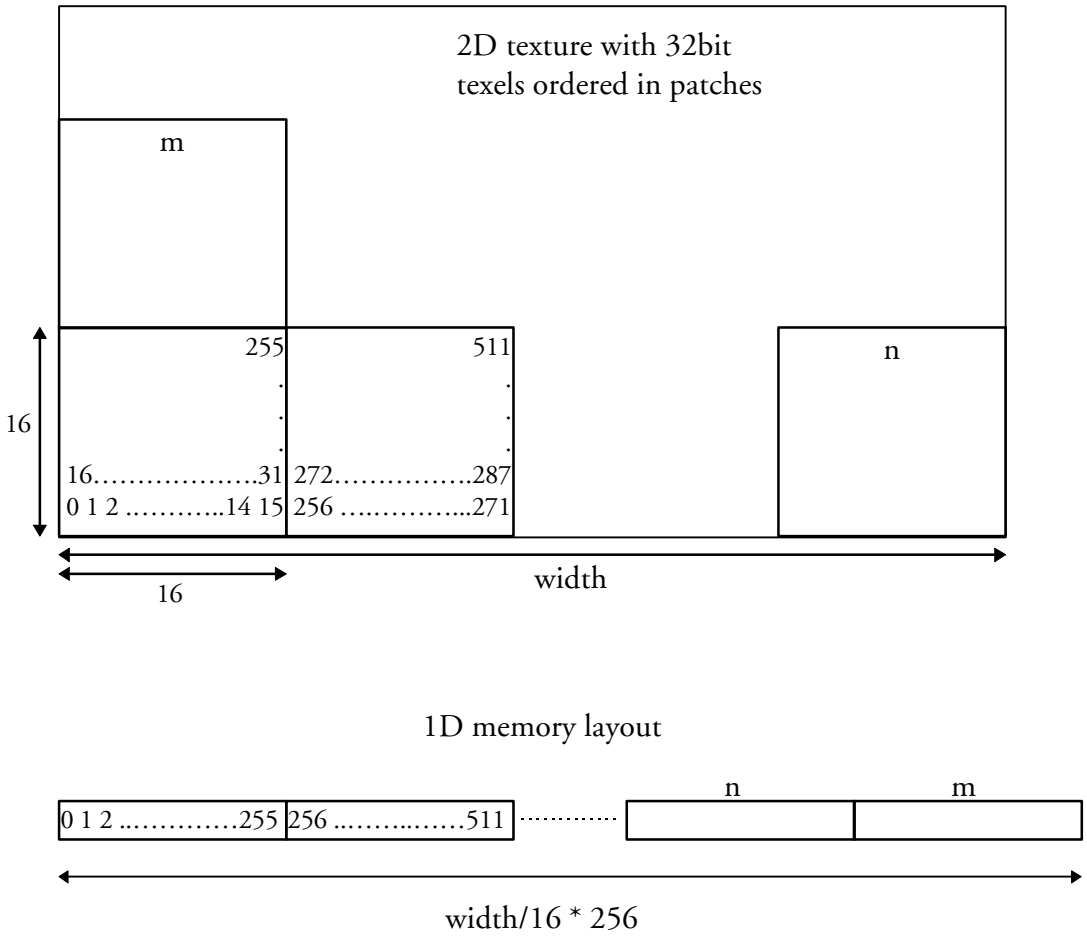


Figure 0.22 Texture Patch Example

The patch mode is only useful when the width of the map exceeds 16 words.

The patch mode works best when the height of the map is greater than 16 texels. For maps which are less than this in height a portion of the patch will not be used so the texel data will be spread out in memory. Consider a 1K word x 4 texture map. This will occupy a quarter of the patch memory so 16K words need to be set aside for 4K of texels. Moving between rows will occur without page breaks, where as in the non patch case it would incur a page break. It is possible to interleave 4 such maps so getting the benefit of less page breaks without the cost of the additional memory.

- *Filter and MapType*. The filter (Nearest or Linear) and map type (1D or 2D) determine how many addresses are generated. Note that the MinFilter is not normally used.

A texel on the map has the integer coordinates i, j and these are calculated from u, v and the width and height values. These integer coordinates are guaranteed to lie

on the texture map (excluding the border texels, if present), so for the nearest filter mode the texel is just read and used.

For the linear filter mode and 2D MapType the four texels (i, j), (i+1, j), (i, j+1) and (i+1, j+1) are read, with obvious reductions for the 1D MapType. The coordinates (i+1) and/or (j+1) may not lie on the texture map. If the texture map has a border (specified in the Border field) then the appropriate texel from the texture map is read, otherwise texel is taken from the **BorderColor** register. The texel color stored in this register is in the normal 8:8:8:8 format.

Texture Memory Layout

The 500TX has dual page detectors in the localbuffer interface so if there are two banks of memory then accesses can toggle between banks without breaking page in either bank. This is important when depth buffering and texture mapping are being done at the same time as their respective accesses are interleaved. Keeping the depth buffer in one bank and the textures in the other will give the best performance. If this separation is not possible because there is only a single bank of memory, or the depth buffer or texture maps have overflowed into the other bank there will be a performance impact, but this is reduced by the texture cache and other features of the memory interface unit.

Texture Cache

The texture data is cached to improve performance by reducing the demand for localbuffer bandwidth. The texture cache is fully associative with a LRU (least recently used) replacement policy and can hold eight 32 bit words. This translates into, for example, a 8x8 by 4 bit texture map. In the cases where a cache doesn't help because there is no re-use of data then the localbuffer is read, however texture reads are grouped together to reduce their impact on other localbuffer accesses by breaking page.

The cache is managed under software control and the **TextureCacheControl** command is used to invalidate the cache (after a texture download, for example), or to disable the cache.

Mipmapping Assistance

The GLINT 500TX does not implement mip mapping directly, but will assist the host in doing mipmapping more efficiently.

A mipmap is an ordered set of arrays representing the same image. Each array has half the linear resolution of the preceding one. This technique allows minification filtering to occur with a constant time overhead irrespective of the size of the projected area.

The first filter name for mipmapping in the Min Filter field specifies the filtering to be done on a level, and the second filter name specifies the filtering to be done between levels.

Mipmap assistance is enabled by setting the MipMapAssist bit in the **TextureReadMode** register. When this bit is set the filter operation is taken from MinFilter field in the same register (normally this field is ignored and only present

for forwards compatibility).

Textured fragments that are filtered using the mipmap technique require the calculation of a value for a variable called rho which is a measure of the projected area of the fragment on the texture map. This variable is used to select the one or two texture maps to use, the intermap interpolation coefficient (if two maps are being used) and the filter mode (which may be different between minification and magnification). These values are used to update the **TextureBaseAddress**, **TextureBaseAddressLR**, **TextureReadMode** registers and the inter-map interpolation coefficient is written to the **Interp0**, **Interp2** or **Interp4** register as directed by the following table.

	Min Filter mode	Host Interp register
1D	Nearest	None
	Linear	None
	NearestMipMapNearest	None
	NearestMipMapLinear	Interp0
	LinearMipMapNearest	None
	LinearMipMapLinear	Interp2
2D	Nearest	None
	Linear	None
	NearestMipMapNearest	None
	NearestMipMapLinear	Interp0
	LinearMipMapNearest	None
	LinearMipMapLinear	Interp4

The interpolants are specified as 9 bit unsigned fixed point numbers. The format is 1 bit integer and 8 bits fraction. The input values will usually lie in the range 0.0 to 1.0.

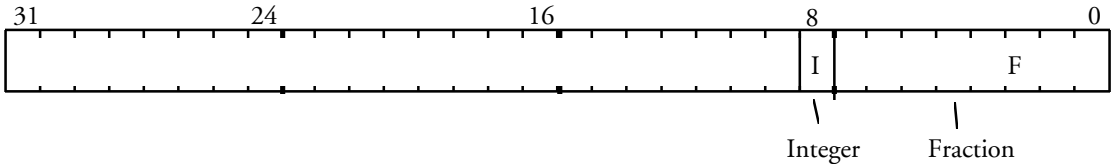


Figure 0.23 Interpolant Fixed Point Format

For the situations where rho is considered to be a constant for all fragments in a primitive, the following state set-up is required before rasterizing the primitive:

```
TextureBaseAddress
TextureBaseAddressLR // if needed
Interp           // depends on the filter mode
TextureReadMode  // width,height and min filter
// set up texture interpolation deltas
// set up primitive interpolation deltas
// render primitive
```

The addresses of the two maps are held in **TextureBaseAddress** and **TextureBaseAddressLR** registers. The second register holds the address of the Low Resolution map whose width and height is half that of the first map. Recall that the width and height values in **TextureReadMode** register are defined as powers of 2 so the values used in the address calculations are one less and clamped to zero if the subtraction would cause either to go negative. All other parameters about the maps (depth, border, patch, U wrap, V wrap, map type) are the same for both maps.

Depending on the 'on map' filtering mode and the map type, up to four texels on each level may be read at (i, j), (i+1, j), (i, j+1) and (i+1, j+1).

For the situations where rho varies for each fragment in a primitive, more interaction with the host is involved. A description for this case is beyond the scope of this manual - please contact 3Dlabs for full details.

Texel Formatting

The texel formatting is controlled by the **TextureFormat** register and it has the following fields:

Field	Width	Function
Order	1	Little endian (0) or big endian (1)
Format	1	Alternative 16 bit format.
ColorOrder	1	BGR (0) or RGB (1)
NumComps	2	1, 2, 3 or 4
OutputFormat	2	Texel (0), Color (1) or Bitmask (2)
MirrorBitMask	1	No (0) or Yes (1)
InvertBitMask	1	No (0) or Yes (1)
ByteSwapBitMask	1	No (0) or Yes (1)

Table 0.9 Texel Format Register

Only the least significant 32 bits of the localbuffer are used for texture storage. If the localbuffer is wider then the additional bits are ignored during texture reads, and overwritten with zeros during texture downloads.

A texel can be 1, 2, 4, 8, 16 or 32 bits in size (depth) and is converted into the internal 32 bit wide texel format.

The first step is to extract the appropriate bits from the data returned by the localbuffer. The texel's coordinates, depth and order determine which texel out of the 32 bit localbuffer word is extracted. If the order is little endian then increasing u (or i) coordinate runs from the most significant end towards the least significant end of the 32 bits and vice versa for big endian order.

The next stage is to take the texel data and extract the RGBA components and format them into the 32 bit internal format. OpenGL defines texture maps as having 1, 2, 3 or 4 components and the formats supported are:

			Number of Components			
Color Order	Localbuffer Texel Width	Format	1	2	3	4
X	1	X	LUT	LUT	LUT	LUT
X	2	X	LUT	LUT	LUT	LUT
X	4	X	LUT	LUT	LUT	LUT
BGR	8	X	L_8	A_4L_4	$B_2G_3R_3$	$A_1B_2G_3R_2$
	16	0	-	A_8L_8	$B_5G_6R_5$	$A_4B_4G_4R_4$
	16	1	-	A_8L_8	$B_5G_5R_5$	$A_1B_5G_5R_5$
	32	X	-	-	-	$A_8B_8G_8R_8$
RGB	8	X	L_8	A_4L_4	$R_3G_3B_2$	$A_1R_2G_3B_2$
	16	0	-	A_8L_8	$R_5G_6B_5$	$A_4R_4G_4B_4$
	16	1	-	A_8L_8	$R_5G_5B_5$	$A_1R_5G_5B_5$
	32	X	-	-	-	$A_8R_8G_8B_8$

Table 0.10 Supported Texel Formats

The 1, 2 and 4 bit texels are converted using a LUT to the internal format. The LUT converts an indexed texel value into an RGBA value. The LUT is loaded by writing to the **TexelLUT**[16] registers. For 3D this allows texture maps to be compressed so that they take up less memory.

For 2D it allows a one bit texture to be used as a stipple to provide a foreground and background color. A 4 bit texture can be used to hold CI dither offsets which the LUT translates into color values (RGBA or CI). This allows Microsoft Windows compatible CI dithering to be implemented.

The table's entries show how the texel data is expanded into the internal format. The subscripts are the number of bits in the corresponding component and the component ordering is with least significant on the right. The luminance values (L) are replicated into the RGB components. When no alpha (A) value is specified it is set to 255. A dash indicates an over specification of the format (i.e. an 8 bit luminance value in a 16 bit wide field). If such a combination occurs then the nearest earlier entry for this number of components is used and the extraneous data is ignored

The 32 bit texel value is now optionally (in this order):

- Byte swapped. If the bytes are labeled ABCD on input then after byte swapping they will have the order DCBA. This allows the normal bit mask format provided by Microsoft Windows to be used directly.
- Mirrored. This swaps bit 0 and bit 31, bit 1 and bit 30, etc. so when no mirroring is enabled the least significant bit in the texel will be the left most pixel in the span. With mirroring the most significant bit in the texel will be the left most pixel in the span.

- Inverted. This simply inverts the texel bits before they are used. This allows the same bit mask to be used to fill in the foreground pixels in one color and then the background pixels in a different color on a second rasterization pass with inversion occurring.

These operations are identical to those provided in the rasterizer for bit mask operations and are intended to allow bit mask data to be held in the localbuffer for use with 2D span processing. However, these operations are available for use in the general 3D case.

The next stage is controlled by the OutputFormat bits in the **TextureFormat** register. They have the following effect:

- Texel. The texel is passed to the Texture Color Unit for further texture processing.
- Color. This would be used by 2D operations as no further texture processing is needed.
- BitMask. The 32 bit texel is held locally and is ANDed with the next span data sent from the rasterizer and subsequently used to read and/or write a span of data.

Texture Read Registers

The **TextureReadMode** register controls the general operation of texel reads and has the following format:

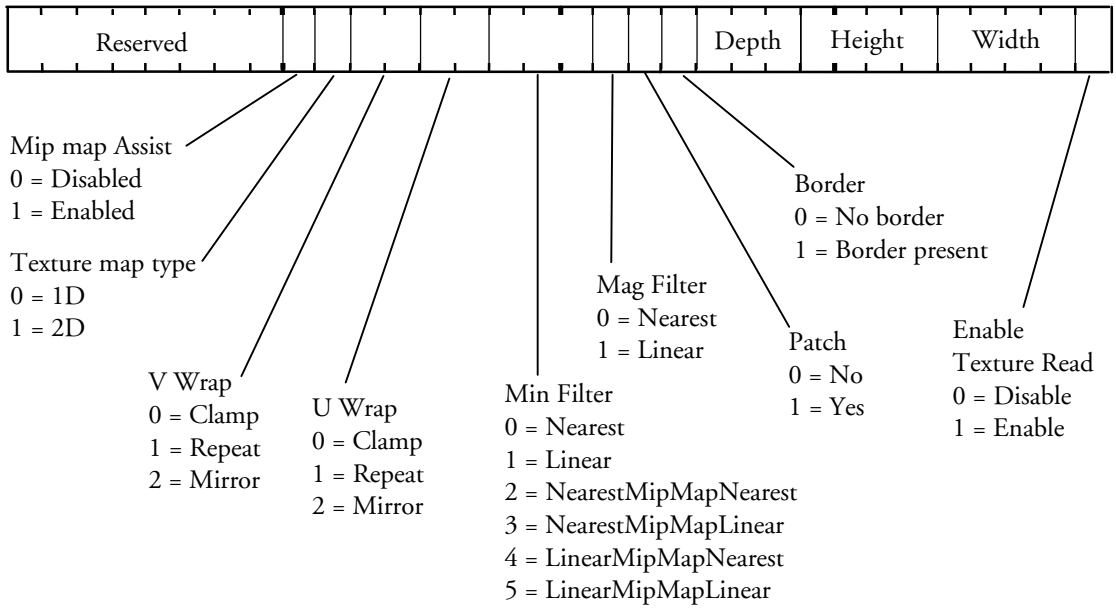


Figure 0.24 TextureReadMode Register

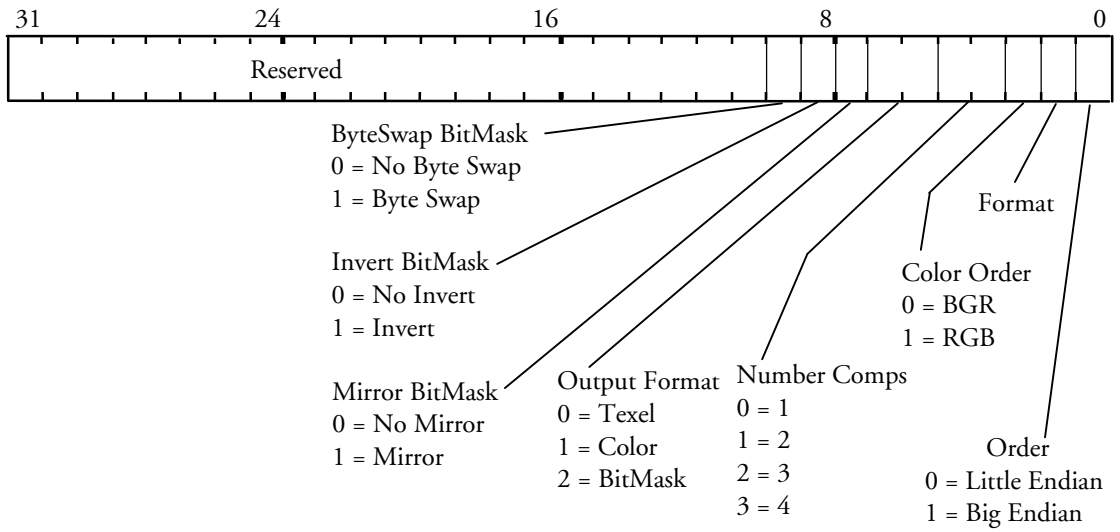


Figure 0.25 TextureFormat Register

Other registers in the texture read unit:

Register	Data Field	Description
TexelCoordU	20 ls bits store coordinate	Only used if host needs to provide a texture coordinate
TexelCoordV	20 ls bits store coordinate	Only used if host needs to provide a texture coordinate
TextureBaseAddress	29 bit address	Lower 5 bits specify address within a word
TextureBaseAddressLR	29 bit address	Lower resolution map address when mip map assistance is enabled. Lower 5 bits specify address within a word.
BorderColor	32 bit color format. Red in lower byte	Only relevant when filter operation is linear
TexelLUT[16]	32 bit texel/color format. Red in lower byte	Relevant for 1, 2 & 4 bit texels
TexelCacheControl	Bit0: 0=No invalidate 1= Invalidate cache Bit 1: 0=Disable cache 1=Enable cache	Allows software control of the texture cache operation

Table 0.11 Other Texture Read Registers

5.7.3 Texture Color Generation

The final phase of the texturing process combines the incoming fragment's color (generated in the color DDA unit) with the texture color value generated from the texture read phase. The function used to combine these two colors is referred to as the texture application mode. The available options are split into two types - OpenGL and QuickDraw3D. The OpenGL options are one of Decal, Blend or Modulate. The QuickDraw3D options are any combination of Decal, Modulate or Highlight.

OpenGL Application Modes

Once the texture value, R, has been calculated it is used in one of three ways: Modulate, Decal, or Blend. The vector equations for these three options are:

Modulate	$C_{rgba} = R_{rgba} * F_{rgba}$
Decal	$C_{rgb} = Lerp(F_{rgb}, R_{rgb}, R_a)$
	$C_a = F_a$
Blend	$C_{rgb} = Lerp(F_{rgb}, K_{rgb}, R_{rgb})$
	$C_a = F_a * R_a$

Where:

R = Texture color
F = Fragment color
K = Texture Environment color
C = Final color
Lerp(A, B, α) linearly interpolates between A and B using α as the interpolation coefficient:

$$Lerp(A, B, \quad) = (1 - \quad) * A + \quad * B$$

and the subscripts identify individual color components.

Apple Texture Modes

These texture application modes support the QuickDraw3D API.

Once the texture value, R, has been calculated it is used in the following ways (any combination of these operations are allowed and they are done in the order given).

Type	Application Mode	Equation
Decal	bit 0	If enabled $R_{rgb} = R_a R_{rgb} + (1 - R_a) F_{rgb}$ $R_a = F_a$ else $R_{rgb} = R_{rgb}$ $R_a = R_a F_a$
Modulate	bit 1	$R_{rgb} = R_{rgb} * Kd$ $R_a = R_a$
Highlight	bit 2	$R_{rgb} = R_{rgb} + Ks$ $R_a = R_a$

Where:

R = Texture color
F = Fragment color
Kd = interpolated diffuse color
Ks = interpolated specular color

and the subscripts identify individual color components.

The Ks and Kd values are taken from the Ks and Kd DDA units respectively. The 6 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 2.22 fixed point format 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 color gradients, when Gouraud shading.

The parameters to control the two DDA units are loaded into the red, green and blue values (there is no alpha value) and are held as 1.8 unsigned fixed point numbers so values greater than 1.0 can be represented.

The final value R is the color used in subsequent color calculations

This style of texture application is used when the TextureType field in the **TextureColorMode** register is Apple.

Compatibility with GLINT 300SX texturing

The following texture registers still exist for backward compatibility: **Texel**[0...7] and **TextureFilter** but are not needed when GLINT 500TX texturing capabilities are being used. Existing code that implements texturing for the GLINT 300SX will operate similarly on the GLINT 500TX provided that texel reads and texture address generation are disabled (by clearing the bit0 of the **TextureReadMode** and **TextureAddressMode** registers respectively).

Texture Color Registers

The application of texture is qualified by the TextureEnable bit in the **Render** command register. The following registers control the application of textures.

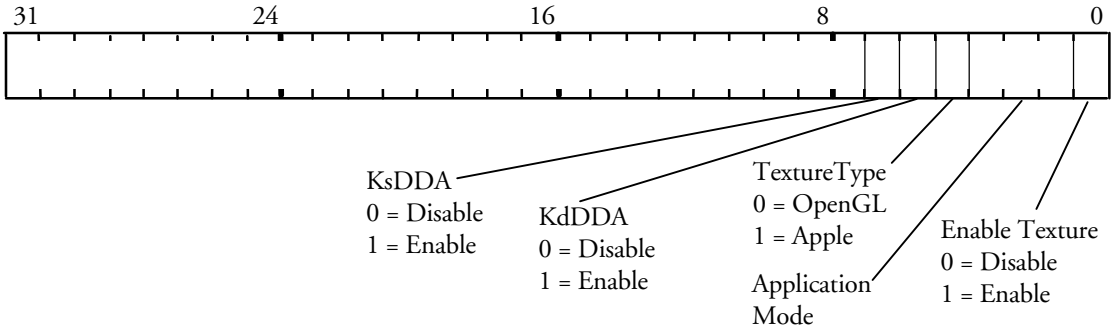


Figure 0.26 TextureColorMode Register

Register	Data Field	Description
TextureEnvColor	32 bit RGBA format, R in least significant byte	standard internal color format, see Figure 0.17
KsStart	24 bit 2's comp fix pt	Ks start value
dKsdx	24 bit 2's comp fix pt	Ks derivative unit X
dKsdyDom	24 bit 2's comp fix pt	Ks derivative unit Y, dominant edge
KdStart	24 bit 2's comp fix pt	Kd start value
dKddx	24 bit 2's comp fix pt	Kd derivative unit X
dKddyDom	24 bit 2's comp fix pt	Kd derivative unit Y, dominant edge

Table 0.12 Other Texture Color Registers

5.7.4 Downloading Texture Maps

Texture maps are downloaded into the localbuffer by simply writing the data to the **TextureData** register (ideally using the on-chip DMA controller). In this mode of operation the peak download rate is 50M words per second [1]. The bypass path to the localbuffer can also be used but this will be much slower.

Texels which are less than 32 bits in size will need to be packed into a 32 bit word before they are downloaded and the packing must be compatible with the way the texels will subsequently be read. Note that the presence of a border in the texture map will complicate the packing as some 32 bit words will now contain texels from adjacent rows.

Texture maps are always stored linearly in memory.

[1]Other system factors such as memory speed, PCI clock frequency, etc. will reduce the system download speed.

The base address of the texture map is specified in the **LBWindowBase** register so it will need to be restored after texture download is complete. The texel word to write to is specified by the **TextureDownloadOffset**. This register auto increments after every texture word is written to the localbuffer, so generally is zeroed at the start of the download, and ignored thereafter. Where the **TextureDownloadOffset** register is useful is if the texture map is downloaded in strips, maybe due to the size of the DMA buffer being smaller than the texture map, or for context switching reasons. At the start of each strip the **TextureDownloadOffset** register is loaded with the offset value for the start of this strip. The offset is 24 bits in size. The **LBWindowBase** register always gives the base address of the texture map and should never be used to give the address of a strip (other than the first one) otherwise any address patching will not work properly.

If the Patch bit in the **LBReadMode** register is set, the texture map address is modified to be compatible with the patching used when reading the texture maps. A patch is always 16 x 16 words, or 16 * (6 - d) by 16 texels where d is the number of bits in a texel expressed as a power of 2. To form the correct address the Localbuffer Unit needs to know the width of the texture map and this is encoded in the PatchCode (in the **LBReadMode** register) as follows:

width in words	Patch code
32	0
64	1
128	2
256	3
512	4
1024	5
2048	6

Two notes on patching:

1. The patch mode is only useful when the width of the map exceeds 16 words.
2. The patch mode works best when the height of the map is greater than 16 texels. For maps which are less than this in height a portion of the patch will not be used so the texel data will be spread out in memory. Consider a 1K word x 4 texture map. This will occupy a quarter of the patch memory so 16K words need to be set aside for 4K of texels. Moving between rows will occur without page breaks, where as in the non patch case it would incur a page break. It is possible to interleave 4 such maps so getting the benefit of less page breaks without the cost of the additional memory.

In summary the only registers which need to be set up for a texture download operation are:

- **LBWindowBase**
- **TextureDownloadOffset**

- **LBReadMode** (PatchEnable and PatchCode fields only).

There is a danger that a texture mapped primitive immediatdy following a texture download may start to read texel data still waiting to be written (texture units before localbuffer units), or conversely a download may overwrite texel data in the process of being read (localbuffer units before texture units). If there is any chance this situation might arise then the **WaitForCompletion** command can be used to prevent a rendering action from starting until all the fragments associated with the previous render action have been written to memory. This command is conceptually similar to the **Sync** command but the host does not need to read from the output FIFO. There is no data field required with the **WaitForCompletion** command.

The texture download mechanism outlined defines what is needed. However the semantics of texture downloading in OpenGL allows for all the fragment formatting operations to be available when downloading images. The normal case will be a straight download with no fragment processing. When this is not so the texture map will need to be processed, maybe into off-screen framebuffer, before loading into the localbuffer as described above.

5.7.5 Texture Order

Any texture operations will cause a loss in performance over the same non-textured rendering, so it is a good idea only to texture those pixels which pass all the depth, stencil and GID tests. OpenGL defines the order in which operations are to be performed on fragments as texture, alpha test, stencil and then depth. It is very likely that in a typical scene many textured fragments will get rejected by the depth test, say, which isn't the most effective use of the texturing capacity. If the alpha test is disabled (or cannot reject fragments) then OpenGL compatible semantics are still maintained if the order is rearranged to be stencil, depth, texture and then alpha test.

The GLINT 500TX has a pipeline which can be re-configured into either of the two orders (TextureDepth or DepthTexture) by writing to the **RouterMode** register. Changing the pipeline order is self synchronizing so the user does not need to wait for the pipeline to empty first.

5.7.6 Texture Download Example

This example shows the state preparation needed to download a texture map in a single block from host memory into the localbuffer with patching enabled .

```
// Texture Download with patching enabled
lbReadMode.Patch = GLINT_TRUE
lbReadMode.PatchCode = widthLog2 + depthLog2 - 10
LBReadMode(lbReadMode)

LBWindowBase(LocalBufferTextureBaseAddress)

TextureDownloadOffset(0)
```

```
// have the texture cache enabled but invalidate
the
// cache
TextureCacheControl
(GLINT_TEXTURE_CACHE_CONTROL_ENABLE|
GLINT_TEXTURE_CACHE_CONTROL_INVALIDATE )

// ensure wait for any outstanding texture reads to
// finish
WaitForCompletion()

// loop through texture data in 32bit steps
for ( i=0 ; i< cDWORDS ; i++ )
     TextureData( textureData )

// ensure wait for outstanding texture writes to
// finish
WaitForCompletion()
```

5.7.7 Texture Mapping Example

This example shows how to prepare GLINT 500TX state render a textured triangle primitive. It assumes the texture has been downloaded using the approach in section §0. This example describes the usual case where the texture filter function (nearest or linear) does not vary across the primitive. In this case there is no involvement required from the host per pixel.

```
// Prepare the texture address unit
textureAddressMode.EnableUnit = GLINT_ENABLE
textureAddressMode.SWrap = GLINT_REPEAT
textureAddressMode.TWrap = GLINT_CLAMP
textureAddressMode.Operation = GLINT_ENABLE // 3D
mode
TextureAddressMode(textureAddressMode)

// Prepare the texture read unit
textureReadMode.EnableUnit = GLINT_ENABLE
textureReadMode.Width = widthLog2
textureReadMode.Height = heightLog2
textureReadMode.Depth = depthLog2
textureReadMode.Patch = GLINT_TRUE
textureReadMode.MagFilter = GLINT_NEAREST
textureReadMode.UWrap = GLINT_REPEAT
textureReadMode.VWrap = GLINT_CLAMP
textureReadMode.TextureType = GLINT_ENABLE // 2D
type
textureReadMode.MipmapAssist = GLINT_DISABLE
TextureReadMode(textureReadMode)

// Prepare the texture format unit
```

```
textureFormat.Data = 0        // set all fields to
0
textureFormat.NumberComps = GLINT_4_COMPONENTS
textureFormat.OutputFormat = GLINT_TEXEL

// Enable the texture application mode
textureColorMode.EnableTexture = GLINT_TRUE
textureColorMode.ApplicationMode = GLINT_DECAL
TextureColorMode(textureColorMode)

// Point at the defined texture in localbuffer
TextureBaseAddress(LocalBufferTextureBaseAddress <<
5)

// Set-up to render into the framebuffer
// Not shown.
// Normalise S, T, Q values from all 3 vertices
// Not shown.
// Calculate the S, T, Q deltas
// Not shown.
// Set-up the S, T, Q delta values
SStart()
dSdx()
dSdyDom()
TStart()
dTdx()
dTdyDom()
QStart()
dQdx()
dQdyDom()

// Render triangle
// rasterization deltas not shown
render.PrimitiveType = GLINT_TRAPEZOID_PRIMITIVE
render.TextureEnable = GLINT_TRUE
Render(render)
```

5.8 Fog Unit

The fog unit is used to blend the incoming fragment's color (generated by the color DDA unit, and potentially modified by the texture unit) with a predefined 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 the 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 one of the equations described below.

The GLINT 500TX performs the fog calculations in parallel to the texture filtering and application so (unlike the GLINT 300SX) there is no degradation in performance when both fog and texture are enabled.

Note that although the fog values are linearly interpolated over a primitive the fog values 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 would typically be an exponential function.

5.8.1 Fog Index Calculation - The Fog DDA

The fog DDA is used to interpolate the fog index (f) across a primitive. The mechanics are similar to those of the other DDA units, as the diagram below illustrates:

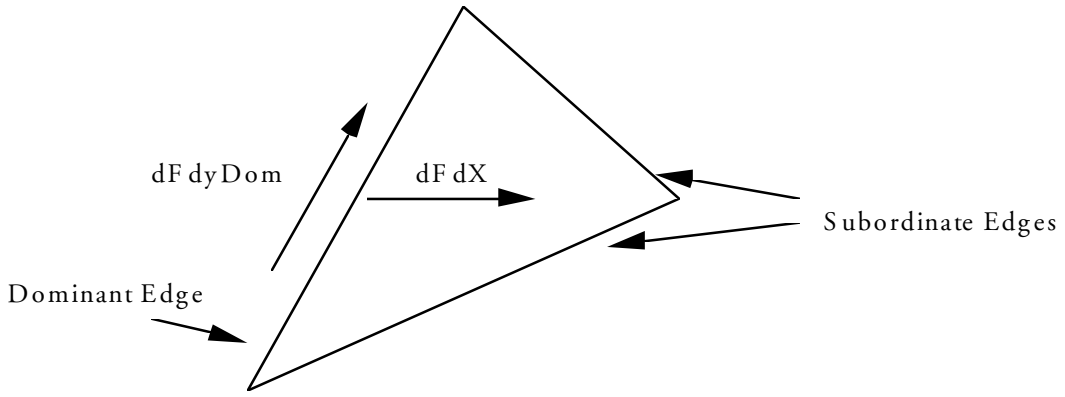


Figure 0.27 Fog Interpolation Over A Triangle

where:

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 a 32bit fixed point value. The format is 2's complement with 10 bits integer and 22 bits fraction.

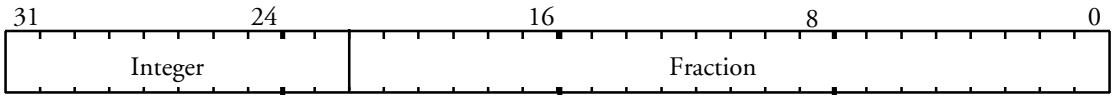


Figure 0.28 Fog Interpolant Fixed Point Format

The DDA has an internal range of approximately +511 to -512, in some cases primitives may exceed these bounds. This problem typically occurs for very large polygons which span the whole depth of a scene. The correct solution is to tessellate the polygon until polygons lie within the acceptable range, however, the visual effect is frequently negligible and can often be ignored.

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.

5.8.2 RGBA Fogging Equation

Fogging is applied differently depending on the color mode. For RGBA mode 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 fragments. Initially no fogging occurs, $f \geq 1.0$, then a region of linear combination of the fragment color and fog color occurs $0.0 < f < 1.0$, followed by a region of constant fog color, $f \leq 0.0$.

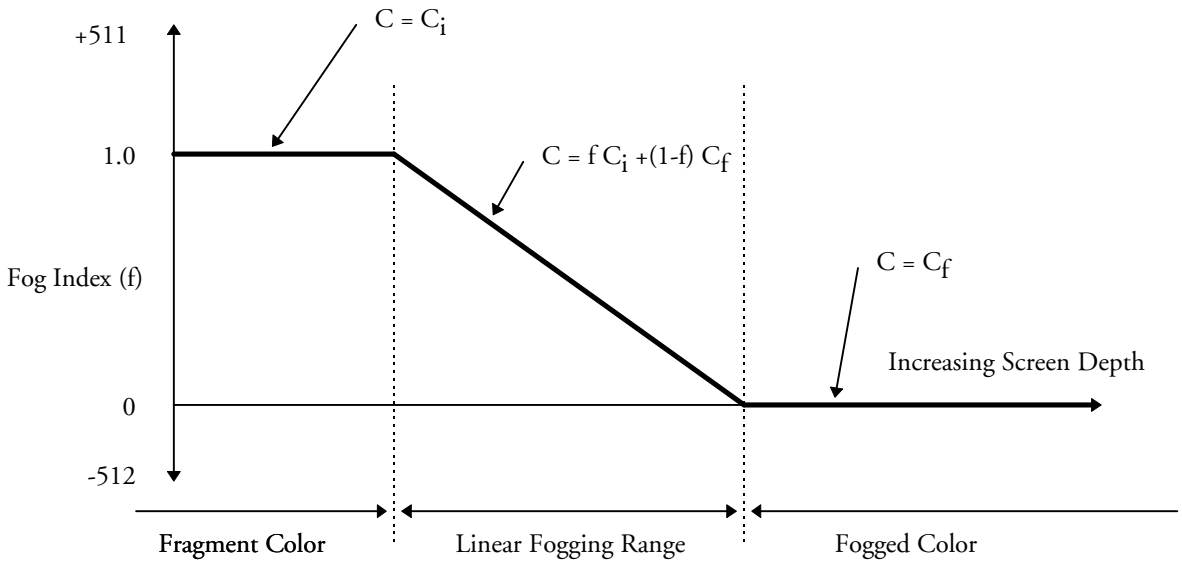


Figure 0.29 RGBA Fogging

5.8.3 CI Fogging Equation

For color index mode the equation is:

$$I = I_i + (1-f)I_f$$

where:

I = outgoing fragment color index
I_i = incoming fragment color index
f = fog index
I_f = fog color index

5.8.4 Registers

The **FogMode** register is used to enable and disable fogging (qualified by the fog application bit in the **Render** command register).

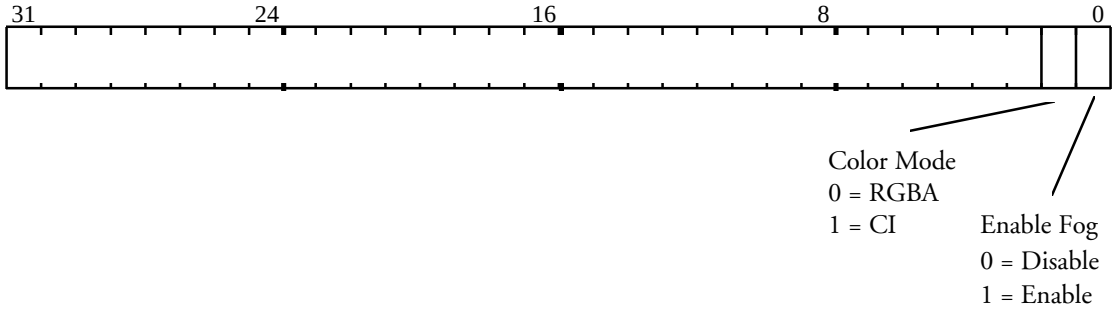


Figure 0.30 FogMode Register

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 10.22 fixed point format as described above.

5.8.5 Fog Example

A Gouraud shaded, fogged RGBA trapezoid, with the fog color set to white (see §0 for details of how to calculate color and fog delta values).

```
// Enable color DDA unit in Gouraud shading mode
colorDDAMode.UnitEnable = GLINT_ENABLE
colorDDAMode.Shade = GLINT_GOURAUD_SHADE_MODE

ColorDDAMode(colorDDAMode)

// Enable the Fog unit
fogMode.FogEnable = GLINT_TRUE
fogMode.ColorMode = GLINT_RGBA_MODE

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 = GLINT_TRUE
```

5.9

Antialias Application Unit

Antialias application controls the combining of the coverage value generated by the rasterizer with the color generated in the color DDA units. The application depends on the color mode, either RGBA or Color Index (CI).

5.9.1 Antialias Application

When antialiasing is enabled this unit is used to combine the coverage value calculated for each fragment with the fragment's alpha value. In RGBA mode the alpha value is multiplied by the coverage value calculated in the rasterizer (its range is 0% to 100%). The RGB values remain unchanged and these are modified later in the Alpha Blend unit which must be set up appropriately. In CI mode the coverage value is placed in the lower 4 bits of the color field. The Color Look Up Table is assumed to be set up such that each color has 16 intensities associated with it, one per coverage entry.

5.9.2 Polygon Antialiasing

A number of issues should be considered when using GLINT to render antialiased polygons. Depth buffering cannot be used with GLINT antialiasing. This is because the order the fragments are combined in is critical in producing the correct final color. Polygons must therefore be depth sorted, and rendered front to back, using the alpha blend modes: **SourceAlphaSaturate** for the source blend function and **One** for the destination blend function. In this way the alpha component of a fragment represents the percentage pixel coverage, and the blend function accumulates coverage until the value in the alpha buffer equals one, at which point no further contributions can made to a pixel.

Although this technique works well in many cases, it is an approximation. Consider the case below which shows three polygons of equal depth which intersect a single pixel. In this case there would ideally be a contribution from each of the polygons. However, if the rendering order is polygon A followed by polygon B, each of which contributes approximately 50% pixel coverage, then polygon C will make no contribution to the pixel as the alpha value is 'saturated' (50%+50%=100%).

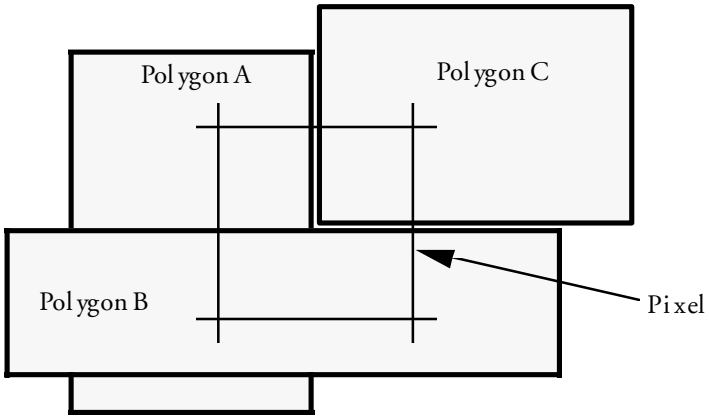


Figure 0.31 Polygon Antialiasing

For the antialiasing of general scenes, with no restrictions on rendering order, the accumulation buffer is the preferred choice. This is indirectly supported by GLINT via image uploading and downloading, with the accumulation buffer residing on the host.

When antialiasing, interpolated parameters which are sampled within a fragment (color, fog and texture), will sometimes be unrepresentative of a continuous sampling of a surface, and care should be taken when rendering smooth shaded antialiased primitives. This problem does not occur in aliased rendering, as the sample point is consistently at the center of a pixel.

See The OpenGL Programming Guide for more details of antialiasing.

5.9.3 Registers

The **AntialiasMode** register controls the unit:

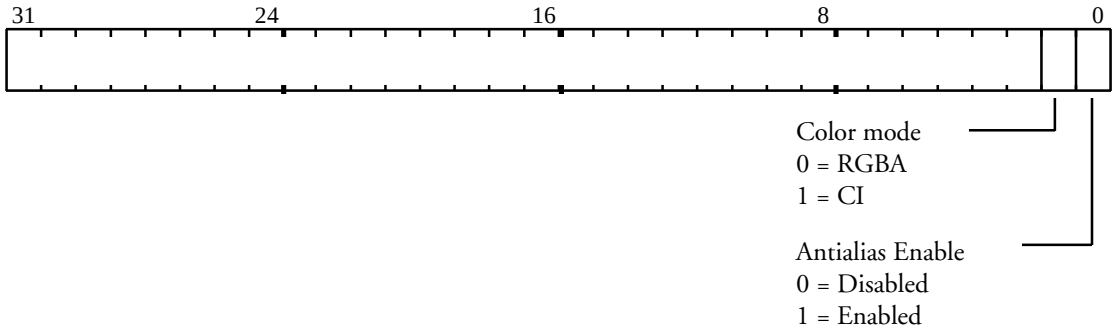


Figure 0.32 AntialiasMode Register

For the coverage application to take place the enable in the AntialiasMode register must be qualified by the CoverageEnable bit in the **Render** command register.

5.9.4 Antialias Example

Enable antialiasing for a RGBA primitive:

```
// Set AA application for RGBA primitive
antialiasMode.AntialiasEnable = GLINT_TRUE
antialiasMode.ColorMode = GLINT_RGBA

AntialiasMode(antialiasMode)

// Set the blend mode to an appropriate value if
// blending is required.  Not shown.

// When issuing a Render command the CoverageEnable
// bit should be set in addition to the antialias
// unit being enabled:
// render.CoverageEnable = GLINT_TRUE
```

5.10 Alpha Test Unit

The alpha test compares a fragment's alpha value with a reference value. Alpha testing is not available in color index (CI) mode.

5.10.1 Alpha Test

The alpha test conditionally rejects a fragment based on the comparison between a reference alpha value and one associated with the fragment, the available tests are:

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

Table 0.13 Alpha Test Comparison Tests

The sense of the test is such that if the comparison mode is set to Less and the reference value is set to 0x80, then fragments with alpha values between 0x0 and 0x7F will pass the test and fragments with alpha values between 0x80 and 0xFF will fail the test and be rejected.

5.10.2 Registers

The **AlphaTestMode** register controls the alpha test:

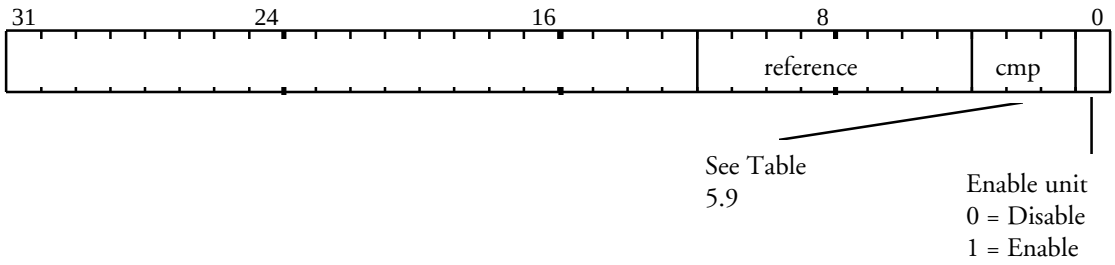


Figure 0.33 AlphaTestMode Register

5.10.3 Alpha Test Example

Set the alpha test mode to be LESS and the reference value to be 0x80:

```
// Enable unit and set modes
alphaMode.UnitEnable = GLINT_ENABLE
alphaMode.Compare = GLINT_ALPHA_COMPARE_MODE_LESS
alphaMode.Reference = 0x80
AlphaMode(alphaMode)     // Load register
// Render primitives
```

5.11 Localbuffer Read/Write Unit

The localbuffer holds the Graphic ID, FrameCount, Stencil and Depth data associated with a fragment. The localbuffer read/write unit controls the operation of GID testing, depth testing and stencil testing.

5.11.1 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, stenc il or GID testing. This requires no localbuffer reads or writes.
- Normal rendering without depth or stencil testing and with GID testing. This requires a localbuffer read to get the GID from the localbuffer.
- 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 with or without GID testing. This requires reads and writes enabled.
- Image upload/download operations. Operations which download depth or stencil information to the localbuffer or read depth, stencil fast clear or GID from the localbuffer.

The address calculation implements the following equations, it applies to reads and writes:

Bottom left origin -

```
Destination address = LBWindowBase - Y/S * W + X
Source address = LBWindowBase - Y/S * W + X +
   LBSourceOffset
```

Top left origin -

```
Destination address = LBWindowBase + Y/S * W + X
Source address = LBWindowBase + Y/S * 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.

`S`	is the Scanline interval for multi-GLINT 500TX systems
`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.

Note: *Turning on Patch addressing introduces additional complexity into the address calculation which is beyond the scope of this manual. Localbuffer bypass accesses are not recommended when Patch mode addressing is enabled.*

The localbuffer can be read in three formats: LBDefault, LBStencil or LBDepth. These tell GLINT 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 [1]. 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 GID, Depth or Stencil enabled.
Disabled	Enabled	Disabled	LBDefault	Rendering with no Stencil or depth tests enabled, but with GID testing enabled
Disabled	Enabled	Enabled	LBStencil LBDepth	Image download. GID testing optional.
Disabled	Enabled	Disabled	LBStencil LBDepth	Image upload. GID testing optional.
Disabled	Enabled	Enabled	LBDefault	Rendering with depth and/or stencil updates enabled. GID testing optional.
Enabled	Enabled	Enabled	LBDefault	Copy operations with GID testing.
Enabled	Disabled	Enabled	LBDefault	Copy operations with no GID testing.

Table 0.14 Localbuffer Read/Write Modes.

[1]Note that these fields are read independently because the width of the localbuffer is greater than the width of the host data bus.

5.11.2 Localbuffer Write

Writes to the localbuffer must be enabled to allow any update of the localbuffer to take place. The **LBWriteMode** register has two data fields, EnableWrite controls the buffer updating and UpLoadData is used for reading back depth and stencil values and for picking.

5.11.3 Localbuffer Data Formats

The four data fields supported in the localbuffer and their allowed lengths and positions are shown in the following table:

Field	Lengths	Start positions
Depth	16, 24, 32	0
Stencil	0, 4, 8	16, 20, 24, 28, 32
FrameCount	0, 4, 8	16, 20, 24, 28, 32, 36, 40
GID	0, 4	16, 20, 24, 28, 32, 36, 40, 44, 48

Table 0.15 Localbuffer Configurations

In addition there is a compact mode for a 32bit wide localbuffer where depth is 24bits, stencil is 1bit, FrameCount is 4bits, and GID is 3bits.

The **LBReadFormat** and **LBWriteFormat** registers must be configured to the appropriate values, see Figure 0.34. The format can be different for different windows.

*Note: The **LBReadFormat** and **LBWriteFormat** registers should not be written to while there are pending reads to the localbuffer. To avoid this a write to these registers should normally be preceded by a **WaitForCompletion** command.*

5.11.4 Registers

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

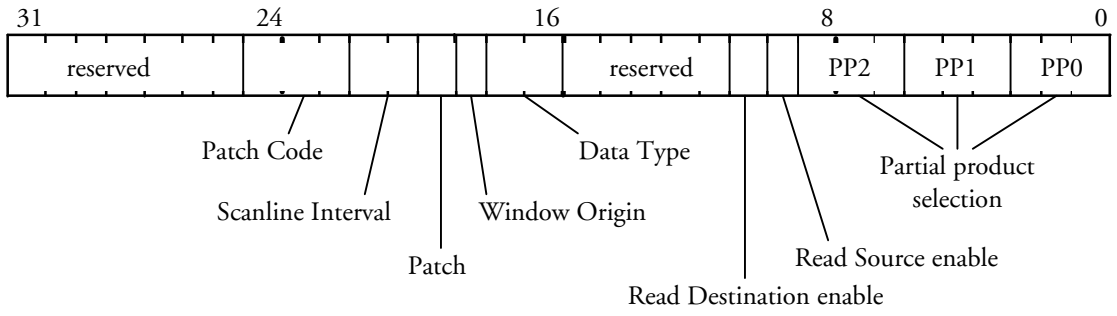


Figure 0.34 LBReadMode Register

The Partial Product fields PP0, PP1, and PP2 define the width of the localbuffer.

They are described in the Hardware Data Structures chapter.

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.

When the Patch bit is set then Patch mode addressing is enabled. This typically results in more efficient memory bandwidth utilization in the localbuffer, as it minimizes the number of page breaks generated when rendering a primitive, and so should be viewed as the normal default case. One case where this mode should not be enabled is when a datastructure needs to be accessed through the localbuffer bypass.

The ScanlineInterval is used in multi-GLINT 500TX systems. See chapter 7. For more details.

The PatchCode controls the address generation for texture mapping . See section 5.7.4 for further details.

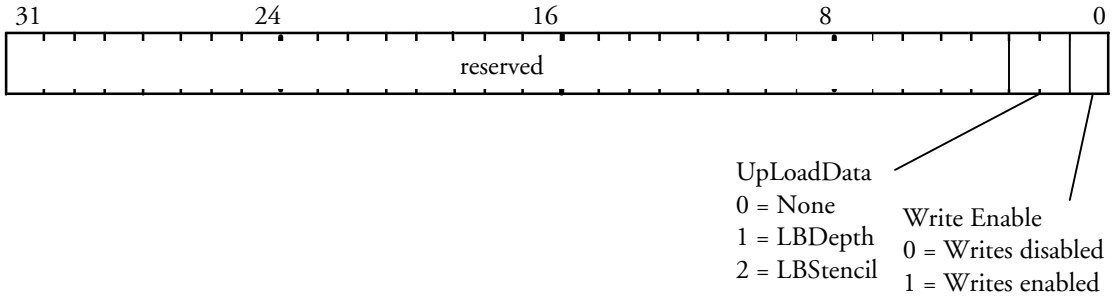


Figure 0.35 LBWriteMode Register

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. In all cases care should be taken to ensure that the field widths and positions are such that the fields do not overlap.

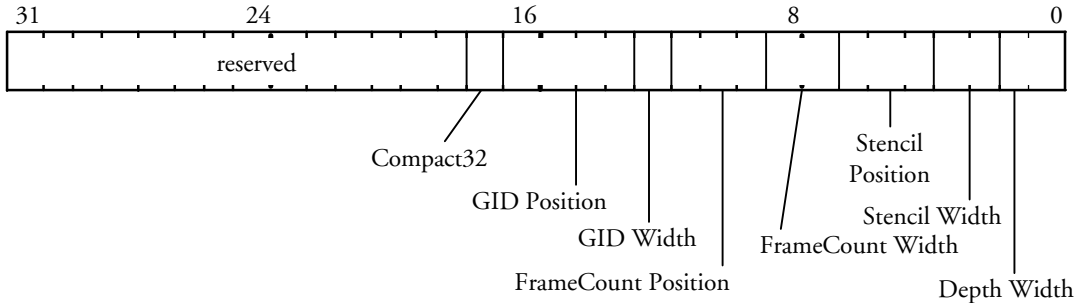


Figure 0.36 LBReadFormat / LBWriteFormat Register Layout

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.

5.11.5 LocalBuffer Example

The following is an example of a rendering operation with localbuffer read and write. GLINT is configured with a 32 bit localbuffer such that 24 bits are used for depth, 4 bits for stencil and 4 bits for fast clear with a screen size of 800x600.

```
// Set the localbuffer read and write formats to be
// 24bit depth, 4 stencil and 4 fast clear.

lbReadFormat.DepthWidth = 1          // 24 bit
lbReadFormat.StencilWidth = 1        // 4 bit
lbReadFormat.StencilPosition = 2     // bit 24
lbReadFormat.FrameCountWidth = 1     // 4 bit
lbReadFormat.FrameCountPosition = 3  // bit 28
lbReadFormat.GIDWidth = 0            // No GID
planes
lbReadFormat.GIDPosition = 0
lbReadFormat.Compact32 = GLINT_FALSE

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

// Set the localbuffer write mode
LBWriteMode (GLINT_ENABLE)

// Set the localbuffer read mode

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

lbReadMode.PP0 = 1  // 32 (<< 5)
lbReadMode.PP1 = 4  // 256 (<< 8)
```

```
lbReadMode.PP2 = 5  // 512 (<< 9)

lbReadMode.ReadSource = GLINT_DISABLE
lbReadMode.ReadDestination = GLINT_ENABLE
lbReadMode.FastClearEnable = GLINT_FALSE
lbReadMode.DataType = GLINT_LBDEFAULT
lbReadMode.WindowOrigin = as appropriate
lbReadMode.Patch = GLINT_FALSE
lbReadMode.ScanlineInterleave = 0
lbReadMode.PatchCode = 0
LBReadMode(lbReadMode)

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

5.12

Pixel Ownership Test Unit

Any fragment generated by the rasterizer may undergo a pixel ownership test. This test establishes the current fragment's write permission to the localbuffer and framebuffer.

5.12.1 Pixel Ownership Test

The ownership of a pixel is established by testing the GID of the current window against the GID of a fragment's destination in the GID buffer. If the test passes, then a write can take place, otherwise the write is discarded. The sense of the test can be set to one of: always pass, always fail, pass if equal, or pass if not equal. Pass if equal is the normal mode. In GLINT the GID planes, if present, are 4 bits deep allowing 16 possible Graphic ID's. The current GID is established by setting the **Window** register.

If the unit is disabled fragments pass through undisturbed.

5.12.2 Register

Pixel ownership is controlled by the **Window** register:

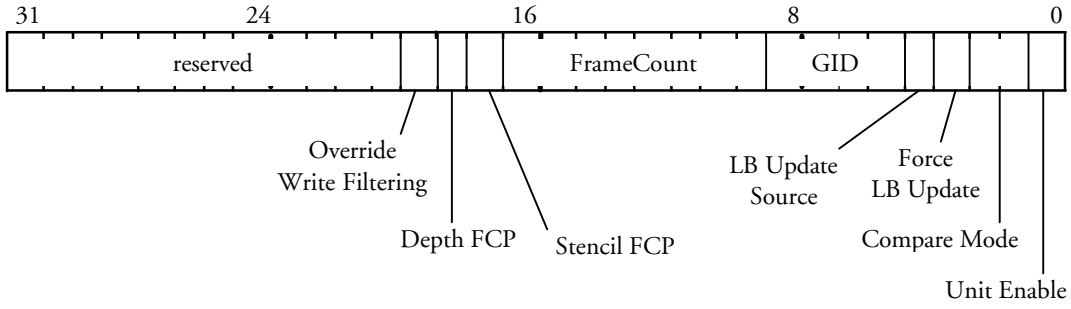


Figure 0.37 Window Register

The CompareMode field will generally be set to 'Pass if Equal' for GID testing, with the current GID in the appropriate field.

The ForceLBUpdate bit is used to allow all the fields in the localbuffer to be updated simultaneously, ForceLBUpdate overrides all GID, stencil and Depth testing.

DepthFCP and StencilFCP bits are used to control the fast clearing of the stencil and depth buffers. FrameCount is the frame counter value for current frame. This is described in more detail in section § 0.

LBUpdate source is used in conjunction with the ForceLBUpdate bit to select whether the source data comes from: the localbuffer, or values held in local registers (**Depth**, **Window**, **Stencil**). The combination of LBUpdateSource being set to LBSourceData, and the force LBUpdate bit being enabled is particularly useful when copying a window from one location on the screen to another. The combination of LBUpdateSource being set to Registers and the force LBUpdate

bit being enabled is particularly useful for initializing the contents of the various localbuffer fields in a window.

Normally GLINT detects the case where the data to be written to the localbuffer is the same as the data read from the localbuffer, and avoids performing the write. Setting the OverrideWriteFiltering bit prevents these writes from being filtered out. This is of value when the localbuffer read format is different from the localbuffer write format since the comparison is done on the internal data format.

5.12.3 Pixel Ownership Example

Setting the **Window** register for normal 3D operations with GID testing but no fast clear planes:

```
// Set Window modes.

window.UnitEnable = GLINT_ENABLE
window.GID = as appropriate
window.CompareMode = GLINT_PASS_IF_EQUAL
window.LBUpdate = GLINT_NO_FORCE
window.FCS = don't care
window.StencilFCP = GLINT_DISABLE
window.DepthFCP = GLINT_DISABLE
window.OverrideWriteFiltering = GLINT_DISABLE
Window(window)

// Note: Window base in framebuffer and localbuffer
// may need updating.
```

5.13 Stencil 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.

5.13.1 Stencil Test

This test only occurs if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership) 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

Mode	Comparison Function
4	Greater
5	Not Equal
6	Greater or Equal
7	Always

Table 0.16 Stencil Functions

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 § 0 to fully understand their relationship.

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

Table 0.17 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
Invert	5	~Source stencil

Table 0.18 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. It should normally be set to exclude the top four bits when the stencil width has been set to 4 bits in the **StencilMode** register.

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

StencilSource	Mode	Use
Test logic	0	This is the normal mode.
Stencil register	1	This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the **Stencil** register. This is used when a constant stencil value is needed, for example, when clearing the stencil buffer when fast clear planes are not available.
LBSourceData: (stencil value read from the localbuffer)	2	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	3	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 either from the localbuffer data, or the **FCStencil** register, depending on whether fast clear operations are enabled.

Table 0.19 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.

5.13.2 Registers

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

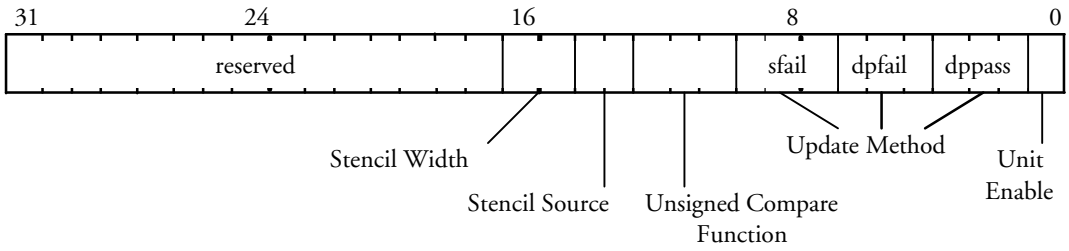


Figure 0.38 StencilMode Register

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

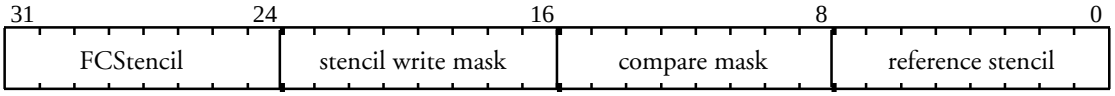


Figure 0.39 StencilData Register.

The stencil writemask is used to control which stencil planes are updated as a result of the test. The FCStencil field holds the stencil fast clear value.

The **Stencil** register holds an externally sourced stencil value. It is a 32bit register of which only the least significant 8bits are used. The unused most significant 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.

5.13.3 Stencil Example

This example sets the stencil unit to use a supplied reference value (0x80) and to test fragments to be LESS than this value. It also sets the stencil planes update function to be Increment if the test passes and the depth test passes (or is not enabled), otherwise it sets the update function to Keep.

```
// Set the localbuffer read and write modes
// See section §0

// Set the stencil modes

stencilMode.UnitEnable = GLINT_ENABLE
stencilMode.DPPass = GLINT_STENCIL_METHOD_INCREMENT
stencilMode.DPFail = GLINT_STENCIL_METHOD_KEEP
stencilMode.SFail = GLINT_STENCIL_METHOD_KEEP
```

```
stencilMode.CompareFunction =
GLINT_STENCIL_COMPARE_LESS
stencilMode.StencilSource = GLINT_SOURCE_TEST_LOGIC
stencilMode.Width = as appropriate
StencilMode(stencilMode)

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

stencilData.ReferenceStencil = 0x80
stencilData.CompareMask = 0xFF
stencilData.StencilWriteMask = as appropriate for
                          width of Stencil buffer
stencilData.FCStencil = don't care

StencilData(stencilData)

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

5.14

Depth Test Unit

The depth (Z) test, if enabled, compares a fragment's depth against the corresponding depth in the depth buffer. The result of the depth test can effect the updating of the stencil buffer if stencil testing is enabled.

5.14.1 Depth Test

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

Mode	Comparison Function
0	Never
1	Less
2	Equal
3	Less Than or Equal

Mode	Comparison Function
4	Greater
5	Not Equal
6	Greater Than or Equal
7	Always

Table 0.20 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.

StencilSource	Mode	Use
Test logic	0	This is the normal mode.
Stencil register	1	This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the **Stencil** register. This is used when a constant stencil value is needed, for example, when clearing the stencil buffer when fast clear planes are not available.
Source stencil value read from the localbuffer	2	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 either from the localbuffer data, or the **FCStencil** register, depending on whether fast clear operations are enabled.
LBSourceData: (stencil value read from the localbuffer)	3	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.

Table 0.21 Depth Sources.

When using the depth DDA for normal depth buffered rendering operations the depth values required are similar to those required for the color values in the color DDA unit:

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.

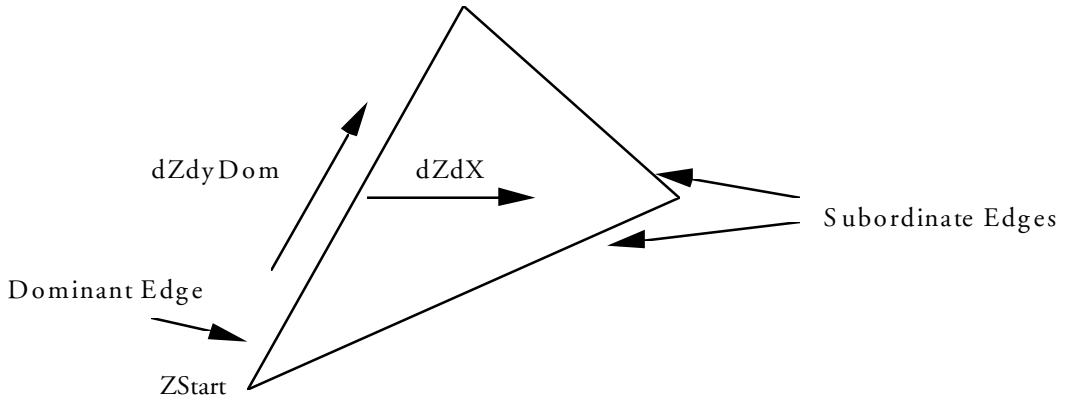


Figure 0.40 Depth Interpolation

The number format for the increment values is 2's complement fixed point integer: 32 bits integer and 16 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:

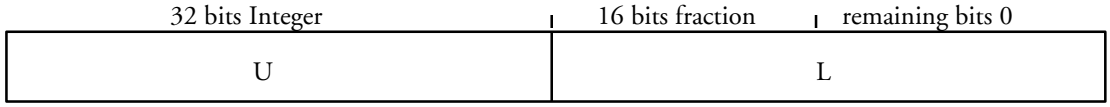


Figure 0.41 Depth Derivative Format.

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.

5.14.2 Registers

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

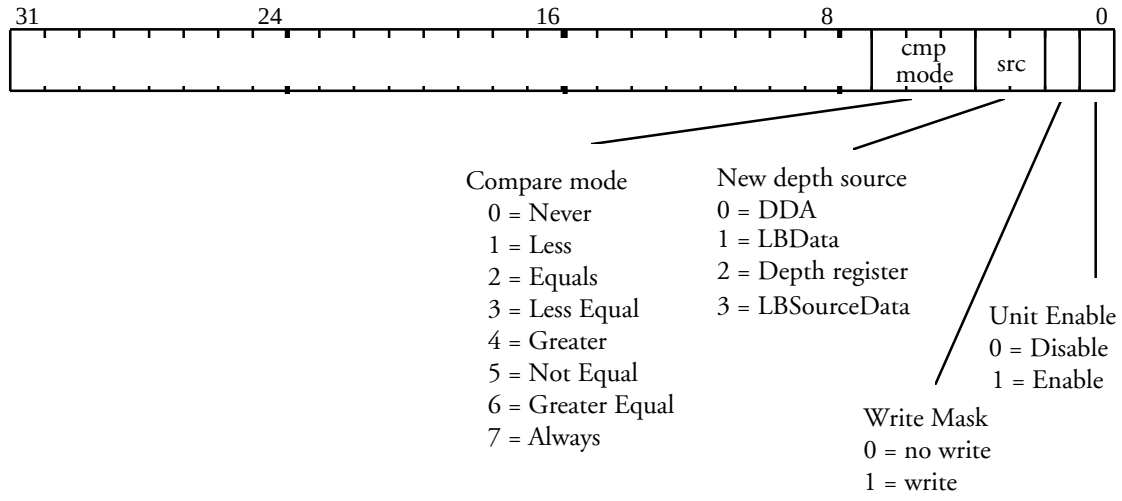


Figure 0.42 DepthMode Register.

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

The **FastClearDepth** register holds the 32 bit fast clear depth (FCDepth) value.

The **Depth** register holds an externally sourced 32 bit depth value. If the depth buffer holds less than 32bits then the user supplied depth value is right justified to the least significant end of the register. The unused most significant bits 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 0.22 Depth Interpolation Registers.

5.14.3 Depth Example

Rendering a Gouraud shaded depth buffered trapezoid.

```
// Set the localbuffer read and write modes
// See section §0

// Set the depth mode

depthMode.UnitEnable = GLINT_ENABLE
depthMode.WriteMask = 1
depthMode.NewDepthSource =
GLINT_NEW_DEPTH_SOURCE_DDA
depthMode.CompareMode =
GLINT_DEPTH_COMPARE_MODE_LESS

DepthMode(depthMode)

// Load the depth start values and deltas for
// 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()

// Enable unit in Gouraud shading mode
colorDDAMode.UnitEnable = GLINT_ENABLE
colorDDAMode.Shade = GLINT_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
dBYDom()

// Render primitive
```

5.15

Framebuffer Read/Write Unit

Before rendering can take place GLINT must be configured to perform the correct framebuffer read and write operations. Framebuffer read and write modes affect the operation of alpha blending, logic ops, writemasks, image upload/download operations and the updating of pixels in the framebuffer.

The framebuffer read and write units are set up in different ways depending on whether Span Operations are being used. Normally, span operations are used for 2D rendering in order to maximize memory bandwidth. Span operations allow multiple pixels to be read and processed in parallel. The following sections discuss the use of the framebuffer read and write units for both standard operation and span operations.

5.15.1 Standard Framebuffer Read Operation

The **FBReadMode** register allows GLINT 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.
- 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.
- Here set-up varies depending what functionality is required. If alpha blending, logic ops or software writemasks are used the framebuffer is read twice i.e. both the source and the destination. When alpha blending and logic ops are not needed, and hardware writemasks are used (or when the software writemask allows updating of all bits in a pixel) 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 that avoiding unnecessary additional reads will enhance performance.

The data read from the framebuffer may be tagged 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). Table 0.23 Framebuffer Read/Write Modes 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 blending.
Disabled	Disabled	Enabled	-	Image download.
Disabled	Enabled	Disabled	FBColor	Image upload.
Enabled	Disabled	Enabled	FBDefault	Image copy with hardware writemasks and no alpha blending orlogical operations
Disabled	Enabled	Enabled	FBDefault	Rendering using logical operations, software writemasks or blending.
Enabled	Enabled	Enabled	FBDefault	Image copy with software writemasks, alpha blending or logic ops.

Table 0.23 Framebuffer Read/Write Modes

5.15.2 Framebuffer Read Span Operations

As well as performing standard, single pixel at a time, read operations the framebuffer read unit can be used to process span operations. The simplest type of operation is where a span mask is presented to the read unit and the ReadSource bit is enabled. This will cause the unit to read a complete span of pixels from the framebuffer in a packed format. The data is always read as a set of 32 bit words. For example, at 8 bits per pixel, up to eight 32 bit words will be read per span; at 16 bits per pixel up to sixteen 32 bit words will be read. In all cases, up to 32 pixels worth of data is read per span. This allows maximum use of both memory and core bandwidth since multiple pixels are being processed.

Since a span mask may not necessarily have all its bits set to 1 (i.e. only a subset of pixels in the span need to be processed), it would be wasteful of memory bandwidth to always read the complete span. For example, at the right hand edge of a rectangle which is being copied, we want the read unit to only read up to the rightmost pixel but not beyond. Whether a 32 bit word is read depends on the corresponding bit values in the span mask. Since each bit in the mask represents a pixel, either 1, 2 or 4 bits will represent a 32 bit word for the depths 32, 16 and 8 bits respectively. If the group of bits representing a 32 bit word is non-zero then the corresponding 32 bits will be read from the framebuffer. Thus:

- at 32 bits per pixel, a single bit in the span mask corresponds to 32 bits in the framebuffer and 32 bit words will be read only at those locations where the corresponding bit in the span mask is a 1.
- at 16 bits per pixel, 2 bits in the span mask represent 32 bits in the framebuffer. A 32 bit word will be read only at those locations where the corresponding 2 span bits form a non-zero value.
- at 8 bits per pixel, a 32 bit word will be read only at those locations where the corresponding 4 span bits form a non-zero value.

The number of 32bit words read from the framebuffer is thus a function of the span mask and the number of bits per pixel, though this is not normally of interest to the programmer. However, the number of 32bit words becomes important for span operations where the data is downloaded from the host. For example, an image download operation using a span operation only requires those 32 bit words which contain required pixel data to be downloaded. Some examples of this are given later.

5.15.3 Merge-copy Span Operations

To understand the way in which the read units works we will examine the way in which a span operation with a logic op works. In particular we consider the case where both ReadSource and ReadDestination bits are set in the **FBReadMode** register. For example, this would be the case when copying data within the framebuffer with an xor logic op.

To perform this operation, the framebuffer read unit must read both a source span of data and a destination span of data. These spans must then be merged so that the data presented to the logic op unit consists of source and destination pairs. Since the logic op unit can combine up to 32 bits at a time, the data can be presented in the form of packed 32 bit words (at 8 bits per pixel this means that the logic op unit can work on 4 pixels at a time).

It would be wasteful of memory bandwidth to read 32 bits from the source followed by 32 bits from the destination. This would result in too many VRAM page breaks. So the read unit reads a complete source span and stores it internally in a data area known as the Pattern RAM. Then the destination span is read. As the destination span is read, it is merged with the saved source span data so that the data which the logical op unit sees comprises corresponding sections of source and destination data. The logic op unit can then combine this data and present a series of 32 bit results to the framebuffer write unit.

The Pattern RAM is so named because it can be used for pattern filling operations as well as a temporary store for source pixel data. This functionality is described below.

5.15.4 PatternRamMode register

To control the operation of the Pattern RAM the GLINT 500TX introduces a new register called the **PatternRamMode**. This register controls whether the Pattern RAM is enabled and how to interpret the contents. Its layout is described below.

The PatternEnable bit is used in conjunction with the ReadSource and ReadDestination bits from the **FBReadMode** register to specify different operations. The following table indicates the different operations that can be specified using these 3 mode bits.

PatternRam Enable	ReadSource Enable	ReadDestination Enable	Operation
0	0	0	No span reads are done, nor is data sourced from the pattern ram. This is the optimal mode for spanfills with constant color. If a span fill with variable color is required then the host must supply the data by writing to the **Color**, **FBData**, or **FBSourceData** registers.
0	0	1	Used for image upload or destination only logical ops. The pattern RAM will be overwritten and left containing indeterminate data by this operation.
0	1	0	Used for a straight blit operation, or source only logical op. The pattern RAM will be overwritten and left containing indeterminate data by this operation.
X	1	1	Span reads of source and destination regions for a ROP2 blit operation. The source span is read first and saved in the pattern ram. The destination span is then read and the data interleaved with data from the pattern ram while sending it to the logical op unit.
1	0	0	No span reads but the data is sourced from the pattern ram for latter use in a span write with variable color.
1	0	1	Span read but the data does not go into the pattern ram. The destination span data is interleaved with data from the pattern ram while sending to the logical op unit. The contents of the pattern ram are left intact so this can be reused without having to load the pattern in again.
1	1	0	The source span is read and saved in the pattern ram. No data is sent to the logical op unit, so this mode can be used to load the pattern ram from the framebuffer for later use as pattern data. Writes would normally be disabled, however if the write mode is set up for variable color then nothing will be drawn, as no color data is provided. A write with constant color will however go ahead if enabled.

5.15.5 Framebuffer Address Calculations

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

Bottom left origin

```
Dest addr = FBWindowBase - Y/S * W + X + FBPixelOffset
Srce addr = FBWindowBase - Y/S * W + X + FBPixelOffset +
                                         FBSourceOffset
```

Top left origin

```
Dest addr = FBWindowBase + Y/S * W + X + FBPixelOffset
Srce addr = FBWindowBase + Y/S * W + X + FBPixelOffset +
                                         FBSourceOffset
```

where:

`Dest addr`	is the address in the framebuffer which is written to if writes are enabled, and is also the address read when ReadDestination is enabled.
`Srce addr`	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,
`S`	is the scanline interval for multi-GLINT systems
`FBWindowBase`	holds the base address in the framebuffer of the current window.
`FBPixelOffset`	is normally zero except when multi-buffer writes are needed[1] 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.

[1]OpenGL, 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.

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 address calculations for span operations are the same as those for non-span operations.

5.15.6 Standard 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.

When uploading images the UpLoadData bit can be set to allow color formatting (which takes place in the Alpha Blend unit). See sections § 0 and § 0 for more details.

5.15.7 Span Operations and Framebuffer Write

If the SpanOperation bit in the **Render** command is zero then the write unit will use the span mask as a block fill mask and will fill the 32 pixel span with the current block color. If the SpanOperation bit is set to indicate variable color span filling, then either the **FBReadMode** register must be set to allow data to be read from the framebuffer or pattern RAM, or the host must provide the data (i.e. the SyncOnHostData bit in the **Render** command must be set). Failure to meet these conditions will NOT hang the chip but will lead to indeterminate results.

For block fills, the fill size is always 32 pixels regardless of the pixel depth or the type of the memory fitted (for the GLINT 300SX the block size depends on the memory configuration). The block fill color is 64 bits wide. This can be specified using the **FBBlockColor** register in which case this 32 bit value is replicated by the write unit to form 64 bits. Or the 64 bits can be explicitly specified using the **FBBlockColorL** (lower 32 bits) and **FBBlockColorU** (upper 32 bits) registers. At 8 and 16 bits per pixel depths it is up to the host software to replicate the block color to fill all 32 bits.

As with the standard mode of operation, in order to write data the WriteEnable bit must be set the **FBWriteMode** register.

To upload span data the UpLoadData bit should be set and the WriteEnable bit should be cleared. This allows image uploads data to be delivered to the host in a packed form. i.e at a pixel depth of 8 bits, 4 pixels per 32 bit word can be read back from the output FIFO; at a depth of 16 bits, 2 pixels per 32 bit word can be read back.

5.15.8 Using the Pattern RAM

As we have seen the pattern RAM can be used as an area in which the GLINT 500TX temporarily stores data read back from the framebuffer. It can also be used explicitly by the host software to perform pattern fills.

The Pattern RAM contains 128 bytes of storage arranged as 32 x 32 bit registers. This is enough to store a full span of data at a depth of 32 bits per pixel. It is also enough space to contain a full 8x8 pattern at both 8 and 16 bits per pixel. At a depth of 32 bits per pixel, half am 8x8 pattern can be stored. It is then possible to pattern fill a region in two passes. This data is stored in the same packed format as span data. The **PatternRamMode** register contains three fields, Xmask, Yshift and Ymask which allow the format of this data to be specified when the PatternEnable bit is set.

The start position in the pattern ram where a spans worth of pattern data is read from is initially determined from the Y coordinate associated with the span mask. The start address is given by:

```
Yoffset = (Y  << Yshift) & Ymask
```

where Yshift and Ymask are in the **PatternRamMode** register. Only the least significant 5 bits of the Y address are of interest.

The X offset is similarly given:

```
Xoffset = offset  & Xmask
```

where the offset is the bit, pair or nibble offset for 32, 16 and 8 bit pixels respectively. The pattern ram address is then:

```
pattern ram addr = Yoffset + Xoffset
```

For an 8x8 bit pattern the values of the X and Y shift and masks are as follows:

Pixel size	Y shift	Y mask	X mask	Notes
8	1	0x0F	0x01	Pattern fills lower half of the pattern ram
16	2	0x1f	0x03	Pattern fills all of ram.
32	2	0x18	0x07	Pattern ram contains even rows of pattern first and then odd rows on second pass.

The Yshift and mask values can be set up for different pattern sizes other than the 8x8 outlined here. The pattern must have a width of 2, 4, 8, 16 or 32, but can have any height. Also in a multi GLINT system there can be one pattern common to all GLINTs or the pattern can be divided up so each GLINT only gets the parts of the pattern it needs for its scanlines.

This pattern filling technique has an advantage over the alternative method of using the texture unit, in that 2 or 4 pixels are processed at a time whereas using

the texture method only one pixel at a time is dealt with. The pattern RAM can be used to supply constant color by setting the **PatternRamMode** register to 0x1 (i.e. the shift and mask are both set to zero), and then loading the color into the Pattern RAM at offset 0. At Pixel depths of 8bpp and 16bpp, the color needs to be replicated to fill all 32bits.

This is useful when performing logical ops where the source is a solid color, as it allows all the benefits of span processing to be achieved.

Note that the pattern is always aligned to the start of a span. This has two consequences:

- If the pattern needs to be aligned relative to some other reference point, then the pattern must be rotated (in X and Y) to give the correct alignment. For example if the pattern is relative to the window origin, and a small rectangle inside the window is to be filled to repair the window background pattern, then the pattern must be rotated.
- Filling trapezoidal areas (as opposed to rectangular areas) will cause the pattern to be sheared. In this case the only alternative is to use the texture unit.

5.15.9 Frame Blank Synchronization

The SuspendUnitFrameBlank command register may be used to stall the GLINT 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 GLINT registers and issue commands. GLINT will continue processing these as long as they do not involve writing to the framebuffer.

The SyncMode data field determines how the buffer swap is to be controlled. Options are:

- wait for vertical frame blank and update external video register
- update external video register immediately
- wait for vertical frame blank then update the VTGFrameRowAddr register immediately.

Note: This command register cannot be used in a multi-GLINT system.

5.15.10 Registers

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

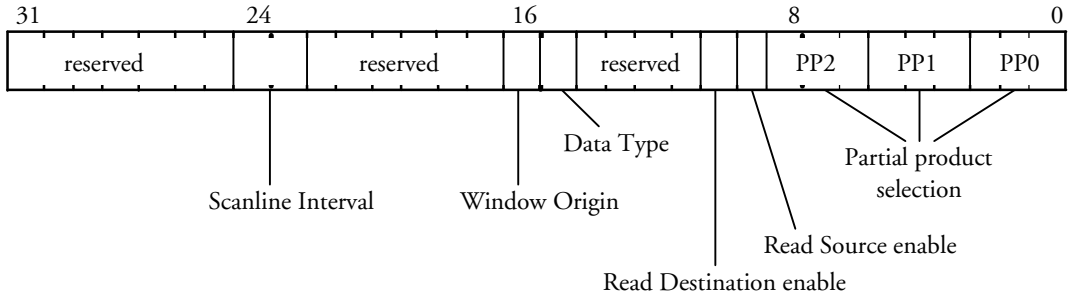


Figure 0.43 FBReadMode Register

See the chapter on Hardware Data Structures for more details of GID, Window Origin, and Partial Products.

The layout of the **PatternRamMode** register is as follows:

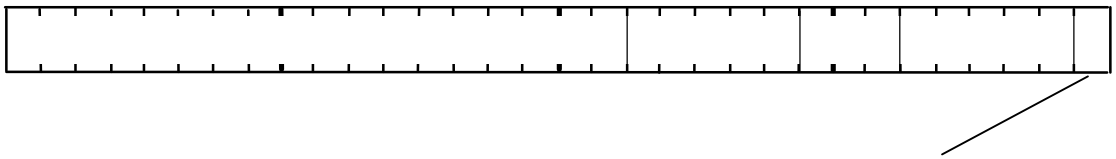


Figure 0.44 PatternRamMode Register

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:

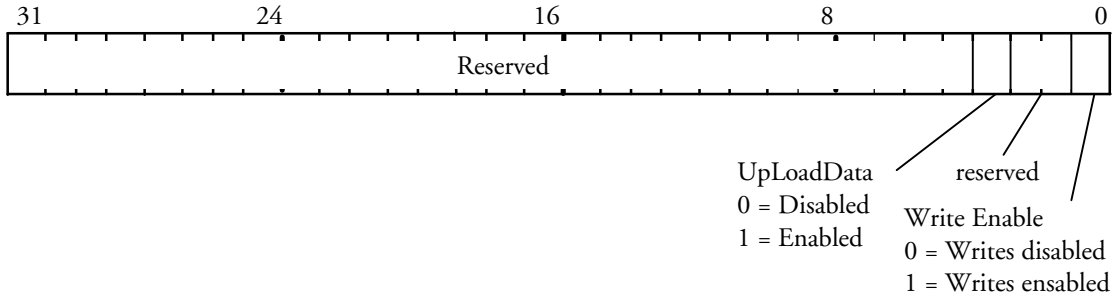


Figure 0.45 FBWriteMode Register

5.15.11 Simple Image Copy Example

This example copies a rectangular region of the framebuffer, without moving any data in the localbuffer. Pixel ownership tests are enabled. 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 = GLINT_ENABLE
fbReadMode.ReadDestination = GLINT_DISABLE
fbReadMode.DataType = GLINT_FBDEFAULT

FBReadMode(fbReadMode)        // Update register

// Now enable framebuffer write
fbWriteMode.WriteEnable = GLINT_ENABLE
FBWriteMode(fbWriteMode)      // Update register

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

// The localbuffer unit should be enabled to allow
// GID testing.
lbReadMode.ReadSource = GLINT_DISABLE
lbReadMode.ReadDestination = GLINT_ENABLE
lbReadMode.DataType = GLINT_LBDEFAULT
lbReadMode.WindowOrigin = as appropriate
LBReadMode(lbreadmode)
```

```
// Disable localbuffer writes
lbWriteMode.WriteEnable = GLINT_DISABLE
LBWriteMode(lbWriteMode)      // Update register

// Enable GID testing.
window.UnitEnable = GLINT_TRUE
window.CompareMode = GLINT_GID_COMPARE_EQUAL
window.ForceLBUpdate = GLINT_FALSE
window.LBUpdate = don't care
window.StencilFCP = GLINT_DISABLE
window.DepthFCP = GLINT_DISABLE
Window(window)

// All the units which could remove the fragment
// must be disabled (Stipple, Alpha, 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 = GLINT_FALSE
ColorDDAMode(colorDDAMode)

// Define the region we wish to copy to.
StartXDom(200<<16)
StartXSub(300<<16)
dXSub(0)
dXDom(0)
StartY(0)
dY(1<<16)
Count(100)

render.PrimitiveType = GLINT_TRAPEZOID

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

5.15.12 Span Operation Image Copy Example

This example copies a rectangular region of the framebuffer, using a span fill operation with an xor logic op. 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.

Note that this is almost identical to how one would copy pixels using the standard rasterization method. The **PatternRamMode** is explicitly disabled and the **Render** command specifies that variable color span filling is to be used. This code will perform an optimal copy at all pixel depths. Also, note that to turn this back into a simple screen-to-screen blt, the ReadDestination bit would be cleared and the logic op unit would be disabled.

```
// First set-up the framebuffer read mode
fbReadMode.ReadSource = GLINT_ENABLE
fbReadMode.ReadDestination = GLINT_ENABLE
fbReadMode.DataType = GLINT_FBDEFAULT

FBReadMode(fbReadMode)        // Update register

// Now enable framebuffer write
fbWriteMode.WriteEnable = GLINT_ENABLE
FBWriteMode(fbWriteMode)      // Update register

// Enable the logic op unit
logicop.UnitEnable = GLINT_TRUE
logicop.LogicOp = XOR
LogicOpMode(logicop)

// Disable the Pattern RAM register
patRamMode.PatternEnable = GLINT_DISABLE
PatternRamMode(patRamMode)      // Update register

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

// Define the region we wish to copy to.
StartXDom(200<<16)
StartXSub(300<<16)
dXSub(0)
dXDom(0)
StartY(0)
dY(1<<16)
Count(100)

render.PrimitiveType  = GLINT_TRAPEZOID
render.FastFillEnable = 1     // use span operation
render.SpanOperation  = 1     // variable color
Render(render)      // Start the rasterization
```

5.15.13 Span Operation Image Copy Example using Pattern RAM

This example assumes that the pixel depth has been set to 8 bits per pixel and uses the pattern RAM to perform a pattern fill using an 8x8 pattern.

```
// First set-up the framebuffer read mode
fbReadMode.ReadSource = GLINT_DISABLE
fbReadMode.ReadDestination = GLINT_DISABLE
fbReadMode.DataType = GLINT_FBDEFAULT

FBReadMode(fbReadMode)        // Update register

// Now enable framebuffer write
fbWriteMode.WriteEnable = GLINT_ENABLE
FBWriteMode(fbWriteMode)      // Update register

// Offsets. No Pixel offset
FBPixelOffset(0x0)

// download the data for the 8x8 pattern. Assume
that // the source data is contained in a byte
array called
// Pat8.
pat = Pat8;
for (i = 0; i < 8; i++)
{
     ulValue = pat[0]          |
               (pat[1] << 8)   |
               (pat[2] << 16)  |
               (pat[3] << 24]);
     PatternRamData[i](ulValue)
     pat += 4;
}

// Enable the Pattern RAM register for an 8x8
pattern
// of depth 8 bits per pixel.
patRamMode.PatternEnable = GLINT_ENABLE
patRamMode.Ymask  = 0x0F
patRamMode.Yshift = 1
patRamMode.Xmask  = 1
PatternRamMode(patRamMode)      // Update register


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

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

render.PrimitiveType  = GLINT_TRAPEZOID
render.FastFillEnable = 1     // use span operation
render.SpanOperation  = 1     // variable color

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

5.15.14 Span Operation Solid Fill Example

This example uses the pattern RAM to perform a solid color fill with logicop. The code works for all 3 color depths. Note that if a logicop is not required we could simply clear the ReadDestination bit and disable the logic op unit, but this would be far slower than using a constant color span fill. To use a span fill we would load the solid color into the **FBBlockColor** register, and clear the SpanOperation bit in the **Render** command.

```
// First set up the framebuffer read mode
fbReadMode.ReadSource = GLINT_DISABLE
fbReadMode.ReadDestination = GLINT_ENABLE
fbReadMode.DataType = GLINT_FBDEFAULT

FBReadMode(fbReadMode)        // Update register

// Now enable framebuffer write
fbWriteMode.WriteEnable = GLINT_ENABLE
FBWriteMode(fbWriteMode)      // Update register

// Offsets. No Pixel offset
FBPixelOffset(0x0)

// Enable the logic op unit for xor
logicop.UnitEnable = GLINT_TRUE
logicop.LogicOp = XOR
LogicOpMode(logicop)

// Enable the Pattern RAM register for a solid
color.
// i.e. Yshift and masks are set to zero.
patRamMode = 1;
PatternRamMode(patRamMode)      // Update register

// replicate the color if necessary and load into
// entry zero of the Pattern RAM.
ulColor = SolidColor;    // start at 32 bit depth
if (PixelDepth < 32)
{
     ulColor |= ulColor << 16;     // 16 bit depth
     if (PixelDepth < 16)
```

```
            ulColor |= ulColor << 8; // 8 bit depth
}
PatternRamData0(ulColor)


// Define the region we wish to write to
StartXDom(200<<16)
StartXSub(300<<16)
dXSub(0)
dXDom(0)
StartY(0)
dY(1<<16)
Count(100)

render.PrimitiveType  = GLINT_TRAPEZOID
render.FastFillEnable = 1     // use span operation
render.SpanOperation  = 1     // variable color

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

5.16 Alpha Blend Unit

Alpha blending combines a fragment's color with those of the corresponding pixel in the framebuffer. Blending is supported in RGBA and BGRA modes only.

5.16.1 OpenGL Alpha Blending

The alpha blend unit, combines the fragment's color value with that stored in the framebuffer, using the blend equation:

$$C_o = C_sS + C_dD$$

where: C_o is the output color, C_s is the source color (calculated internally) and C_d is the destination color read from the framebuffer.

The source blending function, S, and the destination blending function, D, are defined in the following tables. These tables assume a number range of 0.0 to 1.0.

Mode	Value	R	G	B	A
0	Zero	0	0	0	0
1	One	1	1	1	1
2	Destination Color	R_d	G_d	B_d	A_d
3	One Minus Destination Color	$1 - R_d$	$1 - G_d$	$1 - B_d$	$1 - A_d$
4	Source Alpha	A_s	A_s	A_s	A_s
5	One Minus Source Alpha*	$1 - A_s$	$1 - A_s$	$1 - A_s$	$1 - A_s$
6	Destination Alpha	A_d	A_d	A_d	A_d
7	One Minus Destination Alpha	$1 - A_d$	$1 - A_d$	$1 - A_d$	$1 - A_d$
8	Source Alpha Saturate	min of $(A_s, 1 - A_d)$	min of $(A_s, 1 - A_d)$	min of $(A_s, 1 - A_d)$	1

Table 0.24 Source Blending Functions

Mode	Value	R	G	B	A
0	Zero	0	0	0	0
1	One	1	1	1	1
2	Source Color	R_s	G_s	B_s	A_s
3	One Minus Source Color	$1 - R_s$	$1 - G_s$	$1 - B_s$	$1 - A_s$
4	Source Alpha	A_s	A_s	A_s	A_s
5	One Minus Source Alpha	$1 - A_s$	$1 - A_s$	$1 - A_s$	$1 - A_s$
6	Destination Alpha	A_d	A_d	A_d	A_d
7	One Minus Destination Alpha	$1 - A_d$	$1 - A_d$	$1 - A_d$	$1 - A_d$

Table 0.25 Destination Blending Functions

* One Minus Value is sometimes referred to as Inverse Value.

If the blend operations require any destination color components then the framebuffer read mode must be set appropriately, see section 5.14.

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.

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

5.16.2 QuickDraw3D Alpha Blending

When the AlphaType bit in the AlphaBlendMode register is set then QuickDraw3D style alpha blend equations are followed. The OpenGL equations above are used for the RGB components, but the alpha channel is treated differently and has a single source and destination blend functions as follows:

$C_a = 1- (1 - C_{sa}) * (1 - C_{da})$

The source and destination blend functions should be set as follows:

Name	Source Blend	Destination Blend
Pre-multiplied	ONE	ONE_MINUS_SRC_ALPHA
Interpolated	SRC_ALPHA	ONE_MINUS_SRC_ALPHA

Table 0.26 Source Blending Functions

The alpha calculation is the same for both modes.

5.16.3 Image Formatting

The alpha blend and color formatting units can be used to format image data into any of the supported GLINT framebuffer formats, though conversion between CI and RGB modes or vice versa are not supported.

Consider the case where the framebuffer is in RGBA 4:4:4:4 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 (described in section § 0)
- Enable framebuffer reads
- Disable framebuffer writes and set the UpLoadData bit in the **FBWriteMode** register
- Enable the alpha blend unit with a blend function which passes the destination value and ignores the source value (source blend Zero, destination blend One) and set the color mode to RGBA 4:4:4:4
- Set the color formatting unit to format the color of incoming fragments to an 8 bit RGB 3:3:2 framebuffer format.

The upload now proceeds as normal.

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

5.16.4 Registers

The unit is controlled by the **AlphaBlendMode** register:

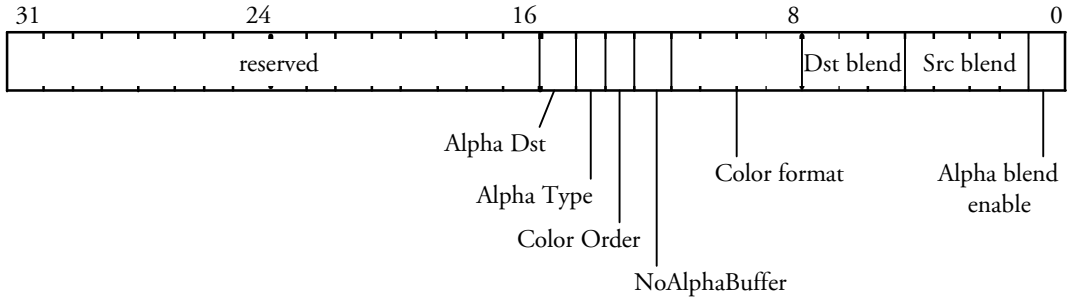


Figure 0.46 AlphaBlendMode Register

The color format and order is needed as the destination color is read from the framebuffer and needs to be converted into the internal GLINT representation, it should therefore be set as appropriate for the framebuffer.

			Internal Color Channel			
	Format	Name	R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4Front	4@0	4@8	4@16	4@24
	4	4:4:4:4Back	4@4	4@12	4@20	4@28
	5	3:3:2Front	3@0	3@3	2@6	255
	6	3:3:2Back	3@8	3@11	2@14	255
	7	1:2:1Front	1@0	2@1	1@3	255
	8	1:2:1Back	1@4	2@5	1@7	255
	13	5:5:5Back	5@16	5@21	5@26	255
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4Front	4@16	4@8	4@0	4@24
	4	4:4:4:4Back	4@20	4@12	4@4	4@28
	5	3:3:2Front	3@5	3@2	2@0	255
	6	3:3:2Back	3@13	3@10	2@8	255
	8	1:2:1Back	1@7	2@5	1@4	255
	7	1:2:1Front	1@3	2@1	1@0	255
	13	5:5:5Back	5@26	5@21	5@16	255
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Table 0.27 GLINT Color Modes

The framebuffer may be configured to be RGBA or Color Index (CI). Table 0.27 shows the full list of color modes supported by GLINT. The R, G, B and A columns show the width of each color component. n@m means that n bits starting at bit position m are read and scaled to fit the 8bit internal color channel format. The least significant bit position is zero. A numerical value (0 or 255) indicates the value substituted when the corresponding channel does not exist in the framebuffer.

For the Front and Back Modes the value to be blended is read only from the low bits or high bits respectively. This is to assist with color space double buffering.

When 5:5:5 bitplane double buffering is required, the 5:5:5:5 mode with the NoAlphaBuffer bit in the **AlphaBlendMode** register set, is used to select the front buffer. The back buffer is selected by using the 5:5:5Back mode, in which case the state of the NoAlphaBuffer bit is ignored.

5.16.5 Alpha Blend Example

This example sets the blend mode to allow antialiasing of polygons, i.e. source blend function = Source Alpha Saturate, destination blend function = One. These

blend functions are suitable for polygon antialiasing when polygons are drawn in front to back order, and the depth test is disabled.

```
// Enable framebuffer reads allow blend operation
// - Not Shown -

// Set the alpha mode.
alphaBlendMode.UnitEnable = GLINT_ENABLE
alphaBlendMode.SourceBlend =
GLINT_BLEND_SRC_ALPHA_SATURATE
alphaBlendMode.DestinationBlend = GLINT_BLEND_ONE
alphaBlendMode.ColorFormat = as appropriate

AlphaBlendMode(alphaBlendMode)     // Load register

// Enable antialias application and disable
// depth testing
// - Not Shown -

// Render polygons sorted front to back with
// Coverage Enable bit set in the Render command
// - Not Shown -
```

5.17

Color Format Unit

The color format unit converts from GLINT's internal color representation to a format suitable to be written into the framebuffer. This process may optionally include dithering of the color values for framebuffers with less than 8 bits width per color component. If the unit is disabled then the color is not modified in any way.

5.17.1 Color Formats

The framebuffer may be configured to be RGBA or Color Index (CI). Table 0.23 Framebuffer Read/Write Modes shows the full list of color modes supported by GLINT.

The R, G, B and A columns show the width of each color component. n@m means that the internal color channel is converted into an n bit number and stored in the framebuffer at bit position m. The least significant bit position is bit zero, and a dash in a column indicates that this component does not exist in the framebuffer for this mode.

For the Front and Back Modes the value is replicated into both buffers, and writemasks may be used to only update one buffer. Note the redundant duplication of the Front and Back modes is retained for symmetry with the Color format field of the **AlphaBlendMode** register.

The 5:5:5 Back format is designed to support multiple independent 15bpp double buffered windows, on systems which have a RAMDAC that can select the front and back buffer on a per pixel basis based on the top bit of the 32bit pixel stream. The front or back buffer may be selected for writing using writemasking.

In CI mode the index is replicated into all streams.

			Internal Color Channel			
	Format	Name	R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4 Front	4@0 4@4	4@8 4@12	4@16 4@20	4@24 4@28
	4	4:4:4:4 Back	4@0 4@4	4@8 4@12	4@16 4@20	4@24 4@28
	5	3:3:2 Front	3@0 3@8	3@3 3@11	2@6 2@14	-
	6	3:3:2 Back	3@0 3@8	3@3 3@11	2@6 2@14	-
	7	1:2:1 Front	1@0 1@4	2@1 2@5	1@3 1@7	-
	8	1:2:1 Back	1@0 1@4	2@1 2@5	1@3 1@7	-
	13	5:5:5 Back	5@0 5@16	5@5 5@21	5@10 5@26	-
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4 Front	4@16 4@20	4@8 4@12	4@0 4@4	4@24 4@28
	4	4:4:4:4 Back	4@16 4@20	4@8 4@12	4@0 4@4	4@24 4@28
	5	3:3:2 Front	3@5 3@13	3@2 3@10	2@0 2@8	-
	6	3:3:2 Back	3@5 3@13	3@2 3@10	2@0 2@8	-
	7	1:2:1 Front	1@3 1@7	2@1 2@5	1@0 1@4	-
	8	1:2:1 Back	1@3 1@7	2@1 2@5	1@0 1@4	-
	13	5:5:5 Back	5@10 5@26	5@5 5@21	5@0 5@16	-
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Table 0.28 GLINT Color Modes

5.17.2 Color Dithering

GLINT uses an ordered dither algorithm to implement color dithering. The

following table shows the exact type of dithering used when dither is enabled. The type of dithering depends on the width of individual color components:

Component Width	Type of Dithering
8	No Dithering
5	2x2 Ordered Dither
4	4x4 Ordered Dither
3	4x4 Ordered Dither
2	4x4 Ordered Dither
1	4x4 Ordered Dither

Table 0.29 Dither Methods

GLINT's ordered dither matrices are shown below:

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

0	2
3	1

Table 0.30 Ordered Dither Matrices, 4x4 and 2x2.

If the color formatting unit is disabled, the color components RGBA are not modified and will be truncated, or rounded, under the control of the RoundingMode bit in the **DitherMode** register, when placed in the framebuffer (assuming that the framebuffer width is less than 8bits per component). In CI mode the value is rounded to the nearest integer. In both cases the result is clamped to a maximum value to prevent overflow.

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

The alpha channel processing is qualified by the AlphaDither control bit. When cleared the alpha channel is processed in the same way as the color channels, as dictated by the DitherEnable bit. When the AlphaDither bit is set however, the alpha channel is not dithered, but is processed according to the state of the RoundingMode bit. The ability to disable dithering on the alpha channel is useful when using the alpha buffer to hold coverage information during antialiasing. In this situation dithering adds noise to the coverage value, leading to artifacts where a pixel which should be fully covered is reported as not fully covered.

See The OpenGL Reference Manual and The OpenGL Programming Guide from Addison-Wesley for more details on dithering.

5.17.3 Registers

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

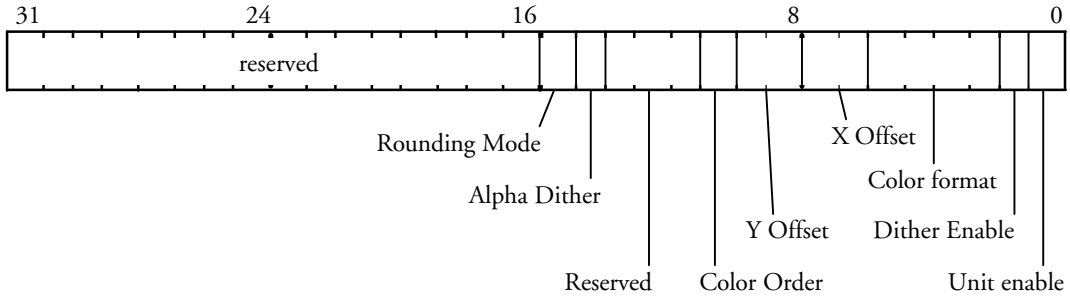


Figure 0.47 DitherMode Register

5.17.4 Dither Example

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

```
// 332 Dithering

ditherMode.UnitEnable = GLINT_TRUE
ditherMode.DitherEnable = GLINT_TRUE
ditherMode.ColorMode = GLINT_COLOR_FORMAT_RGB_332

DitherMode(ditherMode)   // Load register
```

5.17.5 3:3:2 Color Format Example

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

```
// 332 No Dither

ditherMode.UnitEnable = GLINT_TRUE
ditherMode.DitherEnable = GLINT_FALSE
ditherMode.ColorMode = GLINT_COLOR_FORMAT_RGB_332

DitherMode(ditherMode)   // Load register
```

5.17.6 8:8:8:8 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 = GLINT_TRUE
ditherMode.DitherEnable = GLINT_FALSE
ditherMode.ColorMode = GLINT_COLOR_FORMAT_RGBA_8888
```

```
DitherMode(ditherMode)   // Load register
```

The same can be achieved by disabling the color formatting unit as 8 bit components are not dithered:

```
// 8888 No dither
ditherMode.UnitEnable = GLINT_FALSE

DitherMode(ditherMode)   // Load register
```

5.17.7 Color Index Format Example

To set the framebuffer to 4 bit Color Index and enable dithering:

```
// 4 bit CI with dithering

ditherMode.UnitEnable = GLINT_TRUE
ditherMode.DitherEnable = GLINT_TRUE
ditherMode.ColorMode = GLINT_COLOR_FORMAT_CI_4

DitherMode(ditherMode)   // Load register
```

5.18 Logical Op Unit

The logical op unit performs two functions; logic ops between the fragment color (source color) and a value from the framebuffer (destination color), and, optionally control of a special GLINT mode which allows high performance flat shaded rendering.

5.18.1 High Speed Flat Shaded Rendering

On the GLINT 300SX a special rendering mode is available which allows high speed rendering of unshaded images. This mode is still supported on the GLINT 500TX, and is detailed below for completeness, but span processing should be used on the GLINT 500TX in preference to this technique.

To use the mode the following constraints must be satisfied:

- Flat shaded aliased primitive
- No dithering required or logical ops
- No stencil, depth or GID 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 high speed rendering can be achieved by setting the **FBWriteData** register to hold the framebuffer data (formatted appropriately for the framebuffer in use) 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.

5.18.2 Logical Operations

The logical operations supported by GLINT are:

Mode	Name	Operation	Mode	Name	Operation
0	Clear	0	8	Nor	~(S | D)
1	And	S & D	9	Equivalent	~(S ^ D)
2	And Reverse	S & ~D	10	Invert	~D
3	Copy	S	11	Or Reverse	S | ~D
4	And Inverted	~S & D	12	Copy Invert	~S
5	Noop	D	13	Or Invert	~S | D
6	Xor	S ^ D	14	Nand	~(S & D)
7	Or	S | D	15	Set	1

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

Table 0.31 Logical Operations

For correct operation of this unit in a mode which takes the destination color, GLINT must be configured to allow reads from the framebuffer using the **FBReadMode** register. See section § 0 for more details.

GLINT makes no distinction between RGBA and CI modes when performing logical operations.

5.18.3 Registers

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

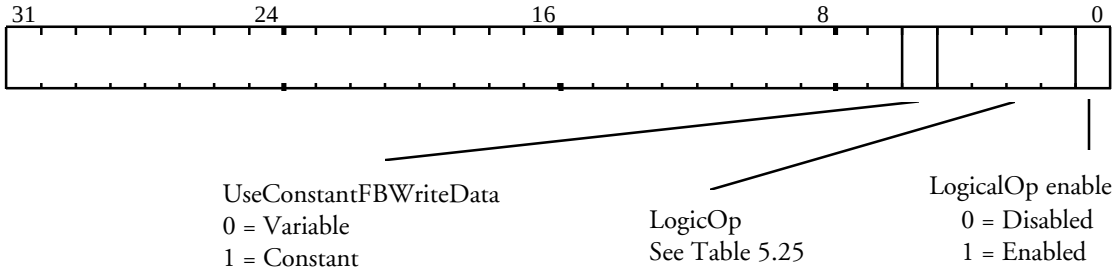


Figure 0.48 LogicalOpMode Register

5.18.4 XOR Example

To set the logical operation to XOR.

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

logicalOpMode.UnitEnable = GLINT_ENABLE
```

```
logicalOpMode.LogicalOp = GLINT_LOGICOP_XOR

LogicalOpMode(logicalOpMode)       // Load register
```

5.18.5 Logical Op and 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 = GLINT_ENABLE
ditherMode.DitherEnable = GLINT_ENABLE
ditherMode.ColorMode = GLINT_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)             // Load register

logicalOpMode.UnitEnable = GLINT_ENABLE
logicalOpMode.LogicalOp = GLINT_LOGICOP_COPY
LogicalOpMode(logicalOpMode)       // Load register

FBSoftwareWriteMask(0xFFFFFFE3)
```

5.19

Framebuffer Writemasks

Two types of framebuffer writemasking are supported by GLINT, software and hardware. Software writemasking requires a read from the framebuffer to combine the fragment color with the framebuffer color, before checking the bits in the mask to see which planes are writeable. Hardware writemasking is implemented using VRAM writemasks and no framebuffer read is required.

5.19.1 Software Writemasks

Software writemasking 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 disables writing to that bit. 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. Note that the ReadDestination bit must be enabled in the **FBReadMode** register when using software writemasks, in which some of the bits are zero.

See the Framebuffer Read/Write section for details of how to enable/disable framebuffer reads.

5.19.2 Hardware Writemasks

Hardware writemasks, if present, are controlled using the **FBHardwareWriteMask** register. If the framebuffer supports 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 replicated to all 4 bytes of the **FBHardwareWriteMask** register. If the framebuffer is in 16 bit packed mode then the 16 bit hardware writemask must be replicated to both halves of the FBHardwareWriteMask register.

See the GLINT Hardware Reference Manual for more details of framebuffer hardware writemasks.

5.19.3 Registers

Both registers **FBHardwareWriteMask** and **FBSoftwareWriteMask** are 32 bit registers in which each bit represents a bit in the framebuffer.

5.19.4 Software Writemask Example

Using software writemasks:

```
// Enable framebuffer reads (not shown)
// Set the writemask
FBSoftwareWriteMask(0x0F0F0F0F)
```

See §0 for another example

5.19.5 Hardware Writemask Example

Using hardware writemasks when neither logic ops, nor alpha blending are enabled:

```
// Disable framebuffer reads (not shown)
// Set the writemasks

FBSoftwareWriteMask(0xFFFFFFFF) // 'Disable'
FBHardwareWriteMask(0xF0F0F0F0) // Actual writemask
```

5.20 Host Out Unit

The Host Out Unit controls which data is available at the output FIFO, and gathers statistics about the rendering operations (picking and extent testing) and the synchronization of GLINT via the **Sync** register.

5.20.1 Filtering

Filtering controls the data available at the output FIFO. There are a number of 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.
- **Synchronization:** A single register, **Sync**, which is used to synchronize GLINT and flush the graphics pipeline.
- **Statistics:** The registers associated with extent and picking.

The filtering is controlled by the **FilterMode** register which is split into 2 bit fields for each category. The 2 bit field selects whether the register tag and/or register data, are passed to the output FIFO. The format of the **FilterMode** register is shown in Table 0.32.

Register Category	Tag Control Bit	Data Control Bit	Description
Diagnostic Use Only	0	1	
Diagnostic Use Only	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**.
Diagnostic Use Only	14	15	

Table 0.32 Filter Modes

Note, the filter unit must be set appropriately before any synchronization can take place, see §0.

5.20.2 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. Spans are handled by GLINT in a fully consistent way for picking and extent checking.

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:

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

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

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 GLINT

3) Draw the primitives.

4) Send a **PickResult** command.

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

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:

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

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

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).

5.20.3 Synchronization

The **Sync** register is filtered and written to the output FIFO in a similar fashion to the other registers. If an interrupt is required to be generated then the most significant bit of the **Sync** command register must be set, and the filtering must be set up to write something into the FIFO. If nothing is written to the FIFO (because of the **FilterMode**) then no interrupt will be generated. The actual interrupt will not be generated until the **Sync** data or tag has passed through, and is on the output of the FIFO, so as to allow low level resynchronization between the 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 data field are free and can be used by the host to identify the reason for the **Sync**.

5.20.4 Registers

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

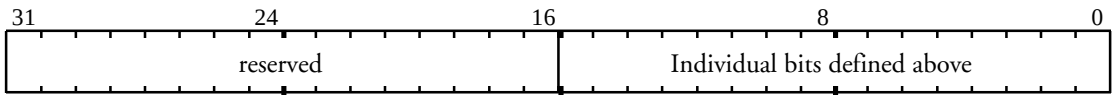


Figure 0.49 FilterMode Register

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

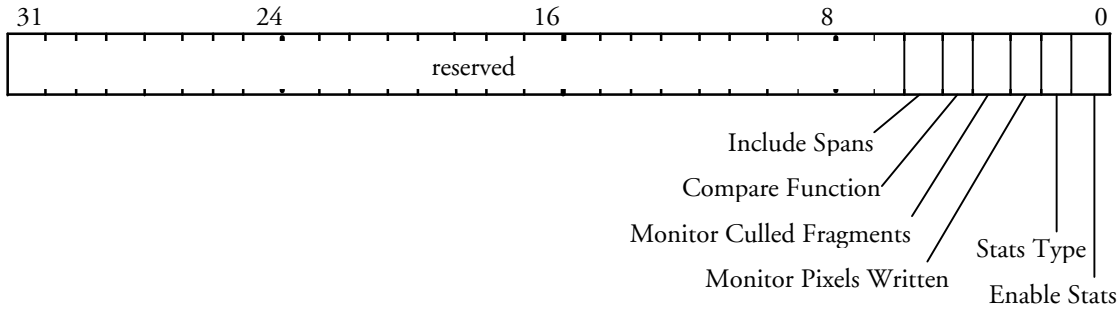


Figure 0.50 StatisticMode Register

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, X in the least significant end of the word.

PickResult is used to read the results of picking, the pick flag is placed in the least significant bit of the 32 bit register. **ResetPickResult** is used to clear the picking flag, the data field is not used.

The **Sync** register is 32 bits with the most significant bit set to indicate an interrupt is to be generated, bits 0-30 are available for the user.

5.20.5 Filter Mode Example

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

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

5.20.6 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)
```

5.20.7 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 GLINT

This section illustrates how to initialize GLINT 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 PCI bus, memory set-up and video output. This information typically is only initialized once following reset.
- Window initialization covers the base address of the current rendering window and its color format. This must be initialized at reset, but will need updating each time GLINT starts drawing to a new window.
- Application initialization covers state that is typically dynamic, enabling & 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 GLINT consult the relevant sections of the Graphics Programming chapter. Examples are given which make use of the pseudocode conventions given in Appendix B.

Note that in general the graphics registers (those listed in Appendix A, as opposed to those documented in the 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 bus

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. However, the subject of PCI bus initialization is beyond the scope of this document. For more details refer to the GLINT 500TX Hardware Reference Manual, and the PCI Local Bus Specification Rev2.1.

6.2.2 Memory Configuration

A part of the GLINT initialization is to specify some of the hardware parameters that define the characteristics of the memory attached to GLINT. In most board designs, these registers are initialized at reset by a set of resistors connected to the chip.

If the GLINT board design does not include these resistors, then these registers will have to be set by software as outlined below.

The content of these registers is dependent upon the board design, and the memory chips that have been used. It is necessary to consult the GLINT Hardware Reference Manual and the board design documentation, to find the correct values for any particular system configuration.

The *Reset* register is initialized automatically at reset as detailed in the GLINT Hardware Reference Manual.

The memory characteristics for the framebuffer and localbuffer are set through three registers. These characteristics include details about the number of banks, page sizes and address strobe requirements. For example, the following will initialize GLINT to operate in a system where the localbuffer comprises 1 bank of memory, with a page size of 2k, and the localbuffer and framebuffer have the RAS/CAS timing values indicated:

```
lbMemoryControl.NBanks =    GLINT_LB_NBANKS_1
lbMemoryControl.PageSize =  GLINT_LB_PAGE_SIZE_2048
lbMemoryControl.RASLow =    GLINT_LB_RAS_LOW_3
lbMemoryControl.RASPrecharge =
                            GLINT_LB_RAS_PRECHARGE_2
lbMemoryControl.CASLow =    GLINT_LB_CAS_LOW_1
lbMemoryControl.PageModeEnable =  GLINT_ENABLE
lbMemoryControl.RefreshCount =    0x20
LBMemoryControl(lbMemoryControl)

fbMemoryControl.RASLow =    GLINT_FB_RAS_LOW_2
fbMemoryControl.RASPrecharge =
                            GLINT_FB_RAS_PRECHARGE_2
fbMemoryControl.CASLow =    GLINT_FB_CAS_LOW_1
fbMemoryControl.PageModeEnable =  GLINT_ENABLE
fbMemoryControl.RefreshCount =    0x20
FBMemoryControl(fbMemoryControl)
```

The refresh count multiplied by 16 represents the number of MClk clock cycles between the start of each refresh. Setting a RefreshCount of 0x20, will cause a refresh every 512 clock cycles.

The *FBModeSel* register contains details of the capabilities & characteristics of the framebuffer and might typically be initialized as follows:

```
// We can use some of the fast VRAM modes
fbModeSel.FastModeEnable =    GLINT_ENABLE

// Buffer is not shared.
fbModeSel.SharedMode = GLINT_FB_SHARED_DISABLED

// Enable VRAM transfer cycles
fbModeSel.XFerEnable =        GLINT_ENABLE

// Select an external timing generator.
fbModeSel.ExtVTG =            GLINT_FB_EXT_VTG
FBModeSel(fbModeSel)
```

6.2.3 Internal Video Timing Registers

If the board design uses the on chip video timing generator, then the video timing registers must be initialized appropriately. For details refer to the GLINT Hardware Reference Manual.

6.2.4 Framebuffer Depth

The size of each pixel to be written into the framebuffer needs to be set up using the PixelSize register. To initialize the pixel size to 32 bits deep the PixelSize register would be loaded as follows.

```
pixelsize = GLINT_FB_PACK_32
PixelSize(pixelsize)
```

6.2.5 Screen Width

The width of the screen is initialized by setting the three partial products fields in the **FBReadMode** and **LBReadMode** 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. Some of the more common values are shown in the table below. A full list is given in Appendix C.

Screen width	PP0	PP1	PP2
640	5	3	0
1024	6	0	0
1152	6	3	0
1280	6	4	0
1600	6	5	2

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

```
fbReadMode.PP0 = 6
fbReadMode.PP1 = 0
fbReadMode.PP2 = 0
FBReadMode(fbReadMode)

lbReadMode.PP0 = 6
lbReadMode.PP1 = 0
lbReadMode.PP2 = 0
LBReadMode(lbReadMode)
```

6.2.6 Screen Clipping Region

GLINT 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 **FBPixelOffset**, **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 = GLINT_ENABLE
scissorMode.UserScissorEnable =   GLINT_DISABLE
ScissorMode(ScissorMode)
```

6.2.7 Localbuffer and Framebuffer Configuration

GLINT supports a range of localbuffer configurations. During initialization, fields in the **LBWriteFormat** and **LBReadFormat** registers should be set to appropriate values which reflect the depth of memory on the board design, and the initial manner in which it is to be used. For example if the hardware is designed to support a 32 bit localbuffer, and initially this is to be divided into a 24 bit Depth buffer, 4 bit stencil, no GID planes and 4 FrameCount planes, then the registers must be set as follows:

```
lbReadFormat.DepthWidth = 1   // 24 bit depth buffer
lbReadFormat.StencilPosition  = 2  // Stencil @ 16
lbReadFormat.StencilWidth  = 1     // 4 bit stencil
lbReadFormat.GIDWidth = 0          // No GID planes
lbReadFormat.GIDPosition = 1       // Does not matter
lbReadFormat.FrameCountPosition=3  // FrameCount @ 20
lbReadFormat.FrameCountWidth = 1
                           // 4 FrameCount plnes
LBReadFormat(lbReadFormat)

lbWriteFormat.DepthWidth = 1  // 24 bit depth buffer
lbWriteFormat.StencilPosition = 2  // Stencil @ 16
lbWriteFormat.StencilWidth  = 1    // 4 bit stencil
lbWriteFormat.GIDWidth = 0         // No GID planes
lbWriteFormat.GIDPosition = 1      // Does not matter
lbWriteFormat.FrameCountPosition=3 // FrameCount @20
lbWriteFormat.FrameCountWidth = 1
                           // 4 FrameCount plnes
LBWriteMode(lbWriteFormat)
```

Note that within the limits of the memory depth that is physically available, it is possible to dynamically change the allocation of the bits, for instance on a per window basis.

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

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

lbReadMode.DataType =   GLINT_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)
```

6.2.8 Host Out Unit

Under some circumstances it is necessary to synchronize with GLINT. This is controlled through the **Sync** command which causes data to be written to the host out FIFO once all processing has completed. The host out FIFO should normally be initialized so as to pass these pieces of 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 =            GLINT_NULL
filterMode.Stencil =          GLINT_NULL
filterMode.Color =            GLINT_NULL
FilterMode.Synchronization =  GLINT_FILTER_TAG_AND_DATA
                              // Allow syncs through
filterMode.Statistics =       GLINT_NULL
FilterMode(filterMode)
```

6.2.9 Disabling Specialized Modes

The Graphic ID, and FrameCount planes, should normally be initially disabled. Refer to the Graphics Programming chapter for more details on their use.

```
window.DepthFCP =       GLINT_DISABLE
window.StencilFCP =     GLINT_DISABLE
window.FrameCount =     0xFF
window.GID =            GLINT_NULL
window.LBUpdateSource = GLINT_GID_LBUPDATE_REGISTER
window.ForceLBUpdate =  GLINT_FALSE
window.CompareMode =    GLINT_GID_ALWAYS_PASS
window.UnitEnable =     GLINT_DISABLE

Window(window)
```

6.3

Window Initialization

GLINT 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 formatting 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 Error! Reference source not found. .

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

```
ditherMode.ColorFormat =
                GLINT_COLOR_FORMAT_RGB_332_FRONT
DitherMode(ditherMode)

alphaBlendMode.ColorFormat =
                GLINT_COLOR_FORMAT_RGB_332_FRONT
AlphaBlendMode(alphaBlendMode)
```

To enable dithering use the following:

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

Note that the color formatting 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.

GLINT 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 OpenGL it would typically be bottom left, whereas for an X windows implementation it would be Top Left. Thus for OpenGL set:

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

lbReadMode.WindowOrigin =
                    GLINT_BOTTOM_LEFT_WINDOW_ORIGIN
LBReadMode(lbReadMode)
```

The window origin for clipping 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). 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(GLINT_ALL_WRITEMASKS_SET)
FBHardwareWriteMask(GLINT_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, stencil, or pixel ownership testing, then reading and writing to the localbuffer may be disabled. The following example initializes the buffers to allow Z buffering and alpha blending:

```
fbWriteMode.UnitEnable =           GLINT_ENABLE
FBWriteMode(fbWriteMode)
lbWriteMode.UnitEnable =           GLINT_ENABLE
LBWriteMode(lbWriteMode)

lbReadMode.ReadSourceEnable =      GLINT_DISABLE
lbReadMode.ReadDestinationEnable = GLINT_ENABLE
LBReadMode(lbReadMode)

fbReadMode.ReadSourceEnable =      GLINT_DISABLE
fbReadMode.ReadDestinationEnable = GLINT_ENABLE
FBReadMode(fbReadMode)
```

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

6.4

Application Initialization

While an application is running it may dynamically use features of GLINT 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    =    GLINT_DISABLE
AreaStippleMode(areaStippleMode)

lineStippleMode.UnitEnable    =    GLINT_DISABLE
LineStippleMode(lineStippleMode);

routerMode.UnitEnable         =    GLINT_DISABLE
RouterMode(routerMode)

window.UnitEnable             =    GLINT_DISABLE
Window(window)

stencilMode.UnitEnable        =    GLINT_DISABLE
StencilMode(stencilMode)

depthMode.UnitEnable          =    GLINT_DISABLE
DepthMode(depthMode)

colorDDAMode.UnitEnable       =    GLINT_DISABLE
ColorDDAMode(colorDDAMode)

textureAddressMode.UnitEnable =    GLINT_DISABLE
TextureAddressMode(textureAddressMode)

textureReadMode.UnitEnable    =    GLINT_DISABLE
textureReadMode(textureReadMode)

textureColorMode.UnitEnable   =    GLINT_DISABLE
TextureColorMode(textureColorMode)

fogMode.UnitEnable            =    GLINT_DISABLE
FogMode(fogMode)

antialiasMode.AntialiasEnable =    GLINT_DISABLE
AntialiasMode(antialiasMode)

alphaTestMode.UnitEnable      =    GLINT_DISABLE
AlphaTestMode(alphaTestMode)

alphaBlendMode.UnitEnable     =    GLINT_DISABLE
AlphaBlendMode(alphaBlendMode)

logicalOpMode.UnitEnable      =    GLINT_DISABLE
LogicalOpMode(logicalOpMode)
```

7. Multi-GLINT Systems

7.1 Overview

This chapter will examine some of the issues and methods that a multi-GLINT 500TX system can employ.

To gain benefit from running multiple GLINTs in parallel the system must be rendering bound. If the system is host or geometry bound then adding in more GLINTs will not improve the system performance.

There are many possible parallel paradigms which can be adopted. The major ones are tabulated below, but this chapter will concentrate on the Scanline Interleaved method. The table is not exhaustive and an interested reader is directed to the book by Whitman [1].

The Scanline Interleaved paradigm is a good all-round method, ideally suited to the simulation market. Boards using this paradigm support all the normal GLINT rendering operations, and operations such as antialiasing, line stipples, image download and bitmasks which typically present problems in a parallel system are fully supported in the GLINT hardware. The only limitation inherent in such architectures, is that Block copies where source and destination are not a multiple of the Scanline Interleave factor apart, are not directly hardware accelerated by GLINT. Such copies must be implemented using a combination of bypass, image upload and image download operations. For the simulation market this is not an issue as on screen copies are rare, however, for desktop machines running in a windowing environment where copies are common a full solution is provided in the dual-GLINT case, by using the shared framebuffer facility. See below for further details.

[1] Multiprocessor Methods For Computer Graphics Rendering by Scott Whitman, ISBN 0-86720-229-7

Paradigm	Description	Advantage	Disadvantage
Frame Interleaving	Frame Interleaving is where a GLINT works on frame n, the next GLINT works on frame n+1, etc.. Each GLINT does everything for its own frame and the video is sourced from each GLINT's framebuffer in turn.	Simple. Good load balancing. Can be implemented with any GLINT product . Doesn't need a broadcast mechanism.	Increase in transport delay. Complete system is duplicated.
Frame Merging or Primitive Parallelism	Frame merging is a similar technique to frame interleaving where each GLINT has a full localbuffer and framebuffer. In this case the primitives are distributed amongst the GLINTs and the resultant partial images composited using the depth information to control which fragment from the multiple buffers is displayed in each pixel position.	Conceptually simple. Average load balancing.	Needs video rate composition using the depth value to select which pixel to display. The localbuffer structure does not readily support this dual port access. Alpha blending and antialiasing are problematical. Block copies don't work.
Screen subdivision (regions)	Here the screen is divided up into large contiguous regions and a GLINT looks after each region. Primitives which overlap between regions are sent to both regions and scissor clipping used. Primitives contained wholly in one region are ideally just sent to the one GLINT. The number of regions and the horizontal and/or vertical division of the screen can be chosen as appropriate, however horizontal bands are usually easier for the video hardware to cope with.	Conceptually simple. Can be implemented with any GLINT product.	Poor load balancing unless regions allocated dynamically. Block copies fail when cross boundaries. Broadcast can not be used effectively.
Screen subdivision (interleaved scanlines)	The interleave factor is every other n^{th} scanline where *n* is the number of GLINTs. Nearly all primitives will overlap multiple scanlines so are ideally broadcast to all GLINTs. Each GLINT only needs enough localbuffer and frame buffer to cover the pixels in its own region, however texture maps are replicated in full.	Load balancing is excellent. Entire Depth and Color buffers not duplicated for each GLINT.	Block copies in Y do not work unless the displacement is a multiple of interleave factor. May be solved using shared framebuffer.

7.2 Setting up the Graphics Processor

In an Interleaved Scanline multi-GLINT system all GLINTs will receive the same command and parameter data with the exception of the one parameter needed to specify which scanline a particular GLINT is owning. This is important as it means that the host does not have the additional burden of calculating different parameters for each GLINT. Ideally the command and parameter data is broadcast to all GLINTs to

economize on system bandwidth and this is trivially done using a GLINT Delta [1] chip.

The first step is to set the MultiGLINT bit in the **RasterizerMode** register. This causes GLINT to operate in an interleaved scan line mode when rasterizing primitives.

Which scanlines a GLINT owns is defined by the **ScanLineOwnership** register and this only has an effect when the MultiGLINT bit is set in the **RasterizerMode** register. The format is as follows:

Bits	Function
0, 1	Scanline Interval This is set to the number of GLINTs and has the values: 0 = 1 GLINT 1 = 2 GLINTs 2 = 4 GLINTs 3 = 8 GLINTs
2, 3, 4	Scanline. This holds which scanline within a Scanline Interval this GLINT owns. For example if the Scanline Interval and this field are both set to 2 then this GLINT owns scanlines 2, 6, 10, etc.

The Scanline Interval is decoded to select the number of least significant bits of Y (generated during rasterization) to compare with the same number of bits in the Scanline field. If these two value are the same then this GLINT owns the scanline.

The value of Y used is whatever the rasterizer has been given so it can be screen relative or window relative. The hardware will naturally force scanlines to be associated to screen relative coordinates. If window relative coordinates are used, the Scanline field will need to be set up to reflect this mapping whenever the window moves.

In some systems it is desirable for each GLINT to have only the memory it needs to hold the depth and color information for its scanlines. Setting the Scanline Interleave factor in the **LBReadMode** and **FBReadMode** registers achieves this. This Scanline Interleave value has the same meaning as in the **ScanLineOwnership** register, and is sometimes useful for it to have different values. Note that the texture addresses are not affected by the Scanline Interval as the texture maps are replicated in full.

After this set up the GLINTs all receive identical command and data streams.

To upload data from the localbuffer or framebuffer the standard upload command sequence is sent to all GLINTs but then the returned data is read a scanline at a time from the successive output FIFOs of the GLINTs.

To sync with the GLINTs the Sync command is broadcast to all GLINTs and then each output FIFO is polled in turn waiting for the **Sync** command. Alternatively interrupts could be used in which case the interrupt handler will collect an interrupt from each GLINT.

[1]The GLINT Delta is a 3D geometry chip from 3DLabs for the GLINT processor range, which offloads the triangle and line set-up calculations from the host and will also handle the broadcasting of commands and data to two GLINT devices.

7.3 The Host Connection

In an ideal system the host will be able to read and write to each GLINT individually and broadcast write to all the GLINTs together. Furthermore, best performance is typically achieved when DMA is used and it is less efficient if each GLINT is using DMA to service itself, with the consequent competition on the PCI bus. A better solution is for an external DMA controller to read the host memory and broadcast the data to the GLINTs. Each GLINT provides a set of signals to show the status of the input FIFO which the external DMA controller can monitor to determine when all GLINTs have enough space in their FIFO to accept the broadcast data.

The GLINT Delta chip provides an economical solution when there are two GLINT 500TX devices in a system as it includes the 'external DMA controller' and can broadcast or selectively write. As an added bonus the GLINTdelta will also do the triangle and line set up calculations which further reduces the host's load.

7.4 The Video Connection

The video stream in a multi-GLINT system with separate framebuffers will need to cycle amongst the framebuffers so each framebuffer provides the data for its own scanlines. The internal video timing generator in GLINT does not have the flexibility to accommodate this so an external video timing generator is needed. This can typically be implemented economically in a few large pals.

7.5 Performance

The following comments assume the system is rendering bound, that each GLINT has its own localbuffer and framebuffer, and that the primitives are large enough to have sufficient pixels for each GLINT to have some to work on.

The pixel throughput of general polygon rendering should increase linearly by the number of GLINTs in the system.

In the case of lines, image download and bitmasks the performance increase will be less as each fragment in the primitive needs to be processed by each GLINT. Fragments on owned scanlines will be processed at the rate appropriate to the rendering modes, while other fragments will only take one cycle.

7.6 A General Purpose Dual GLINT System

The main issue in a scanline interleaved multi-GLINT system is the difficulty of doing a fast arbitrary bit blit. This operation is important for windowing based GUI systems such as Microsoft Windows, or the X Window System. To provide an optimal 3D graphics on the 'desktop' system running in a GUI environment, a different approach can be adopted in the dual-GLINT case.

GLINT has a shared framebuffer interface so it is possible for two GLINTs to share the framebuffer but have separate localbuffers. The shared framebuffer protocol used to arbitrate access is designed to ensure fairer access for both GLINTs than typical master/slave protocols.

Many 3D operations (especially texture mapping) do not place a high burden on the framebuffer bandwidth so the framebuffer can be shared with little impact on the 3D

performance expected from a dual GLINT system.

GUI operations, which tend to be more framebuffer bound, can be implemented using just one GLINT, so the performance will be as good as for a single GLINT system. Note that when a 3D window is moved the localbuffer contents will also need to be moved as well. This raises the same problem that blits are needed in a split localbuffer, however the performance is less critical for 3D so the original blit method using image upload and image download is typically acceptable. Further, in many cases when a 3D window moves, it is acceptable for the application to be required to perform a redraw, or to wait for the next animation frame to be drawn, whereupon the copy becomes superfluous.

8. Performance Tips

The following is a list of software programming tips and techniques which can be applied to maximize GLINT performance.

The list is not exhaustive, nor is this note intended to be a replacement for the information to be found elsewhere in this manual and in the GLINT 500TX Hardware Reference Manual (HRM). Rather it is intended to serve as an introduction to some of the unique or unusual capabilities of the GLINT chip, and a pointer to where more detailed documentation can be found.

The following is a list of the topics which are covered:

- Using VRAM Block Writes - e.g. for clears
- Fast double buffering in a window using 12bit colorspace double buffering
- Incrementing addresses when writing to the FIFO to enable PCI burst transfers
- Using PCI Disconnect under PIO
- Using bus mastership (i.e. DMA)
- Improving DMA bus bandwidth utilization using the indexed FIFO modes
- Disabling units that are not in use (e.g. Framebuffer re ads)
- Use of fast clear planes for clearing the localbuffer
- Clearing all bitplanes of the localbuffer when possible
- Use of the extent register to minimize the area in the localbuffer and framebuffer that needs to be cleared
- Use of the GLINT graphics pipeline in preference to the framebuffer (and/or localbuffer) bypass when possible
- Loading registers in unit order (i.e. Rasterizer first - Host Out last)
- Avoiding unnecessary register updates
- Miscellaneous generic graphics tips

8.1 VRAM Block Writes

Typically GLINT boards are equipped with VRAMs that support block writes. This allows up to 32 pixels at a time to be filled with a constant color by a single framebuffer write access. This can, lead to roughly a 32fold increase in the speed of, for instance, clearing a large area of the framebuffer.

While this technique is most useful when clearing the framebuffer, it can be used to fill any trapezoid.

8.2 Fast double buffering in a window

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

GLINT 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
- Colorspace

For further details see section § 0, §0 and §0 of this manual.

Note that optimal functionality may be achieved by mixing two or more of the above double buffering techniques.

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

The simplest way to program GLINT is by writing data values into the memory mapped registers. 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 GLINT 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 DMA 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.

See section §0.

8.4 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). To facilitate the use of burst transfers when using programmed I/O to load the GLINT FIFOs, GLINT 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.

8.5 Using PCI Disconnect under Programmed I/O

The PCI bus protocol incorporates a feature known as PCI Disconnect, which is supported by GLINT. Once the GLINT is in this mode, if the host processor attempts to write to the full FIFO then instead of the write being lost, the GLINT 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 GLINT, 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 GLINT frees up an entry in its FIFO.

8.6 Using bus mastership (DMA)

Most GLINT boards support PCI bus mastership, allowing the on-board DMA of GLINT to be used to copy data from host memory into the GLINT FIFO.

If a board is DMA capable, then bit `25` of the **FBMemoryCtl** register will be set to `1`, otherwise this bit will be `0`.

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

- PCI bus bandwidth utilization is generally much improved. GLINT has been measured achieving transfer rates of up to 30-40MBytes/sec with a fast host slave (P90 Neptune chipset).
- 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 GLINT and the host, as the host can often perform useful work preparing the next DMA buffer once it has initiated one DMA transfer.

See section §0 for more details on using DMA.

8.7 Disabling units not in use

As a general rule any units within GLINT which are not actively in use for the current rendering should be disabled. Each unit has a bit in a control register for this purpose. This will maximize pixel throughput in the graphics core.

In particular it is important to check that unnecessary reads of the localbuffer are not taking place. For instance it is perfectly possible to set up the localbuffer read unit such that GLINT reads per pixel information (such as Z, stencil and fast clear plane 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.

Similar comments apply for the framebuffer read unit which again should only be enabled to read pixel data when it is essential.

Note that GLINT boards typically support hardware writemasks and these should be used in preference to the software writemasks.

8.8 Rapidly clearing the localbuffer - 1

GLINT supports a special technique for clearing down areas of the DRAM localbuffer, 16 or even 256 times faster than simply writing to every pixel.

When an application is generating animation images, it is normally necessary not only to draw each picture into the framebuffer, but also to first clear down the framebuffer, and to clear down auxiliary buffers such as depth (Z) buffers, stencil buffers, alpha buffer s and others.

In most applications the value written when clearing any given buffer, is the same at every pixel location, though different values may be used in different auxiliary buffers. Thus the framebuffer is often cleared to the value which corresponds to black, while the depth(Z) buffer is typically cleared to a value corresponding to infinity.

This unique capability is referred to as the fast clear mechanism.

Essentially the fast clear mechanism provides a method where the time taken to clear buffers such as the depth(Z) and stencil buffers can be amortized over a number of clear operations issued by the application.

8.9 Rapidly clearing the localbuffer - 2

When clearing the localbuffer it is faster to make accesses to all the bitplanes of the localbuffer e.g. clear the fast clear planes, stencil & depth(Z) buffers simultaneously. This is because just clearing the depth(Z) requires a read-modify-write, whereas clearing all the bitplanes can be done with a write.

8.10 Rapid clear of the localbuffer & framebuffer

GLINT can be instructed to maintain a record of the minimum bounding box that has been rendered to, in a given period. In some circumstances this may be used to limit the area that must be cleared down.

Note that this technique is not appropriate for use in conjunction with the fast clear mechanism for the localbuffer described above.

For further details see the description of the Host Out Unit in this manual.

8.11 Use of the framebuffer (or localbuffer) bypass

Whenever possible rendering should be done through the GLINT graphics pipeline. This is because reading and writing the framebuffer (or localbuffer) using the bypass is relatively slow. In some cases performance may even be improved if a small area of the framebuffer (and/or localbuffer) is uploaded through the graphics pipeline into a bitmap, rendered to, and then downloaded again through the graphics pipeline.

8.12 Loading registers in unit order

To maximize performance, the control registers for the next primitive should be loaded into the GLINT FIFO in unit order. Thus the registers associated with the Rasterizer unit should be loaded first, then Scissor unit, Stipple unit, Color DDA, 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 refer to Figure 0.1.

8.13 Avoiding Unnecessary Register Updates

GLINT control registers retain their value between one primitive and the next. Thus it is not necessary to reload registers that are unchanged between primitives. e.g. the **dY** register usually is set to either +1 or -1 (except when antialiasing).

In addition calculations of register values can often be shared across primitives, for instance between edges in adjacent polygons in meshes.

8.14 Miscellaneous Generic Graphics Tips

The following is a set of miscellaneous tips that are not GLINT specific, but well worth using:

- Avoid polling for Vblank whenever possible, but if you have to poll, consider whether your application is taking just longer than an integer number of Vblank intervals to draw a frame - slightly simplifying the frame to make it just under an integer multiple can dramatically improve performance.
- Another way of looking at the same problem is, if you remove your SwapBuffers() calls, does your application render many more frames per second? If so, you might be spending a lot of time waiting for buffer swaps, and you should tune your app so that it draws just enough to fit in one less frame time.
- When using DMA it may be best to flush the DMA buffer to the chip after entering a large primitive in the buffer (e.g. screen clear), so that the chip is doing useful work while further primitives are being prepared on the host.
- Minimize the use of the **Sync** command.
- Does making your window smaller cause things to speed up? If so, you're probably fill-limited (bottlenecked by filling the pixels in the window). Speed things up by reducing the depth complexity of your scene or by using simpler drawing operations wherever possible (e.g., avoiding depth-buffering for the background or ground plane).
- Does making your window smaller have no effect on the time it takes to draw a frame? If so, you're probably geometry-limited (bottlenecked by transformations, clipping, or lighting) or host-limited.
- Measure the time it takes your application to draw a frame. Now comment out all the drawing calls, and measure again. If most of the elapsed time per frame is spent doing things other that drawing, your application is probably host-limited rather than geometry-limited.
- If you're geometry-limited, you can speed things up by using simpler mod els with fewer vertices, by reducing the amount of clipped geometry, by using fewer light sources, etc. If you're host-limited you should use profiling tools to figure out where your application is spending its time and then tune those areas.

1.

Appendix A. Graphics Register Reference

This Appendix gives details of the format of each of the Graphics registers for GLINT. The GLINT Hardware Reference Manual, details all other registers not given here.

The registers are listed alphabetically by name.

- **Region:** specifies the section of the GLINT memory map in which the register appears. See the GLINT Hardware Reference Manual for more details.
- **Offset:** specifies the address offset from the base address of the region of this register.
- **Tag:** specifies the Tag value used in certain DMA modes. For more details see the Programming Model chapter.
- **Read/write** indicates that the register can be both written and read at the address given by **Offset**.
- **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 reset. In general for the Graphics registers this is undefined.

In the diagrams:

- **reserved** indicates bits that may be used in future members of the GLINT 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 GLINT family, but only to extend the dynamic range of these fields. While reading from these bits is undefined, a good convention to follow is to sign extend the numeric value, rather than masking them to zero before writing the register. This will ensure compatibility if the dynamic range is increased in future members of the GLINT family.
- For enumeration fields which do not specify the full range of possible values, only the specified values should be used. Future members of the GLINT family may define a meaning for the unused values.

AlphaBlendMode

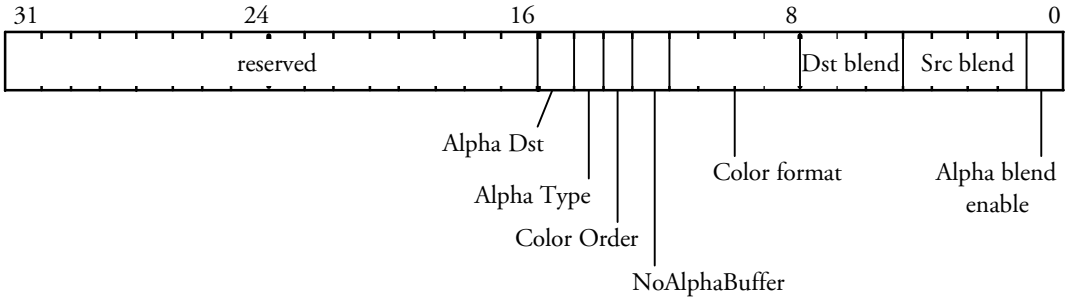
Name: Alpha Blend Mode

Unit: `Alpha Blend`

Region: `0` **Offset:** `0x0000.8810`

Tag: `0x102`

Read/write **Reset Value:** `Undefined`



Controls Alpha Blend ing.

Where the RGB format has an alpha component it may still not exist if those memory planes are not populated. In this case the NoAlphaBuffer bit in the **AlphaBlendMode** register should be set which causes the alpha component to be set to 255 (corresponding to an alpha value of 1.0). The values in the tables below are treated as floating point.

Note that alpha blending is not defined for the Color Index (CI) mode color formats.

Bit0 Alpha Blend Enable:

0 = Disable
1 = Enable

Bit1-4 Source Blend Mode:

Mode	Value	R	G	B	A
0	ZERO	0	0	0	0
1	ONE	1	1	1	1
2	DST_COLOR	R_d	G_d	B_d	A_d
3	ONE_MINUS_DST_COLOR	1 - R_d	1 - G_d	1 - B_d	1 - A_d
4	SRC_ALPHA	A_s	A_s	A_s	A_s
5	ONE_MINUS_SRC_ALPHA	1 - A_s	1 - A_s	1 - A_s	1 - A_s
6	DST_ALPHA	A_d	A_d	A_d	A_d
7	ONE_MINUS_DST_ALPHA	1 - A_d	1 - A_d	1 - A_d	1 - A_d
8	SRC_ALPHA_SATURATE	min of (A_s, 1 - A_d)	min of (A_s, 1 - A_d)	min of (A_s, 1 - A_d)	1

Bit5-7 Destination Blend Mode:

Mode	Value	R	G	B	A
0	ZERO	0	0	0	0
1	ONE	1	1	1	1
2	SRC_COLOR	R_s	G_s	B_s	A_s
3	ONE_MINUS_SRC_COLOR	1 - R_s	1 - G_s	1 - B_s	1 - A_s
4	SRC_ALPHA	A_s	A_s	A_s	A_s
5	ONE_MINUS_SRC_ALPHA	1 - A_s	1 - A_s	1 - A_s	1 - A_s
6	DST_ALPHA	A_d	A_d	A_d	A_d
7	ONE_MINUS_DST_ALPHA	1 - A_d	1 - A_d	1 - A_d	1 - A_d

Bit8-11 Color Format:

			Internal Color Channel			
	Format	Name	R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4Front	4@0	4@8	4@16	4@24
	4	4:4:4:4Back	4@4	4@12	4@20	4@28
	5	3:3:2Front	3@0	3@3	2@6	255
	6	3:3:2Back	3@8	3@11	2@14	255
	7	1:2:1Front	1@0	2@1	1@3	255
	8	1:2:1Back	1@4	2@5	1@7	255
	13	5:5:5Back	5@16	5@21	5@26	255
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4Front	4@16	4@8	4@0	4@24
	4	4:4:4:4Back	4@20	4@12	4@4	4@28
	5	3:3:2Front	3@5	3@2	2@0	255
	6	3:3:2Back	3@13	3@10	2@8	255
	7	1:2:1Front	1@3	2@1	1@0	255
	13	5:5:5Back	5@26	5@21	5@16	255
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

1) n@m means n bits starting at bit m are read from the framebuffer and scaled to fit the 8bit wide internal color channel.
2) Front and Back modes read the color value only from the corresponding low or high bits, to assist with color space double buffering.
3) A numerical value (0 or 255) is the value substituted when that channel does not exist in the framebuffer.

Bit12 **No Alpha Buffer Present:**
0 = Alpha Buffer present
1 = No Alpha Buffer present

Bit13 **ColorOrder:**
0 = BGR
1 = RGB

Bit14 **Alpha Type:**
0 = OpenGL
1 = QuickDraw3D

Bit15 **Alpha Dst:**
0 = FBData
1 = FBSourceData

AlphaTestMode

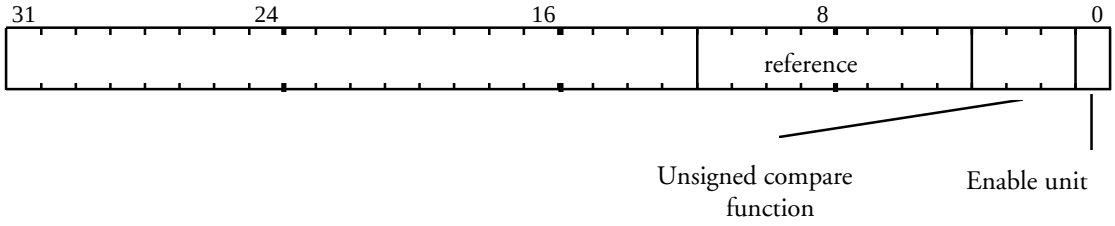
Name: Alpha Test Mode

Unit: `Alpha Test`

Region: `0` **Offset:** `0x0000.8800`

Tag: `0x100`

Read/write **Reset Value:** `Undefined`



When the unit is enabled, the compare operation compares the fragment's alpha value, against the reference alpha value in this register. If the comparison result is false, then the fragment is culled, and will not be drawn.

If the alpha test is disabled then it is as if the alpha test always passes.

The compare operation is done unsigned. The sense of the test is such that if the comparison is LESS and the reference value is 0x80, then fragments with alpha values between 0x0 and 0x7F will pass the test.

Bit0 **Alpha Test Enable:**

0 = Disable
1 = Enable

Bit1-3 **Unsigned Compare 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

Bit4-11 **Reference value**

AntialiasMode

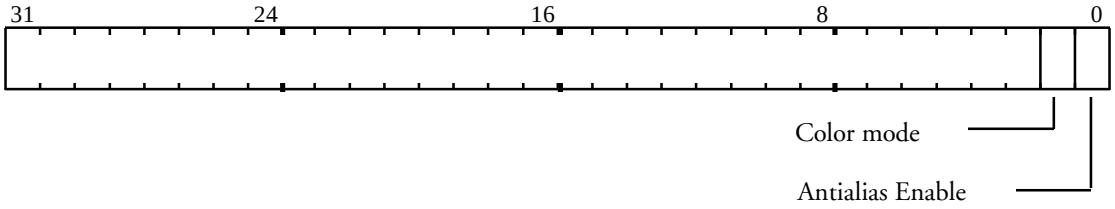
Name: Antialias Mode

Unit: `Antialias Application`

Region: `0` **Offset:** `0x0000.8808`

Tag: `0x101`

Read/write **Reset Value:** `Undefined`



Controls the operation of antialiasing. When the unit is enabled:

- In Color Index (CI) mode the bottom 4 bits of the color index of a fragment is replaced by the coverage value scaled by 15/256, where the result is rounded to the nearest integer.
- In RGBA mode the alpha component of a fragment is multiplied by the coverage value, but the RGB components are not changed.

Note that the CoverageEnable bit in the **Render** command must also be set to enable antialiasing.

Bit0 **Antialias Enable:**
0 = Disable
1 = Enable

Bit0 **Color Mode:**
0 = RGBA
1 = CI

AreaStippleMode

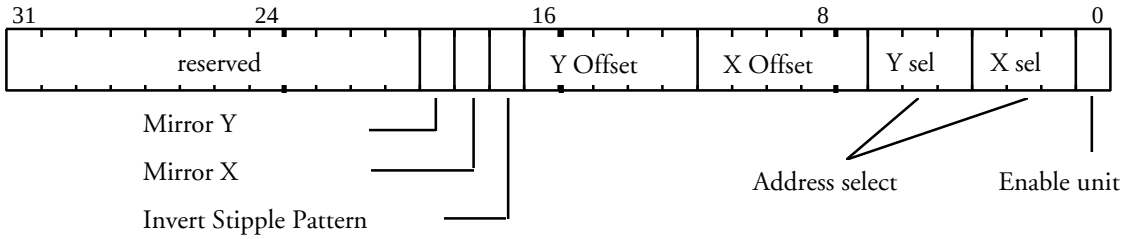
Name: Area Stipple Mode

Unit: `Stipple`

Region: `0` **Offset:** `0x0000.81A0`

Tag: `0x34`

Read/write **Reset Value:** `Undefined`



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

Bit1-3 **X address select:**
0 = 1 bit
1 = 2 bit
2 = 3 bit
3 = 4 bit
4 = 5 bit

Bit4-6 **Y address select:**
0 = 1 bit
1 = 2 bit
2 = 3 bit
3 = 4 bit
4 = 5 bit

Bit7-11 **XOffset**

Bit12-16 **YOffset**

Bit17 **Invert Stipple Pattern:**
0 = No Invert
1 = Invert

Bit18 **Mirror X:**
0 = No Mirror
1 = Mirror

Bit19 **Mirror Y:**
0 = No Mirror
1 = Mirror

AreaStipplePattern[0...31]

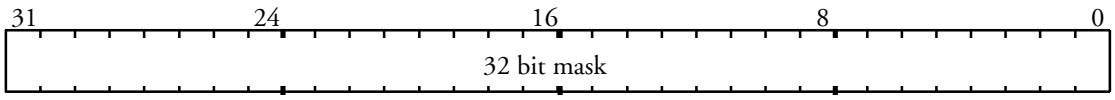
Name: Area Stipple Pattern

Unit: Stipple

Region: 0 **Offset:** 0x0000.8200, ..., 0x0000.82F8

Tag: 0x40, ...,0x5F

Read/write **Reset Value:** Undefined



These 32 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

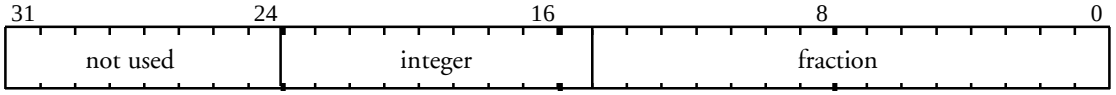
Name: Initial Color - Alpha

Unit: Color DDA

Region: 0 **Offset:** 0x0000.87C8

Tag: 0xF9

Read/write **Reset Value:** Undefined



This register is used to set the initial value for the Alpha value for a vertex when in Gouraud shading mode. The value is 2's complement 9.15 fixed point format.

BitMaskPattern

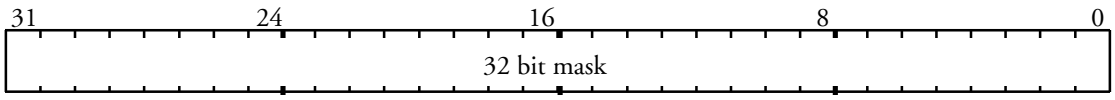
Name: Bit Mask Pattern

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.8068`

Tag: `0xD`

Write **Reset Value:** `Undefined`



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.

BorderColor

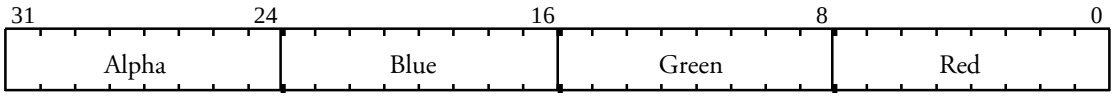
Name: Texture Border Color

Unit: `Texture`

Region: `0` **Offset:** `0x0000.84A8`

Tag: `0x95`

Read/write **Reset Value:** `Undefined`



32bit color value to be used for texture borders.

BStart

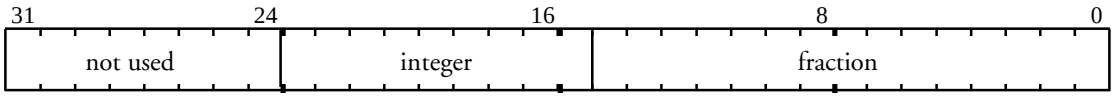
Name: Initial Color - Blue

Unit: Color DDA

Region: 0 **Offset:** 0x0000.87B0

Tag: 0xF6

Read/write **Reset Value:** Undefined



This register is used to set the initial value for the Blue value for a vertex when in Gouraud shading mode. The value is 2's complement 9.15 fixed point format.

Color

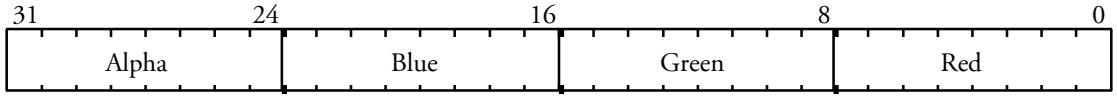
Name: Color

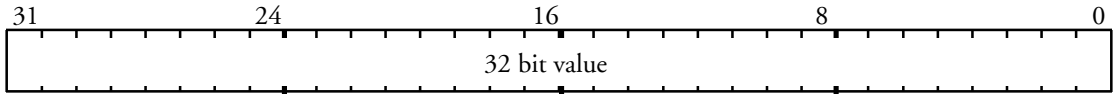
Unit: Color DDA

Region: 0 **Offset:** 0x0000.87F0

Tag: 0xFE

Write **Reset Value:** Undefined





Used for downloading image data to the framebuffer. The format is either the standard color format, or the raw framebuffer format if the color formatting unit is disabled.

In CI mode the color index is placed in bits 0-7. If the color index is less than 8bits then it is left justified in the most significant end of bits 0-7, and the least significant bits should be 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.

It can result in higher performance than using the **ConstantColor** register when rendering flat shaded, depth buffered primitives.

ColorDDAMode

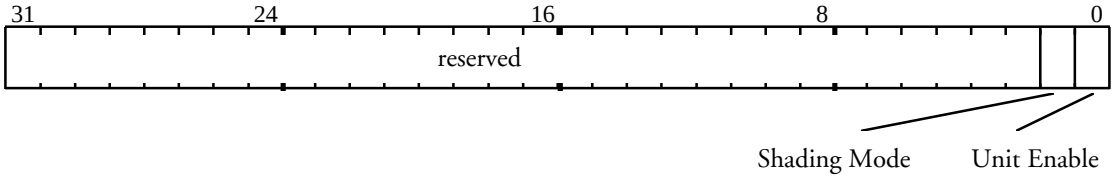
Name: Color DDA Mode

Unit: `Color DDA`

Region: `0` **Offset:** `0x0x0000.87E0`

Tag: `0xFC`

Read/write **Reset Value:** `Undefined`



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

ConstantColor

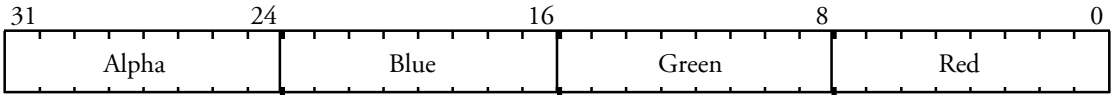
Name: Constant Color

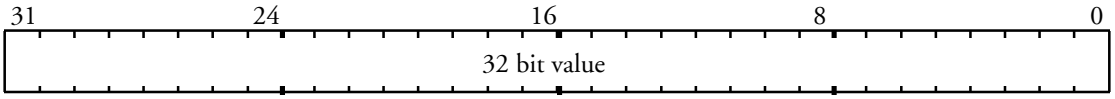
Unit: Color DDA

Region: 0 **Offset:** 0x0000.87E8

Tag: 0xFD

Read/write **Reset Value:** Undefined





Set to either an encoded color RGBA, (or a raw framebuffer data value if the color formatting unit is disabled) when in Flat shading mode (see the **ColorDDAMode** register).

In CI mode the color index is placed in bits 0-7. If the color index is less than 8bits then it is left justified in the most significant end of bits0-7, and the least significant bits should be set to zero.

Continue

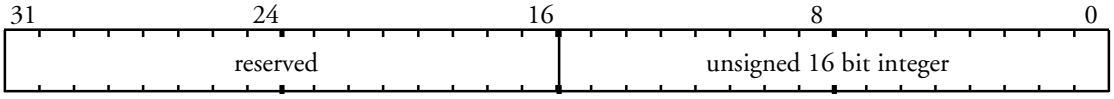
Name: Continue

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8058

Tag: 0xB

Write **Reset Value:** Undefined



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 (or sub scanlines) to fill. Note this count does not get loaded into the **Count** register.

ContinueNewDom

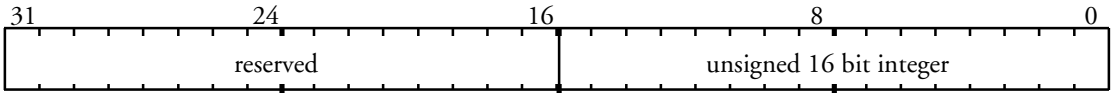
Name: Continue - New Dominant Edge

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8048

Tag: 0x9

Write **Reset Value:** Undefined



This command causes rasterization to continue with a new dominant edge. The dominant edge DDA in the rasterizer is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex 2D 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 any of the other units), it is not suitable for 3D operations.

The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the **Count** register.

ContinueNewLine

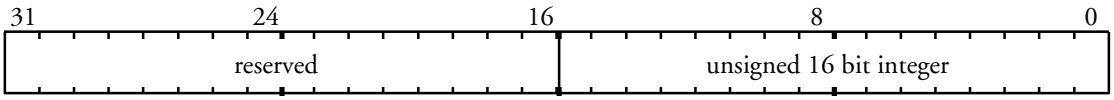
Name: Continue - New Line Segment

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8040

Tag: 0x8

Write **Reset Value:** Undefined



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 (or sub 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

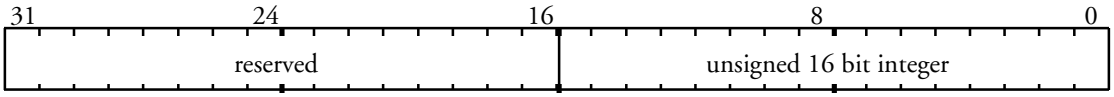
Name: Continue - New Subordinate Edge

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8050

Tag: 0xA

Write **Reset Value:** Undefined



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 (or sub scanlines) to fill. Note this count does not get loaded into the **Count** register.

Count

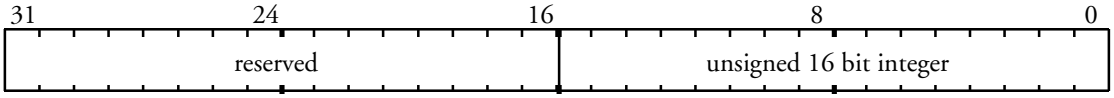
Name: Count

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8030

Tag: 0x6

Read/write **Reset Value:** Undefined



Contents is dependent on the mode set in the Render command:

Number of pixels in a line.

Number of scanlines in a trapezoid.

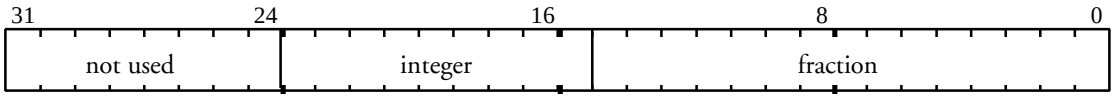
Number of sub scanlines in an antialiased trapezoid.

Diameter of an antialiased point in sub scanlines.

dAdx

dBdx

Name:	X Derivative - Color	
Unit:	`Color DDA`	
Region: `0`	**Offset:**	`0x0000.87D0, 0x0000.87B8, 0x0000.87A0, 0x0000.8788`
	Tag:	`0xFA, 0xF7, 0xF4, 0xF1`
Read/write	**Reset Value:**	`Undefined`



These four registers are used to set the X derivative for the Alpha, Blue, Green, Red values for the interior of a trapezoid when in Gouraud shading mode. The value is in 2's complement 9.15 fixed point format.

dAdyDom
dBdyDom
dGdyDom
dRdyDom

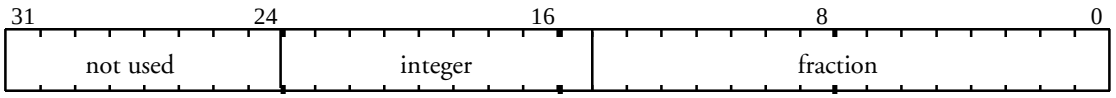
Name: Y Derivative Dominant - Color

Unit: Color DDA

Region: 0 **Offset:** 0x0000.87D8, 0x0000.87C0, 0x0000.87A8, 0x0000.8790

Tag: 0xFB, 0xF8, 0xF5, 0xF2

Read/write **Reset Value:** Undefined



These four registers are used to set the Y derivative dominant, for the Alpha, Blue, Green, Red values along a line, or for the dominant edge of a trapezoid, when in Gouraud shading mode. The value is in 2's complement 9.15 fixed point format.

Depth

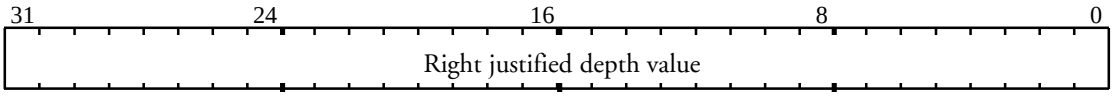
Name: Depth

Unit: Depth

Region: 0 **Offset:** 0x0000.89A8

Tag: 0x135

Read/write **Reset Value:** Undefined



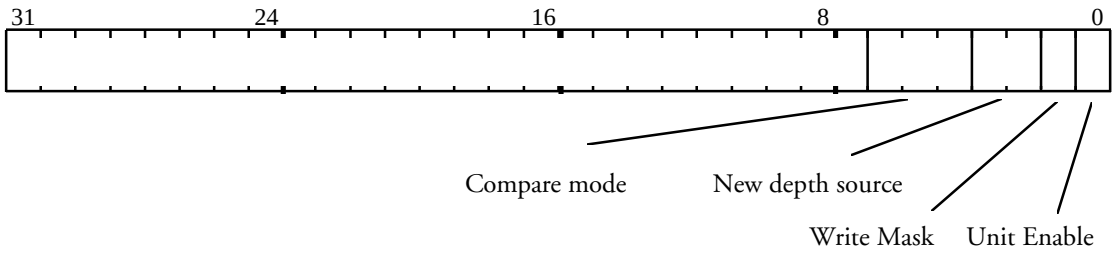
Holds an externally sourced 32 bit depth value. If the depth buffer holds less than 32bits then the user supplied depth value is right justified to the least significant end. The unused most significant bits should be set to zero.

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:	`Depth`	
Region: `0`	**Offset:**	`0x0000.89A0`
	Tag:	`0x134`
Read/write	**Reset Value:**	`Undefined`



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:

Mode	Comparison Function
0	NEVER
1	LESS
2	EQUAL
3	LESS OR EQUAL
4	GREATER
5	NOT EQUAL
6	GREATER OR EQUAL
7	ALWAYS

dFdx

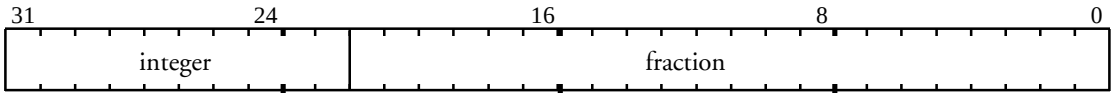
Name: X Derivative - Fog

Unit: Fog

Region: 0 **Offset:** 0x0000.86A8

Tag: 0xD5

Read/write **Reset Value:** Undefined



Fog coefficient derivative per unit X for use in rendering trapezoids. The value is in 2's complement 10.22 fixed point format.

dFdyDom

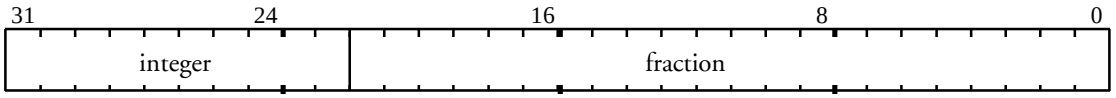
Name: Y Derivative Dominant - Fog

Unit: Fog

Region: 0 **Offset:** 0x0000.86B0

Tag: 0xD6

Read/write **Reset Value:** Undefined



Fog coefficient derivative per unit Y along a line, or for the dominant edge of a trapezoid. The value is in 2's complement 10.22 fixed point format.

DitherMode

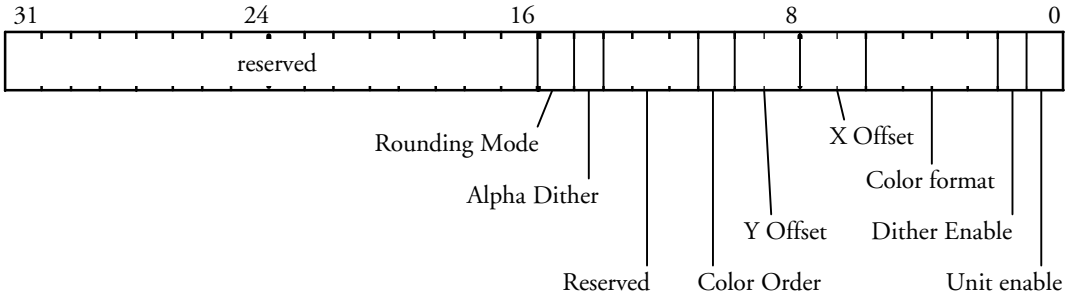
Name: Dither Mode

Unit: `Color Formatting`

Region: `0` **Offset:** `0x0000.8818`

Tag: `0x103`

Read/write **Reset Value:** `Undefined`



Controls the color formatting unit.

Bit0 **Unit Enable:**

0 = Disable
1 = Enable

Bit1 **Dither Enable:**

0 = Disable
1 = Enable

Bit2-5 **Color Format:**

1) n@m means that the internal color channel is converted into an n bit number and stored in the framebuffer at bit m. Bit zero is the least significant bit position.
2) Front and Back modes replicate the color value into the low and high bits to assist with color space double buffering. The modes are redundantly duplicated to mirror the color format field of the **AlphaBlendMode** register. Writemasks should be used to select only the high or low bits for each channel.
3) CI values are replicated into each byte (CI8) or nibble (CI4) to assist with color space double buffering.
4) A dash indicates that this channel does not occur in the framebuffer.

			Internal Color Channel			
	Format	Name	R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4 Front	4@0 4@4	4@8 4@12	4@16 4@20	4@24 4@28
	4	4:4:4:4 Back	4@0 4@4	4@8 4@12	4@16 4@20	4@24 4@28
	5	3:3:2 Front	3@0 3@8	3@3 3@11	2@6 2@14	-
	6	3:3:2 Back	3@0 3@8	3@3 3@11	2@6 2@14	-
	7	1:2:1 Front	1@0 1@4	2@1 2@5	1@3 1@7	-
	8	1:2:1 Back	1@0 1@4	2@1 2@5	1@3 1@7	-
	13	5:5:5 Back	5@0 5@16	5@5 5@21	5@10 5@26	-
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4 Front	4@16 4@20	4@8 4@12	4@0 4@4	4@24 4@28
	4	4:4:4:4 Back	4@16 4@20	4@8 4@12	4@0 4@4	4@24 4@28
	5	3:3:2 Front	3@5 3@13	3@2 3@10	2@0 2@8	-
	6	3:3:2 Back	3@5 3@13	3@2 3@10	2@0 2@8	-
	7	1:2:1 Front	1@3 1@7	2@1 2@5	1@0 1@4	-
	8	1:2:1 Back	1@3 1@7	2@1 2@5	1@0 1@4	-
	13	5:5:5 Back	5@10 5@26	5@5 5@21	5@0 5@16	-
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Bit6-7 **XOffset** to enable window relative dithering.

Bit8-9 **YOffset** to enable window relative dithering.

Bit10 **Color Order:**
0 = BGR

1 = RGB

Bit14 **Alpha Dither:**

0 = Default
1 = No Dither

Bit15 **Rounding Mode:**

0 = Truncate
1 = Round

dKdx

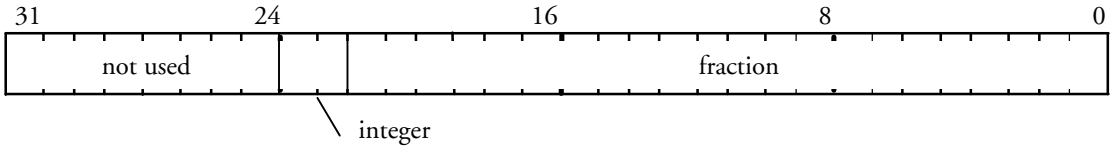
Name: Ks and Kd Derivative unit X

Unit: Texture

Region: 0 **Offset:** 0x0000.86D0, 0x0000.86E8

Tag: 0xDA, 0xDD

Read/write **Reset Value:** Undefined



Derivative unit X for Ks and Kd. The value is 2.22, 2's complement fixed point format.

dKddyDom

dKsdyDom

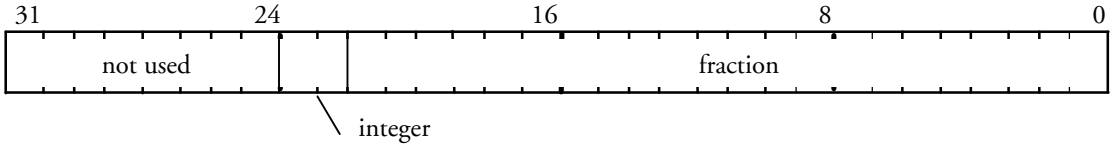
Name: Ks and Kd Derivative unit Y Dominant edge

Unit: Texture

Region: 0 **Offset:** 0x0000.86D8, 0x0000.86F0

Tag: 0xDB, 0xDE

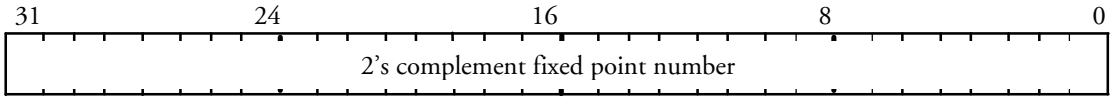
Read/write **Reset Value:** Undefined



Derivative unit Y dominant edge, for Ks and Kd. The value is 2.22, 2's complement fixed point format.

dQdx
dSdx
dTdx

Name:	X Derivative - Texture	
Unit:	`Texture`	
Region: `0`	**Offset:**	`0x0000.8390, 0x0000.83A8 0x0000.83C0`
	Tag:	`0x72, 0x75, 0x78`
Read/write	**Reset Value:**	`Undefined`



Used to set the X derivative for the S, T and Q parameters for texture map interpolation. The value is in 2's complement fixed point format. The binary point is at an arbitrary location, but must be consistent for all S, T and Q values.

dQdyDom
dSdyDom
dTdyDom

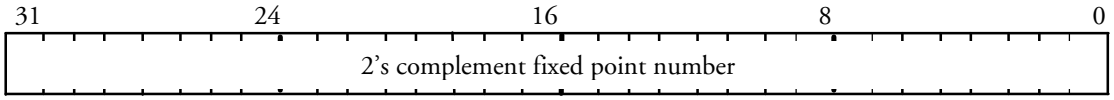
Name: Y Derivative Dominant - Texture

Unit: `Texture`

Region: `0` **Offset:** `0x0000.8398, 0x0000.83B0 0x0000.83C8`

Tag: `0x73, 0x76, 0x79`

Read/write **Reset Value:** `Undefined`



Used to set the Y derivative dominant for the S, T and Q parameters for texture map interpolation. The value is in 2's complement fixed point format. The binary point is at an arbitrary location, but must be consistent for all S, T and Q values.

dXDom

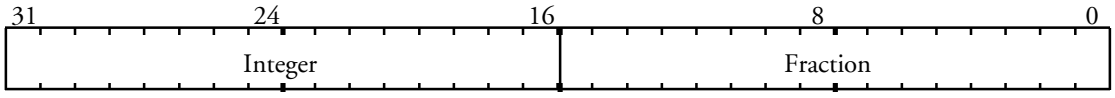
Name: Delta X Dominant

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.8008`

Tag: `0x1`

Read/write **Reset Value:** `Undefined`



Value added when moving from one scanline (or sub scanline) to the next for the dominant edge in trapezoid filling. The value is in 2's complement 16.16 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

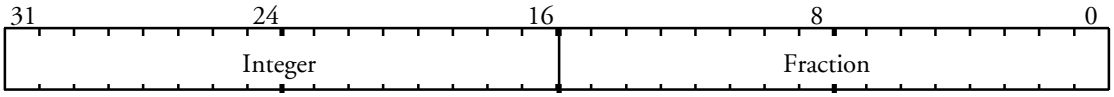
Name: Delta X Subordinate

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.8018`

Tag: `0x3`

Read/write **Reset Value:** `Undefined`



Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling. The value is in 2's complement 16.16 fixed point format.

dY

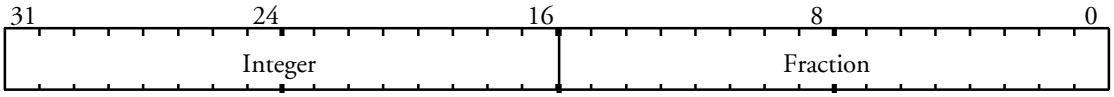
Name: Delta Y

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.8028`

Tag: `0x5`

Read/write **Reset Value:** `Undefined`



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 16.16 fixed point format.

For trapezoids the value will be:

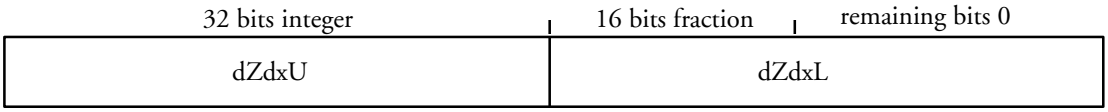
±1.0 if non-antialiased, depending on the scanning direction.

±0.25 when using 4x4 quality antialiasing, depending on the scanning direction.

±0.125 when using 8x8 quality antialiasing, depending on the scanning direction.

dZdxU

Name:	Depth Derivative X	
Unit:	`Depth`	
Region: `0`	**Offset:**	`0x0000.89C8, 0x0000.89C0`
	Tag:	`0x139, 0x138`
Read/write	**Reset Value:**	`Undefined`

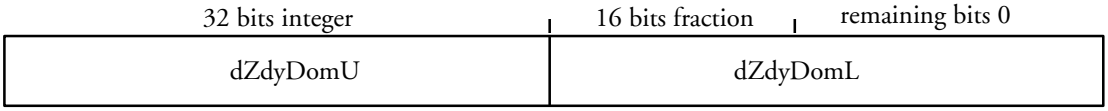


This pair of registers set the depth derivative per unit in X used in rendering trapezoids. **dZdxU** holds the most significant bits, and **dZdxL** the least significant bits. The value is in 2's complement 32.16 fixed point format.

dZdyDomL

dZdyDomU

Name:	Depth Derivative Y Dominant	
Unit:	`Depth`	
Region: `0`	**Offset:**	`0x0000.89D8, 0x0000.89D0`
	Tag:	`0x13B, 0x13A`
Read/write	**Reset Value:**	`Undefined`



This pair of registers set the depth derivative per unit in Y for the dominant edge, or along a line. **dZdyDomU** holds the most significant bits, and **dZdyDomL** the least significant bits. The value is in 2's complement 32.16 fixed point format.

FastClearDepth

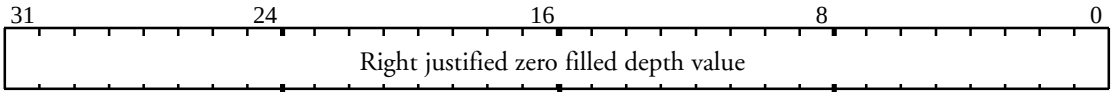
Name: Fast Clear Depth

Unit: Depth

Region: 0 **Offset:** 0x0000.89E0

Tag: 0x13C

Read/write **Reset Value:** Undefined



Depth value to be substituted when using the Frame Count Planes mechanism to provide fast clear of the depth buffer.

FBBlockColor

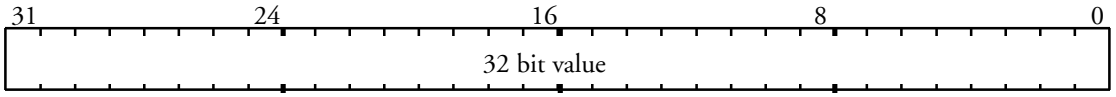
Name: Framebuffer Block Color

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8AC8

Tag: 0x159

Read/write **Reset Value:** Undefined



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 replicated to all 4 bytes of the register.

If the framebuffer is in 16 bit packed mode then the data must be replicated to both halves of the register.

Writing to this register will automatically update the upper and lower 32 bits of the destination 64bit wide register in the VRAMs.

Reading from this register will return the lower 32 bits of the 64 bit wide register in the VRAMs.

FBBlockColorU

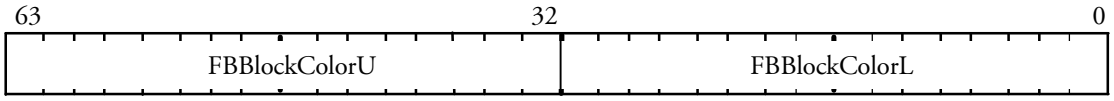
Name: Framebuffer Block Color Lower and Upper

Unit: `Framebuffer R/W`

Region: `0` **Offset:** `0x0000.8C70, 0x0000.8C68`

Tag: `0x18E, 18D`

Read/write **Reset Value:** `Undefined`



Contains the lower and upper respectively, 32 bit words of color data (and optionally alpha value) to be written to the framebuffer during block writes. Note the format is the raw data format of the framebuffer.

Note the lower 32bits are at the higher address.

If the framebuffer is used in 8 bit packed mode, then data should be replicated to all 8 bytes of the register.

If the framebuffer is in 16 bit packed mode then the data must be replicated to all four half words of the register.

The **FBBlockColorL** and **FBBlockColorU** registers are aliased with the **FBBlockColor** register for backwards compatibility.

FBColor

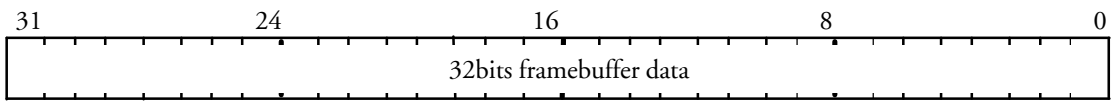
Name: Framebuffer Color Upload

Unit: `Framebuffer R/W`

Region: `N/A` **Offset:** `N/A`

Tag: `0x153`

Read/Output **Reset Value:** `Undefined`



Internal register used in image upload. Note 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 in the Color Format unit.

FBData

Name: Framebuffer Data

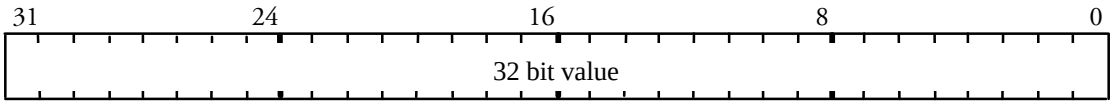
Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8AA0

Tag: 0x0154

Reset Value: Undefined

Write



Supplies the data for situations such as image download where subsequent formatting is required. The formatting can be achieved by means of the **AlphaBlendMode** register to convert to the internal GLINT format, and then via the **DitherMode** register to convert to the required format.

FBHardwareWriteMask

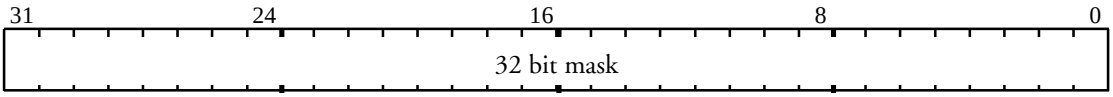
Name: Hardware Writemask

Unit: Framebuffer Writemask

Region: 0 **Offset:** 0x0000.8AC0

Tag: 0x158

Read/write **Reset Value:** Undefined



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.

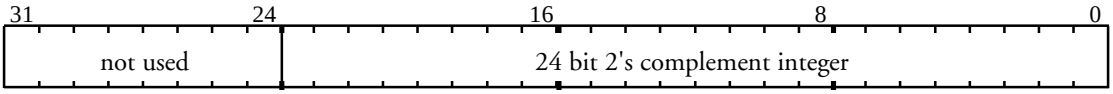
If hardware writemasks are used then all the bits in the software writemask must be set to 1, so that software writemasking is disabled.

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

If the framebuffer is in 16 bit packed mode then the 16 bit hardware writemask must be replicated to both halves of the **FBHardwareWriteMask** register.

FBPixelOffset

Name:	Framebuffer Pixel Offset	
Unit:	`Framebuffer R/W`	
Region: `0`	**Offset:**	`0x0000.8A90`
	Tag:	`0x152`
Read/write	**Reset Value:**	`Undefined`



Offset between buffers when operating on multiple buffers (left/right/front/back) in the framebuffer at the same time.

FBReadMode

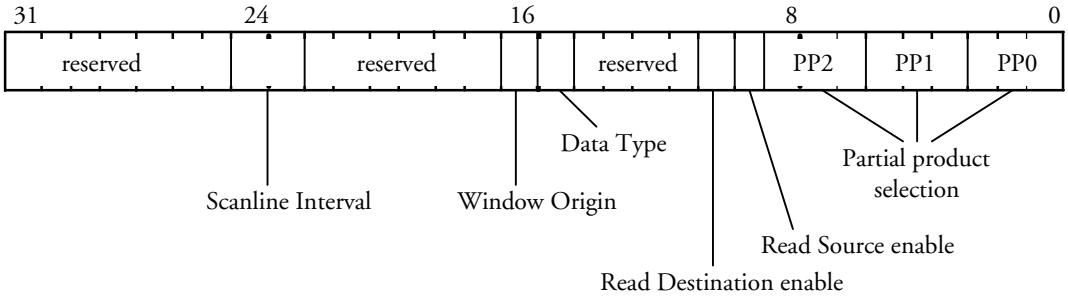
Name: Framebuffer Read Mode

Unit: `Framebuffer R/W`

Region: `0` **Offset:** `0x0000.8A80`

Tag: `0x150`

Read/write **Reset Value:** `Undefined`



Controls reading from framebuffer memory.

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

Bit17-22 **Reserved:**

Bit23-24 **Scanline Interval:**
0 = 1
1 = 2
2 = 4
3 = 8

FBSoftwareWriteMask

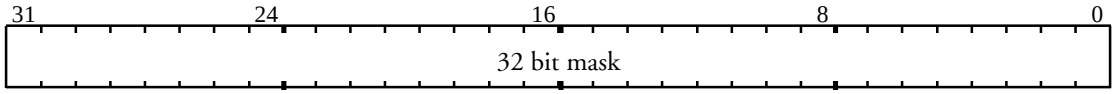
Name: S/w Writemask

Unit: `Framebuffer Writemask`

Region: `0` **Offset:** `0x0000.8820`

Tag: `0x104`

Read/write **Reset Value:** `Undefined`



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 ReadDestinationEnable 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.

FBSourceData

Name: Framebuffer Source Data

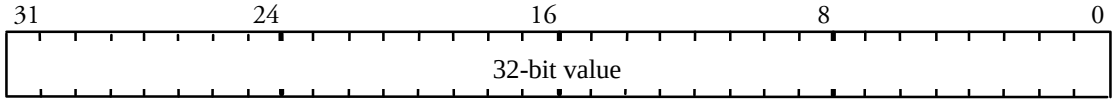
Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8AA8

Tag: 0x0155

Reset Value: Undefined

Write



Supplies the data for situations such as image download with logic ops, where the data is treated as the source rather than the destination parameter. The data supplied should be raw framebuffer format.

FBSourceOffset

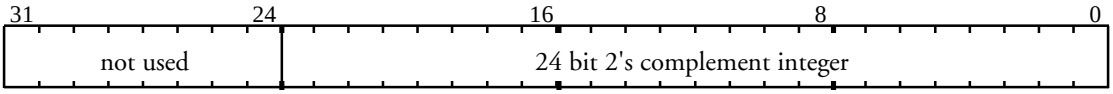
Name: Framebuffer Source Offset

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8A88

Tag: 0x151

Read/write **Reset Value:** Undefined



Sets the offset between the source and destination for a copy operation in the framebuffer.

SourceOffset = SourceAddr - DestAddr

FBWindowBase

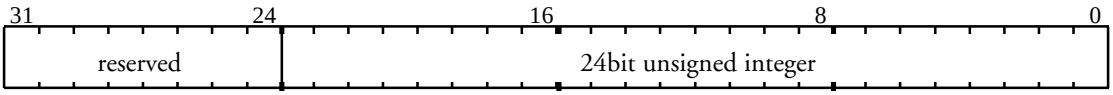
Name: Framebuffer Window Base

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8AB0

Tag: 0x156

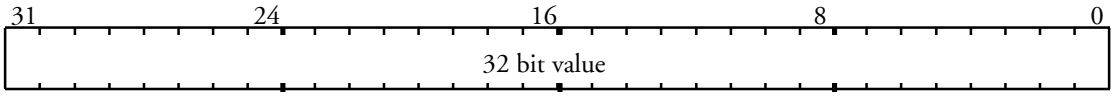
Read/write **Reset Value:** Undefined



Contains the current base address of the window in the framebuffer.

FBWriteData

Name:	Framebuffer Write Data	
Unit:	Logic Op	
Region: 0	**Offset:**	0x0000.8830
	Tag:	0x106
Read/write	**Reset Value:**	Undefined



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, depth or GID testing
- No alpha blending
- No software writemasking

The data is in the raw format of the framebuffer.

Hardware writemasks can be used if available.

FBWriteMode

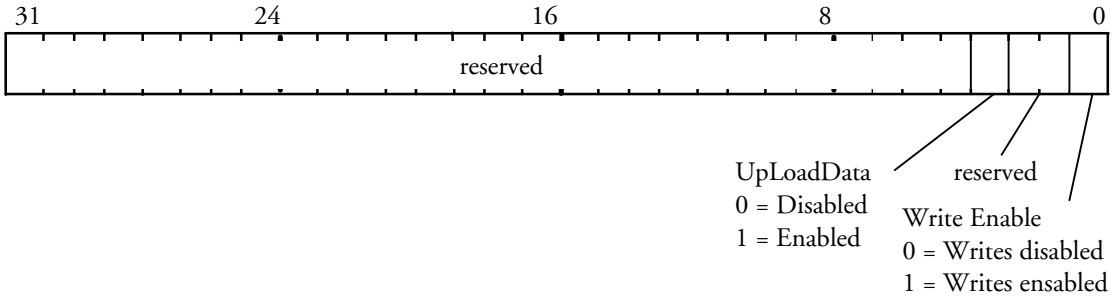
Name: Framebuffer Write Mode

Unit: `Framebuffer R/W`

Region: `0` **Offset:** `0x0000.8AB8`

Tag: `0x157`

Read/write **Reset Value:** `Undefined`



Controls writing to the framebuffer.

Bit0 **Write Enable:**
0 = Disable
1 = Enable

Bit1-2 **Reserved:**

Bit3 **UpLoadData:**
0 = No upload
1 = Upload color to host

FilterMode

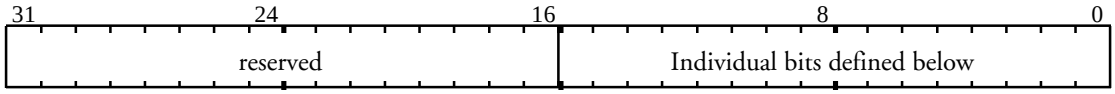
Name: Filter Mode

Unit: `Host Out`

Region: `0` **Offset:** `0x0000.8C00`

Tag: `0x180`

Read/write **Reset Value:** `Undefined`



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 **Diagnostic use only** - 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 outpu t 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 **Diagnostic use only** - set to zero.

FlushSpan

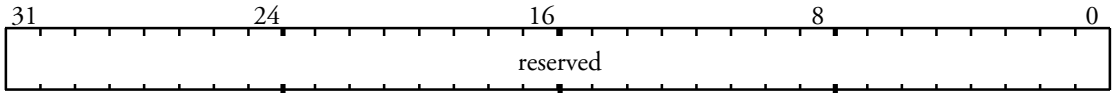
Name: Flush Span

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8060

Tag: 0xC

Write **Reset Value:** Undefined



Command used when antialiasing to force rasterization of any remaining sub-scanlines in a primitive.

FogColor

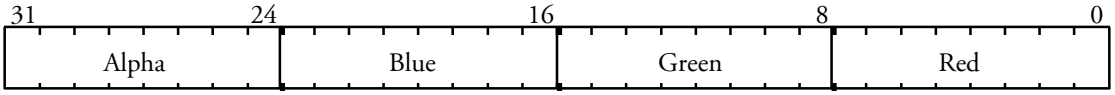
Name: Fog Color

Unit: Fog

Region: 0 **Offset:** 0x0000.8698

Tag: 0xD3

Read/write **Reset Value:** Undefined

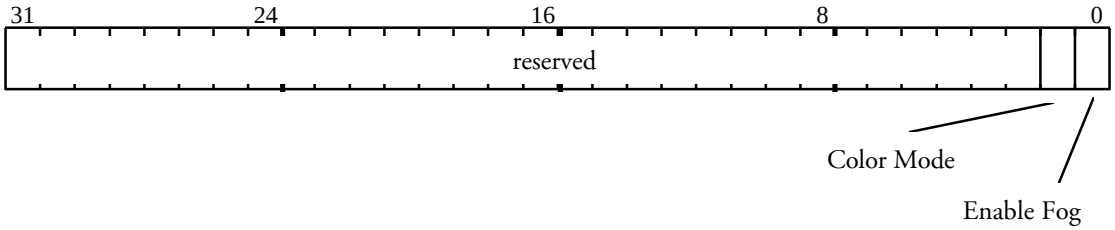


Provides the color to be blended with the fragment's color when fogging is enabled.

In CI mode the color index is placed in bits 0-7. If the color index is less than 8bits then it is left justified in the most significant end of bits0-7, and the least significant bits should be set to zero.

FogMode

Name:	Fog Mode	
Unit:	Fog	
Region: 0	**Offset:**	0x0000.8690
	Tag:	0xD2
Read/write	**Reset Value:**	Undefined



Controls operation of the Fog unit.

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

Bit1 **Color Mode:**
0 = RGBA
1 = CI

FStart

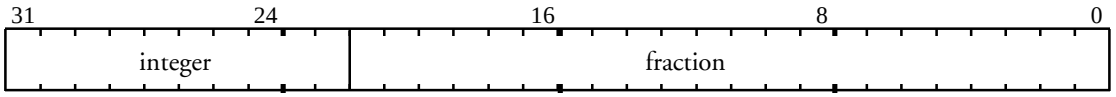
Name: Fog Start Color

Unit: Fog

Region: 0 **Offset:** 0x0000.86A0

Tag: 0xD4

Read/write **Reset Value:** Undefined



Fog coefficient start value. Note the interpolation coefficient is used to blend the fragments color with the color in the **FogColor** register. The value is in 2's complement 10.22 fixed point format.

GStart

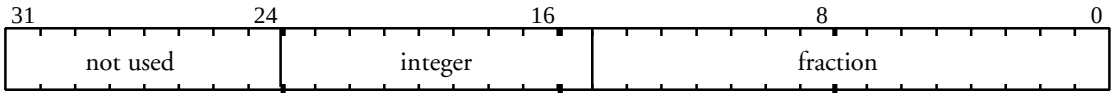
Name: Initial Color - Green

Unit: Color DDA

Region: 0 **Offset:** 0x0000.8798

Tag: 0xF3

Read/write **Reset Value:** Undefined



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 9.15 fixed point format.

Interp[0...4]

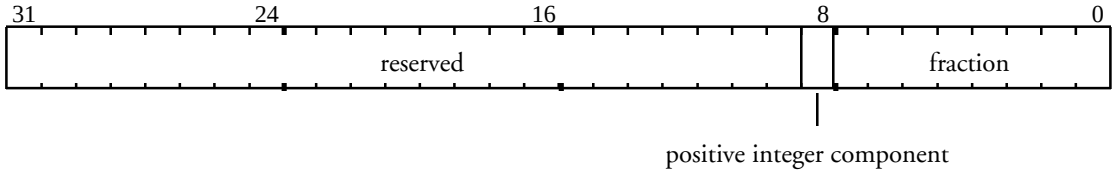
Name: Interpolation Coefficients

Unit: Texture

Region: 0 **Offset:** 0x0000.8640, ..., 0x0000.8660

Tag: 0xC8, ..., 0xCC

Read/write **Reset Value:** Undefined



Interpolation coefficients for texture mapping . The value is in 1.8 fixed point format.

These registers are only present for backwards compatibility with the GLINT 300SX. They should not be used for the GLINT 500TX.

KdStart

KsStart

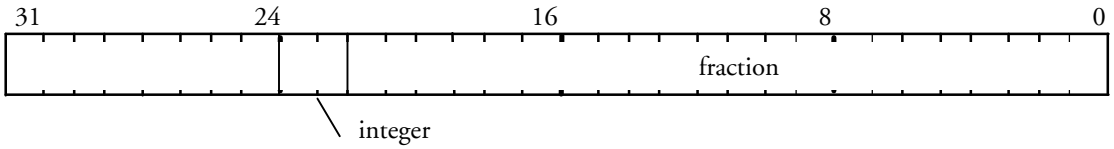
Name: Ks and Kd Start Value

Unit: Texture

Region: 0 **Offset:** 0x0000.86C8, 0x0000.86E0

Tag: 0xD9, 0xDC

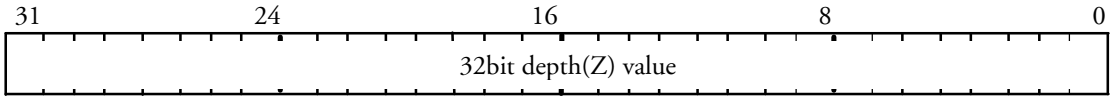
Read/write **Reset Value:** Undefined



Initial values for Ks and Kd. The value is 2.22, 2's complement fixed point format.

LBDepth

Name:	Localbuffer Depth Upload		
Unit:	`Localbuffer R/W`		
Region: `N/A`	**Offset:**	`N/A`	
	Tag:	`0x116`	
Read/Output	**Reset Value:**	`Undefined`	



Internal register used in image upload of the depth buffer. This register should not be written to. It is documented here to give the tag value and format of the data which is read from the Host Out FIFO. Where the depth(Z) buffer width is less than 32bits, the depth value is right justified and zero extended.

LBReadFormat

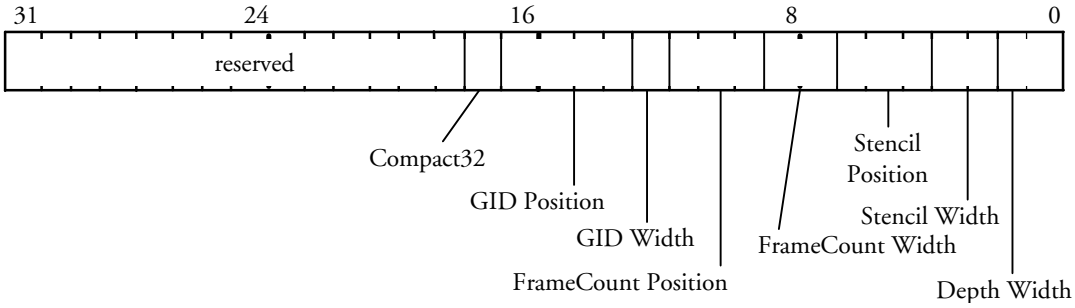
Name: Localbuffer Read Format

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.8888`

Tag: `0x111`

Read/write **Reset Value:** `Undefined`



Specifies the format used when reading from localbuffer memory. The effect of creating a format with overlapping fields is undefined.

Bit0-1 Depth Width:
0 = 16
1 = 24
2 = 32

Bit2-3 Stencil Width:
0 = 0
1 = 4
2 = 8

Bit4-6 Stencil Position:
0 = 16
1 = 20
2 = 24
3 = 28
4 = 32

Bit7-8 Frame Count Width:
0 = 0
1 = 4
2 = 8

Bit9-11 Frame Count Position:

0 = 16
1 = 20
2 = 24
3 = 28
4 = 32
5 = 36
6 = 40

Bit12 GID Width:

0 = 0
1 = 4

Bit13-16 GID Position:

0 = 16
1 = 20
2 = 24
3 = 28
4 = 32
5 = 36
6 = 40
7 = 44
8 = 48

Bit17 Compact32:

0 = No
1 = Yes

LBReadMode

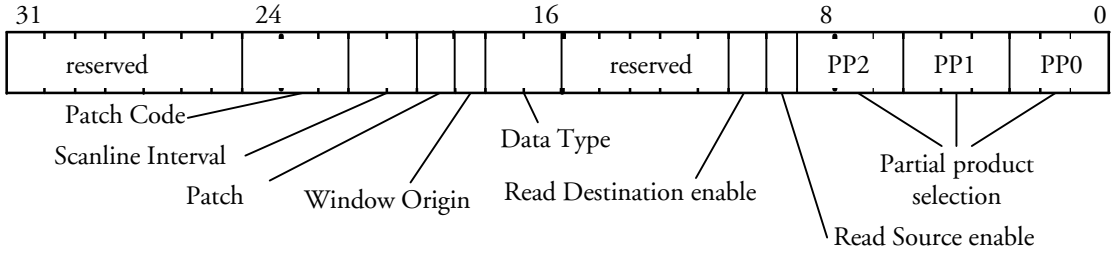
Name: Localbuffer Read Mode

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.8880`

Tag: `0x110`

Read/write **Reset Value:** `Undefined`



Controls reading from localbuffer memory.

Bit0-2 **Partial Product 0**

Bit3-5 **Partial Product 1**

Bit6-8 **Partial Product 2**

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 = Localbuffer Default
1 = Localbuffer Stencil
2 = Localbuffer Depth

Bit18 **Window Origin:**
0 = Top left
1 = Bottom left

Bit19 **Patch:**
0 = No
1 = Yes

Bit20-21 **Scanline Interval:**
0 = 1
1 = 2
2 = 4
3 = 8

Bit22-24 **Patch Code:** Controls texture download address generation

LBSourceOffset

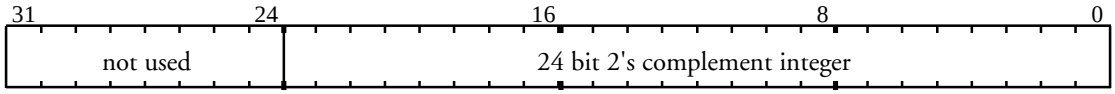
Name: Localbuffer Source Offset

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.8890`

Tag: `0x112`

Read/write **Reset Value:** `Undefined`



Sets the offset between the source and destination for a copy operation in the localbuffer, i.e.:

SourceOffset = SourceAddr - DestAddr

LBStencil

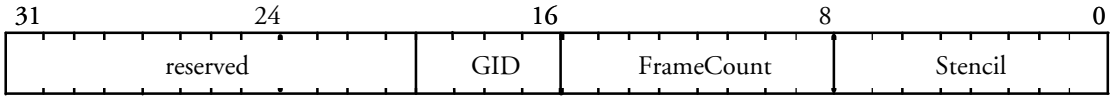
Name: Localbuffer Stencil Upload

Unit: `Localbuffer R/W`

Region: `N/A` **Offset:** `N/A`

Tag: `0x115`

Read/Output **Reset Value:** `Undefined`



Internal register used in image upload of the stencil buffer. This register should not be written to. It is documented here to give the tag value and format of the data which is read from the Host Out FIFO.

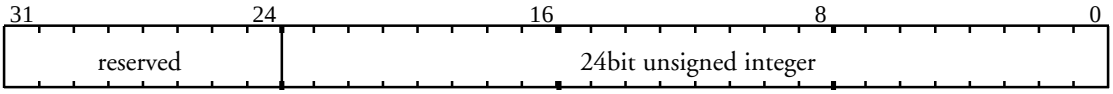
Bit0-7 Stencil:

Bit8-15 FrameCount:

Bit16-19 GID:

LBWindowBase

Name:	Localbuffer Window Base	
Unit:	`Localbuffer R/W`	
Region: `0`	**Offset:**	`0x0000.88B8`
	Tag:	`0x117`
Read/write	**Reset Value:**	`Undefined`



Contains the current base address of the window in the localbuffer.

LBWriteFormat

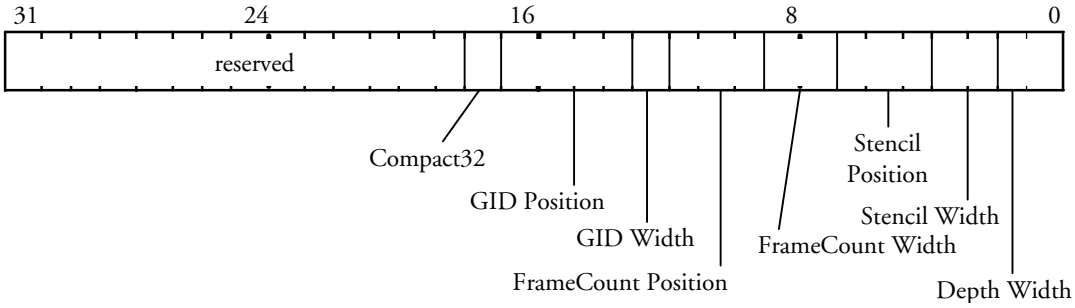
Name: Localbuffer Write Format

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.88C8`

Tag: `0x119`

Read/write **Reset Value:** `Undefined`



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 = 24
2 = 32

Bit2-3 **Stencil Width:**

0 = 0
1 = 4
2 = 8

Bit4-6 **Stencil Position:**

0 = 16
1 = 20
2 = 24
3 = 28
4 = 32

Bit7-8 **Frame Count Width:**

0 = 0
1 = 4
2 = 8

Bit9-11 Frame Count Position:

0 = 16
1 = 20
2 = 24
3 = 28
4 = 32
5 = 36
6 = 40

Bit12 GID Width:

0 = 0
1 = 4

Bit13-16 GID Position:

0 = 16
1 = 20
2 = 24
3 = 28
4 = 32
5 = 36
6 = 40
7 = 44
8 = 48

Bit17 Compact32:

0 = No
1 = Yes

LBWriteMode

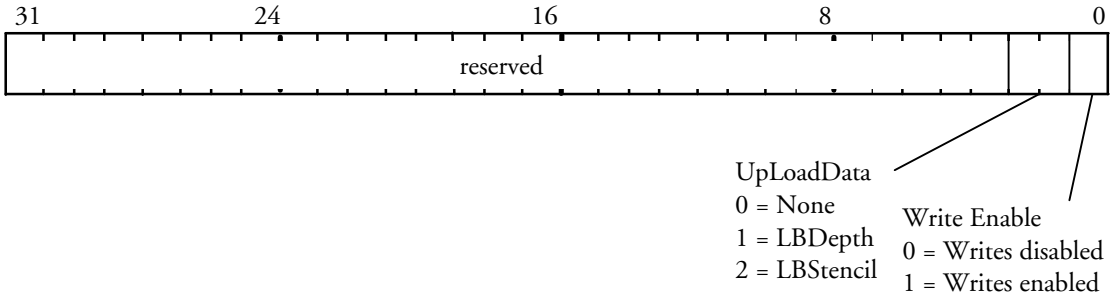
Name: Localbuffer Write Mode

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.88C0`

Tag: `0x118`

Read/write **Reset Value:** `Undefined`



Controls writing to the localbuffer.

Bit0 **Write Enable:**
0 = Disable
1 = Enable

Bit1-2 **UpLoadData:**
0 = None
1 = LBDepth
2 = LBStencil

LineStippleMode

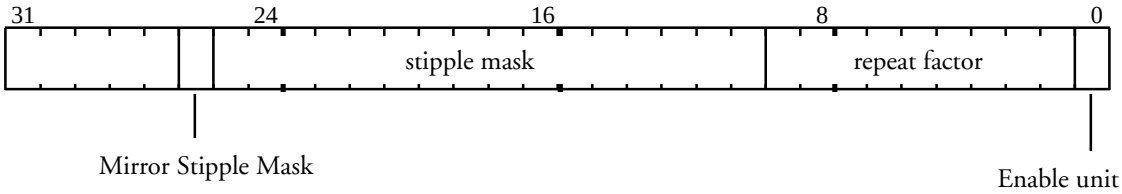
Name: Line Stipple Mode

Unit: `Stipple`

Region: `0` **Offset:** `0x0000.81A8`

Tag: `0x35`

Read/write **Reset Value:** `Undefined`



Controls Line Stippling.

Bit0 **Unit Enable:**
0 = Disable
1 = Enable

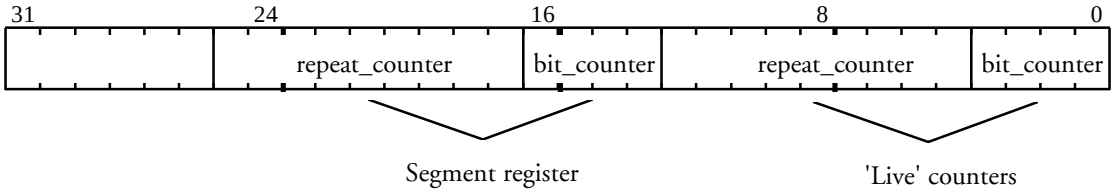
Bit1-9 **Repeat Factor** - set to one less than the required value

Bit10-25 **Stipple Mask**

Bit26 **Mirror Stipple Mask:**
0 = No mirror
1 = Mirror

LoadLineStippleCounters

Name:	Load Line Stipple Counters	
Unit:	`Stipple`	
Region: `0`	**Offset:**	`0x0000.81B0`
	Tag:	`0x36`
Read/write	**Reset Value:**	`Undefined`



Command register used to restore the line stipple counters and segment register after a task switch. The counters are incremented during a line stipple so the value read from them, via the read back path may not match the value loaded into them using this register.

LogicalOpMode

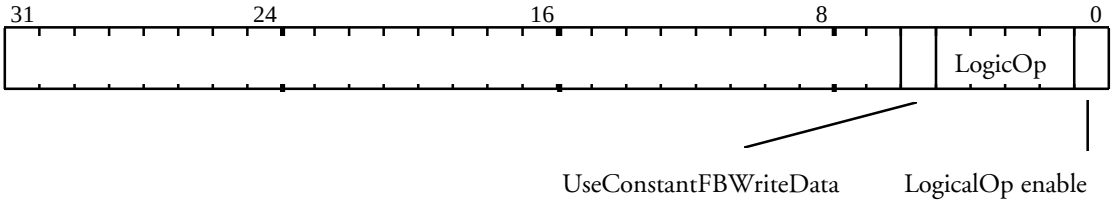
Name: Logic Op Mode

Unit: `Logic Op`

Region: `0` **Offset:** `0x0000.8828`

Tag: `0x105`

Read/write **Reset Value:** `Undefined`



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 (Bit0 set 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	NOOP	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

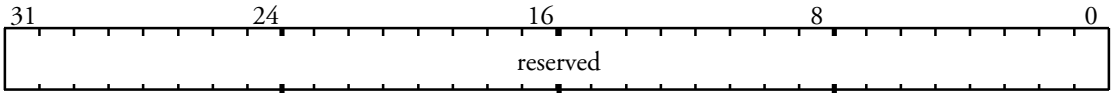
Where: S = Source (fragment) color, D = Destination (framebuffer) color.

Bit5 **UseConstantFBWriteData:**

0 = Variable
1 = Constant

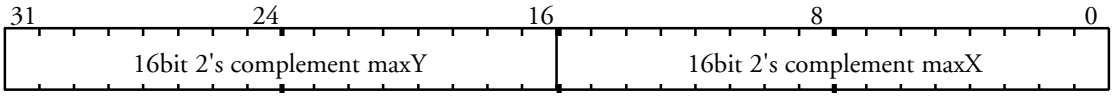
MaxHitRegion

Name:	Max Hit Region	
Unit:	`Host Out`	
Region: `0`	**Offset:**	`0x0000.8C30`
	Tag:	`0x186`
Write	**Reset Value:**	`Undefined`



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.

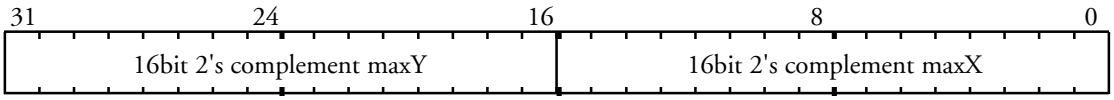
The format of the data output is:



The corresponding tag value output is: `0x186`

MaxRegion

Name:	Max Region	
Unit:	`Host Out`	
Region: `0`	**Offset:**	`0x0000.8C18`
	Tag:	`0x183`
Read/write	**Reset Value:**	`Undefined`



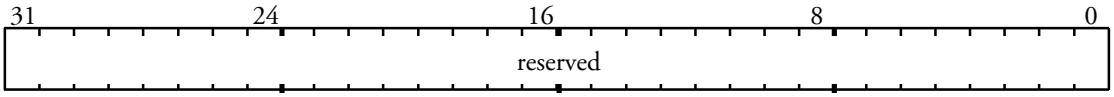
This register has two uses:

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 GLINT during rendering, and so if read back, will not necessarily be the same as when originally stored.

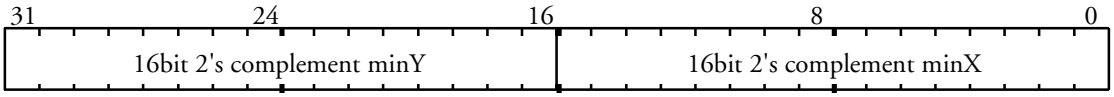
MinHitRegion

Name:	Min Hit Region	
Unit:	`Host Out`	
Region: `0`	**Offset:**	`0x0000.8C28`
	Tag:	`0x185`
Write	**Reset Value:**	`Undefined`



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.

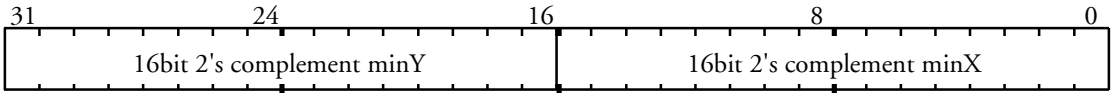
The format of the data output is:



The corresponding tag value output is: `0x185`

MinRegion

Name:	Min Region	
Unit:	`Host Out`	
Region: `0`	**Offset:**	`0x0000.8C10`
	Tag:	`0x182`
Read/write	**Reset Value:**	`Undefined`



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 GLINT during rendering, and so if read back, will not necessarily be the same as when originally stored.

PatternRamMode

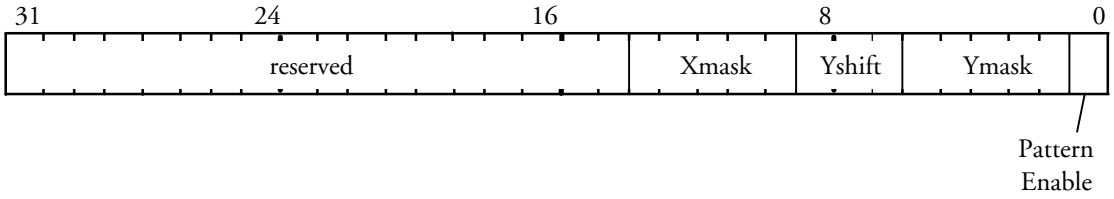
Name: Pattern RAM Mode

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8AF8

Tag: 0x15F

Read/Write **Reset Value:** Undefined



Holds the mode information when the Pattern RAM is used as the source of data during a span operation.

Bit0 **PatternEnable:**
0 = Disable
1 = Enable

Bit1-5 **Y mask:**

Bit6-8 **Y shift:**

Bit9-13 **X mask:**

PatternRamData[0…31]

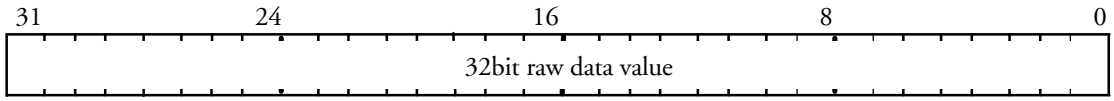
Name: Pattern RAM Data

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8B00, ..., 0000.8BF8

Tag: 0x160, ..., 0x17F

Read/Write **Reset Value:** Undefined



Holds the Pattern RAM data used as the source of data during a span operation.

PickResult

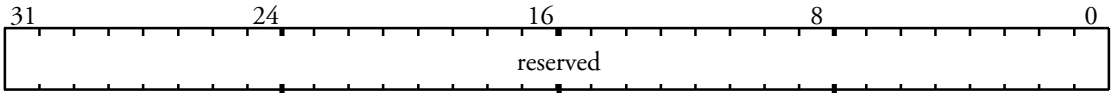
Name: Pick Result

Unit: Host Out

Region: 0 **Offset:** 0x0000.8C38

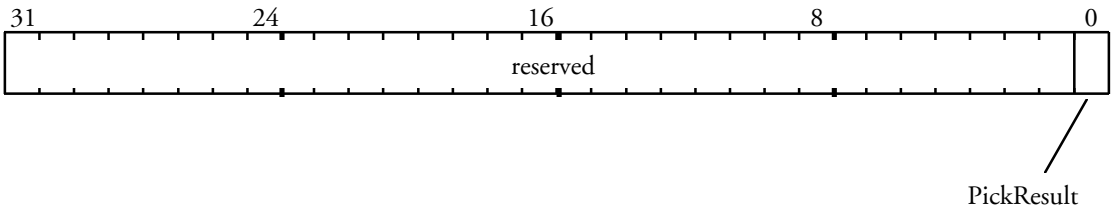
Tag: 0x187

Write **Reset Value:** Undefined



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 format of the data output is:



The corresponding tag value output is: 0x187

Bit0 **PickResult:**

0 = No hit

1 = Hit has occurred

PixelSize

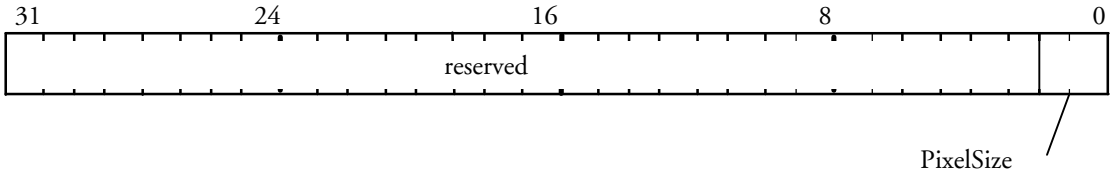
Name: Pixel Size

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.80C0

Tag: 0x18

Read/write **Reset Value:** Undefined



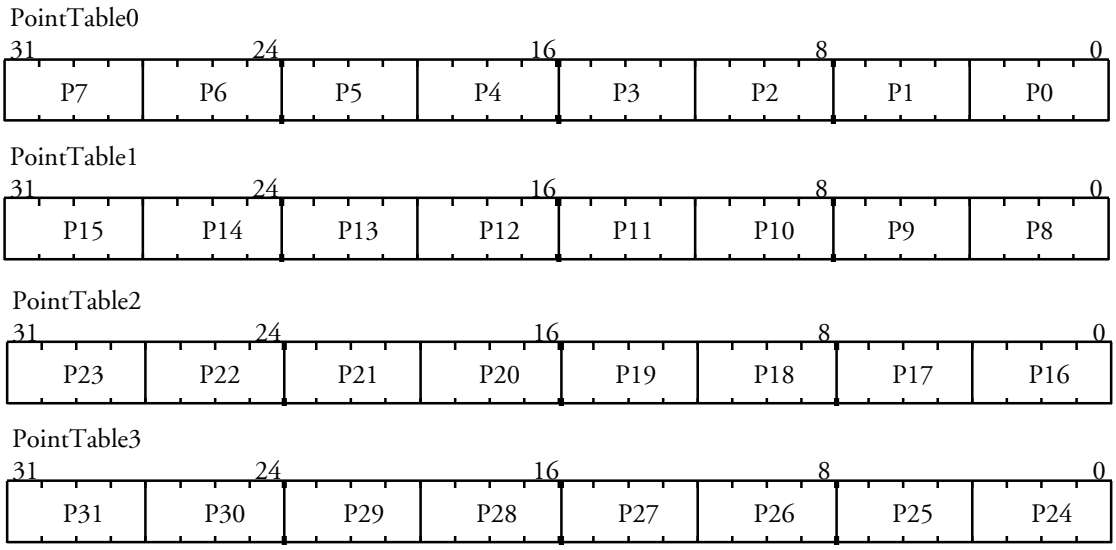
Configures the pixel depth to be used for rendering.

Bit0-1 PixelSize:

0 = 32bpp
1 = 16bpp
2 = 8bpp

PointTable[0...3]

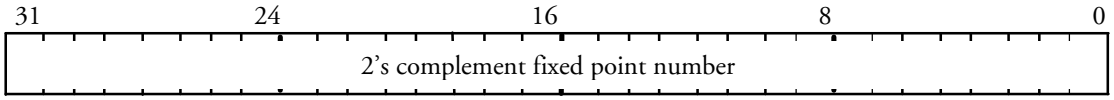
Name:	Point Table	
Unit:	`Rasterizer`	
Region: `0`	**Offset:**	`0x0000.8080, ..., 0x0000.8098`
	Tag:	`0x10, ..., 0x13`
Read/write	**Reset Value:**	`Undefined`



Antialiased point data table. The delta values in the table are held as 1 bit integer and 3 bits fraction. From the host's view the table is organized as four 32 bit words so the overhead of downloading when the point size changes is minimal. Only the parts of the table needed for a particular point size need to be loaded.

QStart

Name:	Q Start Value	
Unit:	`Texture`	
Region: `0`	**Offset:**	`0x0000.83B8`
	Tag:	`0x77`
Read/write	**Reset Value:**	`Undefined`



Initial Q value for texture map. The value is in 2's complement fixed point format. The binary point is at an arbitrary location, but must be consistent for all S, T and Q values.

RasterizerMode

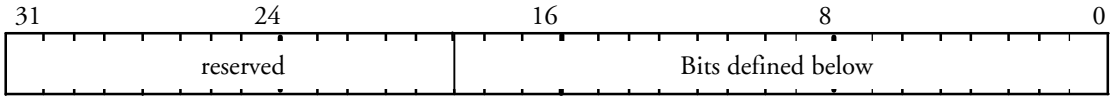
Name: Rasterizer Mode

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.80A0`

Tag: `0x14`

Read/write **Reset Value:** `Undefined`



Defines the long term mode of operation of the rasterizer.

Bit0 **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.

Bit1 **InvertBitMask** When this bit is set the bitmask is inverted first before being tested.

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 affected.

0 = Zero is added,
1 = Half is added
2 = Nearly half is added, i.e. `0x7FFF`

Bit7-8 **BitMaskByteSwapMode** These bits control byte swapping of the BitMask data. If the bytes are ABCD on input, then the swap will leave them as:

0 = ABCD (i.e. no byte swap)
1 = BADC
2 = CDAB
3 = DCBA

Bit9 **BitMaskPacking** This bit controls whether the BitMask data is packed, or if new BitMask data is required on every scanline.

0 = BitMask data is packed
1 = BitMask data is provided for each scanline

Bit10-14 BitMaskOffset

Bit15-16 HostDataByteSwapMode These bits control byte swapping of the BitMask data. If the bytes are ABCD on input, then the swap will leave them as:

0 = ABCD (i.e. no byte swap)
1 = BADC
2 = CDAB
3 = DCBA

Bit17 MultiGLINT Enables mode where GLINT only processes those scanlines allocated to it in a scanline interleaved multi-GLINT system.

0 = Single GLINT mode
1 = Multi-GLINT mode

Bit18 YLimitsEnable This bit enables Ylimits testing as dictated by the **Ylimits** register.

Render

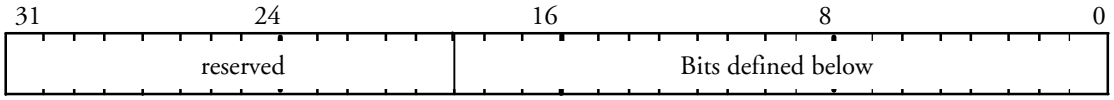
Name: Render

Unit: `Rasterizer`

Region: `0` **Offset:** `0x0000.8038`

Tag: `0x7`

Write **Reset Value:** `Undefined`



Command to start the rendering process.

The data field defines the short term modes required by this primitive. For details see Table 0.1 Command Register Descriptions .

Bit0 **AreaStippleEnable**. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur.
0 = Disable
1 = Enable

Bit1 **LineStippleEnable**. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur.
0 = Disable
1 = Enable

Bit2 **ResetLineStipple**. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit.
0 = Disable
1 = Enable

Bit3 **FastFillEnable** The type of span filling is specified in the SpanOperation field
0 = Disable
1 = Enable

Bit6-7 **PrimitiveType** These bits indicate the type of GLINT primitive to be drawn. The primitives supported and the corresponding codes are:
0 = lines,
1 = trapezoids,
2 = points.

Bit8 **AntialiasEnable**
0 = Disable
1 = Enable

Bit9 **AntialiasingQuality**
0 = 4x4,

1 = 8x8.

Bit10 **UsePointTable**
0 = Disable
1 = Enable

Bit11 **SyncOnBitMask**
0 = Disable
1 = Enable

Bit12 **SyncOnHostData**
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

Bit15 **CoverageEnable**. Note that the Antialiasing Unit must be suitably enabled as well.
0 = Disable
1 = Enable

Bit16 **SubPixelCorrectionEnable**
0 = Disable
1 = Enable

Bit18 **SpanOperation**
0 = Use constant color from **FBBlockColorU**, **FBBlockColorL** registers
1 = Use data from host (SynchOnHostData set) or from framebuffer

ResetPickResult

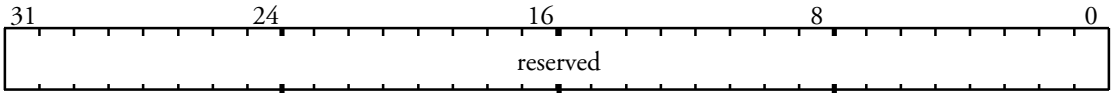
Name: Reset Pick Result

Unit: Host Out

Region: 0 **Offset:** 0x0000.8C20

Tag: 0x184

Write **Reset Value:** Undefined



This command causes the current value of the picking result to be reset to zero. The data field is not used.

RouterMode

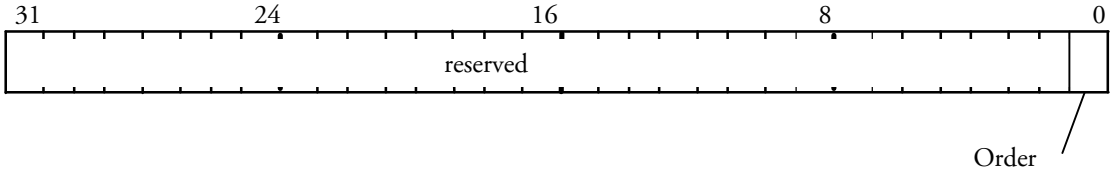
Name: Router Mode

Unit: Router

Region: 0 **Offset:** 0x0000.8840

Tag: 0x108

Read/write **Reset Value:** Undefined



Switches the order of some units in the graphics hyperpipeline.

Bit0 **Order:**

0 = TextureDepth
1 = DepthTexture

RStart

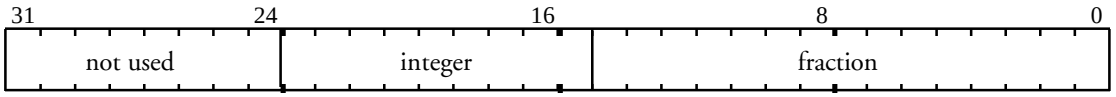
Name: Initial Color - Red

Unit: `Color DDA`

Region: `0` **Offset:** `0x0000.8780`

Tag: `0xF0`

Read/write **Reset Value:** `Undefined`



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 9.15 fixed point format.

SaveLineStippleCounters

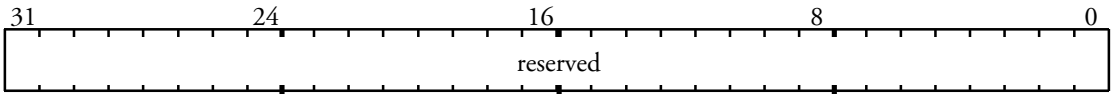
Name: Save Line Stipple State

Unit: `Stipple`

Region: `0` **Offset:** `0x0000.81C0`

Tag: `0x38`

Write **Reset Value:** `Undefined`



Command Register. Copies the current counter values into an internal register for later restoration using the **UpdateLineStippleCounters** command. Useful in drawing stippled wide lines.

ScanLineOwnership

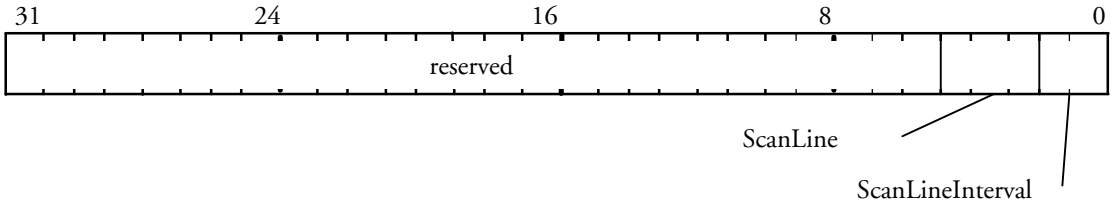
Name: Scanline Ownership

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.80B0

Tag: 0x16

Read/write **Reset Value:** Undefined



Controls which scanlines a GLINT owns in a multi-GLINT system. This register only has an effect if the MultiGLINT bit is set in the **RasterizerMode** register.

Bit0-1 **ScanLineInterval:**

0 = 1
1 = 2
2 = 4
3 = 8

Bit2-4 **Scanline:** Determines which scanline is owned by this GLINT processor.

ScissorMaxXY

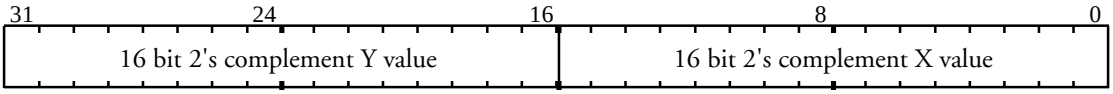
Name: Scissor Rectangle - Maximum XY

Unit: `Scissor`

Region: `0` **Offset:** `0x0000.8190`

Tag: `0x32`

Read/write **Reset Value:** `Undefined`



User scissor rectangle corner with the most positive coordinates relative to the screen origin.

ScissorMinXY

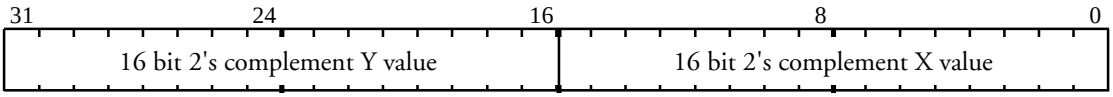
Name: Scissor Rectangle - Minimum XY

Unit: `Scissor`

Region: `0` **Offset:** `0x0000.8188`

Tag: `0x31`

Read/write **Reset Value:** `Undefined`



User scissor rectangle corner with the least positive coordinates relative to the screen origin.

ScissorMode

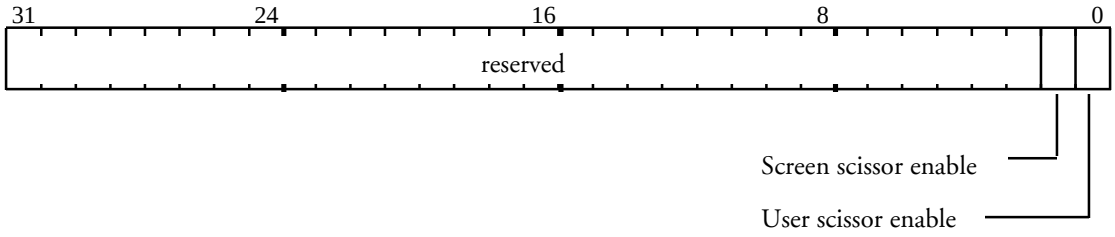
Name: Scissor Mode

Unit: Scissor

Region: 0 **Offset:** 0x0000.8180

Tag: 0x30

Read/write **Reset Value:** Undefined



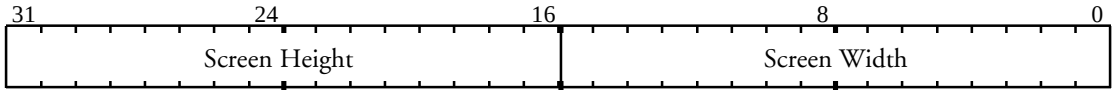
Controls enabling of the screen and user scissor tests.

Bit0 **User Scissor Enable:**
0 = Disable
1 = Enable

Bit1 **Screen Scissor Enable:**
0 = Disable
1 = Enable

ScreenSize

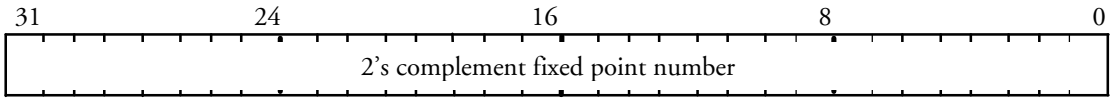
Name:	Screen Size	
Unit:	`Scissor`	
Region: `0`	**Offset:**	`0x0000.8198`
	Tag:	`0x33`
Read/write	**Reset Value:**	`Undefined`



Screen dimensions for screen scissor clip. The screen boundaries are (0, 0) to (width - 1, height - 1) inclusive.

SStart

Name:	S Start Value	
Unit:	`Texture`	
Region: `0`	**Offset:**	`0x0000.8388`
	Tag:	`0x71`
Read/write	**Reset Value:**	`Undefined`



Initial S value for texture map. The value is in 2's complement fixed point format. The binary point is at an arbitrary location, but must be consistent for all S, T and Q values.

StartXDom

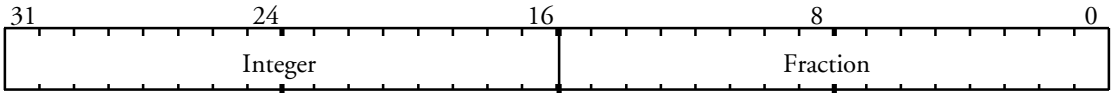
Name: Start X Value - Dominant Edge

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8000

Tag: 0x0

Read/write **Reset Value:** Undefined



Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing. The value is in 2's complement 16.16 fixed point format.

StartXSub

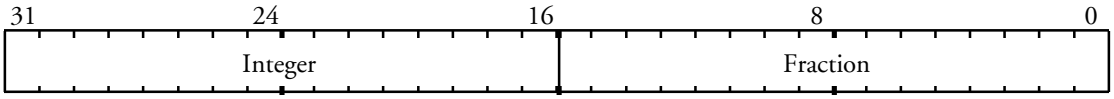
Name: Start X Value - Subordinate Edge

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.8010

Tag: 0x2

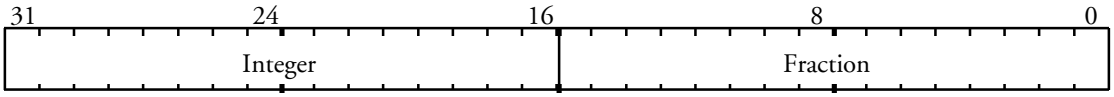
Read/write **Reset Value:** Undefined



Initial X value for the subordinate edge in trapezoid filling. The value is in 2's complement 16.16 fixed point format.

StartY

Name:	Start Y Value	
Unit:	`Rasterizer`	
Region: `0`	**Offset:**	`0x0000.8020`
	Tag:	`0x4`
Read/write	**Reset Value:**	`Undefined`



Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing. The value is in 2's complement 16.16 fixed point format.

StatisticMode

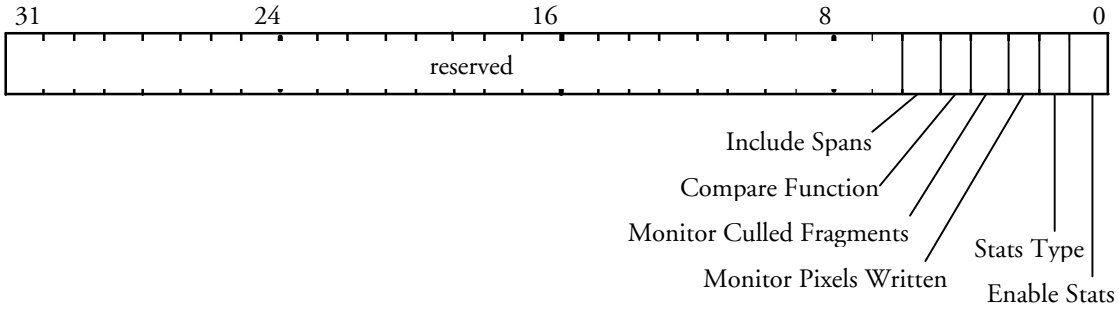
Name: Statistic Mode

Unit: Host Out

Region: 0 **Offset:** 0x0000.8C08

Tag: 0x181

Read/write **Reset Value:** Undefined



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 **MonitorPixelsWritten:**
0 = Excludes Pixels that were drawn
1 = Includes Pixels that were drawn

Bit3 **MonitorCulledFragments:**
0 = Excludes fragments that were culled from being drawn
1 = Includes fragments that were culle d from being drawn

Bit4 **CompareFunction:**
0 = Inside region
1 = Outside region

Bit5 **Include Spans:**
0 = Exclude
1 = Include

Stencil

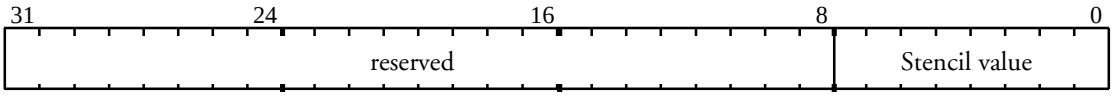
Name: Stencil

Unit: `Stencil`

Region: `0` **Offset:** `0x0000.8998`

Tag: `0x133`

Read/write **Reset Value:** `Undefined`



Set to 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

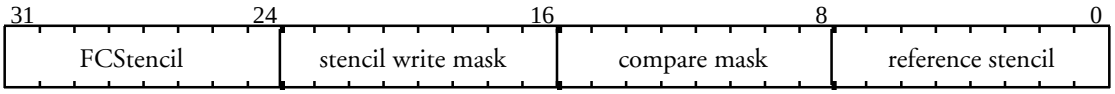
Name: Stencil Data

Unit: `Stencil`

Region: `0` **Offset:** `0x0000.8990`

Tag: `0x132`

Read/write **Reset Value:** `Undefined`



Holds data used in the stencil test.

The stencil writemask controls which stencil planes are updated as a result of the test.

Bit0-7 **Reference Stencil** is the reference value for the stencil test.

Bit8-15 **Compare Mask** is the mask used to determine which bits are significant in the comparison.

Bit16-23 **Stencil Writemask** is the mask used to determine which bits in the localbuffer are updated.

Bit24-31 **FCStencil** is the value written to the localbuffer when using the FrameCount plane clear mechanism.

If the stencil width is 4bits then the value is left justified.

StencilMode

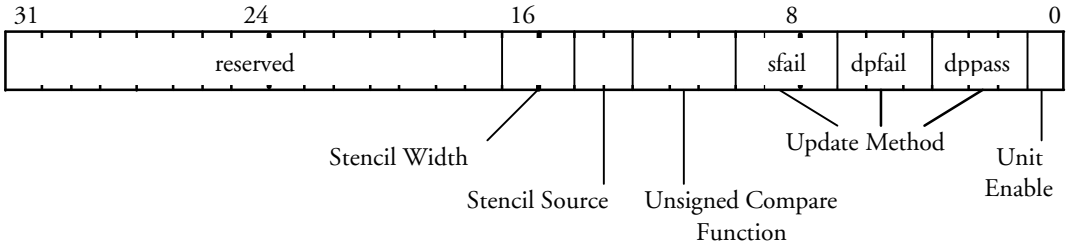
Name: Stencil Mode

Unit: Stencil

Region: 0 **Offset:** 0x0000.8988

Tag: 0x131

Read/write **Reset Value:** Undefined



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

Bit15 Stencil Width:

0 = 4 bits
1 = 8 bits
2 = 1 bit

SuspendUntilFrameBlank

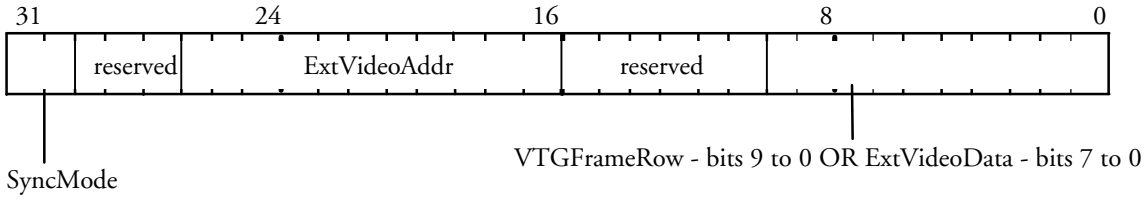
Name: Suspend Until Frame Blank

Unit: Framebuffer R/W

Region: 0 **Offset:** 0x0000.8C78

Tag: 0x18f

Write **Reset Value:** Undefined



NOTE this command is not available in a multi-GLINT 500TX system.

This command causes all outstanding writes to be flushed to the framebuffer, and then any succeeding reads or writes to be suspended until the next vertical frame blank has occurred. The SyncMode data field indicates how the buffer swap is to be controlled.

a) ExtVideoWaitForFB - Wait for the next Vertical frameblank, then write the ExtVideoData field to the address given in ExtVideoAddr.

b) ExtVideoImmediate - As a) but update the register at ExtVideoAddr immediately without waiting for frameblank.

c) FrameRowWaitForFB - Wait for frameblank, then write the new FrameRow address given in the VTGFrameRow field to the GLINT *VTGFrameRowAddr* register.

d) FrameRowImmediate - update the *VTGFrameRowAddr* register immediately without waiting for frameblank.

Bit0-7 **ExtVideoData:** data value to be written to external video register.

OR

Bit0-9 **VTGFrameRow:** new frame row address to be written to the GLINT *VTGFrameRowAddr* register.

Bit16-26 **ExtVideoAddr:** Address of external video register to be updated. Can be derived from (PCI byte address - PCI byte address of the base register in group)/8.

Bit30-31 **SyncMode:**

0 = Wait for Frameblank & update external video register
1 = Update external video register immediately
2 = Wait for Frameblank then update *VTGFrameRowAddr* register
3 = Update *VTGFrameRowAddr* register immediately

Sync

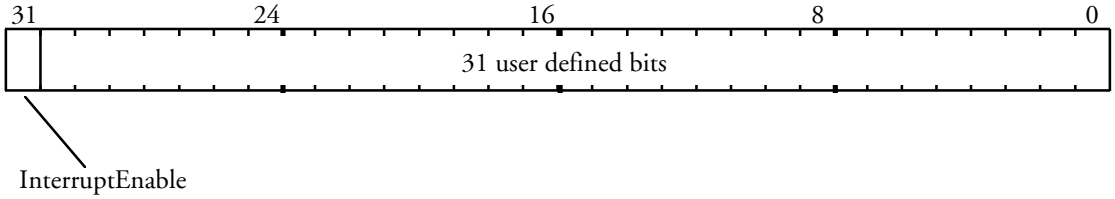
Name: Synchronization

Unit: `Host Out`

Region: `0` **Offset:** `0x0000.8C40`

Tag: `0x188`

Write **Reset Value:** `Undefined`



This command can be used to synchronize GLINT with the host. It is also used to flush outstanding GLINT 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`

Texel[0...7]

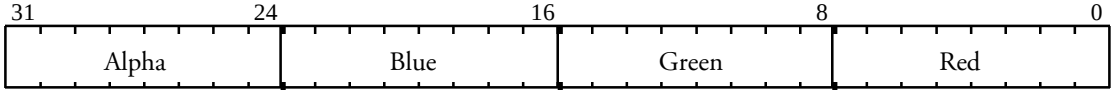
Name: Texel Values

Unit: `Texture`

Region: `0` **Offset:** `0x0000.8600, ..., 0x0000.8638`

Tag: `0xC0, ..., 0xC7`

Read/write **Reset Value:** `Undefined`



Texel values for interpolation.

These registers are only present for backwards compatibility with the GLINT 300SX. They should not be used for new code written for the GLINT 500TX.

TexelLUT[0…15]

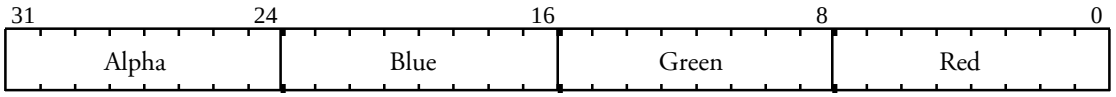
Name: Texel LUT Values

Unit: Texture

Region: 0 **Offset:** 0x0000.8E80, ..., 0x0000.8EF8

Tag: 0x1D0, ..., 0x1DF

Read/write **Reset Value:** Undefined



Texel LUT values for substitution in indexed textures.

TextureAddressMode

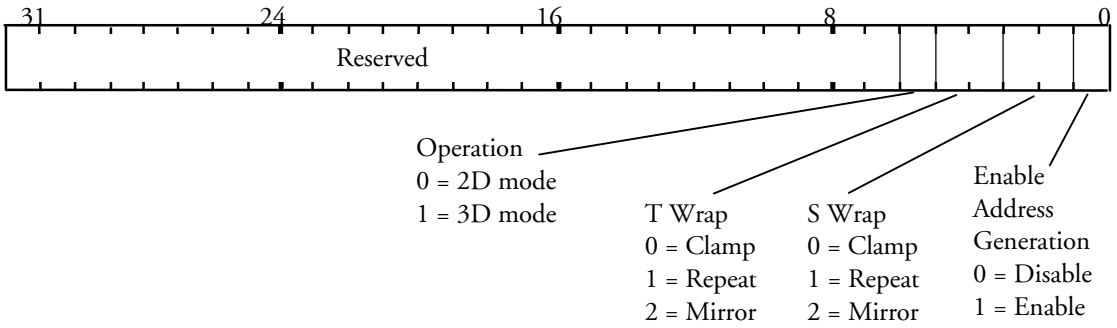
Name: Texture Address Mode

Unit: `Texture`

Region: `0` **Offset:** `0x0000.8380`

Tag: `0x70`

Read/write **Reset Value:** `Undefined`



Provides overall control of the generation of texel addresses.

Bit0 **Texture Address Generation:**
0 = Disable
1 = Enable

Bit1-2 **S Wrap:**
0 = Clamp
1 = Repeat
2 = Mirror

Bit3-4 **T Wrap:**
0 = Clamp
1 = Repeat
2 = Mirror

Bit5 **Operation:**
0 = 2D mode
1 = 3D mode

TextureCacheControl

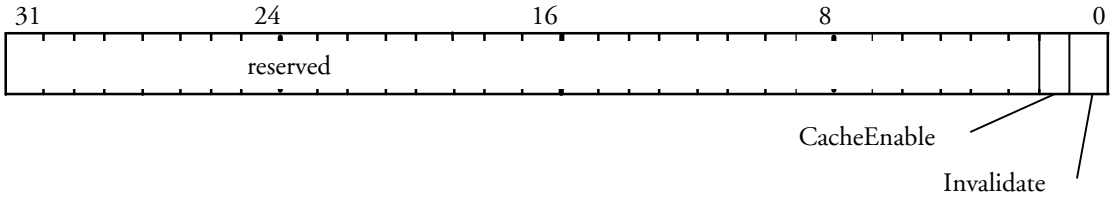
Name: Texture Cache Control

Unit: Texture

Region: 0 **Offset:** 0x0000.8490

Tag: 0x92

Read/write **Reset Value:** Undefined



Allows control of the texture cache operation.

Upon reset the cache is disabled and marked as invalid, however, reading back this register immediately after reset will return undefined values.

The Invalidate bit will always return undefined data when read.

Bit0 **Invalidate:**
0 = No Invalidate
1 = Invalidate Cache

Bit1 **CacheEnable:**
0 = Disable Cache
1 = Enable Cache

TextureColorMode

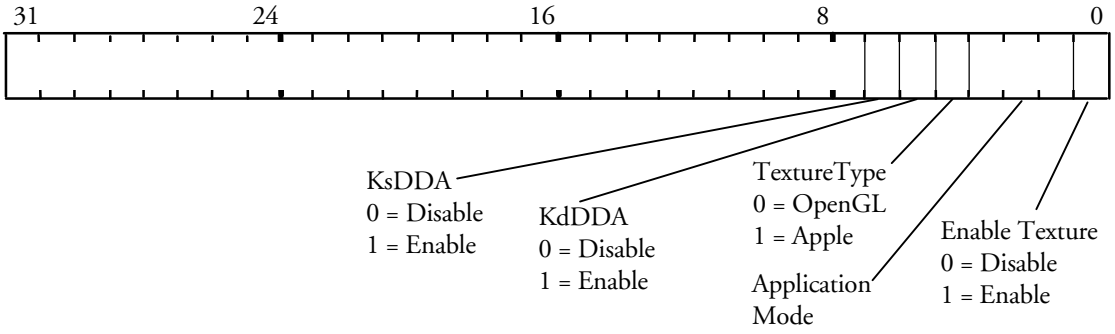
Name: Texture Color Mode

Unit: `Texture`

Region: `0` **Offset:** `0x0000.8680`

Tag: `0xD0`

Read/write **Reset Value:** `Undefined`



Controls the application of texture.

Note that 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

EITHER Where Texture Type is OpenGL:

Bit1-3 **Application Mode:**
0 = Modulate
1 = Decal
2 = Blend

OR Where Texture Type is QuickDraw3D:

Bit1 **DecalEnable:**
0 = Disable
1 = Enable

Bit2 **ModulateEnable:**
0 = Disable
1 = Enable

Bit3 **HighlightEnable:**
0 = Disable
1 = Enable

Bit4 **Texture Type:**
0 = OpenGL

1 = QuickDraw3D

Bit5 **KdDDA:**

0 = Disable
1 = Enable

Bit6 **KsDDA:**

0 = Disable
1 = Enable

TextureData

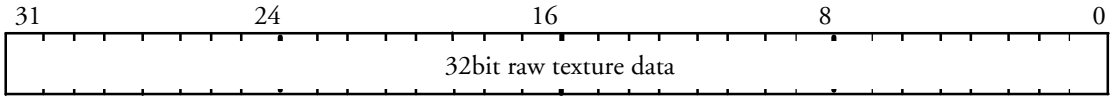
Name: Texture Data

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.88E8`

Tag: `0x11D`

Write **Reset Value:** `Undefined`



32bit raw texture value, formatted to match the format that will be used when the texture is read back from the localbuffer. May include multiple texels (depending on the texel depth), in which case the order of texels within the register will depend on factors such as the byte swap mode, as defined in the **TextureFormat** register when the texture is subsequently read.

TextureDownloadOffset

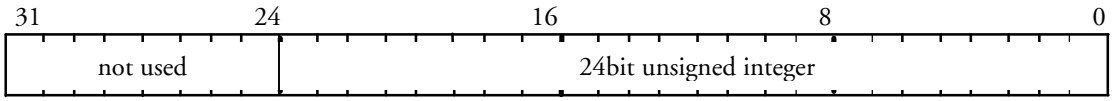
Name: Texture Download Offset

Unit: `Localbuffer R/W`

Region: `0` **Offset:** `0x0000.88F0`

Tag: `0x11E`

Read/write **Reset Value:** `Undefined`



24bit unsigned integer address offset.

Note that when read, the value may have been incremented if any texture download has occurred.

TextureEnvColor

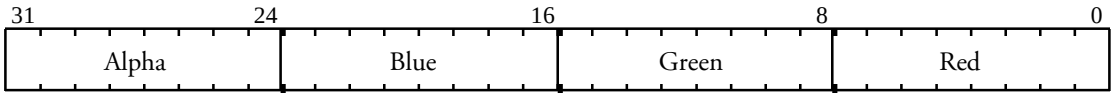
Name: Texture Environment Color

Unit: Texture

Region: 0 **Offset:** 0x0000.8688

Tag: 0xD1

Read/write **Reset Value:** Undefined



Constant color value used in blend texturing mode.

TextureFilter

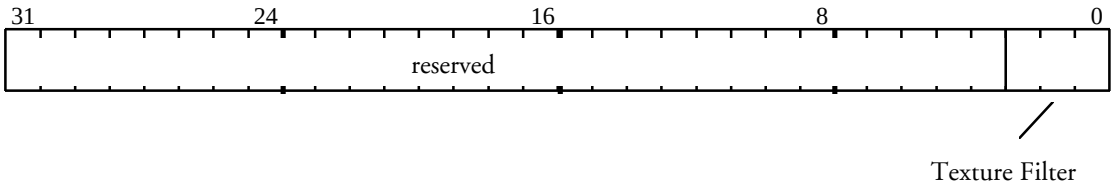
Name: Texture Filter

Unit: Texture

Region: 0 **Offset:** 0x0000.8668

Tag: 0xCD

Read/write **Reset Value:** Undefined



Controls the texture filter mode.

Bit0-2 **Texture Filter:**

0 = Nearest
1 = Linear2
2 = Trilinear4
3 = Linear4
4 = Trilinear8

TextureFormat

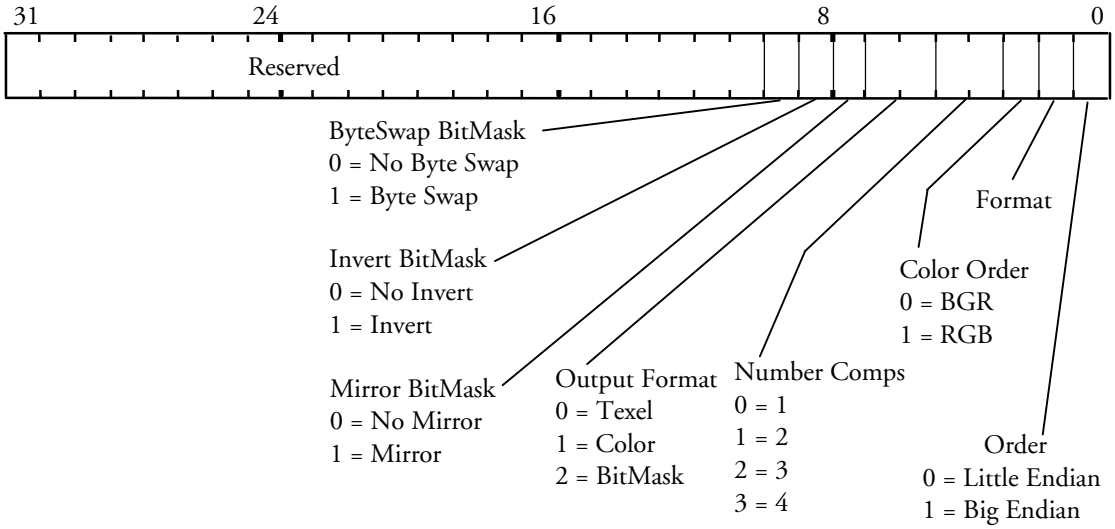
Name: Texture Data Format

Unit: Texture

Region: 0 **Offset:** 0x0000.8488

Tag: 0x91

Read/write **Reset Value:** Undefined



Controls the reading of texture maps.

Bit0 **Order:**
0 = Little Endian
1 = Big Endian

Bit1 **Format:**
0 = 5:6:5 format at 16bpp
1 = 5:5:5 format at 16bpp

Bit2 **Color Order:**
0 = BGR
1 = RGB

Bit3-4 **Number Of Components:**
0 = 1
1 = 2
2 = 3
3 = 4

Bit5-6 **Output Format:**
0 = Texel
1 = Color
2 = BitMask

Bit7 **Mirror BitMask:**
0 = No Mirror
1 = Mirror

Bit8 **Invert BitMask:**
0 = No Invert
1 = Invert

Bit9 **Byte Swap BitMask:**
0 = No Byte Swap
1 = Byte Swap

TextureReadMode

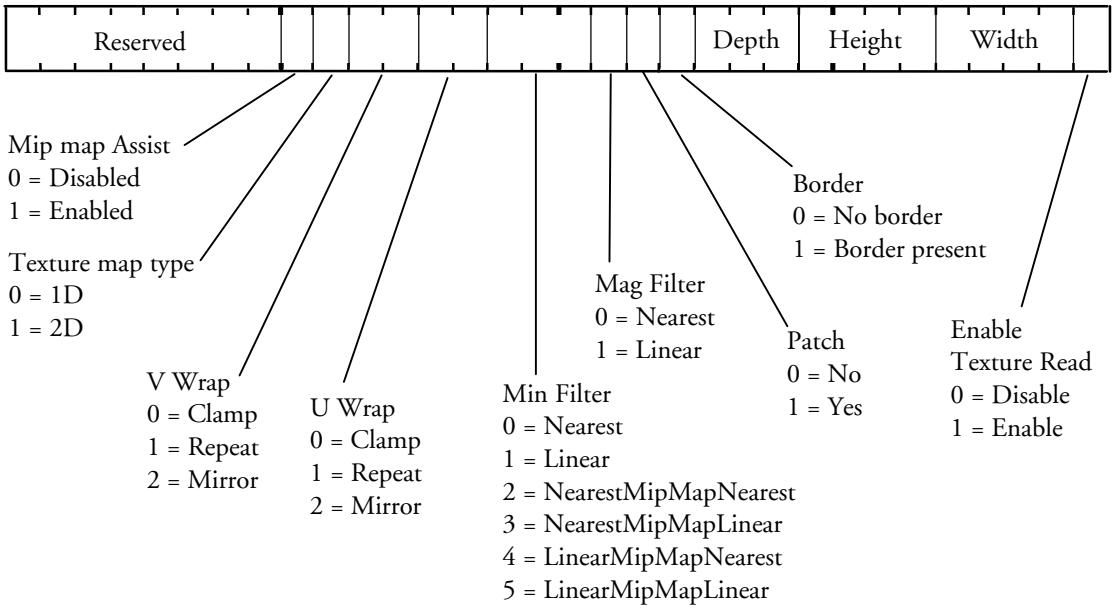
Name: Texture Read Mode

Unit: `Texture`

Region: `0` **Offset:** `0x0000.8480`

Tag: `0x90`

Read/write **Reset Value:** `Undefined`



Controls the reading of texture maps.

Bit0 **Texture Read Enable:**
0 = Disable
1 = Enable

Bit1-4 **Width:** As a power of 2

Bit5-8 **Height:** As a power of 2

Bit9-11 **Depth:** As a power of 2

Bit12 **Border:**
0 = No Border
1 = Border Present

Bit13 **Patch:**
0 = No
1 = Yes

Bit14 **Mag Filter:**
0 = Nearest
1 = Linear

Bit15-17 Min Filter:
0 = Nearest
1 = Linear

Bit18-19 U Wrap:
0 = Clamp
1 = Repeat
2 = Mirror

Bit20-21 V Wrap:
0 = Clamp
1 = Repeat
2 = Mirror

Bit22 Texture Map Type:
0 = 1D
1 = 2D

Bit23 Mip Map Assist:
0 = Disable
1 = Enable

TStart

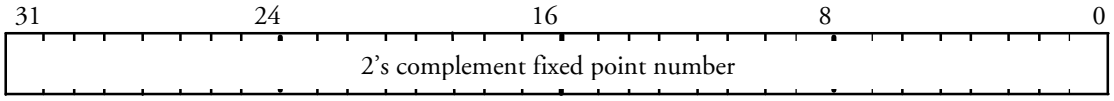
Name: T Start Value

Unit: Texture

Region: 0 **Offset:** 0x0000.83A0

Tag: 0x74

Read/write **Reset Value:** Undefined



Initial T value for texture map. The value is in 2's complement fixed point format. The binary point is at an arbitrary location, but must be consistent for all S, T and Q values.

TxBaseAddr

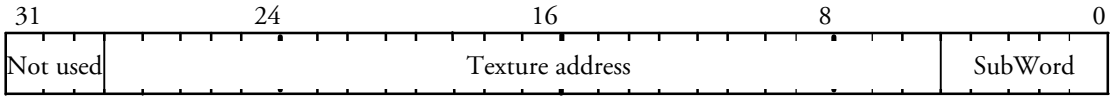
Name: Texture Base Address

Unit: Texture

Region: 0 **Offset:** 0x0000.8500

Tag: 0xA0

Read/write **Reset Value:** Undefined



29bit base address of the texture map. Lower 5bits specify address within a word.

TxBaseAddrLR

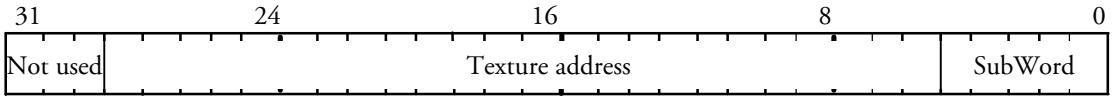
Name: Texture Base Address Low Resolution

Unit: Texture

Region: 0 **Offset:** 0x0000.8508

Tag: 0xA1

Read/write **Reset Value:** Undefined



29bit lower resolution address used for mipmap hardware assistance. Lower 5bits specify address within a word.

UpdateLineStippleCounters

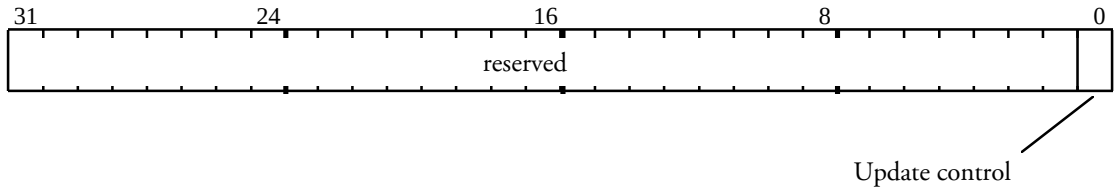
Name: Update Line Stipple Counters

Unit: Stipple

Region: 0 **Offset:** 0x0000.81B8

Tag: 0x37

Write **Reset Value:** Undefined



Command Register. Restores the internally saved stipple counter values saved by the **SaveLineStippleState** command. Useful in drawing stippled wide lines.

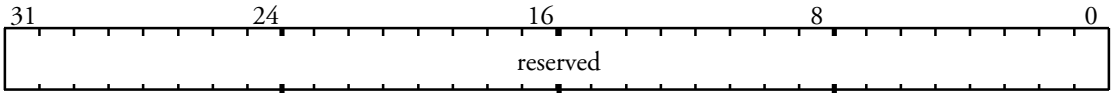
Bit0 **Update counter control:**

0 = Reset counters to zero.

1 = Load counters from segment register.

WaitForCompletion

Name:	Wait For Completion	
Unit:	Rasterizer	
Region: 0	**Offset:**	0x0000.80B8
	Tag:	0x17
Write	**Reset Value:**	Undefined



Command Register. Used to flush all reads and writes to the framebuffer prior to the start of rendering for the next primitive. Useful to separate say a texture download from the surrounding primitives.

Window

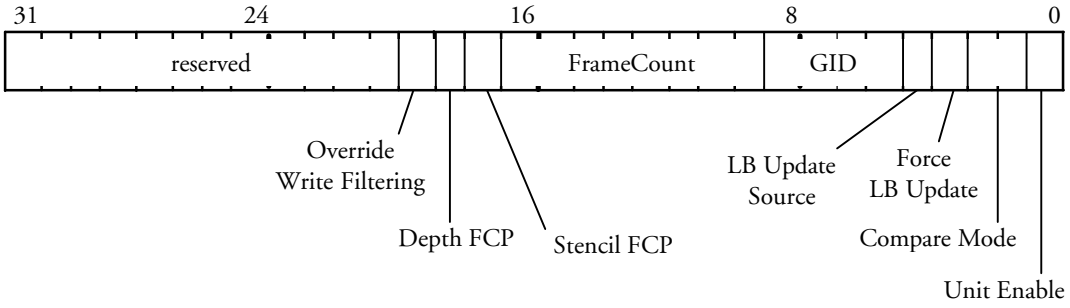
Name: Window

Unit: `Pixel Ownership`

Region: `0` **Offset:** `0x0000.8980`

Tag: `0x130`

Read/write **Reset Value:** `Undefined`



1) Used to set the value and comparison mode for the pixel ownership (GID) test. If the test fails then the fragment will be culled from being drawn.

   If the unit is disabled then it is as if the GID test always passes.

   If the Force LB Update bit is set, this overrides all the tests done in the GID, Stencil and Depth units, and the per unit enables, to force the localbuffer to be updated. However, writes must still be enabled in the **LBWriteMode** register. When this bit is clear any update is conditional on the outcome of the GID, stencil and depth tests.

2) The FrameCount is an eight bit field which is compared with the FrameCount read from the localbuffer. If these are not equal then the fast clear mechanism can be used, however how this is used (if at all) is determined by the Depth FCP and Stencil FCP bits. If these bit(s) are set then the fast clear function is enabled for the corresponding field(s).

Bit0 **Unit Enable:**

0 = Disable
1 = Enable

Bit1-2 **Compare Mode:**

0 = Always Pass
1 = Never Pass
2 = Pass if Equal
3 = Pass if Not Equal

Bit3 **Force LB Update:**

0 = Not Forced

1 = Forced

Bit4 **LB Update Source:**
0 = LBSourceData
1 = Registers

Bit5-8 **GID** to be compared against

Bit9-16 **FrameCount** value for use in fast clears

Bit17 **Stencil FCP:**
0 = Disable
1 = Enable

Bit18 **Depth FCP:**
0 = Disable
1 = Enable

Bit19 **Override Write Filtering:**
0 = No
1 = Yes

WindowOrigin

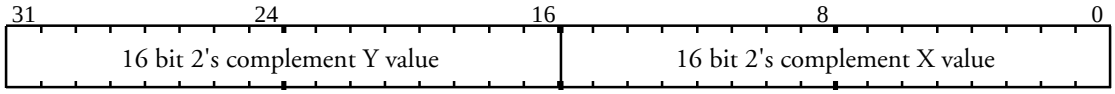
Name: Window Origin

Unit: Scissor

Region: 0 **Offset:** 0x0000.81C8

Tag: 0x39

Read/write **Reset Value:** Undefined



As each fragment is generated by the unit, this origin is added to the coordinates of the fragment to generate its screen coordinates, prior to doing the screen scissor test.

YLimits

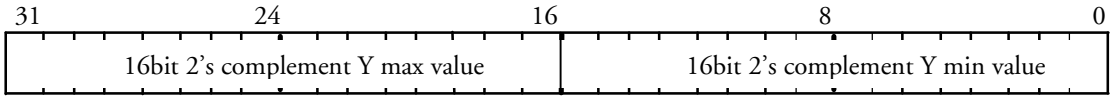
Name: Y Limits

Unit: Rasterizer

Region: 0 **Offset:** 0x0000.80A8

Tag: 0x15

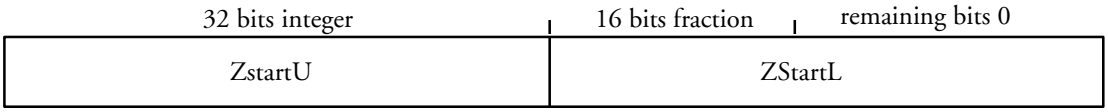
Read/write **Reset Value:** Undefined



Defines the Y extent that the rasterizer should fill between.

ZStartU

Name:	Depth Start Value	
Unit:	`Depth`	
Region: `0`	**Offset:**	`0x0000.89B8, 0x0000.89B0`
	Tag:	`0x137, 0x136`
Read/write	**Reset Value:**	`Undefined`



This pair of registers set the start value for depth interpolation. **ZStartU** holds the most significant bits, and **ZStartL** the least significant bits. The value is in 2's complement 32.16 fixed point format.

1.

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 GLINT in which each 32 bit register is represented as a C structure, potentially split into a series of bit fields. Where in an example 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. This style has been chosen for clarity; there are often more efficient strategies.

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 ¶ 0

The constant definitions and register bit field definitions are based upon those used in the 3Dlabs driver software. Sources including header files are available under source license agreement.

Loading of a GLINT 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 an aliased trapezoid.

render.AreaStippleEnable = GLINT_DISABLE
render.LineStippleEnable = GLINT_DISABLE
render.PrimitiveType = GLINT_TRAPEZOID
render.FastFillEnable = GLINT_DISABLE
render.FastFillIncrement = don't care
render.UsePointTable = GLINT_FALSE
render.AntialiasEnable = GLINT_DISABLE
render.AntialiasingQuality = don't care
render.ResetLineStipple = GLINT_FALSE
render.SyncOnBitMask = GLINT_FALSE
```

```
render.SyncOnHostData = GLINT_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 variable 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. `GLINT_TRUE`[1] , 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 are not readily available, these will typically set *`as appropriate`*.

In some fragments, simply a list of register updates 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 ) ;
```

[1] In the C header files glintreg.h glintdef.h macros and types are generally prefixed by a double underbar, i.e. __GLINT_TRUE and __GlintRenderFmat to avoid name space clashes with other code. In the pseudocode the __ is omitted for clarity.

1.

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 **LBReadMode** register. The range of widths supported by this technique are tabulated below, together with the values for each of the PP fields.

Width	PP2	PP1	PP0
32	0	0	1
64	0	0	2
96	0	1	2
128	0	0	3
160	0	1	3
192	0	2	3
224	1	2	3
256	0	0	4
288	0	1	4
320	0	2	4
352	1	2	4
384	0	3	4
416	1	3	4
448	2	3	4
512	0	0	5
544	0	1	5
576	0	2	5
608	1	2	5
640	0	3	5
672	1	3	5
704	2	3	5
768	0	4	5
800	1	4	5
832	2	4	5
896	3	4	5
1024	0	0	6
1056	0	1	6
1088	0	2	6
1120	1	2	6
1152	0	3	6
1184	1	3	6
1216	2	3	6

1280	0	4	6
1312	1	4	6
1344	2	4	6
1408	3	4	6
1536	0	5	6
1568	1	5	6
1600	2	5	6
1664	3	5	6
1792	4	5	6
2048	0	0	7
2112	0	2	7
2144	1	2	7
2176	0	3	7
2208	1	3	7
2240	2	3	7
2304	0	4	7
2336	1	4	7
2368	2	4	7
2432	3	4	7
2560	0	5	7
2592	1	5	7
2624	2	5	7
2688	3	5	7
2816	4	5	7
3072	0	6	7
3104	1	6	7
3136	2	6	7
3200	3	6	7
3328	4	6	7
3584	5	6	7
4096	0	7	7

Table C.1

1.

Appendix D. Register Table

The following table lists registers by group, giving their tag values and indicating their type. The register groups may be used to improve data transfer rates to GLINT when using DMA.

The following types of register are distinguished:

- **Control:** Set state and control bits ready to draw a primitive.
- **Command:** Initiates drawing of a primitive.
- **Mixed:** A control register which may also be used to supply successive data values during image 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, and whether it is new or has changed in the GLINT 500TX as compared to the GLINT 300SX.

Unit	Register	Major Group (hex)	Offset (hex)	Type	Read/ Write	New (*) Diff. (>)
Rasterizer	StartXDom	00	0	Control	R/W	
	dXDom	00	1	Control	R/W	
	StartXSub	00	2	Control	R/W	
	dXSub	00	3	Control	R/W	
	StartY	00	4	Control	R/W	
	dY	00	5	Control	R/W	
	Count	00	6	Control	R/W	
	Render	00	7	Command	W	>
	ContinueNewLine	00	8	Command	W	
	ContinueNewDom	00	9	Command	W	
	ContinueNewSub	00	A	Command	W	
	Continue	00	B	Command	W	
	FlushSpan	00	C	Command	W	
	BitMaskPattern	00	D	Mixed	W	
Rasterizer	PointTable[0…3]	01	0…3	Control	R/W	
	RasterizerMode	01	4	Control	R/W	>
	YLimits	01	5	Control	R/W	*
	ScanLineOwnership	01	6	Control	R/W	*
	WaitForCompletion	01	7	Command	W	*
	PixelSize	01	8	Control	R/W	*
Scissor Stipple	ScissorMode	03	0	Control	R/W	
	ScissorMinXY	03	1	Control	R/W	

	ScissorMaxXY	03	2	Control	R/W	
	ScreenSize	03	3	Control	R/W	
	AreaStippleMode	03	4	Control	R/W	
	LineStippleMode	03	5	Control	R/W	
	LoadLineStippleCounters	03	6	Control	R/W	
	UpdateLineStippleCounters	03	7	Command	W	
	SaveLineStippleState	03	8	Command	W	
	WindowOrigin	03	9	Control	R/W	
Scissor Stipple	AreaStipplePattern[0…31]	04 05	0…F 0…F	Control	R/W	
Router	RouterMode	10	8	Control	R/W	*
Texture	TextureAddressMode	07	0	Control	R/W	*
	SStart	07	1	Control	R/W	*
	dSdx	07	2	Control	R/W	*
	dSdyDom	07	3	Control	R/W	*
	TStart	07	4	Control	R/W	*
	dTdx	07	5	Control	R/W	*
	dTdyDom	07	6	Control	R/W	*
	QStart	07	7	Control	R/W	*
	dQdx	07	8	Control	R/W	*
	dQdyDom	07	9	Control	R/W	*
	TextureReadMode	09	0	Control	R/W	*
	TextureFormat	09	1	Control	R/W	*
	TextureCacheControl	09	2	Control	R/W	*
	BorderColor	09	5	Control	R/W	*
	TxBaseAddr	0A	0	Control	R/W	*
	TxBaseAddrLR	0A	1	Control	R/W	*
	TexelLUT[0…15]	1D	0…F	Control	R/W	*
Texture Color/Fog	Texel0	0C	0	Control	R/W	
	Texel1	0C	1	Control	R/W	
	Texel2	0C	2	Control	R/W	
	Texel3	0C	3	Control	R/W	
	Texel4	0C	4	Control	R/W	
	Texel5	0C	5	Control	R/W	
	Texel6	0C	6	Control	R/W	
	Texel7	0C	7	Control	R/W	
	Interp0	0C	8	Control	R/W	
	Interp1	0C	9	Control	R/W	
	Interp2	0C	A	Control	R/W	
	Interp3	0C	B	Control	R/W	
	Interp4	0C	C	Control	R/W	
	TextureFilter	0C	D	Control	R/W	
Texture/Fog Color	TextureColorMode	0D	0	Control	R/W	>
	TextureEnvColor	0D	1	Control	R/W	
	FogMode	0D	2	Control	R/W	
	FogColor	0D	3	Control	R/W	
	FStart	0D	4	Control	R/W	
	dFdx	0D	5	Control	R/W	
	dFdyDom	0D	6	Control	R/W	

	KsStart	0D	9	Control	R/W	*
	dKsdx	0D	A	Control	R/W	*
	dKsdyDom	0D	B	Control	R/W	*
	KdStart	0D	C	Control	R/W	*
	dKdStart	0D	D	Control	R/W	*
	dKddyDom	0D	E	Control	R/W	*
Color DDA	RStart	0F	0	Control	R/W	
	dRdx	0F	1	Control	R/W	
	dRdyDom	0F	2	Control	R/W	
	GStart	0F	3	Control	R/W	
	dGdx	0F	4	Control	R/W	
	dGdyDom	0F	5	Control	R/W	
	BStart	0F	6	Control	R/W	
	dBdx	0F	7	Control	R/W	
	dBdyDom	0F	8	Control	R/W	
	AStart	0F	9	Control	R/W	
	dAdx	0F	A	Control	R/W	
	dAdyDom	0F	B	Control	R/W	
	ColorDDAMode	0F	C	Control	R/W	
	ConstantColor	0F	D	Control	R/W	
	Color	0F	E	Mixed	R/W	
Alpha Test	AlphaTestMode	10	0	Control	R/W	
	AntialiasMode	10	1	Control	R/W	
Alpha Blend	AlphaBlendMode	10	2	Control	R/W	>
Dither	DitherMode	10	3	Control	R/W	>
Logical Ops	FBSoftwareWriteMask	10	4	Control	R/W	
	LogicalOpMode	10	5	Control	R/W	
	FBWriteData	10	6	Control	R/W	
LB Read	LBReadMode	11	0	Control	R/W	>
	LBReadFormat	11	1	Control	R/W	>
	LBSourceOffset	11	2	Control	R/W	
	LBStencil	11	5	Output	W	
	LBDepth	11	6	Output	W	
	LBWindowBase	11	7	Control	R/W	
LB Write	LBWriteMode	11	8	Control	R/W	>
	LBWriteFormat	11	9	Control	R/W	>
	TextureData	11	D	Control	W	*
	TextureDownloadOffset	11	E	Control	R/W	*
GID/Stencil/Depth	Window	13	0	Control	R/W	>
	StencilMode	13	1	Control	R/W	>
	StencilData	13	2	Control	R/W	
	Stencil	13	3	Mixed	R/W	
	DepthMode	13	4	Control	R/W	
	Depth	13	5	Mixed	R/W	
	ZStartU	13	6	Control	R/W	
	ZStartL	13	7	Control	R/W	
	dZdxU	13	8	Control	R/W	
	dZdxL	13	9	Control	R/W	
	dZdyDomU	13	A	Control	R/W	

	dZdyDomL	13	B	Control	R/W	
	FastClearDepth	13	C	Control	R/W	
FB Read/Write	FBReadMode	15	0	Control	R/W	>
	FBSourceOffset	15	1	Control	R/W	
	FBPixelOffset	15	2	Control	R/W	
	FBColor	15	3	Output	W	
	FBData	15	4	Mixed	W	
	FBSourceData	15	5	Mixed	W	
	FBWindowBase	15	6	Control	R/W	
	FBWriteMode	15	7	Control	R/W	>
	FBHardwareWriteMask	15	8	Control	R/W	
	FBBlockColor	15	9	Control	R/W	
	PatternRamMode	15	F	Control	R/W	
	PatternRamData[0…31]	16 17	0…F 0…F	Control	R/W	
	FBBlockColorU	18	D	Control	R/W	*
	FBBlockColorL	18	E	Control	R/W	*
	SuspendUntilFrameBlank	18	F	Command	W	*
Host Out	FilterMode	18	0	Control	R/W	
	StatisticMode	18	1	Control	R/W	>
	MinRegion	18	2	Control	R/W	
	MaxRegion	18	3	Control	R/W	
	ResetPickResult	18	4	Command	W	
	MinHitRegion	18	5	Command	W	
	MaxHitRegion	18	6	Command	W	
	PickResult	18	7	Command	W	
	Sync	18	8	Command	W	

Table D.1

1.

Appendix E. Software Compatibility

The GLINT 500TX is a superset of the GLINT 300SX, and will run most software written for the GLINT 300SX unchanged. This appendix documents those areas where 100% compatibility has not been maintained, and the minimum changes that need to be made to software written for the GLINT 300SX, so that it will also run on the GLINT 500TX.

E.1 GLINT 500TX Specific Registers

The GLINT 500TX has many new registers, most of which can be ignored by GLINT 300SX driver software, however the following registers must be initialized:

TextureAddressMode

TextureReadMode

RouterMode

These registers should be set to zero.

Writes to GLINT 500TX registers which do not exist on the GLINT 300SX may be performed safely provided that the bits 14-15 of the **FilterMode** register are set to zero.

E.2 Pixel Size

One fundamental area where software compatibility has not been maintained between the GLINT 300SX and GLINT 500TX concerns the setting of the pixel size.

In the GLINT 300SX a write to the register *FBModeSel* is performed to change the pixel size and the new pixel size takes effect immediately, no matter the state of the graphics core. As a result the host should synchronize with the GLINT 300SX before changing the pixel size to ensure that the core is quiescent. Because the *FBModeSel* register only affects the memory I/O interface and the Graphics Core has no need to know the pixel size. The pixel size is typically initialized on reset from configuration resistors.

In the GLINT 500TX the Graphics Core does need to know the pixel size for the new span filling modes and hence a new register, **PixelSize**, is used to control the pixel size. This has also removed the need to synchronize with the GLINT 500TX before changing the pixel size. Indeed in the GLINT 500TX the pixel size cannot be set by writing to the *FBModeSel* register. Reading the pixel size from the *FBModeSel* register will return 3, i.e. undefined, for the pixel size. The pixel size may only be read back from the **PixelSize** register.

The software implications are:

- On the GLINT 500TX the **PixelSize** register must be initialized by software before any framebuffer accesses are made from the Graphics Core. Bypass accesses do not use the pixel size information and hence

these accesses may be performed before initializing the pixel size.

- If the same software must run on both devices, then the easiest solution is for the software to first synchronize with the GLINT device, then write to the *FBModeSel* register and finally write to the **PixelSize** register. The GLINT 300SX will ignore the write to the **PixelSize** register (assuming that bits 14-15 of the **FilterMode** register are set to zero) and the GLINT 500TX will ignore the write to PixelSize field in the *FBModeSel* register. The synchronize will be an unnecessary overhead in the GLINT 500TX, so ideally the type of the chip should be autosensed by the software.
- On the GLINT 300SX the pixel size is changed dynamically during some fill and BitBlt operations on 8 bpp or 16 bpp framebuffers to improve performance. This will still work in the GLINT 500TX, but the requirement to change the pixel size for performance reasons in the GLINT 500TX has disappeared as the span mechanism provides a much better solution.

E.3 Block Fills

Block fills in the GLINT 500TX are implemented in a radically different way to provide a significant increase in performance and functionality. GLINT 300SX style block fills may be performed on the GLINT 500TX, however several additional registers must to be set up to disable scissoring, area stipple and framebuffer reads. In the GLINT 300SX the block fill bypasses most of the normal processing units and hence the mode of these units is irrelevant during a block fill. In the GLINT 500TX the block fill (now renamed span fill to be more descriptive) is processed by many more units to give additional functionality. It is these units which need to be set up. The registers which need to be written to are:

ScissorMode to disable user and screen scissor operations.

AreaStippleMode to disable area stipple operations. This may also be controlled through the **Render** command.

FBReadMode to disable source and destination framebuffer reads.

StatisticMode to disable statistics collection so that the block fill operation does not grow the extent region or cause picking to occur.

PatternRamMode to disable the Pattern RAM.

The block fill size in the **Render** command and **FBWriteMode** registers are ignored in the GLINT 500TX.

E.4 Device Id

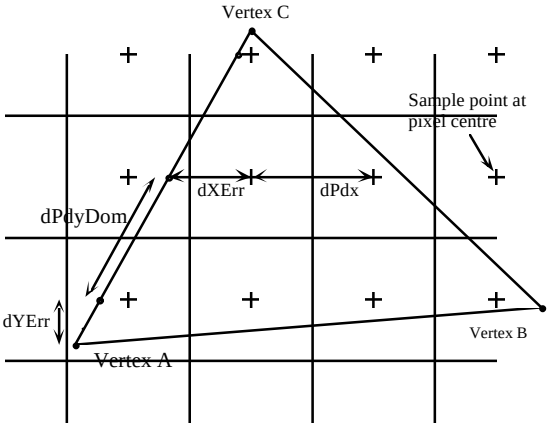
The device number returned by the *CFGDeviceId* register in the GLINT 500TX is 0002h in bits 31-16.

1.

Appendix F. Accurate Rendering

This appendix describes how to calculate the various parameters needed to define a Gouraud shaded triangle. This topic is covered in section 5.2, however in the interest of simplicity some of the finer details were glossed over. The quality of the rasterization and shading suffers where these fine details are not included and will give rise to 'stitch marks' and 'bright edge' artifacts. The main area where simplifications were made earlier relates to the fact that vertices are not, in general, coincident with pixel centers so sub pixel corrections are necessary. The initial values being interpolated (RGB for example) need to be adjusted to account for this. GLINT will do the necessary X corrections when moving from scan line to scan line when the SubPixelCorrection bit is set, but the initial Y correction must be done in software.

Consider a sample triangle, highly magnified to emphasize the sub pixel corrections needed:



The vertices are sorted into Y order and the dominant edge is AC. Scan conversion will start at vertex A and proceed upwards. The origin is bottom left.

The usual parameters to interpolate (denoted P in the diagram) across the triangle would include color (R, G, B and alpha), depth (Z), fog (F), and texture (S, T, Q, Ks and Kd). The source code to set up GLINT to achieve the best quality rendering will only calculate the parameters for RGBA and Z to keep the size of the code down.

```
#include <stdio.h>
#include <float.h>

// A simple macro which just prints out the register name and value.
// Replace this with some code to write to GLINT.

#define LD_GLINT_REG(name, value)                       \
    printf ("%s = %08x\n", #name, value)



// This software is part of the application note which describes
// how GLINT is set up to get the best quality rendering. Particular
// care is taken to avoid cracks, stitch marks and bright edge artifacts
// from occurring. The OpenGL rasterization rules are used.
// The software has not been written with maximum performance in mind,
// but as a clear, well documented example covering the nuances
// which are easily overlooked.

// Simple vertex structure used to interface parameters to the RenderTriangle
// function.

typedef struct {    float   x, y, z;        // in device coords
                    float   r, g, b, a;     // in the range 0.0 to 1.0
                } Vertex;


// Prototypes.

long IntToFixedPoint16 (long i);
long FloatToColor (float f);
long FloatToCoordinate (float f);
void FloatToDepth (float f, long *zi, long *zf);
void RenderTriangle (Vertex *v0, Vertex *v1, Vertex *v2);


// Defines some simple function to convert from floating point numbers
// to various fixed point formats. These can be inlined if necessary.

long IntToFixedPoint16 (long i)
{
    return i << 16;
}

// These functions perform the conversion from floating point numbers
// to the various fixed point format numbers required in GLINT.  They
// are implemented as simple operations on the binary representation
// of IEEE single precision floating point number so the floating
// point rounding mode doesn't need to be set up first and in many
// cases they are faster than using the built in conversion functions,
// especially when the range checking and clamping is taken into account.

//  Format of IEEE single-precision (32-bit) real number.

#define F_BIAS      127
#define F_SIGN_BIT  31
#define F_EXPONENT_BITS 23
#define F_FRACTION_BITS 0


// Convert 32-bit floating-point value to 9.15 fixed-point value used
// for the color parameters.  The input range is assumed to be 0.0
// to 1.0.  The algorithm is:
// If exponent < -15 then return (0x00000000), otherwise
// if exponent < 8 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0xff800000 : 0x007fffff).
```

```
long FloatToColor (float fi)
{
    long            f = *((long *) &fi);
    long            sign;
    unsigned char   exponent;

    sign = (f >> F_SIGN_BIT);
    exponent = (unsigned char)(f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-15))
        return (0);
    if (exponent < (F_BIAS+8))
    {
        f = ((unsigned long)((f | 0x00800000) << 8)
                                              >> ((F_BIAS+16) - exponent));
        if (sign < 0)
            f = -f;
        return (f);
    }
    return (0x007fffff ^ sign);
}


// Convert 32-bit floating-point value to 16.16 fixed-point value used
// for the rasterizer parameters.
// If exponent < 0 then return (0x00000000), otherwise
// if exponent < 31 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0x80000000 : 0x7fffffff).

long FloatToCoordinate (float fi)
{
    long            f = *((long *) &fi);
    long            sign;
    unsigned char   exponent;
    long            res;

    sign = f >> F_SIGN_BIT;
    exponent = (unsigned char) (f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-16))
        return (0);
    if (exponent < (F_BIAS+15))
    {
        res = ((unsigned long)((f | 0x00800000) << 8)
                                                >> ((F_BIAS+15) - exponent));
        if (sign < 0)
            res = -res;
        return (res);
    }
    return (0x7fffffff ^ sign);
}


// Convert 32-bit floating-point value to 24.16 fixed-point value as
// used by the Z values.  Note that this assumes a 24 bit Z buffer.
// If exponent < -16 then return (0x0000000000000000), otherwise
// if CLAMP_24_16 is defined and is non-zero:
// if exponent < 23 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0xff80000000000000 : 0x007fffffffff0000).
// otherwise:
// return (-1**(s) * 1.f * 2**(e - 127)).

void FloatToDepth (float fi, long *zi, long *zf)
{
    long            f = *((long *) &fi);
    long            sign;
```

```
    unsigned char   exponent;
    long            resh;
    unsigned long   resl;

    sign = (f >> F_SIGN_BIT);
    exponent = (unsigned char)(f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-16))
    {
        *zi = 0;
        *zf = 0;
        return;
    }
    if (exponent < (F_BIAS+23))
    {
        f = ((f | 0x00800000) << 8);
        if (exponent < (F_BIAS+0))
        {
            resh = 0;
            resl = ((unsigned long) f >> ((F_BIAS-1) - exponent));
        }
        else
        {
            unsigned char shift;

            shift = ((F_BIAS+31) - exponent);   // 8 <= shift < 32
            resh = ((unsigned long) f >> shift);
            resl = (f << (31 - shift));         // shifts >= 32 undefined
            resl <<= 1;                         // so we must shift twice
        }
        if (sign < 0)
        {
            unsigned long old_resl;

            resl = ~resl;
            resh = ~resh;
            old_resl = resl;
            resl += 0x00010000;
            if (resl < old_resl)        // overflow
                ++resh;
        }
    }
    else
    {
        resh = (0x007fffff ^ sign);
        resl = (0xffff0000 ^ sign);
    }
    resl &= 0xffff0000;
    *zi = resh;
    *zf = resl;
}


#define SAME 0
#define REVERSED ~SAME
#define ORDER(v0, v1, v2, order) {a = v0; b = v1; c = v2; windingOrder = order;}


void RenderTriangle (Vertex *v0, Vertex *v1, Vertex *v2)
{
    float   dxAB, dyAB, dxBC, dyBC, dxAC, dyAC; // Diff in x,y for each edge.
    float   drAC, dgAC, dbAC, daAC, dzAC;      // Diff in rgbz for dominant edge
    float   drBC, dgBC, dbBC, daBC, dzBC;       // Diff in rgbz for the BC edge.
    float   dxdyAC, dxdyAB, dxdyBC;             // Edge gradients for unit
                                                // set in y
```

```
float   drdxdy, dgdxdy, dbdxdy;
float   dadxdy, dzdxdy;
float   drdx, dgdx, dbdx, dadx, dzdx;       // Gradients for unit step in x.
float   r0, g0, b0, a0, z0;                 // Start values
float   area, oneOverArea, t1, t2;
float   oneOverdyAC;
Vertex  *a, *b, *c;                         // Sorted vertices.
long    xDomFixed, xSubFixed;
float   dyErr, yBottom, yTop;
long    iyBottom, iyTop;
int     windingOrder;                       // Not used.
long    zi, zf;
long    temp;

// Sort vertices into ascending Y order. *a points to the vertex with the
// lowest y value. Compare winding order of the pre and post sorted vertices
// and set winding order flag as appropriate (this is only needed if culling
// based on the winding order is to be done).

if (v0->y < v1->y)
{
    if (v1->y < v2->y)
        ORDER (v0, v1, v2, SAME)
    else
        if (v0->y < v2->y)
            ORDER (v0, v2, v1, REVERSED)
        else
            ORDER (v2, v0, v1, SAME)
}
else
{
    if (v1->y < v2->y)
    {
        if (v0->y < v2->y)
            ORDER (v1, v0, v2, REVERSED)
        else
            ORDER (v1, v2, v0, SAME)
    }
    else
        ORDER (v2, v1, v0, REVERSED)
}

// Compute signed area of the triangle.
// Form vectors for two edges of the triangle.
dxAC = a->x - c->x;
dxBC = b->x - c->x;
dyAC = a->y - c->y;
dyBC = b->y - c->y;

// Form the cross product of the two edges.
area = dxAC * dyBC - dxBC * dyAC;

if (area == 0.0)
    return;             // Reject zero area triangles.

// A negative area just means the order of the vertices, after sorting, was
// clockwise. Note this may be different from original input order.
if (area < 0.0)
    area = -area;           // Make positive.

// The dx/dy value (change in x for unit change in y) are needed for
// each edge so the rasterizer can compute the new left and right hand
// x coordinates as it steps from one scan line to the next. Horizontal
// or near horizontal edges will have very large gradients but these will
// be handled later. Values for AC and BC have already been calculated so
// just do the remaining edge.
```

```
dxAB = a->x - b->x;
dyAB = a->y - b->y;

// The dominant edge is always AC (i.e. the edge with the maximum Y extent).
// Compute the change in rgbaz along this edge for unit change in y.
oneOverdyAC = 1.0 / dyAC;

// Differences along edge AC
drAC = a->r - c->r;
dgAC = a->g - c->g;
dbAC = a->b - c->b;
daAC = a->a - c->a;
dzAC = a->z - c->z;

// Gradient along edge AC for each parameter.
drdxdy = drAC * oneOverdyAC;
dgdxdy = dgAC * oneOverdyAC;
dbdxdy = dbAC * oneOverdyAC;
dadxdy = daAC * oneOverdyAC;
dzdxdy = dzAC * oneOverdyAC;
dxdyAC = dxAC * oneOverdyAC;

// Difference along edge BC
drBC = b->r - c->r;
dgBC = b->g - c->g;
dbBC = b->b - c->b;
daBC = b->a - c->a;
dzBC = b->z - c->z;

// Compute the change in rgbaz when taking unit steps in x.
oneOverArea = 1.0 / area;

t1 = dyAC * oneOverArea;
t2 = dyBC * oneOverArea;

drdx = drAC * t2 - drBC * t1;
dgdx = dgAC * t2 - dgBC * t1;
dbdx = dbAC * t2 - dbBC * t1;
dadx = daAC * t2 - daBC * t1;
dzdx = dzAC * t2 - dzBC * t1;

// A general triangle will need to be split into two trapezoids for
// rendering. Either of these trapezoids may have a zero height in
// which case the triangle has a flat top or bottom.  The rasterizer
// and DDAs are still set up, however the count may be zero.

// Fill lower trapezoid.
yBottom = a->y;
yTop = b->y;

// The y coordinates are converted to integer values, taking into
// account the openGL rules which determine which pixels fall within
// the boundary.

temp = FloatToCoordinate (yBottom);     // float to 16.16 fixed point
temp += 0x00007fff;                         // add in nearly a half
iyBottom = temp >> 16;                      // extract integer part

temp = (int) FloatToCoordinate (yTop);  // float to 16.16 fixed point
temp += 0x00007fff;                         // add in nearly a half
iyTop = temp >> 16;                         // extract integer part

dyErr = iyBottom + 0.5 - yBottom;

// Check for the case when AB is a true horizontal edge to prevent a divide
```

```
// by zero.
if (dyAB == 0.0)
    dyAB = FLT_MIN;             // set to a very small number.

dxdyAB = dxAB / dyAB;

// Move the rgbaz values at vertex a along the edge AC in proportion
// to how far the vertex a is from the pixel center in the y direction
// to do the sub pixel adjustment in Y.  GLINT will do the sub pixel
// adjustment in X automatically, if enabled.

r0 = a->r + dyErr * drdxdy;
g0 = a->g + dyErr * dgdxdy;
b0 = a->b + dyErr * dbdxdy;
a0 = a->a + dyErr * dadxdy;
z0 = a->z + dyErr * dzdxdy;

// Similarly for the start values for the left and right hand edges.
xDomFixed = FloatToCoordinate (a->x + dyErr * dxdyAC);
xSubFixed = FloatToCoordinate (a->x + dyErr * dxdyAB);

// Load up GLINT with the parameters.

// Rasterizer. Note that the RasterizerMode is set to add
// __GLINT_START_BIAS_ALMOST_HALF to the XDom, XSub and
// Y Start values to conform to the OpenGL rasterization rules.

LD_GLINT_REG(StartXDom, xDomFixed);
LD_GLINT_REG(dXDom,     FloatToCoordinate (dxdyAC));
LD_GLINT_REG(StartXSub, xSubFixed);
LD_GLINT_REG(dXSub,     FloatToCoordinate (dxdyAB));
LD_GLINT_REG(StartY,    IntToFixedPoint16 (iyBottom));
LD_GLINT_REG(dy,        IntToFixedPoint16 (1));
LD_GLINT_REG(Count,     (iyTop - iyBottom));

// Color DDA.
LD_GLINT_REG(RStart,    FloatToColor (r0));
LD_GLINT_REG(dRdx,      FloatToColor (drdx));
LD_GLINT_REG(dRdyDom,   FloatToColor (drdxdy));
LD_GLINT_REG(GStart,    FloatToColor (g0));
LD_GLINT_REG(dGdx,      FloatToColor (dgdx));
LD_GLINT_REG(dGdyDom,   FloatToColor (dgdxdy));
LD_GLINT_REG(BStart,    FloatToColor (b0));
LD_GLINT_REG(dBdx,      FloatToColor (dbdx));
LD_GLINT_REG(dBdyDom,   FloatToColor (dbdxdy));
LD_GLINT_REG(AStart,    FloatToColor (a0));
LD_GLINT_REG(dAdx,      FloatToColor (dadx));
LD_GLINT_REG(dAdyDom,   FloatToColor (dadxdy));

// Depth DDA.
FloatToDepth (z0, &zi, &zf);
LD_GLINT_REG(ZStartU,   zi);
LD_GLINT_REG(ZStartL,   zf);

FloatToDepth (dzdx, &zi, &zf);
LD_GLINT_REG(dZdxU,     zi);
LD_GLINT_REG(dZdxL,     zf);

FloatToDepth (dzdxdy, &zi, &zf);
LD_GLINT_REG(dZdyDomU,  zi);
LD_GLINT_REG(dZdyDomL,  zf);

// Render the trapezoid ...
LD_GLINT_REG(Render,    0x00014041);

// Fill upper trapezoid.
```

```
    yBottom = b->y;
    yTop = c->y;

    // The y coordinates are converted to integer values, taking into
    // account the openGL rules which determine which pixels fall within
    // the boundary.

    temp = FloatToCoordinate (yBottom); // float to 16.16 fixed point
    temp += 0x00007fff;                  // add in nearly a half
    iyBottom = temp >> 16;               // extract integer part

    temp = FloatToCoordinate (yTop);     // float to 16.16 fixed point
    temp += 0x00007fff;                  // add in nearly a half
    iyTop = temp >> 16;                  // extract integer part

    // Find the dyErr value for vertex B so that the start value for x can be
    // corrected.
    dyErr = iyBottom + 0.5 - yBottom;

    // Check for the case when BC is a true horizontal edge to prevent a divide
    // by zero.
    if (dyBC == 0.0)
        dyBC = FLT_MIN;             // set to a very small number.

    dxdyBC = (dxBC / dyBC);

    // Set up the rasterizer for the upper trapezoid.  All other DDA units
    // can carry on with their parameters as they are walking up the same
    // edge.
    xSubFixed = FloatToCoordinate (b->x + dyErr * dxdyBC);
    LD_GLINT_REG(StartXSub, xSubFixed);
    LD_GLINT_REG(dxSub, FloatToCoordinate (dxdyBC));
    LD_GLINT_REG(ContinueNewSub, (iyTop - iyBottom));
}
```

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 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 and antialiasing.

alpha test A test used to cull selected fragments from being drawn, based on a comparison of a fixed value with the alpha value of the fragment.

antialiasing A rendering style which weights the color of a pixel by the fraction of its area that is covered by primitives, leading to reduction or elimination of jagged edges.

bitblt Bit aligned block transfer. Copy of a rectangular array of pixels in a bitmap from one location to another.

block write A feature provided in some VRAM devices which allows multiple pixels to be set to a given value by a single write. See also fast fill which is an alternative name for the same feature.

command register A register which when loaded triggers activity in GLINT. For instance the **Render** command register when loaded will cause GLINT 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 GLINT to render primitives. Software on the host must save away the current contents of the GLINT 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 GLINT 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.

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. See also 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.

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.

fast fill A feature provided in some VRAM devices which allows multiple pixels to be set to a given value by a single write. See also block write which is an alternative name for the same feature.

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. See also depth-cueing.

Frame Count Planes(FCP) Used to allow higher animation rates by enabling DRAM localbuffer pixel data, such as depth (Z), to be cleared down quickly.

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), alpha, 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.

Graphic ID (GID) A component of a pixel containing information used for per pixel clipping.

host The processor which controls GLINT.

localbuffer An area of memory which may be used to store the following non-displayable pixel information: depth(Z), stencil, Fast Clear Planes, Graphic ID. This memory is typically separate from the framebuffer.

passive fragment A fragment which fails one or more of the various culling tests, such as scissor, depth(Z), alpha, etc., is nor written to/combined with the corresponding pixel in the framebuffer. See also "fragment" and "active

fragment".

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 GLINT primitives are points, lines, trapezoids (including triangles as a subset), and bitmaps.

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.

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 binar y pattern which is used to cull fragments from being drawn.

task A process, or thread on the host which uses the GLINT coprocessor. Typically tasks assume that they have sole use of GLINT and rely on a device driver to save and restore their GLINT 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 col or 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.

window control buffer A buffer containing control bits used by display hardware to select between multiple hardware LUTs or display buffers (such as overlay and underlay) on a per pixel basis. Usually a given value in the buffer corresponds to a single window on the screen.

writemask A bit pattern used to enable or inhibit the writing of the corresponding bits of a fragment's color into the framebuffer.

1.

Index

S

T

U

V

W

Z