TEXAS INSTRUMENTS

TVP4020 PERMEDIA® 2

Programmer's Reference

User's Guide

1998

Mixed-Signal Products

Proprietary and Confidential

TEXAS INSTRUMENTS

Printed in U.S.A.
4/98

User's
Guide

TVP4020 PERMEDIA®2
Programmer's Reference

1998

TEXAS INSTRUMENTS

TVP4020 PERMEDIA® 2
Programmer's Reference

1998 *Mixed-Signal Products*

Proprietary and Confidential

Printed in U.S.A.
4/98

TVP4020 PERMEDIA®2

Programmer's Reference

1998

User’s Guide

User’s Guide

[redacted]

[redacted]

[redacted]

TVP4020 PERMEDIA® 2
Programmer's Reference

User's Guide

Proprietary and Confidential

SLAU011A
April 1998

PRINTED WITH SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

The material in this document is the intellectual property of 3Dlabs. It is provided solely for information. You may not reproduce this document in whole or in part by means. While every care has been taken in the preparation of this document, 3Dlabs accepts no liability for any consequences of its use. Our products are under continual improvement and we reserve the right to change their specification without notice.

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

This document contains copyrighted, proprietary and confidential information of 3Dlabs which is reproduced herein with the permission of 3Dlabs Inc, Ltd.; no copying or other distribution of this material is permitted without the written permission of Texas Instruments (TI) and 3Dlabs.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

The TVP4020 is a high-performance PERMEDIA® 2 graphics processor that balances high quality 3-D texturing and graphics performance with leading edge windows, video, and SVGA acceleration. Based on a proven low-cost and scaleable architecture, the TVP4020 accelerates a broad range of applications including: game, animation, authoring, web browser, design visualization, publishing, and general multimedia.

The TVP4020 sets the standard for 3-D and multimedia acceleration. It meets the increasing need for balanced 3-D and multimedia acceleration, in a single, low-cost PCI device.

This document is meant to function as the primary reference for programmers and system designers who develop software to drive the TVP4020. Information on programming the I/O registers can be found in the *TVP4020 3D Graphics Processor Data Manual.*

A knowledge of the principles of 2-D and 3-D graphics programming is useful for understanding this document.

How to Use This Manual

This document is composed of the following chapters:

Chapter 1 gives an overview of the TVP4020.

Chapter 2 details the programming model for the chip.

Chapter 3 describes the memory I/O and organization of the TVP4020 support systems in the framebuffer, localbuffer and texture buffer.

Chapter 4 describes how to use the TVP4020 for graphics rendering.

Chapter 5 describes how to initialize the TVP4020.

Chapter 6 provides tips for programming the TVP4020.

Chapter 7 tabulates the TVP4020 registers.

Chapter 8 gives lists of registers and their addresses.

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

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

Appendix C summarizes the differences between the TVP4020 and TVP4010.

Appendix D is a glossary of technical terms.

An extensive index is included.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

  ```
  0011  0005  0001         .field    1, 2
  0012  0005  0003         .field    3, 4
  0013  0005  0006         .field    6, 3
  0014  0006               .even
  ```

  Here is an example of a system prompt and a command that you might enter:

  ```
  C:  csr −a /user/ti/simuboard/utilities
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect “*section name*”**,** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don’t enter the brackets themselves. Here’s an example of an instruction that has an optional parameter:

  LALK *16–bit constant [, shift]*

  The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here’s an example of a list:

  ```
  { * | *+ | *– }
  ```

  This provides three choices: `*`, `*+`, or `*–`.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value*$_1$ *[, ... , value*$_n$*]*

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Information About Cautions and Warnings

This book may contain cautions and warnings.

CAUTION

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to <u>you</u>.

WARNING

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation From Texas Instruments

The following books describe the TV4020 and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477−8924. When ordering, please identify the book by its title and literature number.

TVP4020 Architecture Overview, Literature No. SLAU010

TVP4020 3D Graphics Processor Data Manual, Literature No. SLAS161

Other Documentation

The following documentation is referenced in this manual:

PCI Local Bus Specification, Revision 2.1, Peripheral Component Interconnect Special Interest Group (PCI SIG) Portland, Ore.

The following referenced books are available through Addison−Wesley, Reading, Mass.:

Programmer’s Guide to the EGA, VGA and Super VGA Cards

OpenGL Reference Manual

OpenGL Programming Guide

If You Need Assistance . . .

If you want to . . .	**Contact Texas Instruments at . . .**	
Visit TI online	World Wide Web:	http://www.ti.com
Receive general information or assistance	World Wide Web:	http://www.ti.com/sc/docs/pic/home.htm
	North America, South America:	(214) 644−5580
	Europe, Middle East, Africa	
	Dutch:	33−1−3070−1166
	English:	33−1−3070−1165
	French:	33−1−3070−1164
	Italian:	33−1−3070−1167
	German:	33−1−3070−1168
	Japan (Japanese or English)	
	Domestic toll-free:	0120−81−0026
	International:	81−3−3457−0972 or 81−3−3457−0976
	Korea (Korean or English):	82−2−551−2804
	Taiwan (Chinese or English):	886−2−3771450
Ask questions about Mixed Signal Processor (MSP) product operation or report suspected problems		(713) 274−2320
	Fax:	(713) 274−2324
	Fax Europe:	+33−1−3070−1032
	Email:	4389750@mcimail.com
	World Wide Web:	http://www.ti.com/dsps
	BBS North America:	(713) 274−2323 8−N−1
	BBS Europe:	+44−2−3422−3248
	320 BBS Online:	ftp.ti.com:/mirrors/tms320bbs (192.94.94.53)
Ask questions about microcontroller product operation or report suspected problems		(713) 274−2370
	Fax:	(713) 274−4203
	Email:	*H370@msg.ti.com
	World Wide Web:	http://www.ti.com/sc/micro
	BBS:	(713) 274−3700 8−N−1
Request tool updates	Software:	(214) 638−0333
	Software fax:	(214) 638−7742
	Hardware:	(713) 274−2285
Order Texas Instruments documentation (see Note 1)	Literature Response Center:	(800) 477−8924
Make suggestions about or report errors in documentation (see Note 2)	Email:	comments@books.sc.ti.com
	Mail:	Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251−1443

Notes: 1) The literature number for the book is required; see the lower-right corner on the back cover.

2) Please mention the full title of the book, the literature number from the lower-right corner of the back cover, and the publication date from the spine or front cover.

Trademarks

3Dlabs is the worldwide trading name of 3Dlabs Inc., Ltd.

3Dlabs, GLINT, and PERMEDIA are registered trademarks of 3Dlabs.

Macintosh, QuickDraw, and QuickDraw3D are trademarks of Apple Computer, Inc.

OpenGL is a trademark of Silicon Graphics Inc.

RAMDAC is a trademark of Brooktree Corp.

TI is a trademark of Texas Instruments Incorporated.

Windows, Win32, Windows NT, and Windows 95 are trademarks of Microsoft Corp.

Contents

Figures

T ables

Chapter 1

Overview

This chapter presents a functional overview of the TVP4020 operation. Subjects covered include the external interfaces, the memory subsystem, the host interface, and task switching.

Topic **Page**

1.1 Functional Overview

The TVP4020 is a high-performance peripheral-component interconnect/ accelerated graphics port (PCI/AGP) graphics processor. It balances high quality, three-dimensional (3-D) polygon and textured graphics acceleration; windows acceleration; and state-of-the-art MPEG1/MPEG2 playback, with a fast integrated super-video graphic array (SVGA) core, integrated random-access memory digital-to-analog converter (RAMDAC), and video ports.

The TVP4020 features a full line of advanced graphics processing functions that include:

❑ Enhanced 3-D graphics features and performance at 83 MHz

- 83M perspective-correct, bilinear filtered, texture mapped pixels/s
- 42M perspective-correct, bilinear filtered, texture mapped, depth buffered pixels/s
- 800K texture mapped polygons/s
- True-color 3-D graphics
- Polygon based with Z buffer
- Texture decompression
- Full scene antialiasing
- Integrated 100-MFLOP geometry pipeline setup processor
- Slope and setup information calculation
- Floating- to fixed-point conversion

❑ Full support for the Intel™ AGP and PCI

- 66-MHz operation
- Direct memory access (DMA) and execute mode support
- Sideband addressing
- Enhanced graphical user interface (GUI) acceleration
- Ultra-fast block transfer (BLT) engine and two-dimensional (2-D) rasterizer
- Stretch BLTs, monochrome/color expansion, and logic operations

❑ Fast SVGA with 8, 16, 24, and 32-bit packed framebuffer storage

❑ MPEG2 compatible video playback acceleration

    - YUV 4:4:4, YUV 4:2:2, and YUV 4:2:0 (native MPEG2 format)
    - Unlimited multiple playback windows (occluded)
    - Independent x- and y-scaling and mirroring

- Integrated true-color 233-MHz RAMDAC
    - DPMS, DDC1, and DDC2AB+
    - Clock synthesizer and hardware cursor
    - 320×200 to 1600×1200 screen resolutions

- Multimode video streams
    - Simultaneous input and output video
    - Optional scaling and filtering
    - Optional color space conversion and gamma correction

- Flexible multifunction synchronous dynamic random-access memory (SDRAM) or synchronous graphic random-access memory (SGRAM)

- Comprehensive suite of optimized software drivers

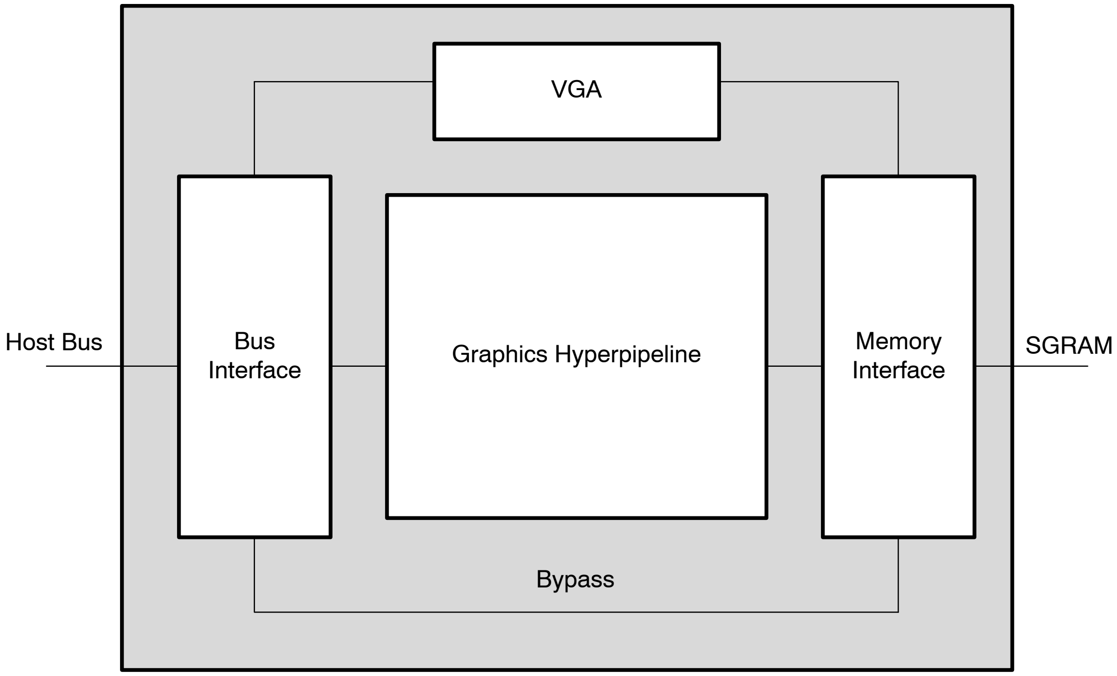
- Reference board designs and manufacturing kits

1.2 Memory Subsystem

The TVP4020 provides flexible support for the memory subsystem (see Figure 1−1). This allows the system designer a wide choice of price/ performance tradeoffs.

The same physical memory holds all data used by the TVP4020. Internally, the data types are divided into texture, localbuffer, and framebuffer. The localbuffer holds depth and stencil data; the framebuffer holds color data for display.

Figure 1−1. External Interfaces



1.3 Host Interface

The TVP4020 functions as a register file. The host initializes the control registers with information to draw a primitive. To start the chip drawing, the host writes commands to a command register.

The host accesses the TVP4020 registers directly through the memory map. Registers are accessed either individually or in groups.

The chip also supports a bypass route to the memory to allow direct reading/writing of pixels and implementation of algorithms not directly supported by the TVP4020.

1.4 Task Switching

Where multiple applications simultaneously access the TVP4020, the software driving the chip handles the loading of the correct state. The TVP4020 supports a number of software architectures with the following functions:

- Synchronous operation, which allows a new task to load its context without waiting for current rendering to complete
- All loadable states, which can be read back
- A sync command that flushes all renderings, which can be polled or can return an interrupt

1.4.1 SVGA

The TVP4020 contains a fast video graphics adapter (VGA) core. The TVP4020 super VGA, known as SVGA, is used for DOS VGA applications and for boot time before it switches to the graphics hyperpipeline.

This document does not cover VGA programming. Specific information on the TVP4020 VGA is described in the *TVP4020 3D Graphics Processor Data Manual*. Video graphics adapter information, such as standard registers, is described in the *Programmer's Guide to the EGA, VGA and Super VGA Cards*.

Table 1–1 shows the standard VGA modes that are supported.

Table 1–1. SVGA Unit

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

Table 1−2 describes the supported VESA SVGA modes.

Table 1−2. VESA SVGA Modes

Mode (Hex)	**Pixels**	**Colors**
100	640×400	256
101	640×480	256

Chapter 2

Programming Model

This chapter describes the programming model for the TVP4020. It provides a conceptual description of the interface rather than the details and exact usage of specific registers. In-depth descriptions of how to program the TVP4020 for specific drawing operations are in later chapters.

Topic **Page**

2.1 Memory Regions

The TVP4020 is divided into memory regions as shown in Table 2−1. The Region 0 address map is shown in Table 2−2.

Table 2−1. Memory Regions

Region	Address Space	Bytes	Description	Comments
Config	Configuration	256	PCI configuration	PCI special
Zero	Memory	128K	Control registers	Relocatable
One	Memory	8M	Memory region one	Relocatable
Two	Memory	8M	Memory region two	Relocatable
ROM	Memory	64K	Expansion ROM	Relocatable
SVGA	Memory and I/O	-	SVGA addresses	Optional and fixed

Table 2−2. Region 0 Address Map

Address Range (hex)	Description	Byte Swap
0000.0000 - 0000.0FFF	Control and status	No
0000.1000 - 0000.1FFF	Memory control	No
0000.2000 - 0000.2FFF	GP FIFO access	No
0000.3000 - 0000.3FFF	Video control	No
0000.4000 - 0000.4FFF	RAMDAC	No
0000.5000 - 0000.57FF	Video streams general purpose bus	No
0000.5800 - 0000.5FFF	Video streams control	No
0000.6000 - 0000.6FFF	SVGA control	No
0000.7000 - 0000.7FFF	Reserved	No
0000.8000 - 0000.FFFF	GP registers	No
0001.0000 - 0001.0FFF	Control and status	Yes
0001.1000 - 0001.1FFF	Memory control	Yes
0001.2000 - 0001.2FFF	GP FIFO access	Yes
0001.3000 - 0001.3FFF	Video control	Yes
0001.4000 - 0001.4FFF	RAMDAC	Yes
0001.5000 - 0001.57FF	Video streams general purpose bus	No
0001.5800 - 0001.5FFF	Video streams control	No
0001.6000 - 0001.6FFF	SVGA control	Yes
0001.7000 - 0001.7FFF	Reserved	Yes
0001.8000 - 0001.FFFF	GP registers	Yes

2.2 TVP4020 as a Register File

The simplest way to view the interface to the TVP4020 graphics processor is as a flat block of memory-mapped registers (that is, a register file). This register file is part of the address map for the TVP4020.

When a TVP4020 host software driver initializes, it maps the register file into its address space. Each register has an associated address tag, which gives its offset value from the base of the register file (since all registers reside on a 64-bit boundary, the tag offset value is measured in multiples of eight bytes). The most straightforward way to load a value into a register is to write the data to its mapped address. In reality, the chip interface comprises a 256-entry-deep, first-in-first-out (FIFO) buffer, and each write to a register includes the value and the register's address tag, as a new entry in the FIFO.

Programming the TVP4020 to draw a primitive consists of writing values to the appropriate registers, followed by a write to a command register. This last write triggers the start of a drawing.

The TVP4020 has approximately 200 registers. All registers are 32 bits wide and 32-bit addressed. Many registers are split into bit fields with bit 0 as the least significant bit.

In future chip revisions, the register file may be extended, and unused bits in certain registers may be assigned new meanings. Be certain to write to defined registers only and always write zeros to undefined bits in registers. The only exception is that in certain registers, you can write unmasked values to registers that hold numeric data. These fields are marked as *not used* in Chapter 7, *Graphics Register Reference*, and elsewhere.

2.2.1 Register Types

TVP4020 has three main types of registers:

❑ Control registers
❑ Command registers
❑ Internal registers

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

Command registers are those which, when written to, create an action*.* Typically, the host initializes the appropriate control registers and then writes to a command register to initiate drawing. Some command registers, such as ResetPickResult or Sync, do not initiate rendering. Apart from these, there are two types of command registers: begin-draw and continue-draw. Begin-draw commands start the rendering operation using those values specified by the control registers. Continue-draw commands continue the drawing operation, using internal register values, as they were when the previous drawing operation completed. Using continue-draw commands can significantly reduce the amount of data to be loaded into the TVP4020 when drawing multiple connected objects such as polylines. Examples of command registers include the Render and ContinueNewLine registers.

For convenience, this document uses the phrase, *sending a Render command to TVP4020* rather than 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 to the TVP4020, and before rendering starts, the internal registers are updated with the values in the corresponding control registers. If a continue-draw command is sent to the TVP4020, this update does not occur and drawing continues with the current values in the internal registers. For example, if a line is being drawn, 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 to the TVP4020, 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 is completed, the internal registers contain the X-, Y-coordinates of the next point to be drawn. If a continue-draw command is now given, these final X-, Y-internal values are not modified, and subsequent drawing operations use these values. Alternatively, if a begin-draw command is used, the internal registers reload from the StartXDom and StartY registers.

Other than for continue-draw commands, internal registers can be ignored.

2.2.2 Efficiency Issues and Register Types

Software developers wishing to write device drivers for the TVP4020 should become familiar with the different types of registers. Some control registers, such as the StartXDom and StartY, have to be updated for almost every primitive, whereas other control registers such as those for scissor clip or logical operations can be updated less frequently. Preloading of the appropriate control registers can reduce the amount of data 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.

Chapter 8, *Register Tables*, lists the graphics registers according to their type, name, and address.

2.3 TVP4020 I/O Interface

There are four ways of loading the TVP4020 registers:

❑ The host writes a value to the mapped address of the register.

❑ The host writes address-tag/data pairs to the FIFO.

❑ The host writes address-tag/data pairs to the FIFO, via direct memory access (DMA).

❑ The host writes to raw, memory-mapped, graphics processor (GP) FIFO addresses.

In cases where the host writes data values directly to the chip through the register file, the FIFO overflow must be considered (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 host can read the values in this register at any time. When using DMA, the DMA controller automatically ensures that there is room in the FIFO before it performs further transfers. Thus, the host can pass a buffer of up to 64K 32-bit words to the DMA controller. Details of the FIFO and DMA controller operation are further described in subsections 2.3.3, *FIFO Control* and 2.3.4, *The DMA Interface*.

2.3.1 PCI Disconnect

The PCI bus protocol incorporates a feature known as PCI Disconnect, which is supported by the TVP4020. PCI Disconnect is enabled when the host writes to bit zero of the DisconnectControl register, which is at offset 0×68 in PCI Region 0. If the TVP4020 is in this mode and the host processor attempts to write to the full FIFO, instead of losing the write transaction, the TVP4020 chip asserts PCI Disconnect, and the host processor reinitiates the write cycle until it succeeds.

This feature allows faster downloading of data to the TVP4020 because the host does not need to poll the InFIFOSpace register. However, in this mode, the bus is saturated by the host processor until the TVP4020 frees up an entry in its FIFO. As a result, the PCI Disconnect should only be used for operations where either the TVP4020 can consume data faster than the host can generate it, or where there are no time-critical peripherals sharing the PCI bus.

2.3.2 Graphics Processor Active Bit

The graphics processor active bit is typically used when the data being transferred is expected to be absorbed quickly and the disconnect is not expected to be used often. In some systems, PCI Disconnect can lose interrupts if it is used too often or for too long. When transferring data, check the graphics processor active bit in the PCI Disconnect register to make certain that the graphics processor is not processing a large primitive. If this bit is set, disconnect stays disabled.

2.3.3 FIFO Control

Section 2.2, *TVP4020 as a Register File*, referred to the TVP4020 interface as a register file. More precisely, when the host writes a data value 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 does not update until the TVP4020 processes this entry. In the case where the TVP4020 is busy performing a time-consuming operation (for example, drawing a large texture-mapped polygon) and not draining the FIFO very quickly, the FIFO can become full. If the host writes to a register when the FIFO is full, no entry is put into the FIFO and that write transaction is lost.

The input FIFO is 32 entries deep, and each entry consists of a tag/data pair, an address word that addresses the register to be updated, followed by the data to be sent to the register. The host can read the InFIFOSpace register to determine how many entries are free. The value this register returns is never greater than 32.

An example of loading the TVP4020 registers using the FIFO follows. The pseudocode fills a series of rectangles. Details of the conventions used in the pseudocode examples are in Appendix B, *Screen Widths Table*.

In the following example, the data to draw a single rectangle consists of five words (including the render command).

```
dXDom(0x0);  // common set-up
dXSub(0x0);
dY(1);
for (i = 0; i < nrects; ++i) {
   while (*InFIFOSpace < 8)
      ;  // wait for room
   StartXDom (rect->x1);
   StartXSub (rect->x2);
   Count (rect->y2 - rect->y1);
   YStart(rect->y1);
   Render (TVP4020_TRAPEZOID_PRIMITIVE);
                                }
```

The InFIFOSpace FIFO control register contains 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 each time the host puts an entry into the FIFO. Before the host writes to the input FIFO, the user must check that there is sufficient space by reading the InFIFOSpace register.

The graphics core (GC) FIFO interface provides a port through which both GC register addresses and data can access the input FIFO. A range of 4K bytes of host space is provided although all data may flow through one address in the range. All accesses go directly to the FIFO; the range is provided to allow for data transfer schemes, which force the use of incrementing addresses.

This interface cannot read the GC registers. The TVP4020 reads the command buffers sent to the input FIFO interface via the DMA controller.

A data formatting scheme allows multiple data words to access the input FIFO with one address word where adjacent or grouped registers are being written to or when one register is being written to many times.

Note:

The FIFO interface is accessed at 32-bit boundaries. This allows a direct copy from a DMA format buffer.

2.3.4 The DMA Interface

Loading registers directly through the FIFO is often an inefficient way to download data to the TVP4020. Since the FIFO can accommodate only a small number of entries, the host must query the TVP4020 frequently to determine how much space is left. In situations where an API function requires a large amount of data to be sent to the TVP4020, if the host writes to the FIFO directly, a return from this function is not possible until the TVP4020 consumes almost all the data. This can take time, depending on the types of primitives being drawn.

To avoid these problems, the TVP4020 provides an on-chip DMA controller that loads data from arbitrary-sized (< 64K 32-bit words) host buffers into the FIFO. The host software prepares a host buffer that contains register address tag descriptions and data values. It then writes the base address of this buffer to the DMAAddress register and then counts the number of words to transfer to the DMACount register. Writing to the DMACount register starts the DMA transfer and the host can 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. At the same time, the TVP4020 reads data from the host buffer and loads it into its FIFO. FIFO overflow never occurs because the DMA controller automatically waits until there is room in the FIFO before transferring data.

There is one restriction on the use of DMA control registers. Before attempting to reload the DMACount register, the host software must wait until the previous DMA process is complete. The host can, however, load the DMAAddress register while the previous DMA process is in progress, because the address is latched internally at the start of the DMA transfer. Many display driver functions can be implemented using the following skeleton structure:

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

Using the DMA control register frees the host to return to the application. In parallel, the TVP4020 controls the DMA operation and drawing. This can increase performance significantly over loading a FIFO directly. In addition, some algorithms require that data be loaded multiple times (for example, drawing the same object across multiple clipping rectangles). Because the TVP4020 DMA control register only reads the buffer data, the data can be downloaded many times by restarting the DMA operation. This can be beneficial when composing the buffer data, which is a time-consuming task.

You can further optimize the DMA process by using a double-buffered mechanism with two DMA buffers, as shown in the format below. This allows the second buffer to fill before waiting for the previous DMA buffer to complete, thus, improving the parallelism between the host and the TVP4020 processor.

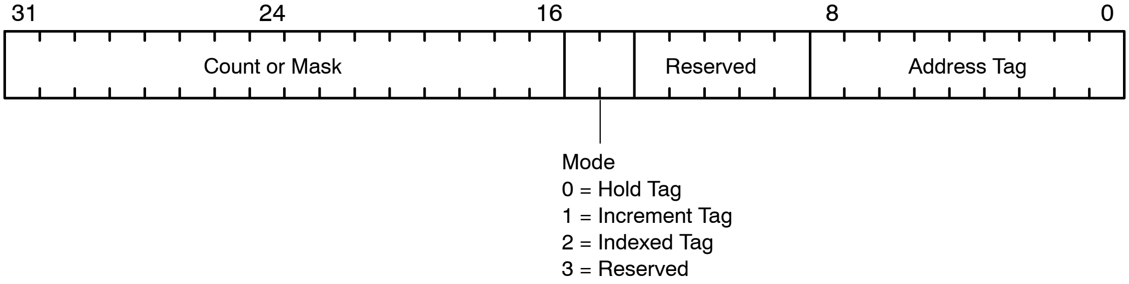
```
do any pre-work
get free DMA buffer and mark as in use
put render data into this new buffer
DMAAddress(address of new buffer)
while (TRUE) {
   count = *DMACount; // note this is volatile
   if (count) {
       while (--count)
           ; // wait for count to expire
              }
   else
       break;   // DMA completed
             }
DMACount(number of words in new buffer)
mark the old buffer as free
return
```

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.

2.3.4.1 DMA Tag Description Format

When the DMA process is performed, each 32-bit tag description in the DMA buffer conforms to the format shown in Figure 2−1.

Figure 2−1. DMA Tag Description Format



There are three tag addressing formats for DMA: hold, increment, and indexed. The formats reduce the amount of data that needs to be transferred, making better use of the available DMA bandwidth. Each of these formats is described in the following sections. Each row in the following formats represents a 32-bit value in the DMA buffer. The address tag for each register is given in Chapter 7, *Graphics Register Reference*.

2.3.4.2 Hold Format

The hold format is as follows:

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

value 1

...

value n
```

Set the SyncOnHostData bit in the render command to use the hold format for image download. In this format, the 32-bit tag description contains a tag value and a count that specifies the number of data words following in the buffer. The DMA controller writes each of the data words to the same address tag. This is useful for image download when the host continuously writes pixel data to the color register. The bottom nine bits specify the register where the data is written; the high-order 16 bits specify the number of data words (minus 1) that follow in the buffer, which are written to the address tag.

Note:

The 2-bit mode field for this format is zero. Thus, a given tag value can be easily loaded into the low-order 16 bits.

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

```
StartXDom

2 << 16

StartY

5 << 16

StartXSub12 << 16

Count

1

Render

(trapezoid render command)
```

2.3.4.3 Increment Format

The increment format is as follows:

```
address-tag with Count=n-1, Mode=1

value 1

...

value n
```

The increment 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; the TVP4020 updates an internal copy). Thus, this format allows contiguous TVP4020 registers to load by specifying a single 32-bit tag value followed by a data word for each register. The low-order nine bits specify the address tag of the first register to load. 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.

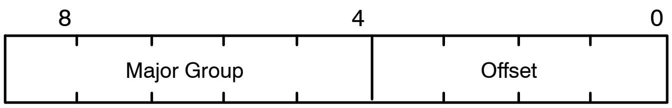
To enable this format, the TVP4020 register file is organized so that registers frequently loaded together have adjacent address tags. For example, the eight AreaStipplePattern registers can be loaded as follows:

```
AreaStipplePattern0, Count=7, Mode=1

row 0 bits

row 1 bits

...

row 7 bits
```

2.3.4.4 Indexed Format

The TVP4020 address tags are nine-bit values. For the indexed DMA format, these nine-bit values are organized into major groups. Within each group there are up to 16 tags. The low-order four bits give the offset value within the group. The high-order five bits give the major group number. See Chapter 8, *Register Tables*, for a listing of the individual registers with their major group and offset value.

Figure 2–2. Indexed Register Format



The indexed register format allows up to 16 registers within a group to be loaded while specifying only a single address-tag description word.

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

If the address-tag description word is set to indexed format, the high-order 16 bits are used as a mask to indicate which registers within the group are to be used. The bottom four bits of the address-tag description word are not used. The group is specified by bits 4 through 8. Each bit in the mask represents a unique tag within the group. If a bit is set, then the corresponding register is loaded. The number of bits set in the mask determines the number of data words to follow the tag description word in the DMA buffer. The data is stored in order of the increasing, corresponding address tag. For example:

```
0x003280F0
value 1
value 2
value 3
```

The format bits are set to 2, corresponding to the indexed format. The mask field (0×0032) has three bits set so that three data words follow the tag description word. Bits 1, 4, and 5 are set so that the tag offset values are 1, 4, and 5. The major group is set by bits 4−8, which are 0×0F (in indexed format, bits 0−3 are ignored). Thus, the actual registers to update have address tags 0×0F1, 0×0F4 and 0×0F5. These are updated with value 1, value 2, and value 3, respectively.

2.3.4.5 DMA Example

The following pseudocode duplicates the previous example of drawing a series of rectangles, but it uses the DMA controller. This example uses a single DMA buffer and the simplest hold format 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++ = TVP4020TagdXDom;
*pbuf++ = 0;
*pbuf++ = TVP4020TagdXSub;
```

```
*pbuf++ = 0;
*pbuf++ = TVP4020TagdY;
*pbuf++ = 1 << 16;
for (i = 0; i < nrects; ++i) {
   *pbuf++ = TVP4020TagStartXDom;
   *pbuf++ = rect->x1 << 16;// Start dominant edge
   *pbuf++ = TVP4020TagStartXSub
   *pbuf++ = rect->x2 << 16;// Start of subordinate edge
   *pbuf++ = TVP4020TagCount;
   *pbuf++ = rect->y2 - rect->y1;
   *pbuf++ = TVP4020TagYStart;
   *pbuf++ = rect->y1 << 16;
   *pbuf++ = TVP4020TagRender;
   *pbuf++ = TVP4020_TRAPEZOID_PRIMITIVE;
                               }
// initiate DMA
DMACount((int)(pbuf - dma_buffer))
```

In the above example, a previously allocated host buffer is pointed at by `dma_buffer`. For this example, significantly less data would be required using indexed tags.

2.3.4.6 DMA Buffer Addresses

The host software must generate the correct DMA buffer address for the TVP4020 DMA controller. This typically means that the address passed to the TVP4020 must be the physical address of the DMA buffer in the host memory. The buffer must also reside at contiguous physical addresses, as accessed by the TVP4020. On a system that 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 noncontiguous physical memory, the buffer must divide into sets of contiguous physical memory pages and each of these sets must transfer separately. In this situation, the whole DMA buffer cannot transfer at once; the host software must wait for each set to transfer. Often, the best way to handle these fragmented transfers is through an interrupt handler.

2.3.4.7 DMA Interrupts

The TVP4020 provides interrupt support as an alternative means of determining when a DMA transfer is complete. This interrupt support can provide a considerable speed advantage. If enabled, the interrupt generates whenever the DMACount register changes from a nonzero to a zero value. Since the DMACount register decrements every time a data item transfers from the DMA buffer, it also occurs when the last data item transfers from the DMA buffer.

To enable the DMA interrupt, the DMAInterruptEnable bit must be set in the IntEnable register. The interrupt handler checks the DMAFlag bit in the IntFlags register to determine that a DMA interrupt actually occurred. To clear the interrupt, the DMA handler program writes a word 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 pseudocode above only hints at the system details.

This scheme frees the processor for other work while the DMA process is finishing. Because the overhead of handling an interrupt is often quite high for the host processor, you must tune the scheme to allow the host a period of polling, before it sleeps on the interrupt.

2.3.5 Output FIFO and Graphics Processor FIFO Interface

To read data back from the TVP4020, an output FIFO is provided. Each entry in this FIFO is 32 bits wide and can hold either tag or data values. Thus, its format is unlike the input FIFO whose entries are always tag/data pairs (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 that the TVP4020 writes to the output FIFO is controlled by the FilterMode register. This register allows filtering of output data in various categories, including the following:

- Depth: The output in this category results from an image upload of the depth buffer.
- Stencil: The output in this category results from an image upload of the stencil buffer.
- Color: The 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 two bits per category. If the least significant of these two bits is set (0×1), the output of that category's register tag is enabled. If the most significant bit is set (0×2), the output of that category's data is enabled. Both the tag and data output can be enabled at the same time. In this case, the tag is written first to the FIFO and then followed by the data. The FilterMode register is described in more detail in Section 4.16, *Host Out Unit*.

For example, to perform an image upload from the framebuffer, the FilterMode register enables output data for the color category. Then the rectangular area to be uploaded is described to the rasterizer. Each pixel that the TVP4020 reads from the framebuffer is placed into the output FIFO. When the output FIFO is full, the TVP4020 blocks pixel output, internally, until space becomes available. The programmer is responsible for reading all data from the output FIFO; for example, by first calculating how many pixels will result from an image upload and then reading exactly that amount from the FIFO.

The Output FIFO Words register must be read first by the host to determine the number of entries, or 32-bit data items, in the FIFO (reading from the FIFO when it is empty returns undefined data). This procedure repeats until all the expected data or tag items are read. The address of the output FIFO is described below.

All expected data must be read back by the host. The TVP4020 blocks the data when the output FIFO is full. You must take precautions to avoid the deadlock condition that results when the host is waiting for space to become free in the input FIFO while, at the same time, the TVP4020 is waiting for the host to read data from the output FIFO.

2.3.6 Graphics Processor FIFO Interface

The TVP4020 has a sequence of 1k×32-bit addresses in the PCI Region 0 address map called the graphics processor FIFO access**.** The TVP4020 can read any address in this output FIFO range (typically a program chooses the first address and uses 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 that force the use of incrementing addresses.

Writing to a location in this address range provides raw access to the input FIFO. Again, the TVP4020 typically chooses the first address in which to write. Thus, the same address can be used for both input and output FIFOs. Reading this location 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. Because the register file has a unique address for each register, writing to this unique address allows the TVP4020 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 TVP4020 DMA buffer, it is possible to transfer data to the TVP4020 by constructing a DMA-style data buffer and then copying each item from this buffer to the raw input FIFO address. Based on the tag descriptions and data written, the TVP4020 constructs tag/data pairs to enter the input FIFO as real FIFO entries. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

When the host writes to the raw FIFO address, it must still check the FIFO-full condition by reading the InFIFOSpace register. Writing tag descriptions, however, does not move data entries into the FIFO; such a write transaction simply establishes a set of tags to be paired with the subsequent data. Thus, free space is necessary 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 and the FIFO is empty, 32 write transactions are possible before there is a need to check again for free space.

See the *TVP4020 3D Graphics Processor Data Manual* for more details on the graphics processor FIFO interface address range.

2.4 Interrupts

All interrupts can be individually enabled and disabled. See the *TVP4020 3D Graphics Processor Data Manual* for more details.

2.5 Synchronization

There are two main cases when the host must synchronize with the TVP4020:

❑ Before reading back from the TVP4020 registers
❑ Before directly accessing the memory via the bypass mechanism

Also, the host must synchronize with the TVP4020 for framebuffer management tasks such as double buffering, though this may be better handled using the SuspendUntilFrameBlank command. Synchronizing with the TVP4020 means waiting for any pending DMA process to finish and the chip to complete any processing currently being performed. The following pseudocode shows the general scheme:

```
TVP4020Data  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, wait for the DMA process to finish as normal. Then wait for space to become free in the FIFO (because the DMA controller actually loads the FIFO). Space for two registers is needed: one register to enable generation of an output sync value, and the second register for the sync command itself. The enable flag can be set at initialization time. The output value generates only when a sync command is sent and the TVP4020 completes all processing.

Rather than polling, you can use a Sync interrupt. 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 generates when the tag or data reaches the output end of the host-out FIFO. You must carefully consider using the sync interrupt because the TVP4020 generally empties the FIFO more quickly than it sets up and handles the interrupt.

2.6 Host Memory Bypass

Typically, the host accesses memory indirectly through commands sent to the TVP4020 FIFO interface. However, the TVP4020 provides the whole memory as part of its address space so that it can be memory mapped by an application. Access to the memory through this route is independent of the TVP4020 FIFO.

Drivers use direct access to memory for algorithms that are not supported by the TVP4020 or for better performance in some specific cases. One case, for example, might be when multiple pixels are written simultaneously and there is minimal host software overhead.

A driver using the bypass mechanism synchronizes memory accesses made through the FIFO with those made directly through the memory map. If the TVP4020 writes data to the FIFO and then accesses the memory, the memory access occurs before the commands in the FIFO are fully processed. This lack of temporal ordering is generally undesirable.

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

2.7 DMA Controller

The DMA controller allows data transfer from the PCI bus to the TVP4020 memory. This controller is independent of the DMA controller that feeds the graphics processor FIFO and supports rectangular data structures and data formatting.

2.8 Register Read Back

Under some operating environments, multiple tasks access the TVP4020 chip. Sometimes a server task or driver arbitrates access to the TVP4020 on behalf of multiple applications. In these circumstances, the state of the TVP4020 chip may need to be saved and restored on each context switch. To facilitate this, the TVP4020 registers can be read back. Internal and command registers cannot be read back. For details on which registers are readable, see Chapter 8, *Register Tables*.

To perform a context switch, the host must first synchronize with the TVP4020. This means sending a sync command and waiting for the sync output data to appear in the output FIFO. Following this, the registers can be read back.

To read the TVP4020 register, the host reads the same address that is used for a write transaction, that is, the base address of the register file plus the offset value for the register.

Since internal registers cannot be read back, the programmer must take precautions when context switching a task that is making use of continue-draw commands. Continue-draw commands rely on the internal registers to maintain their previous state. This state will be destroyed by any rendering work performed by a new task. To prevent this, continue-draw commands must be performed through DMA processing since the context switch code has to wait for outstanding DMA processes to complete. Alternatively, continue-draw commands can be performed in a nonpreemptable code segment.

Reading back individual registers should be avoided because the process can adversely affect chip synchronization and performance. It is more appropriate for your program to keep a software copy of the register and update whenever the register changes.

2.9 Byte Swapping

Internally, the TVP4020 operates in little-endian mode. However, the TVP4020 works with both big-endian and little-endian host processors. Since the PCI bus specification indicates that byte ordering is preserved regardless of the size of the transfer operation, the TVP4020 provides facilities to handle byte swapping. See the *TVP4020 3D Graphics Processor Data Manual* for more details of byte swapping on the PCI bus.

Additional support is provided within the graphics core of the chip-to-byte swap images and bitmasks, as they are transferred to and from the host. These support mechanisms are documented in Chapter 4, *Graphics Programming*.

2.10 Red and Blue Swapping

For a given graphics board, the RAMDAC™ and/or API forces a given interpretation for true-color pixel values. For example, 32-bit pixels are interpreted as either RGB (red at byte 2, green at byte 1 and blue at byte 0) or BGR (blue at byte 2 and red at byte 0). The byte position for red and blue may be important previously written software. Expect one byte order or the other, particularly when handling image data stored in a file.

The TVP4020 provides three registers to specify the byte positions of blue and red internally. In the texture/fog/blend unit, the AlphaBlendMode register contains a one-bit field called ColorOrder. If the register sets the ColorOrder bit to zero, the byte order is BGR; if it sets the bit to one, the order is RGB. This bit must be set in the color format unit, and in the texture read unit through the DitherMode and TextureDataFormat registers.

Chapter 3

Memory I/O and Organization

This section describes the arrangement of data stored in memory. The TVP4020 has a single unified memory space that is divided into three buffers: the localbuffer, the framebuffer, and the texture buffer. These buffers can be of any size and at any position in the memory.

For 3-D operations associated with the framebuffer, a localbuffer holds depth and/or stencil information. A texture buffer is present, when needed. For 2-D operations, the localbuffer is not generally used, but the texture buffer may be used to store pixel maps.

Topic **Page**

3.1 Patching

The TVP4020 supports an optional scheme, known as “patching,” for organizing memory. Data is usuallly stored linearly in memory, so that incrementing addresses move from left to right along a scanline of the appropriate buffer. The TVP4020 memory type uses a page structure that allows fast access within a 2K-byte region but imposes a penalty for moving to a new 2K-byte region. This page structure favors access patterns that move along a scanline but is not efficient for moving patterns vertically. The large change in address can cause a page break.

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

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

3.2 Localbuffer

The localbuffer holds depth and stencil information that corresponds to each displayed pixel. The depth field can be either 15 or 16 bits wide and the stencil field either one or zero bits wide. The total width of the localbuffer data cannot be greater than 16 bits. If a stencil field is defined, then it occupies bit 15; the depth field always starts at bit zero.

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

3.2.1 Localbuffer Coordinates

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

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

Bottom-left origin:

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

Top-left origin:

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

where:

X = the pixel X coordinate.

Y = the pixel Y coordinate.

LBWindowBase = the base address in the localbuffer of the current window.

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

W = the screen width. Only a subset of widths is supported and encoded into the PP0, PP1, and PP2 fields in the LBReadMode register. See Appendix B, *Screen Widths Table*, for more details.

This produces the localbuffer address in pixels. For the TVP4020, the localbuffer data is always 16 bits, so the physical byte address is twice the pixel address. The destination address is the address to which the data is written; data may also be read from this address if read-modify-write operations are needed, such as depth testing. The source address is mainly used for copy operations and reading data.

3.3 Framebuffer

The framebuffer holds color data produced by the TVP4020. The framebuffer may hold both displayed and nondisplayed data. Color buffers can be placed anywhere in memory; there is no restriction on areas where data can be displayed.

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

To access alternative buffers, load either the FBPixelOffset register or redefine the base address of the window held in the FBWindowBase register.

3.3.1 Framebuffer Coordinates

Generating coordinates for the framebuffer is similar to the process for the localbuffer except for the addition of FBPixelOffset. The WindowOrigin bit in the FBReadMode register selects either the top-left or bottom-left corner as the origin for the framebuffer.

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

Bottom-left origin:

```
Destination address = FBWindowBase – Y * W + X +
   FBPixelOffset
Source address = FBWindowBase – Y * W + X + FBPixelOffset
   +  FBSourceOffset
```

Top-left origin:

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

where:

X = the pixel X coordinate.

Y = the pixel Y coordinate.

FBWindowBase = the base address in the framebuffer of the current window.

FBPixelOffset = zero except when multibuffer write operations are needed to access pixels in alternative buffers without changing the FBWindowBase register. This is useful as the window system may asynchronously change the window's position on the screen. The FBPixelOffset is held in the FBPixelOffset register.

FBSourceOffset = zero except during a copy operation when data is read from one address and written to another. The FBSourceOffset is held in the FBSourceOffset register.

W = the screen width. Only a subset of widths is supported and encoded into the PP0, PP1 and PP2 fields in the FBReadMode register. See Appendix B, *Screen Widths Table*, for more details.

These address calculations translate a 2-D address into a linear address to create a smaller framebuffer width to a nonpower of two (for example, 640). The address is in pixels; the physical byte address is calculated by multiplying the pixel address by the number of bytes in the pixel.

The width is specified as the sum of partial products selected by the fields PP0, PP1, and PP2 in the FBReadMode register. This is the same mechanism used to set the width of the localbuffer; however, the widths may be set independently. The range of supported widths is tabulated in Appendix B, *Screen Widths Table*, 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 bitmaps, the TVP4020 chooses the next largest width from the table. The difference between the table width and the bitmap width is the unused strip of pixels down the right-hand side of the bitmap.

Such bitmaps can only be copied to the screen as a series of scanlines rather than as a rectangular block, unless the texture read unit is used. In this case, the read stride can be set differently from the write stride using partial products. However, windowing systems often store offscreen bitmaps in rectangular regions that use the same stride as the screen. In this case, normal bit-aligned block transfers (BitBlts) can be used.

3.3.2 Framebuffer Color Formats

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

❑ As a collection of nonmeaningful and unformatted fields of up to 32 bits

Bit planes may be allocated to the control cursor and color look-up tables (LUTs), and to multibuffer visibility or priority functions. In this case, the TVP4020 sets and clears bit planes quickly but does not perform any color processing such as interpolation or dithering. All color processing can be disabled so that raw read and write operations, writemasking, and logical operations are performed. This allows the framebuffer to update and modify the control planes, as necessary.

❑ As a collection of one or more color components

All the processing of color components, except for the final writemask and logical operations, is done using the internal color format. The final stage before writemask and logical operations processing converts the internal color format to that required by the physical configuration of the framebuffer and video logic. The range of supported formats is given in Table 3–1. The nomenclature n@m means this component is *n* bits wide and starts at bit position m in the framebuffer. The least significant bit position is zero, and a dash in a column indicates that this component does not exist for this mode.

In addition, there are some important points to note:

❑ When present, the alpha channel is always associated with the RGB color channels rather than identified as a separate buffer.

This allows it to move in parallel with the color channels and function correctly during multibuffer updates and double buffering.

❑ For the front and back modes, the data value is duplicated in both buffers.

In general, if the data format does not take 32 bits, the data is repeated in the empty bit planes. If the data format requires eight bits, the same value is repeated in all four bytes of the word. The pixel size determines how many of the bytes are written to memory. In a 16-bit format (for example, 5:5:5:1), the data is repeated in the upper and lower halves of the word. If the pixel size is set to 16 bits, only half the word is written to memory; if the pixel size is set to 32 bits then both halves are written to, with the same data in each. A writemask selects which bits are written. This is used for certain types of double buffering. The front and back modes are used in the alpha blend unit to extract the appropriate buffer.

❑ The offset modes (10 and 11) format the colors into a 7-bit value and then add 64 to the result.

This format eliminates reserved entries in window-system color tables.

- YUV (luminance, chrominance) formats are only available as textures.

  The TVP4020 can convert YUV textures to RGB and apply them to polygons; it cannot convert RGB to YUV for storage. If a YUV texture is loaded into the chip, it should be loaded as raw data or converted to RGB as it is loaded.

- The CI4 format is only available as a texture.
- When reading the framebuffer, RGBA (red, green, blue, alpha) components are scaled to their internal width for alpha blending, if needed.
- The color format of the framebuffer is independent of the color format of the texture buffer; the texture buffer supports the same formats as the framebuffer plus additional formats for YUV color.

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

The color format information must be stored in three places: the DitherMode register, the AlphaBlendMode register, and the TextureDataFormat register.

Table 3−1. Supported Color Formats

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

Notes: 1) The DitherMode register does not support the YUV444, YUV422, or CI4 formats.
2) The AlphaBlendMode register does not support the YUV444, YUV422, or CI4 formats.

3.3.3 Special Memory Modes

The TVP4020 uses synchronous graphics RAM (SGRAM) to store data. SGRAM devices usually have special features that are particularly useful for graphics.

3.3.3.1 Hardware Writemasks

Hardware writemasks support writemasking in the framebuffer and do not incur a performance penalty. If hardware writemasks are not available, the TVP4020 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, clear the ReadSource and ReadDestination enable modes in the FBReadMode register. This writes the required writemask to the FBHardwareWriteMask register, sets the FBSoftwareWriteMask register to all ones, and sets the number of framebuffer read operations to zero (for normal rendering).

To use software writemasking (if hardware masks are not available), set the ReadDestination enable mode in the FBReadMode register. This writes the required writemask to the FBSoftwareWriteMask register and sets the number of framebuffer read operations to one (for normal rendering).

3.3.3.2 Block Write Operations

Block writing writes to consecutive pixels in the framebuffer simultaneously. Writing is useful when filling large areas but X-has some restrictions:

❏ No depth or stencil testing can be performed.

❏ All the pixels must be written with the same value so no color interpolation, alpha blending, dithering, or logical operations can be performed.

Block writing is not restricted to rectangular areas and can be used for any trapezoid. Hardware writemasking is available during block write operations, but software writemasking is not. The scissor tests and the extent checking process operate correctly with block write operations, and bitmask patterns can be applied.

The FBBlockColor register holds the value to write to each pixel. This register must not be updated immediately following any render command that performs a block write operation.

Sending a render command with the PrimitiveType field set to trapezoid and the FastFillEnable field set results in block filling of the area. During a block fill operation, any inappropriate state is ignored. Thus, whenever stippling, color interpolation, depth testing, and/or logical operations are enabled, the inappropriate states are ignored and have no effect. However, scissor clipping does function correctly when block writing.

The TVP4020 always writes 32 pixels per block fill. It takes care of any partial blocks at the beginning or end of spans.

3.4 Double Buffering

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

For further details see subsections 4.13.3, *Alpha Blending*; 4.13.7, *Fog Example*; and 4.15, *Logical Op Unit*; and refer to the *TVP4020 3D Graphics Processor Data Manual*.

3.4.1 BITBLT Double Buffering

Bit-aligned block-transfer (BITBLT) double buffering provides independent, windowed, double buffering by BLTing (transferring a bit-aligned block that copies a rectangular array of bitmap pixels from one location to another) an area from one buffer to the other. It requires the maintenance of a complete duplicate buffer of nondisplayed RAM. To swap buffers, perform BITBLT double buffering to the displayable area. The characteristics of BITBLT double buffering include:

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

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

3.4.2 Full-Screen Double Buffering

This section describes how to implement full-screen double buffering with the TVP4020 when using the video timing generator. To perform full-screen double buffering, the available display RAM must be partitioned into two parts: buffer zero and buffer one, each of which contains enough memory to display a full screen of pixel information. The partitioning consists of deciding the offset value of the RAM at which a given buffer starts. This offset value is used to program the various TVP4020 registers. For a given resolution and pixel depth, there must be enough display adapter RAM to make this possible. For example, with 32-bit deep pixels and 4M bytes of RAM, you can implement full-screen double buffering at 800×600 resolution, but not at 1024×768.

There are two factors to consider for full-screen double buffering. First, the video output hardware must be configured to display the pixels from the correct buffer. Second, the TVP4020 chip must be programmed to render the correct buffer. To achieve smooth animations, the buffer being rendered is usually different from the buffer being displayed.

3.4.2.1 Video Output

To display a given buffer, the video output hardware must be programmed with the offset value of that buffer in RAM. In the TVP4020 internal timing generator, this is controlled by the ScreenBase register located in the TVP4020 control space at offset 0×3000.

3.4.2.2 TVP4020 Rendering

When determining the memory location of a pixel being rendered, the TVP4020 operates in screen coordinates.

To simplify the calculation of pixel coordinates loaded into the TVP4020, you may load this value into the FBPixelOffset register. The last thing the TVP4020 does before passing a pixel address to the framebuffer interface is to add the value in the FBPixelOffset register to its address. Thus, it is possible to move the rendering origin to any pixel location in memory. When swapping buffers, this can be the pixel location at which a given buffer starts.

These values can be precalculated at system start-up, and loaded as required.

3.4.2.3 Synchronization

Double buffering allows the displaying of one buffer (the front buffer) while rendering the other (the back buffer). When rendering to the back buffer finishes, the buffers are swapped and rendering continues into what was previously the front buffer, now the new back buffer. As a general rule, buffers should not be swapped until all rendering to the back buffer is completed. This eliminates visible tearing or screen break-up during the buffer swap.

The TVP4020 reads the ScreenBase register at the end of each vertical blanking period to determine the starting pixel for the next displayed frame. Thus, the TVP4020 can write to this register at any time in order to swap buffers. This swap takes effect on the next frame. The same is not true for loading the FBPixelOffset register. This register updates as soon as the command to load works its way through the input FIFO. As a result, any rendering that takes place after the FBPixelOffset loads occurs in the new buffer. This can result, detrimentally, in rendering being displayed before the buffers are swapped. Thus, to avoid picture breakup, avoid the following scheme:

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

```
FBPixelOffset = 0               // draw to buffer 0 now
Render Commands                 // draw next frame
```

There are two problems with this scheme. First, even though the write to the ScreenBase register occurs immediately, the TVP4020 does not actually swap the buffers until the end of the next vertical blanking period. Thus, the start of the next frame rendering may be viewed in the front buffer prior to the buffer swap. Second, once a command is loaded into the input FIFO, the host is free to continue with other work while the TVP4020 executes the command. All accesses to the ScreenBase register bypass the FIFO. Thus, the host can update the FIFO and the buffer swap can occur before the TVP4020 completes a rendering of the last frame.

The TVP4020 includes the SuspendUntilFrameBlank command. It solves the above problems without the host needing to synchronize with the TVP4020. Here is the preferred version of the above example:

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

The SuspendUntilFrameBlank command flushes all outstanding reads and writes to the framebuffer, and prevents any further framebuffer memory accesses until after the buffers are swapped.

The data that the host loads into the SuspendUntilFrameBlank command enables the TVP4020 to swap the buffers automatically when the vertical blank (VBLANK) occurs. The TVP4020 loads a new buffer offset value into the ScreenBase register, as discussed previously. See the detailed description in Chapter 7, *Graphics Register Reference*.

Thus, a single command register access ensures that:

- All rendering completes to the back buffer.
- The chip will wait for VBLANK before carrying out the swap.
- The host can continue sending rendering commands to the TVP4020 without affecting the displayed buffer.

To enhance performance, send nonframebuffer related commands to the TVP4020 following the SuspendUntilFrameBlank command. This allows better overlap between the host and the TVP4020. In general, any commands that do not render to the framebuffer can be queued in the TVP4020 FIFO before waiting on VBLANK.

Eventually, the host sends more framebuffer rendering commands, and the TVP4020 stalls its hyperpipeline until the buffer swap is completed. Ideally, the host uses this time to perform nonrendering operations (for example, to prepare additional DMA buffers for rendering and swapping).

With this scheme, the host does not usually need to wait for VBLANK, unless it is accessing framebuffer memory through the bypass.

The LineCount register can be polled to wait for VBLANK. A VBLANK interrupt is also available (see *the TVP4020 3D Graphics Processor Data Manual* for details). The LineCount register is reset at the start of the VBLANK period and is incremented by one, for each scanline, as the video scanner moves down the screen. As a result, polling this register for a value less than the value held in the VbEnd register indicates that the TVP4020 is in the VBLANK period.

3.4.3 Bitplane Double Buffering

Use bitplane double buffering at 32-bits-per-pixel framebuffer depth with 32768 colors in 5:5:5:1 true-color mode. A RAMDAC board selects between the high and low 16 bits of input stream based on whether bit 31 is set or clear. The front and back buffers for each pixel become interleaved within the same 32-bit word in the framebuffer; that is, buffer zero becomes the lower 16 bits, and buffer one becomes the upper 16 bits.

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

When rendering graphical user interface (GUI) data (such as window borders, titles etc.), bit 31 must always be set to the same value so that the pixels are always displayed from the same buffer. The hardware writemask can then write to either the highest or lowest 16 bits, when rendering the animated contents of a window.

The characteristics of bitplane double buffering are:

- ❑ Almost instantaneous buffer swapping
- ❑ No offscreen buffer required (for example, 1152×900 would be the maximum resolution on a 4MB framebuffer at 32-bpp depth.)

- ❑ Multiple windows can be double buffered. A GUI can write information with no performance penalty.
- ❑ Useful at 5:5:5:1 RGB color depth

In order to allow the Microsoft™ Windows™ 95 device independent bitmap (DIB) engine to render directly to the framebuffer in the 5:5:5:1 format, a special framebuffer bypass option is supported. It presents the front and back buffers uninterleaved, that is, as a 5:5:5:1 16-bpp packed framebuffer. This allows rarely used complex primitives to be rendered by software.

3.4.4 Panning

Use display panning by setting the ScreenBase and ScreenStride registers. The ScreenBase register defines where in the framebuffer the image starts. For panning to work, the image in the framebuffer must be larger than the one to display. The ScreenStride holds this difference in terms of 64-bit units per scanline. For example, with a screen width of 640 pixels and a framebuffer image width of 660 32-bit pixels, the ScreenStride must be set to 10.

3.5 Texture Buffer

The texture buffer is very similar to the framebuffer. The framebuffer write unit stores textures in formats that the framebuffer supports and loads them into memory. If you find that the texture format is different from the framebuffer format, temporarily set the DitherMode register to the texture format during texture loads. The texture read unit reads the textures.

If the texture is already in the correct format, you can use a fast texture load by writing raw texture data to the TextureData register. Raw data is 32 bits wide, with the correct bit pattern for memory storage. No data formatting or packing is performed, so the texture must be preprocessed for a fast texture load. The texture is stored linearly in memory from the address specified in TextureDownLoadOffset, which is automatically incremented. No patching is done, so if the texture is patched it must be done by the host. This method eliminates setting up the rasterizer and changing the state of the pipeline.

3.5.1 Texture Buffer Coordinates

The Texture Address unit forms texture coordinates and passes them to the Texture Read unit. In place of the rasterizer x- and y-coordinate system, the texture address unit generates S and T values.

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

Bottom-left origin

```
Texture address = TextureBaseAddress – T * W + S
```

Top-left origin

```
Texture address = TextureBaseAddress + T * W + S
```

where:

S = the texel S coordinate.

T = the texel T coordinate.

TextureBaseAddress = the base address in the framebuffer of the current window.

W = the texture map width. Only a subset of widths is supported and encoded into the PP0, PP1, and PP2 fields in the TextureReadMode register. See Appendix B, *Screen Widths Table*, for more details.

These address calculations translate a 2-D address into a linear address to create a small texture width to a nonpower of two (for example, 640). The width of the texture map used for these calculations is independent of the width and height used for texture effects such as repeat or clamp. The address is in texels; the physical byte address is calculated by multiplying the texel address by the number of bytes in the texel.

3.5.2 Texture Color Formats

Texture maps use the same formats as the framebuffer, plus YUV and four-bit color index formats (see subsection 3.3.2, *Framebuffer Color Formats*, for details). The formats of the texture map and framebuffer do not have to be the same.

Chapter 4

Graphics Programming

This chapter describes the TVP4020 graphics hyperpipeline and its units, and shows how to render a simple graphic primitive.

Topic **Page**

4.1 The Graphics Hyperpipeline

The graphics hyperpipeline, or graphics processor, supports:

- Point, line, triangle and bitmap primitives
- Flat and Gouraud shading
- Texture mapping, fog, and alpha blending
- Scissor and stipple
- Stencil test, depth (*Z*) buffer test
- Dithering
- Logical operations

The units in the graphics hyperpipeline are:

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

Figure 4−1. Graphics Hyperpipeline

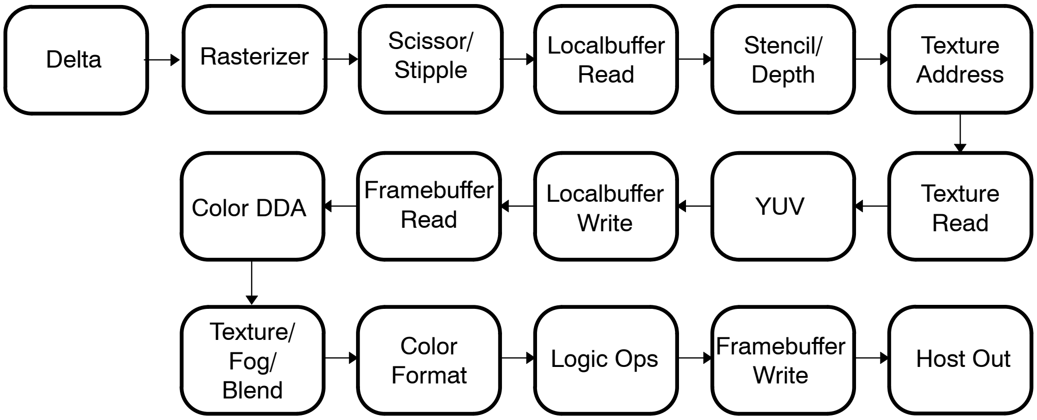
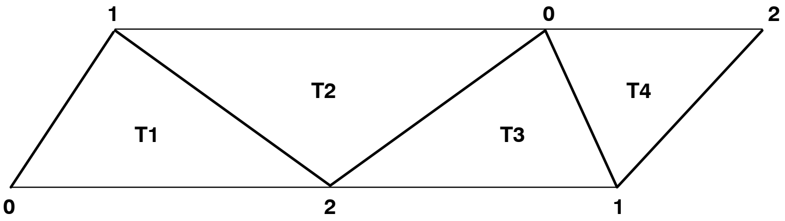


Figure 4−1 shows the order in which hyperpipeline operations are performed. The Scissor/Stipple unit operates before the texture address generator so that any fragments which fail a stipple test do not create a texture access. All units in the hyperpipeline operate independently. For example, enabling the XOR Logic Op does not automatically enable a framebuffer read mode; you must perform this explicitly.

Parameters are not corrupted by the calculations. As a result, you can initiate the sharing of parameters between primitives by not reloading those parameters. For example, if you load the first triangle in a triangle strip into V0, V1, and V2, then the next triangle will load into V0, the next V1, and so on as shown in Figure 4−2.

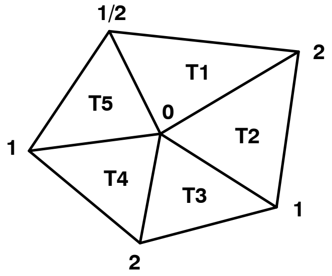
Figure 4−2. Triangle Mesh



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

Similarly, a triangle fan may be implemented by initially loading V0, V1, and V2 and then loading V1 and V2 as shown in Figure 4−3. Note that T1 and T5 share a vertex that is loaded first in V1 and then in V2.

Figure 4−3. Triangle Fan



Individual triangles, strips, or fans may be backface culled such that triangles that face away from the viewer are not drawn. The sign of the triangle area detects backfacing triangles but whether it rejects positive or negative areas depends on the definition of the triangle format (whether the vertices are clockwise or counterclockwise). The format may also vary when drawing meshed primitives, such as a strip where the area sign alternates triangle by triangle. When you enable backface culling in the Delta unit, you can set the sign to reject areas in each triangle as the triangle is drawn.

Lines are handled slightly differently in that only V0 and V1 are used for drawing. The direction of the line is defined as part of the command. As a result, a line can run either from V 0 to V1 or from V1 to V0. A polyline may be drawn by loading the first vertex into V0, the second vertex into V1, the third vertex into V0, the fourth vertex into V1, and so on.

4.2 Delta Unit

For best performance, use the Delta unit to calculate the edge deltas used by the graphics processor. The Delta unit accepts vertex parameters as shown in Table 4−1.

Table 4−1. Vertex Parameters

Offset	Category	Parameter	Fixed Point Format	IEEE Single Precision Floating Point Range
0	Texture	s	2.30 s (see Note 1)	−1.0...1.0 (see Note 2)
1		t	2.30 s	−1.0...1.0
2		q	2.30 s	−1.0...1.0
3		Ks	2.22 us	0.0...2.0
4		Kd	2.22 us	0.0...1.0
5	Color	red	1.30 us	0.0...1.0
6		green	1.30 us	0.0...1.0
7		blue	1.30 us	0.0...1.0
8		alpha	1.30 us	0.0...1.0
9	Fog	f	10.22 s	−512.0...512.0
10	Coordinate	x	16.16 s	−32K...+32K (see Notes 3 and 4)
11		y	16.16 s	−32K...+32K
12		z	1.30 us	0.0...1.0
14	Packed Color	Packed Color	8888	8888

Notes:

1) This is the range when normalize is not used. When normalize is enabled, the fixed-point format can be anything, providing it is the same for the s, t, and q parameters. The numbers are interpreted as if they had 2.30 format for the purpose of conversion to floating point. If the fixed-point format (2.30) is different from what the user had in mind, then the input values are prescaled by a fixed amount (i.e., the difference in binary point positions) prior to conversion.
2) This is the range when normalize is not used. When normalize is enabled, the range is extended to $2^{\pm 32}$ approximately. This also applies to the t and q values as well.
3) The normal range here is limited by the size of the screen.
4) K = 1024

While you can write values to vertex storage in either floating- or fixed-point format, any values returned through the readback mechanism result in the clamped floating point (IEEE single precision) version of the value written. The returned value of a parameter may be different from the value written under the following conditions:

- ❑ Any clamping occurs
- ❑ The input number was a NaN or denormalized IEEE number
- ❑ The input value exceeds the internal range (approximately $\pm 2^{32}$)

The texture parameters (S, T, and Q) must be handled differently than the other parameters as their range must be constrained to get the best results from the finite precision DDA and the perspective division hardware in the graphics processor. Any operation on the texture parameters before they are used is controlled by the TextureParameterMode in the DeltaMode register.

The options are NoClamp, Clamp, or Normalize. The NoClamp and Clamp options work the same as with the other parameters. The Normalize option finds the maximum absolute value of the texture S, T, and Q values for the primitive and normalizes all the values for the range −1.0 through 1.0, inclusive, prior to being used in the set-up calculations. The normalize option, which allows normalization to work on a triangle-by-triangle basis across a triangle mesh, does not change the texture values in vertex storage.

4.2.1 Drawing Commands

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

The Draw and Repeat commands prompt Delta to calculate the required rendering device data and update the Start, dX and dyDom registers in the graphics processor Rasterizer, Color, Depth, Texture and Fog units. Any additional registers in the Rasterizer unit are also loaded (the RasterizerMode register is not updated). Finally, the Render and ContinueNewSub commands are sent to the rendering devices.

The data field that accompanies the DrawTriangle or DrawLine command controls some of the aspects of the Delta operation, in conjunction with the DeltaMode register. The relevant bits in the Draw command and their affect on the Delta unit are described in Table 4−2. The values in the remaining bits must be compatible with the desired operation.

Table 4−2. Draw Command Bit Field Assignments Affecting Delta

Bit. No.	Name	Description
13	TextureEnable	When set (and qualified by the TextureEnable bit in the DeltaMode register), it calculates the texture values (S, T, and Q).
14	FogEnable	When set (and qualified by the FogEnable bit in the DeltaMode register), it calculates the fog values.
16	SubPixelCorrection-Enable	When set (and qualified by the SubPixel-CorrectionEnable bit in the DeltaMode register), it enables the subpixel correction of any value interpolated in the Y direction. The rendering devices perform the subpixel corrections in the X direction.
20	RejectNegativeFace	Qualified by the BackFaceCull field in the DeltaMode register. If set, it rejects triangles with a negative area. If cleared, it rejects triangles with a positive area.

4.2.2 DrawLine Commands

The DrawLine01 command prompts Delta to draw a line from vertex 0 −V0 to vertex 1 − V1. Conversely, DrawLine10 prompts Delta to draw a line from V1 to V0. These two commands allow the drawing of polylines by alternately updating V0 and V1. The alternate use of DrawLine01 and DrawLine10 continues the correct line stipple pattern across segments in a polyline.

Due to the DDA algorithm, the drawing direction can affect the rendering pixels. With the same data in V0 and V1, the two DrawLine commands can render different pixels. This is important for XOR or patterned line operations.

4.2.3 Repeat Commands

The RepeatTriangle and RepeatLine commands repeat the previously set up triangle or line, most often when a rendering state changes. For example, when a scissor region is updated, the primitive is redrawn to implement window clipping.

A RepeatTriangle command must follow a DrawTriangle command only, not a DrawLine command. Mixing incorrect Repeat and Draw commands results in undefined visual effects.

4.2.4 DeltaMode Register

The DeltaMode register holds long-term state information. The per-primitive control information comes from the Draw command as previously described.

Table 4−3 lists the DeltaMode register bit field assignments and describes their functions.

Table 4−3. DeltaMode Register Bit Field Assignments

Bit. No.	Name	Description
0, 1	Reserved	
2, 3	DepthFormat	The following options apply: 0: 15-bit depth 1: 16-bit depth 2: Reserved 3: Reserved
4	FogEnable	When set, it enables the fog calculations. This field is qualified by the FogEnable bit in the Draw command.
5	TextureEnable	When set, it enables the texture calculations. This field is qualified by the TextureEnable bit in the Draw command.
6	SmoothShadingEnable	When set, it enables the color calculations.
7	DepthEnable	When set, it enables the depth calculations.
8	SpecularTextureEnable	When set, it enables the specular texture calculations.
9	DiffuseTextureEnable	When set, it enables the diffuse texture calculations.
10	SubPixelCorrection-Enable	When set, it provides the subpixel correction in Y. This is qualified by the SubPixelCorrection Enable in the Draw command.
11	DiamondExit	When set, it enables the application of the OpenGL “Diamond-exit” rule to modify the start and end coordinates of lines.
12	NoDraw	When set, it prevents a Render command from being sent to the rendering devices. This field only affects the Draw commands. This field allows the host to alter the setup parameters before sending a Render command.
13	ClampEnable	When set, it causes the input values to be clamped to a parameter specific range. Note that the texture parameters are not affected by this field.

Table 4.3 DeltaMode Register Bit Field Assignments

Bit. No.	Name	Description
14, 15	TextureParameter-Mode	This field causes the texture parameters to be: 0: Used as given 1: Clamped to lie in the range −1.0 to 1.0 2: Normalize to lie in the range −1.0 to 1.0
16	Reserved	
17	BackFaceCull	When set, it enables backface culling of triangles. Rejection is based on the sign of the area of triangle, whether +ve or −ve is controlled by the draw command.
18	ColorOrder	Specifies order of colors in V*PackedColor messages. Bit 31 Bit 0 0 = Alpha, Blue, Green, Red 1 = Alpha, Red, Green, Blue Each color component is 8 bits.

Set any unused bits in the DeltaMode register to zero. Any Repeat commands use the DeltaMode values that were in effect when the corresponding Draw command was issued.

4.2.5 Rasterizer Modes

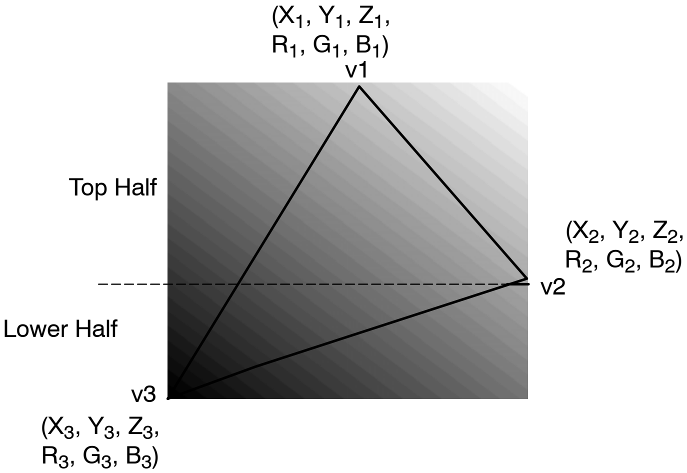
There is one Delta-specific requirement when setting the Rasterizer unit rendering modes. Set the BiasCoordinates bits in the RasterizerMode register (bits 4 and 5) to zero. This selects a zero bias for any addition to the start X and Y values.

4.3 A Gouraud-Shaded Triangle Without Using the Delta Unit

This section illustrates how to use the TVP4020 to render a typical 3-D graphics primitive: the Gouraud-shaded, depth-buffered triangle, without using the Delta Unit. Figure 4−4 assumes a coordinate origin at the bottom left of the window with a drawing function that occurs from top to bottom. The TVP4020 can draw from top to bottom or bottom to top.

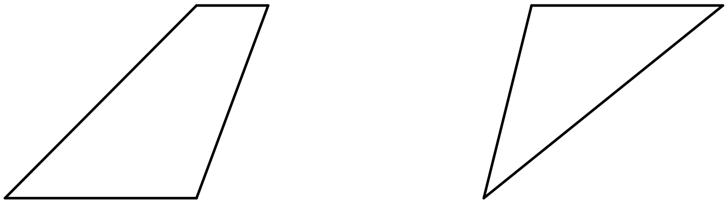
Consider a triangle with vertices, v_1, $v_{2,}$ and v_3 where each vertex comprises X, Y, and Z coordinates, as shown in Figure 4−4 below. Each vertex has a different color made up of red, green, and blue (RGB) components.

Figure 4−4. Example Triangle



This diagram makes a distinction between top and bottom halves because the TVP4020 rasterizes screen-aligned trapezoids and flat-topped or flat-bottomed triangles, as shown in Figure 4−5 below.

Figure 4−5. Screen-Aligned Trapezoid and Flat-Topped Triangle



4.3.1 Initialization

The TVP4020 requires many of its registers to initialize in a particular way, regardless of what is to be drawn. For instance, the screen size and appropriate clipping must be set up. Usually this needs to be done only once. For clarity, this example assumes that all initialization is complete. For more details on initialization, see Chapter 5, *Initialization*.

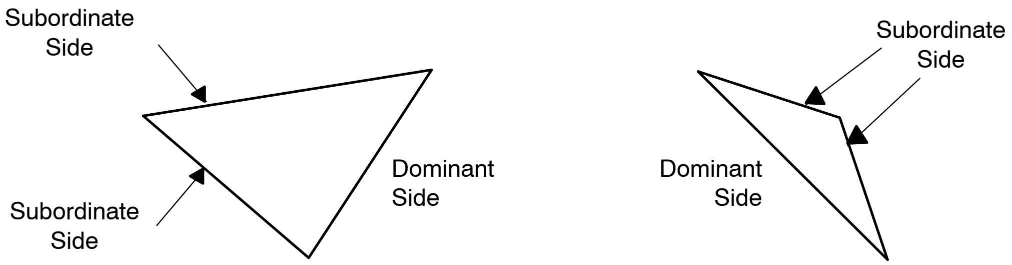
Other initialization states occasionally change, though not usually on a per-primitive basis, for instance, for enabling Gouraud-shading and depth buffering. A detailed treatment is provided later in this chapter.

4.3.2 Dominant and Subordinate Sides of a Triangle

The dominant side of a triangle is the side with the greatest range of Y values. The choice of dominant side is optional when the triangle is either flat bottomed or flat topped.

The TVP4020 always draws triangles starting from the dominant edge toward the subordinate edges (see Figure 4−6). This simplifies the calculation of setup parameters, as described in the following paragraphs.

Figure 4−6. Dominant and Subordinate Sides of a Triangle



4.3.3 Calculating Color Values for Interpolation

To draw from left to right and top to bottom, the color gradients (or deltas) required are:

$$dRdy_{13} = \frac{R_3 \pm R_1}{Y_3 \pm Y_1} \qquad dGdy_{13} = \frac{G_3 \pm G_1}{Y_3 \pm Y_1} \qquad dBdy_{13} = \frac{B_3 \pm B_1}{Y_3 \pm Y_1}$$

and from the plane equation:

$$dRdx = \left[(R_1 - R_3)\,\Theta\,\frac{(Y_2 - Y_3)}{a}\right] - \left[(R_2 - R_3)\,\Theta\,\frac{(Y_1 - Y_3)}{a}\right]$$

$$dGdx = \left[(G_1 - G_3)\,\Theta\,\frac{(Y_2 - Y_3)}{a}\right] - \left[(G_2 - G_3)\,\Theta\,\frac{(Y_1 - Y_3)}{a}\right]$$

$$dBdx = \left[(B_1 - B_3)\,\Theta\,\frac{(Y_2 - Y_3)}{a}\right] - \left[(B_2 - B_3)\,\Theta\,\frac{(Y_1 - Y_3)}{a}\right]$$

where:

$$a = ABS\,\{[(X_1 - X_3)\,\Theta\,(Y_2 - Y_3)] - [(X_2 - X_3)\,\Theta\,(Y_1 - Y_3)]\}$$

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 can be calculated by adding $dRdy_{13}$, for each scanline between Y_1 and Y_n, to R_1, and then adding *dRdx* for each fragment along scanline Y_n from the left edge to X_n.

The example used has the knee; that is, vertex 2, on the right-hand side, and drawing is performed from left to right. If the knee were on the left side (or drawing was performed 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, the TVP4020 always draws triangles starting from the dominant edge toward the subordinate edges. For the example triangle in Figure 4−4, this means left to right.

4.3.4 Register Set Up for Color Interpolation

For the example triangle, shown in Figure 4−4, the TVP4020 registers must be set as follows. Details of register formats are provided later.

```
// Load the color start and delta values to draw
// a triangle
RStart (R1)
GStart (G1)
BStart (B1)
dRdyDom (dRdy13)   // To walk up the dominant edge
dGdyDom (dGdy13)
dBdyDom (dBdy13)
dRdx (dRdx)        // To walk along the scanline
dGdx (dGdx)
dBdx (dBdx)
```

4.3.5 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:

$$dZdx = \left[(Z_1 - Z_3) \, \Theta \frac{(Y_2 - Y_3)}{a} \right] - \left[(Z_2 - Z_3) \, \Theta \frac{(Y_1 - Y_3)}{a} \right]$$

where:

$$a = ABS \, \{[(X_1 - X_3) \, \Theta \, (Y_2 - Y_3)] - [(X_2 - X_3) \, \Theta \, (Y_1 - Y_3)]\}$$

The divisor, shown here as *a*, is the same as for color gradient values. The two deltas, *dZdy₁₃* and *dZdx,* allow the *Z* value of each fragment in the triangle to be determined by linear interpolation, as described for the color interpolation.

4.3.6 Register Setup for Depth Testing

Internally, the TVP4020 uses fixed-point arithmetic. The formats for each register are described later. Each depth value must be converted into a 2s-complement, fixed-point number and then loaded into the appropriate pair of registers. The upper or U registers store the integer portion, and the lower, or L, registers store the fractional bits, left justified and zero filled.

For the example triangle in Figure 4−4, the TVP4020 needs its registers set 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)
```

4.3.7 Calculating the Slopes for Each Side

The TVP4020 draws filled shapes, such as triangles, as a series of spans with one span per scanline. Therefore, it needs to know the start-X and end-X coordinates of each span. It determines this by a process called 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 is drawn in two parts, top down to the knee, that is, beginning at vertex 2 and ending at the bottom. The dominant side is the left side, so the top half is:

$$dXDom = dX_{13} \qquad dXSub = dX_{12}$$

The start X and Y coordinates, the number of scanlines, and the above deltas give the TVP4020 enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat-topped triangle (the TVP4020 rasterizes 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, 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 must be provided. 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.

4.3.8 Rasterizer Mode

The TVP4020 rasterizer has a number of modes that remain effective from the time they are set until they are modified and can, thus, affect many primitives. In the case of the Gouraud-shaded triangle, the default values for these modes are suitable.

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

4.3.9 Subpixel Correction

The TVP4020 can perform subpixel correction of all interpolated values when rendering aliased trapezoids. This correction ensures that any parameter (color/depth/texture/fog) is correctly sampled at the center of a fragment. In general, subpixel correction is always enabled when rendering any trapezoid with interpolated parameters. Control of subpixel correction in the Render command register is described in the following section. It is selectable on a per-primitive basis. It does not need to be enabled for any primitive that does not use interpolation, including copy operations. If it is disabled and interpolators are used, the values calculated for the primitive may not be exactly correct; enabling subpixel correction may reduce the performance of the chip, particularly for small primitives.

4.3.10 Rasterization

At this point, the TVP4020 is almost ready to draw the triangle. Setting up the registers as described here and sending the Render command draws the top half of the example triangle.

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 = TVP4020_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE
// Draw the top half of the triangle
Render (render)
```

After the Render command is issued, the registers in the TVP4020 can immediately be altered to draw the lower half of the triangle. Only two registers need to be loaded, and the ContinueNewSub command needs to be sent. Once the TVP4020 receives ContinueNewSub, the drawing of this subtriangle begins.

```
// 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
```

4.4 Rasterizer Unit

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

The TVP4020 can directly rasterize:

❏ Aliased screen-aligned trapezoids
❏ Aliased single-pixel-wide lines
❏ Aliased single-pixel points

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

4.4.1 Trapezoids

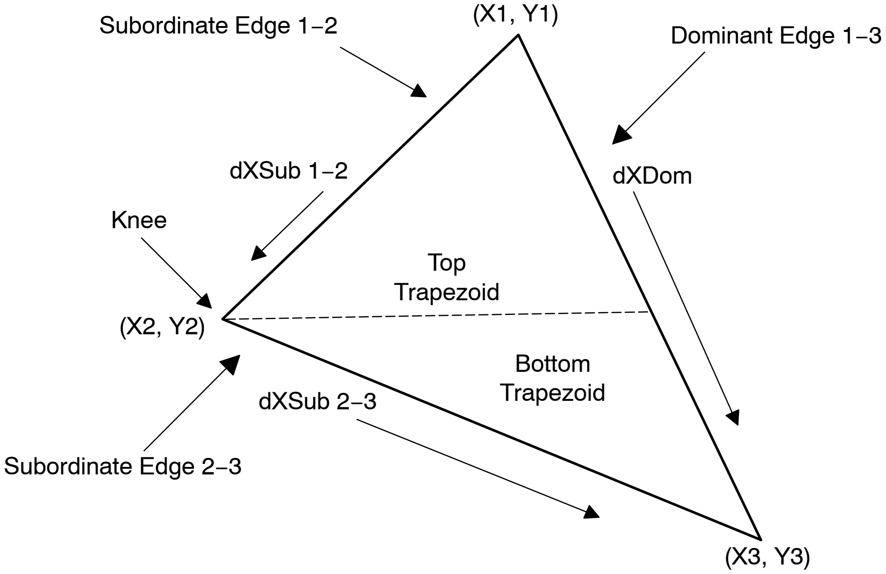
The TVP4020 is a basic area, screen-aligned trapezoid primitive characterized by top and bottom edges that are parallel to the X-axis. The side edges may be vertical (a rectangle) but, in general, are diagonal. The top or bottom edges can degenerate into points that form either flat-topped or flat-bottomed triangles. Any polygon can decompose into screen-aligned trapezoids or triangles. Usually, polygons decompose into triangles because the interpolation of values over nontriangular polygons is ill defined. The rasterizer does handle flat-topped and flat-bottomed bow-tie polygons, which are special types of screen-aligned trapezoids.

The approach to render a triangle and determine which fragments to draw is called edge walking. For example, an aliased triangle, as shown in Figure 4–7, is rendered from top to bottom. The origin is bottom left of the window. Starting at (X1, Y1), and 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. The results are 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.

Usually, adjacent triangles or polygons that share an edge or vertex must be drawn to ensure that pixels, which make up the edge or vertex, are drawn only once. You can achieve this by omitting the pixels down the left or the right sides and along the top or lower sides. The TVP4020 omits the pixels down the right-hand edge. 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. For example, if StartY = Y1 and the number of scanlines is set to Y1–Y2, the lower edge of the top half of the triangle is excluded. This excluded edge is 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 that has the maximum range of Y values. Rendering always proceeds from the dominant edge toward the relevant subordinate edge. In the example in Figure 4−7, the dominant edge is 1 − 3, so rendering is from right to left.

Figure 4−7. Rasterizing a Triangle



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

1) Load the edge parameters and derivatives for the dominant edge and the first subordinate edges into the first triangle.

2) Send the Render command. This starts the scan conversion of the first triangle, working from the dominant edge. This means that triangles with the knee on the left scan from right to left, and triangles with the knee on the right scan from left to right.

3) Load the edge parameters and derivatives for the remaining subordinate edge into the second triangle.

4) Send the ContinueNewSub command. This starts the scan conversion of the second triangle.

Pseudocode for the above example is as follows:

```
// Set the Rasterizer mode to the default,
// see section 4.4.11
RasterizerMode (0)
// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format
StartXDom (X1<<16)
dXDom (((X3- X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2- X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<<16)           // Down the screen
Count (Y1 - Y2)
// Set the render mode to aliased primitive with
// subpixel correction.  See section 4.4.7
render.PrimitiveType = TVP4020_TRAPEZOID_PRIMITIVE
render.SubpixelCorrectionEnable = TVP4020_TRUE
// Draw top half of the triangle
Render (render)
// Set the start and delta for the second half of the
// triangle.
StartXSub (X2<<16)
dXSub (((X3- X2)<<16)/(Y3 - Y2))
// Draw lower half of triangle
ContinueNewSub (abs(Y2 - Y3))
```

After the Render command is sent, the registers in the TVP4020 can be altered immediately to draw the second half of the triangle. To do this, only two registers need to be loaded and the ContinueNewSub command needs to be sent. Once the first triangle drawing is complete and the TVP4020 receives the ContinueNewSub command, subtriangle drawing begins. The ContinueNewSub command register is loaded with the remaining number of scanlines to be rendered.

A continue command can be used instead of the ContinueNewSub command to avoid reloading the rasterizer-edge Digital Differential Analyzers (DDAs). However, rasterization errors can accumulate, which may result in imprecise rendering.

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

4.4.2 Lines

Single-pixel-wide aliased lines are drawn using a DDA algorithm, so all that the TVP4020 needs for input data is *StartX*, *StartY*, *dX*, *dY*, and length. The algorithm calculates:

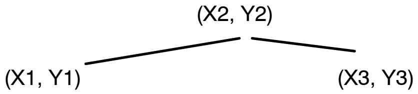
```
while (length--)
{
   X = X + dx
   Y = Y + dy
   plot ((int)X,  (int)Y)
}
```

Consider rendering a two-segment polyline from (X_1, Y_1) to (X_2, Y_2) to (X_3, Y_3) (see Figure 4−8).

Both segments are X major so:

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

Figure 4−8. Polyline



The pseudocode to render the line in Figure 4−8 is as follows:

```
// Set the Rasterizer mode to the default,
// see section 4.4.11
RasterizerMode (0)
// Load the delta values for the first segment.
StartXDom (X1<<16)
dXDom (1.0<<16)
StartY (Y1<<16)
dY (((Y2- Y1)<<16)/(X2 - X1))
Count (abs (X2 - X1))
// Set the render mode
render.PrimitiveType = TVP4020_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))
```

The mechanism to render the second segment with the ContinueNewLine command is analogous to the ContinueNewSub command used at the knee of a triangle. Take precautions when using a continue command for lines. Incorrect rendering can occur with logical operations and alpha blending if a segment draws back over the previous line segment, attempting to reuse pixels that were just updated. The solution is to send a Sync command prior to the ContinueNewLine. This ensures that pending write operations are flushed before the framebuffer reads the new line segment. There is no need to poll for the Sync command here; the act of loading this command register is sufficient.

Note:

When you use a continue command other than ContinueNewLine, an undefined error is propagated along the line; thus, this command is rarely used for lines. To minimize this error, a number of actions are available. To restart the DDA units on receipt of a ContinueNewLine command, see subsection 4.4.11, *Rasterizer Mode*.

For OpenGL rendering, the ContinueNewLine command is not used and, thus, individual segments are rendered.

4.4.3 Points

The TVP4020 supports a single-pixel, aliased-point primitive. For points larger than one, pixel trapezoids should be used for rendering. The fields in the Render command register are described in detail later; however, in this case, the PrimitiveType field in the Render command should equal TVP4020_POINT_PRIMITIVE. The pseudocode portion to render an aliased unity-sized point is:

```
// Set the Rasterizer mode to the default,
// See section 4.4.11
RasterizerMode (0)
// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format
StartXDom (X<<16)
StartY (Y<<16)
// Set-up the render command.
render.PrimitiveType = TVP4020_POINT_PRIMITIVE
```

```
// Render the point
Render (render)
```

4.4.4 Rectangles

The rectangle primitive is restricted to integer pixel positions only. (Use the trapezoid primitive for rectangles that require subpixel positioning.)

The rectangle is defined using two registers: RectangleOrigin, which defines the X and Y start point; and RectangleSize, which defines the width and height. You can control the direction in which the rectangle is filled using the Render command, and separately control the fill direction in X and Y to make the primitive suitable for copy operations.

4.4.5 Spans

Shapes that are more complex than points, lines, or trapezoids may be drawn as a series of spans. Each span may be drawn as a horizontal line or as a single-pixel-high trapezoid. Both are special cases; the loading of certain registers (for example, *dXDom*, d*XSub*, and *dY*) can be omitted. (See subsections 4.4.2, *Lines*, and 4.4.3, *Points*, for details.) Trapezoids, however, can optionally use block-write operations for constant color spans, which may be preferable.

4.4.6 Block Write Operation

The TVP4020 supports synchronous graphics RAM (SGRAM) block-write operations using block sizes of 32 pixels. Any screen-aligned trapezoid, not just rectangles, can be filled using a block-write operation. The SGRAM hardware writemasks can be used in conjunction with this operation.

Block-write operations are enabled by setting the FastFillEnable field in the Render command register.

Only the rasterizer and framebuffer write units are involved in block filling. The other units ignore block-write fragments, so they need not be disabled.

4.4.7 Subpixel Precision and Correction

The rasterizer has a fractional precision of 15 bits in X and Y. The maximum screen width is 2048 pixels. Thus, a number of subpixel precision bits are available. The extra bits are required for two reasons:

❑ For using an accumulation buffer (where scans are rendered multiple times with jittered input vertices)

❑ For correct interpolation of parameters, to provide high-quality shading, as described below

The TVP4020 supports subpixel correction of interpolated values when rendering trapezoids. Subpixel correction ensures that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment’s center. This correction is required to ensure consistent shading of objects made from many primitives. Subpixel correction should be enabled for all rendering that uses interpolated parameters.

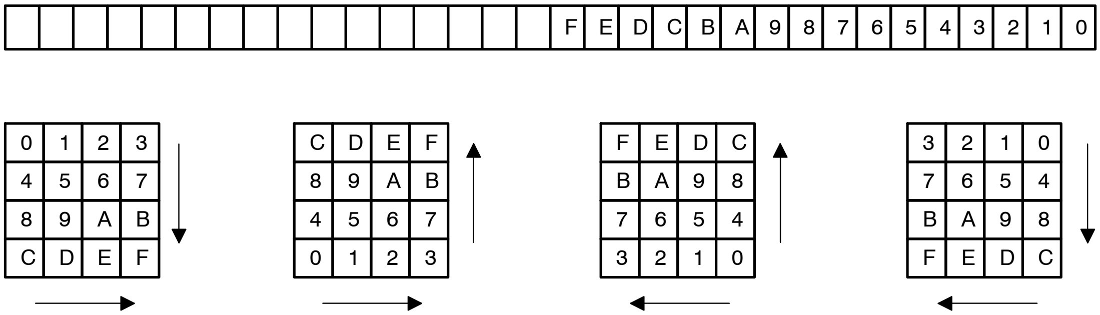
4.4.8 Bitmaps

A bitmap primitive is a trapezoid or line of ones and zeros that control which fragments are generated by the rasterizer. Only fragments where the corresponding bitmap bit is set are submitted for drawing. Normal use is for drawing characters, although the mechanism is available for all primitives. The bitmap data is packed contiguously into 32-bit words so that rows are adjacent to each other. By default, bits in the mask word are used from the least significant end toward the most significant end and are applied to pixels in the order in which they are generated. The relationship between bits in the mask and the scanning order is shown in Figure 4−9.

Instead of rejecting fragments that fail the bitmask, the RasterizerMode register can set them to the background color. The background color comes from the Texel0 register, which may be static or dynamically loaded through the Texture Read unit.

The rasterizer scans through the bits in each word of the bitmap data and increments the X,Y coordinates to trace out the rectangle of the given width and height. By default, any set bits (1) in the bitmap generate a fragment; any reset bits (0) reject the fragment.

Figure 4−9. Relationship Between Bitmask and Scanning Directions



																F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

0	1	2	3
4	5	6	7
8	9	A	B
C	D	E	F

C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3

F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0

3	2	1	0
7	6	5	4
B	A	9	8
F	E	D	C

The selection of bits from the BitMaskPattern register can be mirrored; that is, the pattern traverses from most significant bit (MSB) to least significant bit (LSB) rather than from LSB to MSB. Also, the sense of the test can be reversed such that a set bit rejects a fragment and a failed bit rejects a fragment. This control resides in the RasterizerMode register, described in subsection 4.4.11, *Rasterizer Mode*.

When one bitmap word is exhausted and pixels in the rectangle still remain, then rasterization is suspended until the next write to the BitMaskPattern register, or until the bitmask can be reused. If the bitmask is still valid when a new line is started, it can continue to the next line or be discarded and a new one started; the start position of the mask can be specified to ignore the first bits. It is also possible to index the mask using the X position of the rasterizer, which allows a 32-bit wide, window-aligned bit pattern. When used with a new mask for every scanline, it supports a 32 x 32 stipple pattern.

For example, a five-pixel-wide, eight-pixel-high bitmap requires that a register be set as follows:

```
// Set the Rasterizer mode to the default,
// see section 4.4.11
RasterizerMode (0)
// Set-up the start values and the deltas.
// Note that the X and Y coordinates are converted to
// 16.16 format
StartXDom (X<<16)
dXDom (0)
StartXSub ((X + 5)<<16)       // Right hand edge pixels
                              // get missed off.
StartY (Y<<16)
dY (1<<16)
Count (8)
// At least the following bits require setting for the
// Render command.
render.PrimitiveType = TVP4020_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = TVP4020_TRUE
render.ReuseBitMask = TVP4020_FALSE
// Issue render command. First fragment will be
// generated on receipt of the BitMaskPattern
Render (render)
```

```
// 8x5 pixel bitmap requires 40 bits, and so 2
// 32 bit words.
BitMaskPattern (patternWord0)
BitMaskPattern (patternWord1)
```

Rendering starts as soon as the first patternWord is loaded into the BitMaskPattern register.

4.4.9 Block Writing and Bitmaps

The fastest way to render downloaded bitmap data that does not require logical operations processing is to use block filling. Set the rasterizer up as normal, setting the FastFillEnable bit. If it is also necessary to plot the background color and repeat the operation for the background color with the InvertBitMask bit set in the RasterizerMode register.

Because the downloaded bitmask data is ANDed with masks generated by the rasterizer without any realignment, the host software must 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 rasterizer. A faster way is with the User Scissor, which is the recommended method (see Section 4.5, *Scissor/Stipple Unit*, for additional details). However, 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, the scissor need not be used.

For example, download data to fill a rectangle with the left edge at 10 and right edge at 200. Assume that the host bitmap data is to be loaded from an offset of 35 within the bitmap. Match the bit at offset 35 with the pixel at offset 10.

To do the least amount of work on the host and avoid shifting the data, use a block operation such as this:

1) Download the host bitmap data at the previous 32-bit boundary.

2) Set up the TVP4020 to discard the first three bits of data by rasterizing a rectangle whose left edge is three pixels less than that required. In this case, rasterize the left edge to start at pixel 7. The source bitmap data will correctly align with the mask data produced by the rasterizer.

3) In order to protect the three pixels that would otherwise be overwritten, use the scissor clip and set its bounds to those of the original rectangle.

When using a block-write operation like this, the rasterizer will wait for new bitmask data to be downloaded at the start of each scanline, eliminating the need to perform the alignment operation on the right-hand edge.

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

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

4.4.10 Copy/Upload/Download

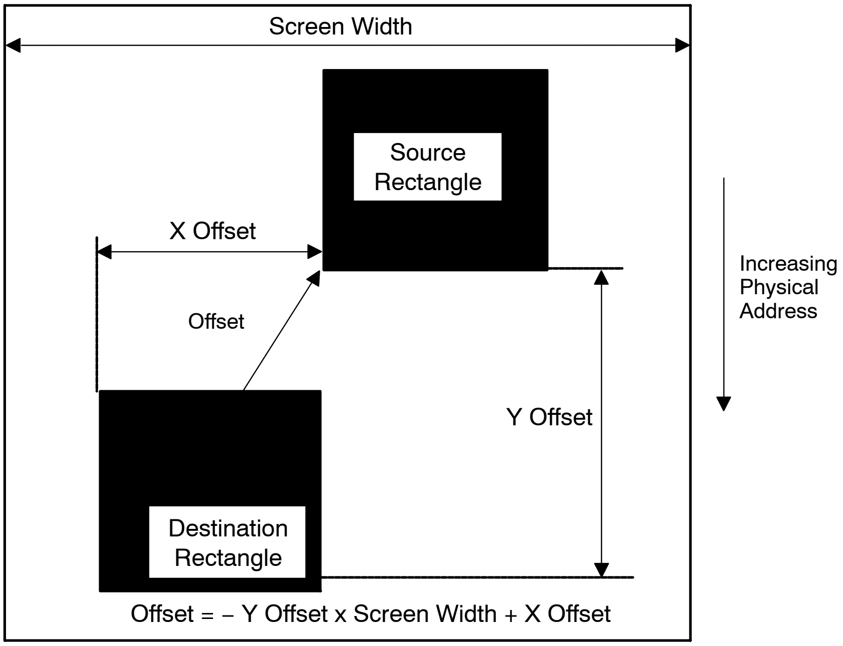
The TVP4020 supports three pixel-rectangle operations: copy, upload, and download. These apply to all buffer types.

Typically, the TVP4020 copy operation moves raw blocks of data around buffers. To zoom or reformat data, external software must upload the data, process it, and download it again, or the TVP4020 must use the texture part of the Texture/Fog/Blend unit.

To copy a rectangular area, configure the rasterizer to render the destination rectangle to generate fragments for the area to copy. The TVP4020 copy operation works by adding a linear offset value to the destination fragment address to find the source fragment address. The calculation of the offset value is shown in Figure 4−10.

The offset value is independent of the origin of the buffer or window, as it is added to the destination address. When the source and destination overlap, be certain to choose the source scanning direction so that the overlapping area is not overwritten before it is moved. Do this by swapping the values written to the *StartXDom* and *StartXSub*, or by changing the sign of *dY* and setting *StartY* to the opposite side of the rectangle.

Figure 4−10. TVP4020 Copy Operation


Screen Width
Source Rectangle
X Offset
Offset
Increasing Physical Address
Y Offset
Destination Rectangle
Offset = − Y Offset x Screen Width + X Offset

The TVP4020 buffer upload/download operations are similar to copy operations where the region of interest is generated in the rasterizer. However, you must configure the localbuffer and framebuffer to read or to write only, rather than simultaneously read and write. Set the Host Out unit to output data to the FIFO for image uploads. For downloads, set the rasterizer to synchronize on the appropriate data type. This means that the rasterizer does not generate the next fragment address until data is supplied from the host processor.

Disable units that generate fragment values (for example, the Color DDA unit) for any copy/upload/download operations.

Note:

During image upload, all the returned fragments must be read from the Host Out FIFO; otherwise, the TVP4020 hyperpipeline will stall. In addition, any units that can discard fragments (for instance the following tests: bitmask, user scissor, screen scissor, stipple, depth, and stencil), must be disabled; otherwise, a shortfall in the number of pixels returned may occur, leading to deadlock.

Because the area of interest during copy, upload, and download operations is defined by the rasterizer, it is not limited to rectangular regions.

Color formatting can be used for image copy, upload, and download operations. Data can be formatted from or to any of the supported TVP4020 color formats. Section 4.14, *Color Format Unit*, fully describes this operation.

4.4.11 Rasterizer Mode

Using the RasterizerMode register, you can set the following long-term modes. These are:

- ❑ Mirror BitMask. This is a single-bit flag that specifies the direction from which bits are checked in the BitMaskPattern register. If the bit is reset, the direction is from least significant to most significant (bit 0 to bit 31); if the bit is set, it is from most significant to least significant (from bit 31 to bit 0).
- ❑ Invert BitMask. This is a single bit that controls the sense of the accept/reject test when using a bitmask. When the bit is reset and the BitMask bit is set, the fragment is accepted. When it is reset, the fragment is rejected. When the bit is set, the sense of the test is reversed.
- ❑ Fraction Adjust. These two bits control the action taken by the rasterizer when it receives a ContinueNewLine command. As the TVP4020 uses a DDA algorithm to render lines, an error accumulates in the DDA value. The TVP4020 controls the error by doing one of the following:
  - ■ Leaving the DDA running, which means errors will propagate along a line
  - ■ Setting the fraction bits to either zero, a half, or almost a half (0×7FFF)
- ❑ Bias Coordinates. Only the integer portion of the DDA values generate fragment addresses. Often a rounding of values is required. Set the bias coordinate bit to true. This will automatically add almost a half (0×7FFF) to all input coordinates.
- ❑ ForceBackgroundColor. When set, if a fragment fails, the bitmask test it is not discarded. It uses the contents of the Texel0 register in place of the normal color. This provides foreground/background color selection.
- ❑ BitMaskByteSwapMode. Controls how or whether the bitmask is byte swapped as it is loaded. Four byte orders are supported.
- ❑ BitMaskPacking. Controls whether a bitmask is discarded at the end of a scanline or continued onto the next. It is not supported for block-write operations.
- ❑ BitMaskOffset. Sets the position of the first bit in the test bitmask. It is not supported for block write operations.

- ❏ HostDataByteSwapMode. Controls byte swapping of host data being sent to the chip. This applies to any Render register operation using the SyncOnHostData. Four byte orders are supported.
- ❏ LimitsEnable. When enabled, LimitsEnable allows quick rejection of fragments outside the defined area.
- ❏ BitMaskRelative. When enabled, BitMaskRelative specifies that the bitmask should be accessed by an index made up of the lower five bits of the X coordinate of the current fragment.

4.4.12 Synchronization

In most circumstances, the TVP4020 automatically synchronizes between primitives so that data for the first primitive is written before data for the second primitive is read. This is handled by data type so that localbuffer read and write operations are synchronized, as are framebuffer read and write operations. However, localbuffer read operations are not synchronized with framebuffer write operations.

If a unit modifies data that is not the normal type, you may need to explicitly synchronize the graphics hypeppipeline. If the Framebuffer Write unit clears the localbuffer with block fills, you must synchronize the graphics hyperpipeline before localbuffer data is read. If the Framebuffer Write unit downloads a texture map, you must synchronize the graphics hyperpipeline before the Texture Read unit accesses the texture.

You can perform explicit synchronization of the graphics hyperpipeline by using the WaitForCompletion command. It has no data field, and you can insert it into a stream of commands. There is no need to wait for the TVP4020 to report that synchronization occurred.

Alternatively, you must perform synchronization using the Sync command. This requires the host processor to poll the chip until it reports that the graphics hyperpipeline is idle, as described in Section 4.16, *Host Out Unit*.

4.4.13 X and Y Limits Clipping

The rasterizer usually rasterizes all pixels on every scanline, generating a fragment per pixel. When large numbers of scanlines are subsequently clipped out (for example, by the scissor unit), a lot of time is wasted. The YLimits register provides a way to quickly eliminate whole scanlines for a given primitive. This register effectively provides a Y scissor clip in the rasterizer.

If limits testing is enabled in the RasterizerMode register and if a scanline being rasterized falls outside the Y limits bounds, then the rasterizer moves directly onto the next scanline without rasterizing in X.

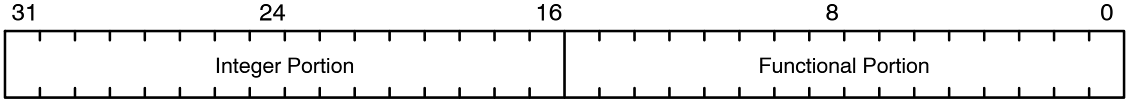
The XLimits register avoids unnecessary rasterization, but does not act as a true X scissor clip. This is to ensure correct interpolation of color, fog, etc. The limits registers provide efficiency.

The XLimits and YLimits scissor clipping functions are automatically disabled when SyncOnHostData or SyncOnBitMask is used.

4.4.14 Registers

The TVP4020 provides real coordinates with fractional parts to the rasterizer in 2s complement fixed-point format, as shown in Figure 4−11. The point stays consistent with a 16.16 format even though some of the integer and fractional bits may not be significant. You must sign-extend the integer portion to fill unused bits, and also set unused bits in the fraction to zero.

Figure 4−11. Real Coordinate Representation



When reference is made to signed-fixed-point format, the sign bit is included in the integer section. For example, a signed-fixed-point format of 12.15 implies 1 sign bit followed by 11 integer bits and 15 fraction bits. Table 4−4 shows the rasterizer command registers and Table 4−5 shows the rasterizer control registers.

Table 4−4. Rasterizer Command Registers

Register Name	Data Field	Description
Render	See Table 4−6	Starts the rasterization process
ContinueNewDom	12-bit integer	Allows rasterization to continue with a new dominant edge. The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to break down into a collection of trapezoids with continuity maintained across boundaries. Because this command only affects the rasterizer DDA and not that of any other units, it is not suitable for 3-D operations. The data field holds the number of scanlines to fill. This count does not get loaded into the Count register.
ContinueNewSub	12-bit integer	Allows 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 (that is, two subordinate edges). The data field holds the number of scanlines to fill. This count does not get loaded into the Count register.
Continue	12-bit integer	Allows the rasterization to continue after new delta value(s) are loaded, but does not reload either of the primitive-edge DDAs. This can result in the accumulation of rasterization errors and cause imprecise rendering. The data field holds the number of scanlines to fill. This count does not get loaded into the Count register.
ContinueNewLine	12-bit integer	Allows the rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line; however, the fraction bits in the DDAs can be kept and set to zero, half, or nearly one half, under control of the RasterizerMode register. The data field holds the number of pixels in a line and this count does not get loaded into the Count register. ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error, in contrast to the value with which they would be loaded for the second and subsequent segments.
WaitForCompletion	Not applicable	Suspends the TVP4020 core until all outstanding reads and writes in framebuffer memory units are complete. It prevents a new primitive from rasterizing before the previous primitive is finished. It is used, for example, to separate texture downloads from the surrounding primitives. The same functionality is achieved by using the Sync command and waiting for it in the Host Out FIFO. However, WaitForCompletion doesn’t involve the host and, thus, it can be inserted into a DMA buffer.

Table 4–5. Rasterizer Control Registers

Register Name	Data Field	Description
RasterizerMode	See Table 4–7	Defines the long-term mode of operation of the rasterizer
StartXDom	Signed fixed-point 12.15 format	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing
dXDom	Signed fixed-point 12.15 format	The value added when moving from one scanline to the next for the dominant edge in trapezoid filling. Also holds the change in X when plotting lines, so for Y major lines this will be a fraction (dx/dy). Otherwise, it is normally ±1.0, depending on the required scanning direction.
StartXSub	Signed fixed-point 12.15 format	Initial X value for the subordinate edge
dXSub	Signed fixed-point 12.15 format	The value added when moving from one scanline to the next for the subordinate edge in trapezoid filling
StartY	Signed fixed-point 11.14 format	Initial scanline in trapezoid filling, or initial Y position for line drawing
dY	Signed fixed-point 12.15 format	The value added to Y to move from one scanline to the next. For X major lines, this will be a fraction (dy/dx). Otherwise, it is normally ±1.0, depending on the required scanning direction.
Count	12-bit integer	Number of pixels in a line. Number of scanlines in a trapezoid.
XLimits	Xmax: 2s complement 12-bit value in the upper word Xmin: 2s complement 12-bit value in the lower word	Defines the X extents that the rasterizer should fill. A span is rasterized if its X value satisfies: Xmin £ X < Xmax.
YLimits	Ymax: 2s complement 12-bit value in the upper word Ymin: 2s complement 12-bit value in the lower word	Defines the Y extents that the rasterizer should fill. A scanline is filled if its Y value satisfies: Ymin £ Y < Ymax.
RectangleOrigin	Y: 2s complement 12-bit value in the upper word X: 2s complement 12-bit value in the lower word	Defines the origin of a rectangle primitive. The corner of the rectangle that this refers to is controlled by the rectangle fill direction fields in the Render command.
RectangleSize	Height: 2s complement 12-bit value in the upper word Width: 2s complement 12-bit value in the lower word	

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 the TVP4020. These bits are AreaStippleEnable, TextureEnable, FogEnable, ReuseBitMask and SubpixelCorrection.

You can use the render operation when a window is cleared to a background color. For normal 3-D primitives, stippling and fog operations may be enabled, but these are ignored for window clears. If initially the FogMode and AreaStippleMode registers are enabled through the unit Enable bits, the bits need only be set or cleared within the Render command to achieve the required result. This removes the need to load the FogMode and AreaStippleMode registers for every Render operation.

The bit fields of the Render command register are shown in Table 4–6.

Table 4–6. Render Command Register Fields

Bit No.	Name	Description
0	AreaStippleEnable	Enables area stippling
1,2	Reserved	
3	FastFillEnable	Enables fast fill using VRAM block mode
4, 5	Reserved	
6, 7	PrimitiveType	Sets the primitive type: 0 = Line 1 = Trapezoid 2 = Point 3 = Rectangle
8,9,10	Reserved	
11	SyncOnBitMask	Enables the bitmask test. Waits for a new mask when the current one expires unless SyncOnHostData or ReuseBitmask is enabled.
12	SyncOnHostData	Waits for host data before sending step message
13	TextureEnable	Enables texturing
14	FogEnable	Enables fogging
15	Reserved	
16	SubPixelCorrection-Enable	Enables subpixel correction
17	ReuseBitMask	Reuses bitmask when last bit is used
18. 19	Reserved	
20	RejectNegativeFace	Used by Delta unit
21	IncreaseX	Direction of fill for rectangle
22	IncreaseY	Direction of fill for rectangle

Several long-term rasterizer modes are stored in the RasterizerMode register as shown in Table 4–7.

Table 4–7. Rasterizer Mode Register

Bit No.	Name	Description
0	MirrorBitMask	When this bit is set, the bitmask bits are consumed from the most significant end toward the least significant end. When this bit is reset, the bitmask bits are consumed from the least significant end toward the most significant end.
1	InvertBitMask	When this bit is set, the bitmask is inverted before being tested.
2, 3	FractionAdjust	These bits are for the ContinueNewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted: 0: No adjustment is made. 1: Set the fraction bits to zero. 2: Set the fraction bits to half. 3: Set the fraction to *nearly half*, i.e., 0×7fff.
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. 0×7fff is added.
6	ForceBackgroundColor	If the bitmask test fails, this bit takes the color from the Texel0 register instead of using the normal color.
7, 8	BitMaskByteSwapMode	Controls byte swapping of the bitmask. If input is ABCD, 0: ABCD 1: BADC 2: CDAB 3: DCBA
9	BitMaskPacking	If enabled, the current bitmask is discarded at the end of every scanline even if it is not finished. 0: Enabled 1: Disabled
10 – 14	BitMaskOffset	Position of first bit to test in bitmask.
15,16	HostdataByteSwapMode	Controls byte swapping of host data. If input is ABCD, 0: ABCD 1: BADC 2: CDAB 3: DCBA
17	Reserved	
18	LimitsEnable	If enabled, quickly rejects areas of primitive outside defined area 0: Enabled 1: Disabled
19	BitMaskRelative	Controls whether bitmask is indexed by counter or by lower 5 bits of X value 0: Disabled 1: Enabled

Note: The register BitMaskPattern holds the 32-bit mask for bitmask stippling.

4.5 Scissor/Stipple Unit

Two scissor tests are provided in the TVP4020: the user scissor test and the screen scissor test. The user scissor checks each fragment against a user supplied scissor region; the screen scissor checks that the fragment lies within the screen. The stipple test checks each fragment against an 8×8 pattern.

4.5.1 User Scissor Test

The user scissor test tests each fragment, as follows:

$\mathtt{XMin} \leq \mathtt{X} < \mathtt{XMax}$

$\mathtt{YMin} \leq \mathtt{Y} < \mathtt{YMax}$

where:

X and Y = the coordinates for the fragments

XMin, *XMax*, *YMin* and *YMax* = the user supplied scissor region

When a fragment fails the test, it is discarded. The test may be screen or window relative. This test applies to normal pixels and block-fill operations.

4.5.2 Screen Scissor Test

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

The following test is made:

$0 \leq (\mathtt{X} + \mathtt{WX}) < \mathtt{SW}$

$0 \leq (\mathtt{Y} + \mathtt{WY}) < \mathtt{SH}$

where:

X = Fragment X coordinate

Y = Fragment Y coordinate

SW = Screen Width

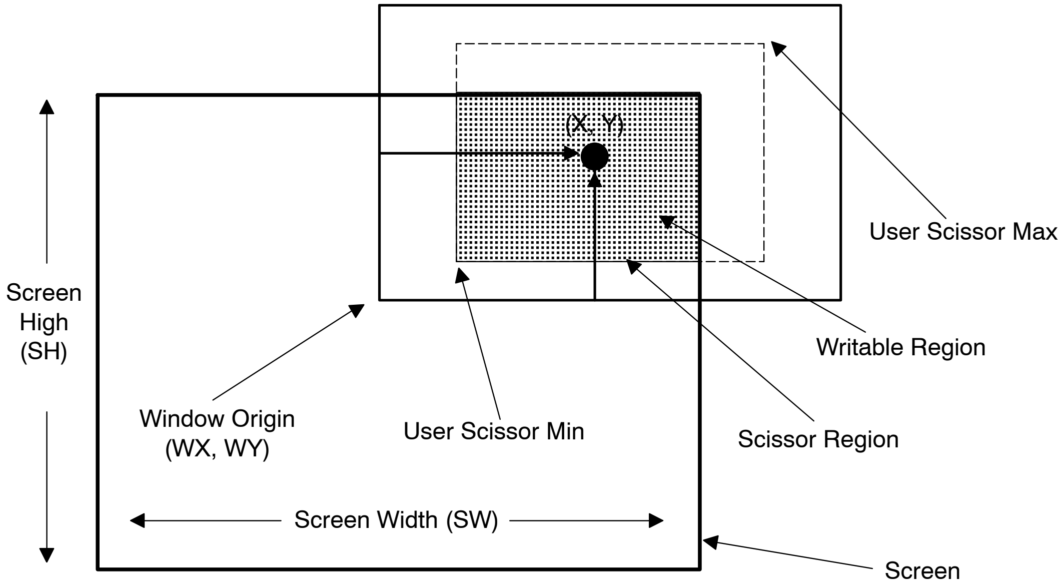
SH = Screen Height

WX = Window origin X coordinate

WY = Window origin Y coordinate

Figure 4−12 shows a simple scenario of a screen with a single window and a user-defined scissor region. The shaded area shows the region where fragments pass the user and screen scissor tests and can progress in the hyperpipeline. Fragments outside this region are culled from the pipeline. This test applies to normal pixels and block-fill operations.

Figure 4−12. Screen Scissor and User Scissor Tests



This test may reject fragments when a part of a window moves off the screen. It will not reject fragments when part of a window is overlapped by another window.

This test is normally enabled. The most common exception is during image uploading.

4.5.3 Area Stippling

An 8×8-bit-area stipple pattern can be applied to fragments. The least significant three bits of the fragment (X,Y) coordinates index into a 2-D stipple pattern. If the selected bit in the pattern is set, then the fragment passes the test; otherwise, it is rejected. In addition, the bit pattern can be inverted or mirrored. Inverting the bit pattern changes the sense of the accept/reject test. If the mirror bit is set, the most significant bit of the pattern falls toward the left of the window, but the default is the converse.

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

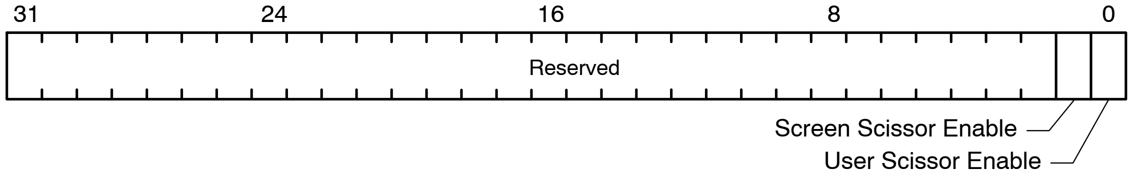
If the ForceBackgroundColor bit is set in the AreaStippleMode register, fragments that fail the area stipple test are not discarded. Instead, the Texel0 register contents are used in place of the normal color for that pixel.

The AreaStippleMode register enables area stippling. It must be qualified by the AreaStippleEnable bit in the Render command register. Area stippling may be used with block filling, but in this case the backgound color is not available.

4.5.4 Registers

The ScissorMode register controls the scissor operation, as shown in Figure 4−13.

Figure 4−13. ScissorMode Register



Typically, the screen scissor test is always enabled. An exception is during image uploading.

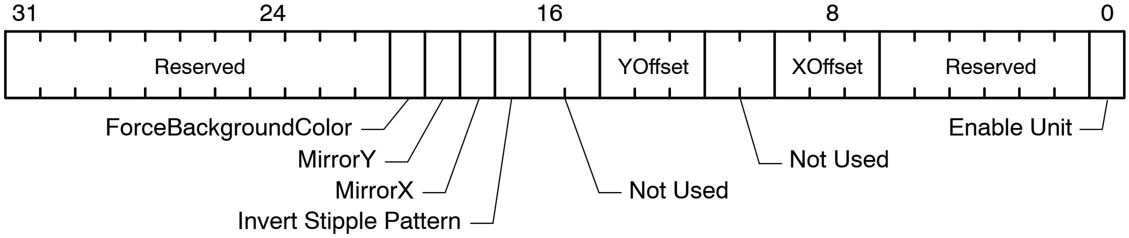
Two registers specify the user scissor region: ScissorMinXY and ScissorMaxXY. The X values are stored in the least significant 16 bits of the register and the Y values in the most significant 16 bits of the register.

The WindowOrigin register has the X-origin coordinate stored in the least significant 16 bits of the register and the Y-origin coordinate stored in the most significant 16 bits of the register. As the rasterizer unit generates each fragment, it adds this origin to the coordinates of the fragment to further generate screen coordinates.

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

The AreaStippleMode register controls the area stipple operation, as shown in Figure 4−14.

Figure 4–14. AreaStippleMode Register



The AreaStippleEnable bits in the Render command register qualify the Enable Unit bit. The AreaStipplePattern *n* register sets up the area stipple, where *n* represents an integer between 0 and 7.

4.5.5 Scissor Example

The following pseudocode shows how to enable the screen scissor for a region: $10 \le X < 500$, $100 \le Y < 200$ with a screen size of 1280×1024 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 = TVP4020_ENABLE
scissorMode.ScreenScissorEnable = TVP4020_ENABLE
ScissorMode(scissorMode)
// Render primitives
```

4.5.6 Area Stipple Example

The following example shows a repeating area stipple pattern of 2×2 pixels that produces a 50-percent gray area:

```
AreaStipplePattern0(0xAA)
AreaStipplePattern1(0x55)
AreaStipplePattern2(0xAA)
AreaStipplePattern3(0x55)
AreaStipplePattern4(0xAA)
AreaStipplePattern5(0x55)
AreaStipplePattern6(0xAA)
AreaStipplePattern7(0x55)
// Set-up mode register
areaStippleMode.UnitEnable = TVP4020_ENABLE
areaStippleMode.XOffset = 0
areaStippleMode.YOffset = 0
areaStippleMode.Invert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0
// Load mode register
AreaStippleMode(areaStippleMode)
// When issuing a Render command, the AreaStippleEnable
// bit should be set in addition to the area stipple test
// being enabled:
// render.AreaStippleEnable = TVP4020_TRUE
```

4.6 Localbuffer Read and Write Units

The localbuffer holds the stencil and depth data associated with a fragment. The localbuffer read and write units are considered a pair, but function as separate units in the graphics hyperpipeline.

4.6.1 Localbuffer Read

The LBReadMode register can be configured to make zero, one, or two reads of the localbuffer. The most common modes of access to the localbuffer are as follows:

- Normal rendering without depth or stencil testing. This requires no localbuffer reading or writing.
- Normal rendering with depth and/or stencil testing required, which conditionally requires updating the localbuffer. This requires enabling the localbuffer read and write modes.
- Copy operations. Operations that copy all or part of the localbuffer. This requires enabling the read and write modes.
- Upload and download operations. Operations that download depth or stencil information to the localbuffer, or read back depth or stencil values from the localbuffer to the host.

The following address calculation implements the following equations:

Bottom-left origin :

```
Destination address = LBWindowBase – Y * W + X
Source address = LBWindowBase – Y * W + X +
   LBSourceOffset
```

Top-left origin :

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

where:

Destination address = the address for any write and read

Source address = the address for a source read

X = the pixel X coordinate

Y = the pixel Y coordinate

LBWindowBase = the base address in the localbuffer of the current window

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

W = the screen width. Only a subset of widths is supported and encoded into the PP0, PP1 and PP2 fields in the LBReadMode register. See Appendix B, *Screen Widths Table*, for more details.

The localbuffer can be read in three formats: LBDefault, LBStencil, or LBDepth. These tell the TVP4020 which areas of the localbuffer are required. LBDefault is used for all copy and rendering operations; LBStencil and LBDepth are used for image upload of the stencil and depth planes. Table 4−8 summarizes the common rendering operations and the required read modes.

Table 4−8. Localbuffer Read/Write Modes

Read Source	Read Destination	Writes	Data Type	Rendering Operation
Disabled	Disabled	Disabled	–	Rendering with no depth or stencil enabled
Disabled	Disabled	Enabled	LBStencil LBDepth	Download to localbuffer from host
Disabled	Enabled	Disabled	LBStencil LBDepth	Upload from localbuffer to host
Disabled	Enabled	Enabled	LBDefault	Rendering with depth and/or stencil updates enabled
Enabled	Disabled	Enabled	LBDefault	Localbuffer copy operations

4.6.2 Localbuffer Write

You must enable write modes to the localbuffer to allow any localbuffer update. The LBWriteMode register is a single-bit flag that controls updating of the buffer.

4.6.3 Localbuffer Data Formats

LBWriteMode is a single-bit register, as shown in Figure 4−16. When the least significant bit is set, the register enables write operations to the localbuffer.

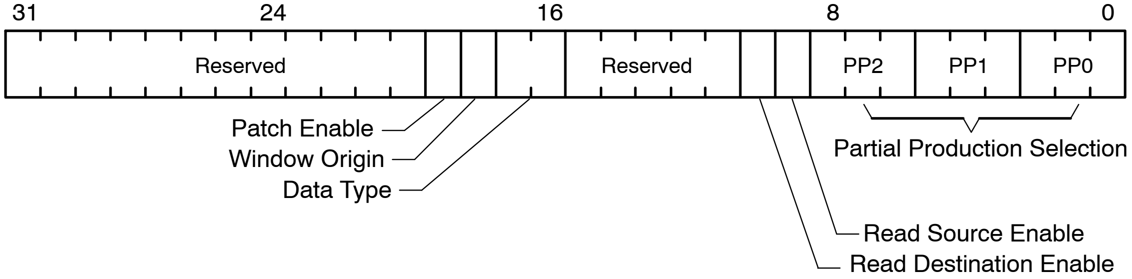
The depth field can be either 15 or 16 bits wide and the stencil field either one or zero bits wide. The total width of the localbuffer data must not be greater than 16 bits. If a stencil field is defined, then it occupies bit 15; the depth field always starts at bit 0.

You must configure the LBReadFormat and LBWriteFormat registers to the appropriate values. The format can be different for different windows.

4.6.4 Registers

The LBReadMode register is shown in Figure 4−15.

Figure 4−15. LBReadMode Register



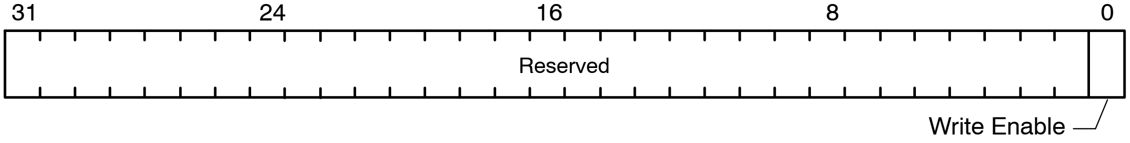
When set, PatchEnable enables normal patch addressing of the localbuffer. This typically results in more efficient memory bandwidth use.

The partial product fields PP0, PP1, and PP2 define the width of the localbuffer; they are described in Appendix B, *Screen Widths Table*.

ReadSourceEnable and ReadDestinationEnable control localbuffer readings 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.

LBWriteMode is a single-bit register, as shown in Figure 4−16. When the least significant bit is set, the register enables write operations to the localbuffer.

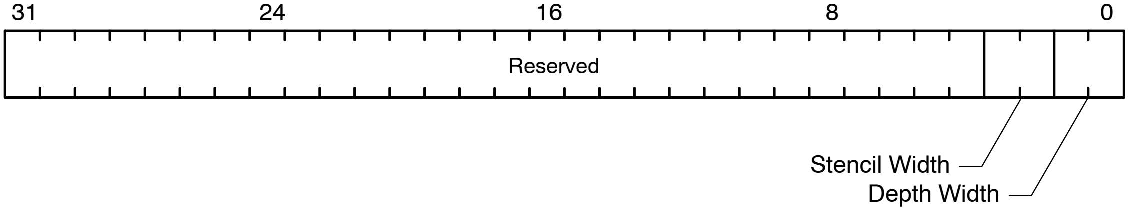
Figure 4−16. LBWriteMode Register



The localbuffer format must be specified for both read and write operations using the LBReadFormat and LBWriteFormat registers, as shown in

Figure 4−17. Usually, you would set these register values identically. Setting them to different values may be useful when copying between two windows that use different depth widths.

Figure 4−17. LBReadFormat/LBWriteFormat Register



LBSourceOffset holds a 24-bit 2s complement value used during copy operations.

LBWindowBase contains the current base address of the window in the localbuffer.

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

4.6.5 Localbuffer Example

The following is an example of a rendering operation with localbuffer read and write modes. The TVP4020 is configured with an 800×600 screen size, and a 16-bit localbuffer such that 15 bits are used for depth and 1 bit is used for stencil.

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

lbReadFormat.StencilWidth = 3        // 1 bit

LBReadFormat(lbReadFormat)           // Load read format

LBWriteFormat(lbReadFormat)          // Write is same as
read

// Set the localbuffer write mode

LBWriteMode(TVP4020_ENABLE)

// Set the localbuffer read mode

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

lbReadMode.PP0 = 5 // 512 (<< 9)

lbReadMode.PP1 = 4 // 256 (<< 8)

lbReadMode.PP2 = 1 // 32  (<< 5)
lbReadMode.ReadSource = TVP4020_DISABLE
lbReadMode.ReadDestination = TVP4020_ENABLE
lbReadMode.DataType = TVP4020_LBDEFAULT
lbReadMode.WindowOrigin = as appropriate
lbReadMode.PatchMode = TVP4020_DISABLE
LBReadMode(lbReadMode)
LBWriteMode(TVP4020_DISABLE)
// Now ready to render with localbuffer read and write
// suitable for stencil and depth buffering operations.
```

4.7 Stencil Test and Depth Test Units

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 results of the stencil and depth tests. Stencil testing can be used in many different ways, for example, for hidden line removal, decals, masking areas of the screen, and stippling.

When enabled, the depth (*Z*) test compares the fragment depth against the corresponding depth in the depth buffer. If the test fails, the fragment is rejected.

4.7.1 Stencil Test

This test occurs only when all the preceding tests (bitmask, scissor, stipple) have passed. The stencil test is controlled by the stencil function and the stencil operation. The stencil function controls the test between the reference stencil value and the value in the stencil buffer. If the test mode 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 results of the stencil and depth tests.

Table 4−9 shows the available stencil functions.

Table 4−9. Stencil Comparison Modes

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

Some of these comparison modes are redundant as the TVP4020 only uses 1-bit stencil values. They are included here, however, to help with GLINT® software compatibility and compatibility with future devices.

If the stencil test is enabled, then the stencil buffer is updated depending on the outcome of both the stencil and the depth tests (if the depth test is disabled, the depth result sets to pass). See Table 4−10, Table 4−11, and Table 4−12, and subsection 4.7.3, *Registers*, for a definition of the StencilMode register and its relationship with the stencil and depth tests.

Table 4−10. Possible Update Operations for Stencil Planes

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

In Table 4−11, the entries dppass, dpfail, and sfail are set to one of the update methods as noted, and source stencil is the value in the stencil buffer.

Table 4−11. Stencil Operations

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

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

The source stencil value can come from a number of places and is controlled by a field in the StencilMode register.

Table 4−12. Stencil Sources

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

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

4.7.2 Depth Test

This test is only performed if all the preceding tests (bitmask, scissor, stipple) have passed. The comparison tests are shown in Table 4−13.

Table 4−13. Depth Comparison Modes

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

The test compares the fragment depth against a source depth value. If the comparison mode is less and the result is true, then the fragment value is less than the source value. The source value comes from a number of places and is controlled by a field in the DepthMode register.

Table 4−14. Depth Sources

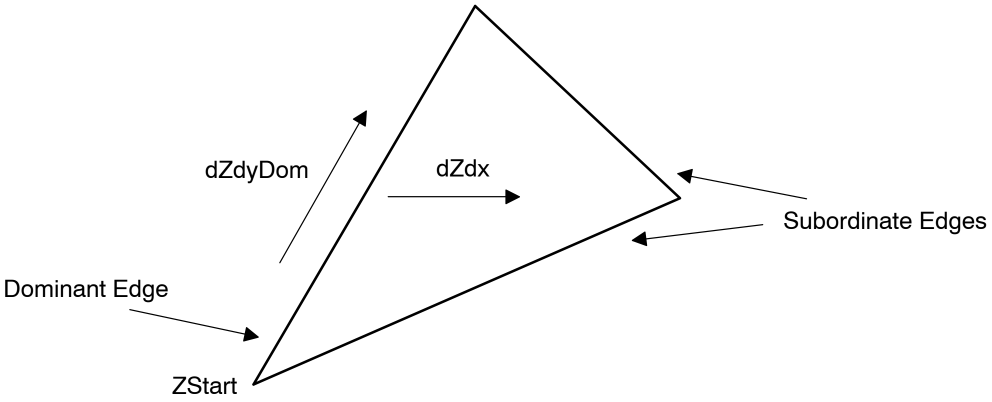
Source	Use
DDA	This is used for normal, depth-buffered 3-D rendering.
Depth register	This is used, for example, in the OpenGL draw-pixels function when the host supplies the depth values through the Depth register. Alternatively, this is used when a constant depth value is needed, for example, when clearing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

For a depth-buffered trapezoid, the TVP4020 interpolates from the dominant edge of the trapezoid to the subordinate edges. This means that two increment values are required: one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in Figure 4−18. The rendering direction chosen here is bottom to top.

```
ZStart = Start Z value
dZdyDom = Increment along dominant edge.
dZdx = Increment along the scan line.
```

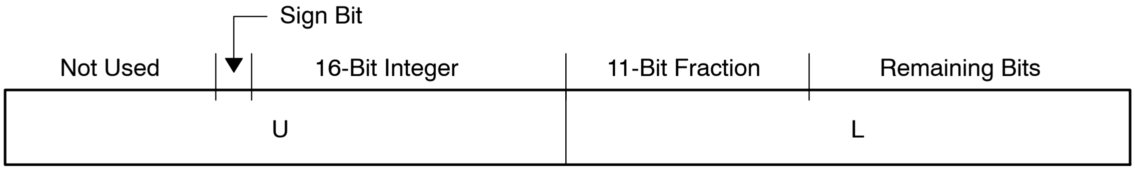
The *dZdx* value is not required for *Z*-buffered lines.

Figure 4−18. Depth Interpolation



The number format for the increment values is the 2s complement fixed-point integer: a 16-bit integer and 11-bit fraction. All of the start, derivative, and internal data are in this format. This format is mapped into the upper and lower registers (U and L), as shown in Figure 4−19.

Figure 4−19. Depth Derivative Format



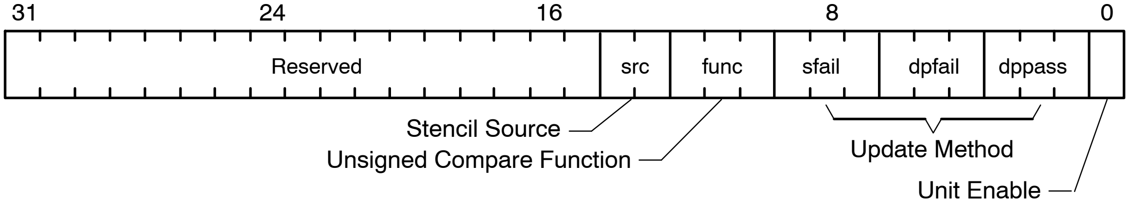
In many instances, the fractional part can contain zero, which eliminates the need to continually update *ZStartL***,** *dZdxL***,** and *dZdyDomL*.

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

4.7.3 Registers

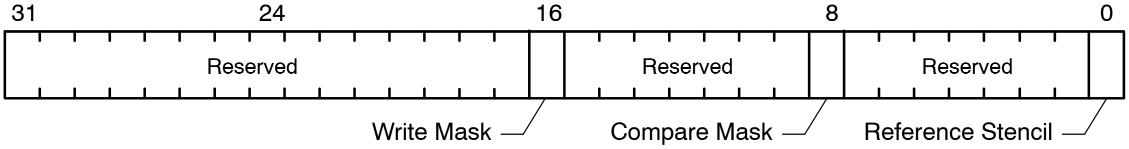
The StencilMode register controls the stencil test, as shown in Figure 4−20.

Figure 4−20. StencilMode Register



The StencilData register holds the other data associated with the test, as shown in Figure 4−21.

Figure 4−21. StencilData Register

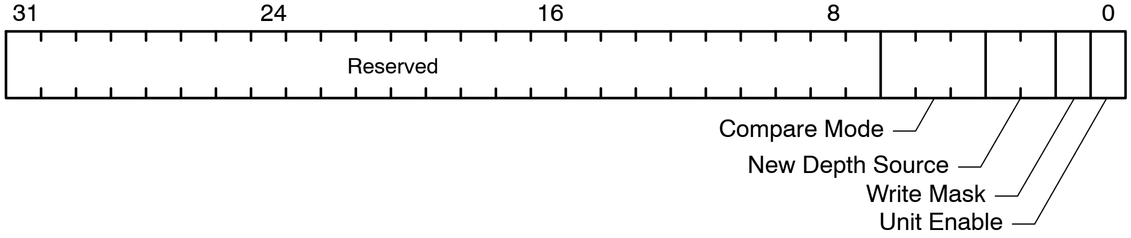


The stencil writemask controls which stencil planes are updated as a result of the test. The Stencil register holds an externally sourced stencil value; a 32-bit register in which only the least significant bit is used. Set the unused bits to zero.

The Stencil unit must be enabled to update the stencil buffer. When it is disabled, the stencil buffer will update only if ForceLBUpdate is set in the Window register.

The DepthMode register controls operation of the Depth unit, as shown in Figure 4−22.

Figure 4−22. DepthMode Register



The single-bit writemask controls the updating of all the bits in the depth buffer.

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

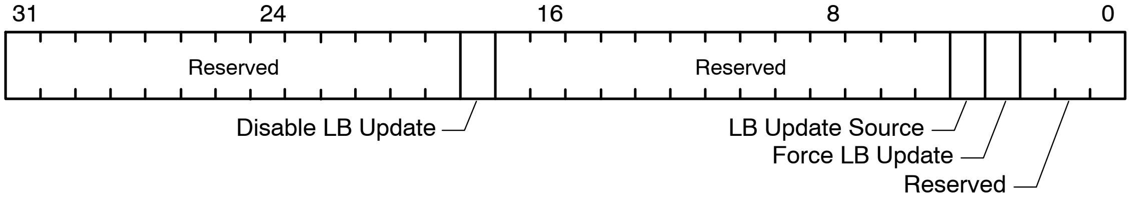
The DDA and other registers are shown in Table 4−15, where increment values are split into two registers.

Table 4−15. Depth Interpolation 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	

The Window register controls the update of the localbuffer, as shown in Figure 4−23.

Figure 4−23. Window Register



4.7.4 Stencil Example

In the following stencil example, the Stencil unit sets a supplied reference value (0×1) and tests fragments that are less than this value. It also sets the stencil planes update function to Decrement mode when the test is successful and when the depth test is successful or is not enabled. Otherwise, it sets the update function to Keep mode. Because Decrement is the selected mode, this example does not require that the Stencil register be loaded.

```
// Set the localbuffer read and write modes
// See section 4.6
// Set the stencil modes
stencilMode.UnitEnable = TVP4020_ENABLE
stencilMode.DPPass = TVP4020_STENCIL_METHOD_DECREMENT
stencilMode.DPFail = TVP4020_STENCIL_METHOD_KEEP
stencilMode.SFail = TVP4020_STENCIL_METHOD_KEEP
stencilMode.CompareFunction = TVP4020_STENCIL_COMPARE_LESS
stencilMode.StencilSource = TVP4020_SOURCE_TEST_LOGIC
StencilMode(stencilMode)
// Set the reference stencil value and set the
// compare and writemasks to 0x1
stencilData.ReferenceStencil = 0x1
stencilData.CompareMask = 0x1
stencilData.StencilWriteMask = 0x1
StencilData(stencilData)
// Enable the depth test here if required, if not enabled
// the result of the depth test is set to pass.
```

4.7.5 Depth Example

The following depth example shows the required setup for drawing a depth-buffered primitive:

```
// Set the localbuffer read and write modes
// See section 4.6
depthMode.UnitEnable = TVP4020_ENABLE
depthMode.WriteMask = 1
depthMode.NewDepthSource = TVP4020_NEW_DEPTH_SOURCE_DDA
depthMode.CompareMode = TVP4020_DEPTH_COMPARE_MODE_LESS
DepthMode(depthMode)
// Load the depth start values and deltas for the dominant
edge
// and the body of the trapezoid
ZStartU()    // Load upper and lower start values
```

```
ZStartL()
dZdxU()      // Load upper and lower dZdx deltas
dZdxL()
dZdyDomU()   // Load upper and lower dominant edge deltas
dZdyDomL()
// Render primitive
```

4.8 Texture Address Unit

The Texture Address unit calculates the address of the texel that maps to the current fragment XY position. You can apply perspective correction as part of the operation.

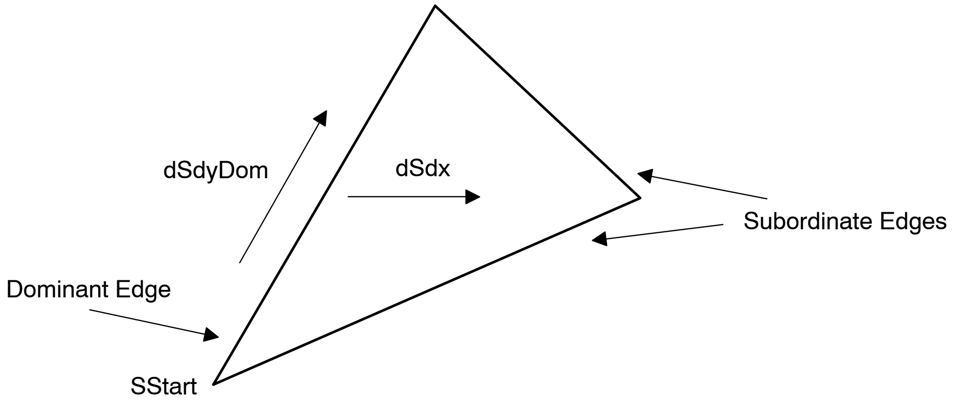
The texture coordinates are S and T, where S is analogous to X and T to Y. The S and T values are generated by interpolation; a third component, Q, may also be interpolated and used in perspective correction.

4.8.1 Texture Interpolation

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

The TVP4020 interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required per texture component, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated for the S component in Figure 4–24.

Figure 4–24. Texture Address Interpolation



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

The calculation for the delta values is the same as other parameters, such as depth values. Refer to subsection 4.3.5, *Calculating Depth Gradient Values,* for additional information.

When perspective correction is not enabled, then the S and T values are the texture coordinates of the appropriate vertex. When perspective correction is enabled, the texture coordinates are divided by the homogenous coordinate W, and Q is formed from 1/W. S and T are then normalized with respect to Q so that Q lies in the range 1 to 1/127. These values are then used to calculate delta values in the same way as for color or depth. If the dynamic range of Q is such that it cannot be normalized to the supported range, the software must either tessellate the triangle into smaller regions to reduce the range, or accept a reduction in accuracy; a Q value of zero is handled in a normal manner.

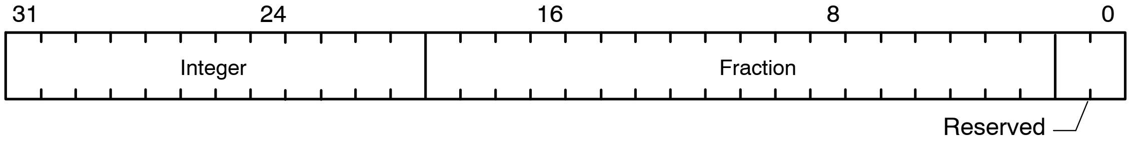
If perspective correction is enabled, each interpolated S and T value is divided by the interpolated Q value. If fast perspective correction is enabled, the faster but less accurate division is used. The result is passed to the Texture Read unit, which reads the texel from memory.

If subpixel correction is enabled for a primitive, then any correction required is applied to the texture coordinates.

4.8.2 Registers

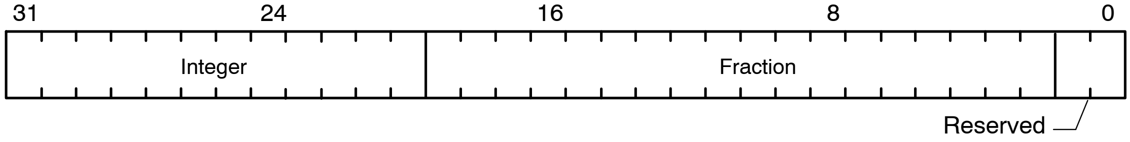
The S and T values are in a 30-bit 2s complement format, as shown in Figure 4−25.

Figure 4−25. Fixed-Point S and T Format



The Q values are in a 29-bit 2s complement format, as shown in Figure 4−26.

Figure 4−26. Fixed-Point Q Format



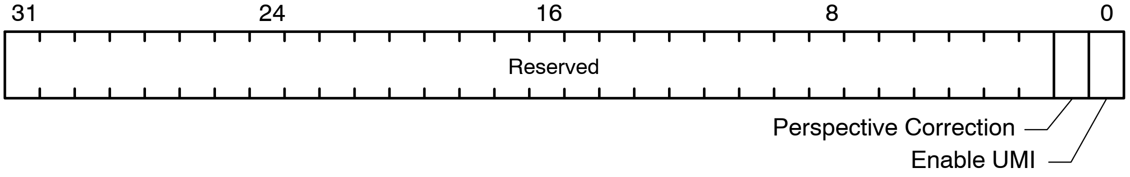
The registers to set up texture interpolation are shown in Table 4−16.

Table 4–16. Texture Interpolation Registers

Register	Data Field	Description
SStart	30-bit 2s comp fix pt	S start value
dSdx	30-bit 2s comp fix pt	S derivative per unit X
dSdyDom	30-bit 2s comp fix pt	S derivative per unit Y, dominant edge
TStart	30-bit 2s comp fix pt	T start value
dTdx	30-bit 2s comp fix pt	T derivative per unit X
dTdyDom	30-bit 2s comp fix pt	T derivative per unit Y, dominant edge
QStart	29-bit 2s comp fix pt	Q start value
dQdx	29-bit 2s comp fix pt	Q derivative per unit X
dQdyDom	29-bit 2s comp fix pt	Q derivative per unit Y, dominant edge

The TextureAddress Mode is shown in Figure 4–27.

Figure 4–27. TextureAddressMode



4.8.3 Texture Interpolation Example

This texture interpolation example shows how to set up the parameters for 2-D texture mapping. To achieve 1-D texture mapping, set *TStart*, *dTdx*, and *dTdyDom* to zero.

```
// Load the start values and deltas for the dominant edge
// and the body of the trapezoid
SStart()    // Load S start value
TStart()    // Load T start value
QStart()    // Load Q start value
dSdx()      // Load S delta for X
dTdx()      // Load T delta for X
dQdx()      // Load Q delta for X
```

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

4.9 Texture Read Unit

The texture buffer holds texture data. The buffer shares the same memory as the localbuffer and framebuffer. The Framebuffer Write unit typically writes texture maps to memory in a manner similar to image download.

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

4.9.1 Read Unit

The address calculation implements the following equations:

Bottom-left origin:

```
Address = TextureBaseAddress − T * W + S
```

Top-left origin:

```
Address = TextureBaseAddress + T * W + S
```

where:

Address = the address from which any read will be made

S = the texel S coordinate

T = the texel T coordinate

TextureBaseAddress = the base address of the current texture

W = the texture width. Only a subset of widths is supported and encoded into the PP0, PP1 and PP2 fields in the TextureReadMode register. See Appendix B, *Screen Widths Table*, for more details.

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

❑ Clamp: clamps the coordinate to 0 or the maximum value

❑ Repeat: accesses the map modulo for the width or height. This results in the texture map being repeated.

❑ Mirror: accesses the map modulo for the width or height and mirror alternate texture maps

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

4.9.2 Texture Base Address

The texture map base address is set in the TextureBaseAddress register. The lower 24 bits of this field specify the map address in texels. Bit 30 specifies that the texture is in system memory instead of in local memory and must be executed directly across the PCI bus without first copying the texture to local memory. See the *TVP4020 3D Graphics Processor Data Manual* for more details.

You can load the texture base address indirectly from memory using the TextureID register. Use the memory address of the texture base address (specified in 32-bit units). Loading the TextureID register prompts the real base address to load from memory. If bit 31 of the value loaded is set, the value is interpreted as invalid, the graphics processor is halted, and an interrupt is issued to the CPU. This mechanism indicates that the required texture is not resident in local memory and must be copied. Once copying is complete and the texture base address in memory is updated with its invalid bit cleared, the graphics processor rereads this value and restarts the process. See the *TVP4020 3D Graphics Processor Data Manual* for details on loading textures while the graphics processor is stalled.

4.9.3 Texture Filtering

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

4.9.4 Texture Formatting

The texture map can be held in memory in a variety of formats that correspond to the formats supported by the framebuffer. Two additional formats store texture maps in YUV color format. When a texel is read into the TVP4020, it is converted to the internal color format. External color formats are shown in Table 3–1.

Note:

The color format value is made up of the four bits of the TextureFormat field and the one bit TextureFormatExtension field in the TextureDataFormat register.

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

If the texture is in color index mode (either four or eight bits), the single value repeats for all color components. If the framebuffer format is also in color index mode, the single value is used as the pixel color; if the framebuffer is in RGBA mode, then the texture value becomes gray scale.

The texture values can be indexed through a 256 entry look-up table (LUT) inside the TVP4020. Each table entry holds a 32 bit RGB value. If you use the CI8 texture mode, then the whole LUT is used for each texture. If you use the CI4 texture mode, then each texture uses 16 entries, thus, 16 separate LUTs may be loaded and the appropriate one indexed (the upper four bits of the index are supplied by the upper four bits of TexelLUTIndex).

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

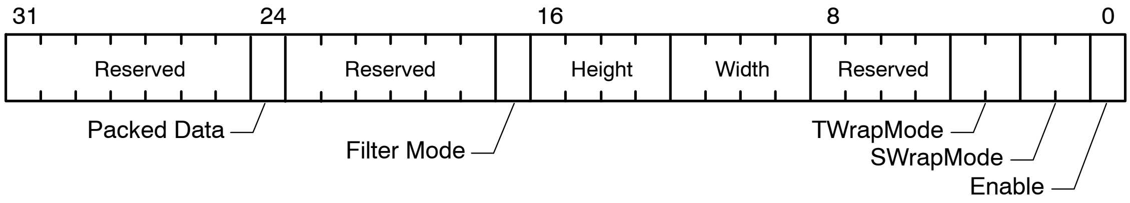
4.9.5 Registers

The TextureReadMode register, as shown in Figure 4−28, controls the way that textures are read from memory. With the filter mode bit disabled, nearest-neighbor texture mapping is performed. With this bit set, bilinear filtering is enabled.

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

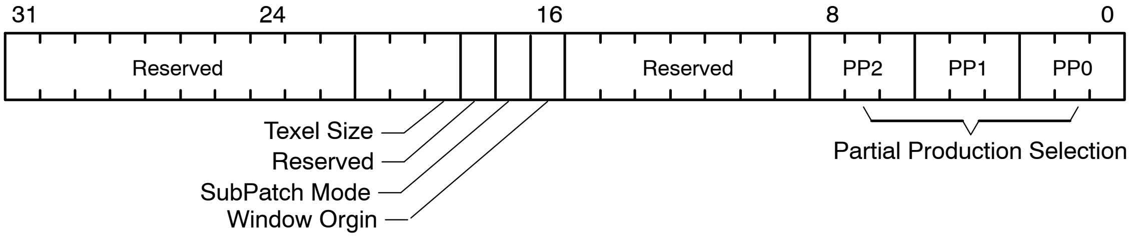
The packed data bit defines how texels are read from memory. If this bit is cleared, each texel is read one at a time; if set, several texels can be read simultaneously, improving efficiency. The actual number of texels read in this case is dependent on the texel size. See subsection 4.11.4, *Packed Copies*, for a description of how this case can be used for packed copies.

Figure 4−28. TextureReadMode Register



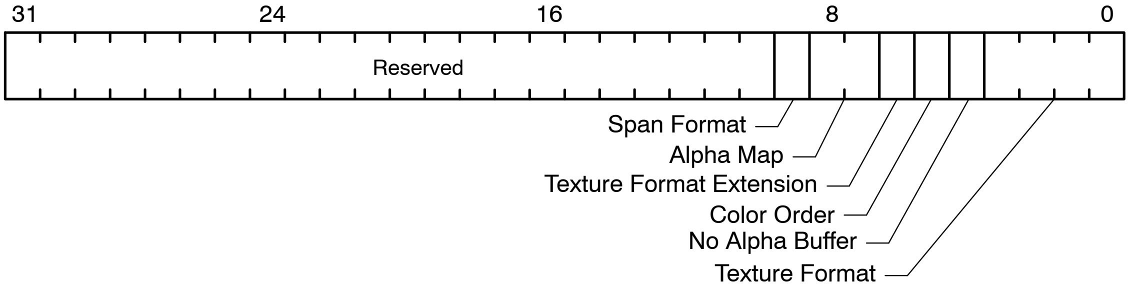
The TextureMapFormat register, as shown in Figure 4−29, specifies the way the texture map is held in memory. The partial product codes are detailed in Appendix B, *Screen Widths Table*. The window origin specifies the origin as either top-left or bottom-left. When enabled, SubPatchMode improves the performance of typical texture mapping.

Figure 4−29. TextureMapFormat Register



The TextureDataFormat register, as shown in Figure 4−30, specifies the color format of the texture. The TextureFormat, combined with the TextureFormat Extension, contains one of the modes described in Table 3−1. The color order specifies whether the texture is in RGB or BGR color format.

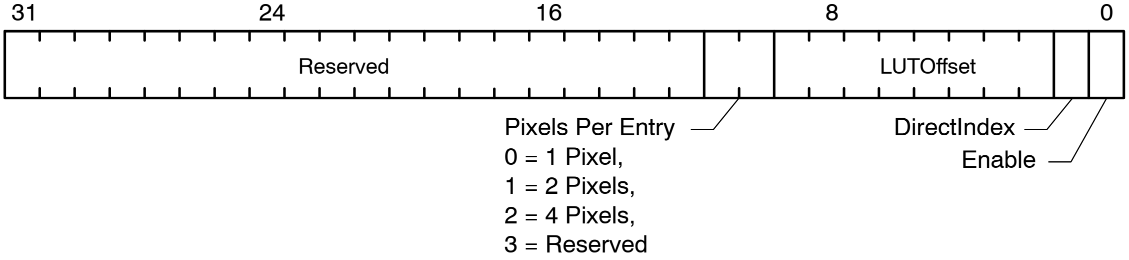
Figure 4−30. TextureDataFormat Register



4.9.6 Using the Texel LUT

The TexelLUT0-15 registers contain the texture color look-up table. Each register contains eight-bit fields for red, green and blue color components. The TexelLUTMode register, as shown in Figure 4−31, allows use of the TexelLUT0-15 registers. When enabled, the texel value becomes an index to the look-up table.

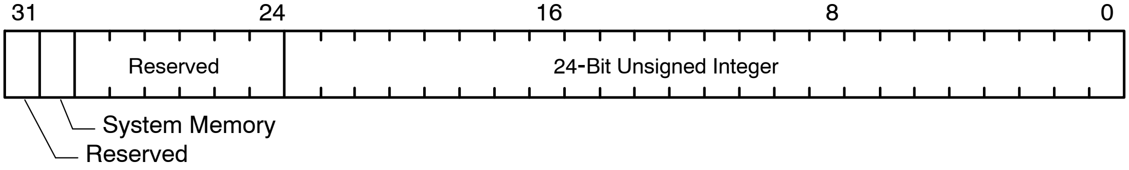
Figure 4−31. TexelLUTMode Register



You must enable the LUT before looking up a texture color. The other fields of this register control LUT 2-D operations. Enabling DirectIndex indexes the LUT by address fragment, not by reading from data memory. If using a block fill operation, the LUT is indexed at the start of every scanline. This is based on the lower three bits of the Y value (X is ignored), the LUTOffset added to the index, and the PixelsPerEntry field. Two consecutive entries in the LUT fill the upper and lower halves of the 64-bit block color register.

If block fill operations are not used, the lower three bits of each fragment's X and Y values are used to index the LUT. The PixelsPerEntry field scals the X and Y values so that an 8 x 8 pixel pattern is supported, and the LUTOffset field is added to the index before it is used. Figure 4−32 shows the TexelLUTAddress register.

Figure 4−32. TexelLUTAddress Register



If you use the LUT for 2-D operations, enable the Texture Application unit and set it to copy mode so that the texture color generated by the LUT converts to a color that can be plotted on the screen.

If all 256 LUT entries need to be filled, use the TexelLUTData and TexelLUTIndex registers. Write the LUT offset for the first entry to the TexelLUTIndex register, and then the succession of LUT entries to the TexelLUTData register. The LUT offset automatically increments after each entry is written.

It is also possible to load the LUT directly from memory by loading the TexelLUTAddress register with the LUT address in memory (in 32-bit units) and the TexelLUTTransfer register. The index field specifies the first entry to load, while the count filed specifies the number of entries to load. Bit 30 of the TexelLUTAddress register specifies that the LUT is resident in system memory and can be read across the PCI bus.

You can load the TextureLUTAddress register indirectly using the TexelLUTID register. This register operates identically to the TexelLID register. Some latency may occur between the register value being written to the TVP4020 and the interrupt being asserted, while it is possible that both registers were loaded before the interrupt was received. To determine which register caused the interrupt, you can read them back. They will hold the value read from the memory.

To read back the LUT entries, first read from the TexelLUTIndex register, which resets the readback index to zero. Then read from the TexelLUTData register as many times as necessary.

4.9.7 Block Fill Textures

If you enable texture mapping (and disable DirectIndex) when a block fill operation is performed, the mask for the filled block is read from memory as a texture map. The Texture Address unit must be set appropriately so that the S value increments or decrements by one, for each block of 32 pixels, while T stays at zero. The calculated texture address is used to index a texture map and returned data is used as a mask to control which pixels are plotted during a block fill operation. This feature can be used to draw text for which a font was previously loaded into a memory font cache.

You must byte align the layout of the data in memory, so that if the character is up to 8 pixels wide, specify a texel size of 8 bits. If it is up to 16 pixels, specify a size of 16 bits; up to 24 pixels, specify at 24 bits; and up to 32 pixels, specify at 32 bits. If the character is wider than 32 pixels, change to a word aligned bitmask and keep the pixel size to 32 bits. To match the normal data format for fonts, set the SpanFormat field in the TextureDataFormat register, which stores the data along with the bits in each mirrored byte.

4.9.8 Alpha Mapping

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

As the bilinear zoom magnification factor is increased, the variable alpha spreads across more destination pixels. The range of alpha values rejected by the chroma key test in the YUV unit can be adjusted to allow fine control over the exact size of the cutout. If blending is enabled, then the varying alpha values smooth the transition of the sprite edge to the background.

The AlphaMapUpperBound and AlphaMapLowerBound registers control the range over which the test is performed. The test is enabled by the TextureDataFormat register.

4.9.9 Texture Download Example

The following pseudocode is an example of texture downloading:

```
fbReadMode.PatchMode = TVP4020_TRUE
fbReadMode.SubPatchMode = TVP4020_SUBPATCH
FBReadMode(fbReadMode);
fbWriteMode.Enable = TVP4020_TRUE
FBWriteMode(fbWriteMode)
// Set format to 8 bits
ditherMode.UnitEnable = TVP4020_TRUE
ditherMode.Enable = TVP4020_FALSE
ditherMode.ColorMode = TVP4020_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)
// Do image download
```

4.9.10 Texture Mapping Example

The pseudecode for texture mapping a trapezoid is as follows:

```
textureAddressMode.Enable = TVP4020_TRUE
textureAddressMode.PerspectiveCorrection = TVP4020_TRUE
TextureAddressMode(textureAddressMode)
// Load texture address parameters
SStart()
dSdx()
dSdyDom()
TStart()
```

```
dTdx()
dTdyDom()
QStart()
dQdx()
dQdyDom()
// Configure texture read
textureReadMode.Enable = TVP4020_TRUE
textureReadMode.SWrapMode = TVP4020_TEXTURE_WRAP_REPEAT
textureReadMode.TWrapMode = TVP4020_TEXTURE_WRAP_REPEAT
textureReadMode.Width = width
textureReadMode.Height = height
textureReadMode.FilterMode = TVP4020_FALSE
TextureReadMode(textureReadMode)
textureMapFormat.PP0 = partialProduct0
textureMapFormat.PP1 = partialProduct1
textureMapFormat.PP2 = partialProduct2
textureMapFormat.SubPatchMode = TVP4020_TRUE
textureMapFormat.TexelSize = TVP4020_8_BITS_PER_TEXEL
TextureMapFormat(textureMapFormat)
textureDataFormat.TextureFormat = TVP4020_COLOR_FORMAT_RGB_332
TextureDataFormat(textureDataFormat)
// Enable texture/fog/blend unit, load other parameters
// and render
```

4.10 YUV Unit

The YUV unit converts the YUV color format, also known as *YCbCr*, to RGB format. It also performs chroma-key testing. Chroma-key testing may be performed either before or after the conversion.

The YUV conversion is performed on data that is being loaded into the Texel0 register. The data for this may come from the TextureRead unit or from the host, so YUV conversion can be performed either during texture download or on a texture as it is applied to a primitive. The YUV data can be in either 4–4–4 or 4–2–2 format. The chroma test may be done with either YUV or RGB data.

4.10.1 Chroma Test

The chroma test specifies upper and lower bounds, against which the Texel0 value is tested. The test may be set to pass when the components of Texel0 are either all inside or all outside the bounds. This is controlled by the accept/reject TestMode options of the YUVMode register. If the test passes, the Texel0 data may be used, as normal, in the Texture/Fog/Blend unit. If the test fails, then the fragment to which the texture data maps, may be rejected (not plotted). This is useful for cutouts and sprites.

Alternatively, on test failure, the Texel0 value may be rejected and the texture operation on the fragment suppressed. To achieve this, set the RejectTexel bit in the YUVMode register. In this case, the underlying color, provided by the TVP4020, is used without modification by the texture color. This is useful for applying a logo to a shaded polygon where the underlying color is provided by the Color DDA unit.

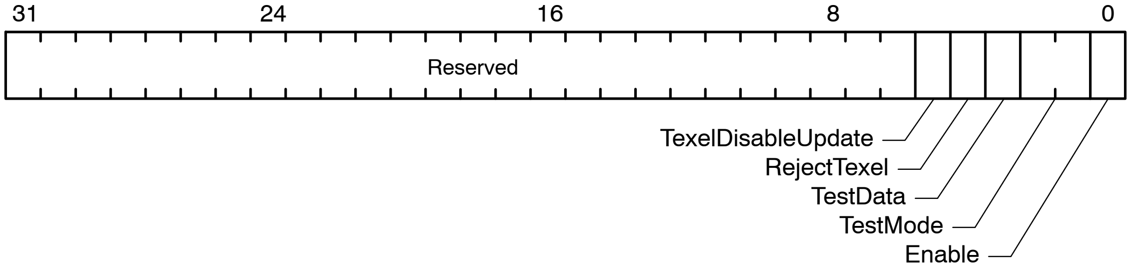
The test modes are shown in Table 4–17.

Table 4–17. Chroma Test Modes

Mode	Test Mode
0	No test
1	Accept
2	Reject

Chroma-key testing can be performed without involving texture mapping. To achieve this, set the TexelDisableUpdate field in the YUVMode register, as shown Figure 4–33. This allows fragments to be rejected by chroma testing as part of a copy operation. If chroma testing is required against the destination color of a copy (that is, only to overwrite pixels of the specified color), then the destination region of the screen is used as the texture map and the framebuffer units are set up to do a normal copy. The texels are read in and tested, and fragments are rejected if the colors do not match. If the fragment is rejected, then the pixel does not copy. Setting the TexelDisableUpdate bit discards the texel as soon as the test is complete, improving performance.

Figure 4−33. YUVMode Register

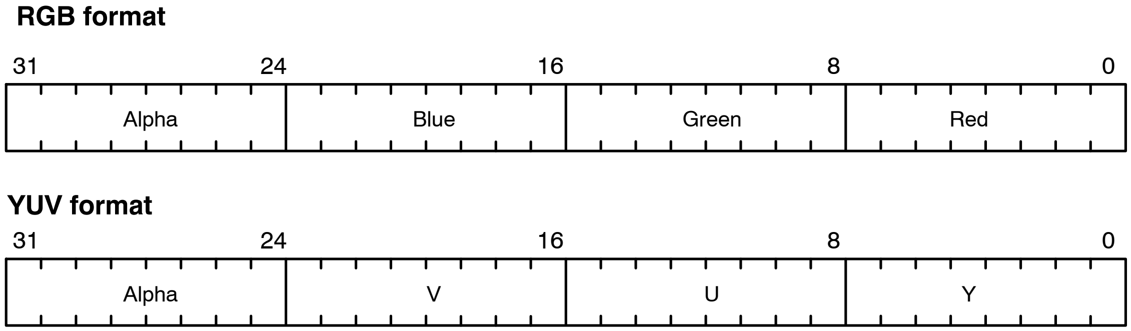


The TestData bit controls when the chroma test occurs in relation to the color conversion. Setting this bit causes the chroma test to occur on the output of the unit; clearing it causes the chroma test to occur on the input; that is, after or before color conversion, respectively, assuming the Enable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

The TestMode, as shown in Figure 4−34, can be set as follows:

- ❑ Accept; that is, pass test if upper bound ≤ color ≥ lower bound.
- ❑ Reject; that is, fail test if upper bound ≤ color ≥ lower bound.

Figure 4−34. ChromaUpperBound and ChromaLowerBound Registers



RGB format

YUV format

4.11 Framebuffer Read and Write Units

Before drawing can begin, the TVP4020 must be configured to perform the correct framebuffer read and write operations. Framebuffer read modes affect alpha blending, logic operations, software writemasks, image uploading, and image copy operations. Framebuffer write modes affect all drawing in the framebuffer.

4.11.1 Framebuffer Read

The FBReadMode register allows the TVP4020 to be configured to perform zero, one, or two read operations 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 reading of the framebuffer is required, and framebuffer write modes must be enabled. Framebuffer read modes must be disabled for maximum efficiency.

- ❑ Rendering operations that use logical operations, software writemasks, or alpha blending. In these cases, the destination pixel must be read from the framebuffer and framebuffer write modes must be enabled.

- ❑ Image copy operations. Here, setup depends on whether logical operations, software writemasks, and/or alpha blending are occurring with the copy. If any of these are occurring, the framebuffer must perform two read operations: one for the source and one for the destination. Otherwise, only one read operation is required.

- ❑ Image upload. This requires enabling the read mode of the destination framebuffer pixels, and disabling the framebuffer write mode.

- ❑ Image download. This case requires no framebuffer reading (as long as software writemasking, alpha blending, and logical operations are disabled), and the write mode must be enabled.

To enhance performance, avoiding unnecessary read operations.

For both the read and the write operations, an offset value is added to the calculated address. The source offset value (FBSourceOffset) performs copy operations. The pixel offset value (FBPixelOffset) allows multibuffer updates. For normal rendering, set the offset values to zero. The address calculation implements the following equations:

Note:

The OpenGL specification allows any combination of the front, back, left, and right color buffers to update simultaneously. In this case, a scene can be rendered multiple times, changing the FBPixelOffset as appropriate. When in this mode, ensure that the buffers that affect the rendering are updated only once. For example, when rendering with depth buffering enabled, make certain that localbuffer write modes are enabled only for the last buffer updated.

Bottom-left origin:

```
Destination address = FBWindowBase – Y * W + X +
   FBPixelOffset
Source address = FBWindowBase – Y * W + X +
   FBPixelOffset +  FBSourceOffset
```

Top-left origin:

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

where:

Destination address = the write address in the framebuffer when write modes are enabled, and also the read address when ReadDestination is enabled.

Source address = the read address in the framebuffer when ReadSource is enabled.

X = the pixel X coordinate.

Y = the pixel Y coordinate.

FBWindowBase = the base address in the framebuffer of the current window.

FBPixelOffset = zero except when multibuffer write modes provide a way to access pixels in alternative buffers, without changing the FBWindowBase register. This is useful since the window system may asynchronously change the window’s position on the screen. It is held in the FBPixelOffset register.

FBSourceOffset = zero except during a copy operation when data is read from one address and written to another. The FBSourceOffset is held in the FBSourceOffset register and is the offset from destination to source.

W = the screen width. Only a subset of widths is supported and encoded into the PP0, PP1, and PP2 fields in the FBReadMode register. See Appendix B, *Screen Widths Table*, for more details.

The data read from the framebuffer may be either FBDefault (data that may be written back into the framebuffer or used in some manner to modify the fragment color) or FBColor (data that will be uploaded to the host). Table 4−18 summarizes the framebuffer read/write control for common rendering operations.

Table 4−18. Framebuffer Read/Write Modes

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

Incorrect data can be read when read modes are enabled but the same data was just written with read modes disabled. To avoid this problem, send a WaitForCompletion command after enabling the read modes, but prior to the next primitive.

4.11.2 Framebuffer Write

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

The Framebuffer Write unit also controls the operation of fast block-fill operations, when they are supported by the framebuffer. Fast fill rendering is enabled via the FastFillEnable bit in the Render command register. The block color is 64-bits wide; normally the same values are used in the upper and lower halves of the register, so they are both set with the FBBlockColor register. If different data is required in both halves of the register, use the FBBlockColor Upper and FBBlockColorLower registers. The data put into the color registers must be in raw framebuffer format. When using the framebuffer in 8-bit packed mode, the TVP4020 repeats the data in each byte. When using the framebuffer in 16-bit packed mode, the TVP4020 repeats the data in the top 16 bits.

Due to restrictions in the way that the memory devices implement block-fill operations, a packed 24-bit RGB framestore can only use block filling for colors that have all pixel bytes set to the same value.

When uploading images, set the UpLoadData bit to allow color formatting. (see subsection 4.13.4, *Image Formatting*).

4.11.3 Patching

Under certain circumstances, data in the framebuffer can use patched addressing to improve performance. However, only nonvisible data is usually patched. Patch mode organizes data for efficient drawing of scanline primitives; it also aids in line drawing. This form is typically used in the localbuffer for patching the depth buffer. See subsection 4.6.4, *Registers*, for additional information. The SubPatch mode reorganizes data for efficient texture operations, as explained further in subsection 4.9.5, *Registers*. SubPatchPack mode is used when four-bit textures are loaded as eight bits; that is, the subpatch packing accounts for the two texels per byte.

4.11.4 Packed Copies

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

4.11.5 Image Downloads

An image download can be performed in one of three ways:

- By loading the data in standard color format into the Color register and using the Color Format unit to organize it into the framestore format
- By loading the data in raw framebuffer format into either the Color register or the FBData register. The former requires that the Color Format unit be disabled while the latter ignores this unit.
- By loading the data in another raw format into the FBSourceData register while the Texture/Fog/Blend unit converts it into the internal color format. The Color Format unit then converts it into the arrangement to be stored in the framebuffer.

All techniques require an appropriate rasterizer set up.

4.11.6 Fast Texture Download

Normal texture is downloaded as an image. This involves setting up the rasterizer to draw a rectangle, and changing the state of a number of units. This is the preferrable procedure when any processing needs to be done, such as color format conversion, color space conversion, or patching.

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

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

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

4.11.7 Hardware Writemasks

Hardware writemasks, when available, are controlled using the FBHardwareWriteMask register. If the framebuffer memory devices support hardware writemasks and you plan to use them, then disable the software writemask by setting all the bits in the FBSoftwareWriteMask register. This will result in using fewer framebuffer read modes when no logical operations or alpha blending is needed.

If the framebuffer is used in an 8-bit packed mode, then an 8-bit hardware writemask must repeat in all four bytes of the FBHardwareWriteMask register. If the framebuffer is in a 16-bit packed mode, then the 16-bit hardware writemask must repeat in both halves of the FBHardwareWriteMask register.

Note:

Since there is no overall enabling for this feature, the hardware writemask must be set to all ones, except when hardware writemasking is explicitly required.

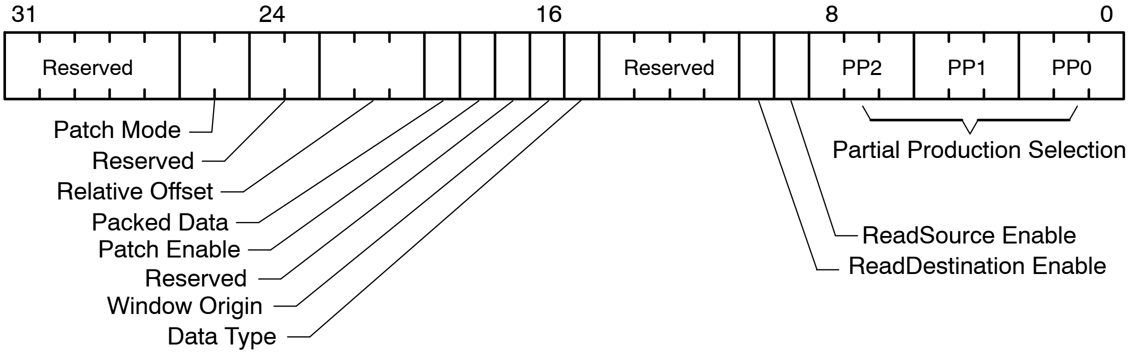
4.11.8 Frame Blank Synchronization

You may use the SuspendUntilFrameBlank command register to stall the TVP4020 graphics hyperpipeline until the next frame blank. You can maximize system performance by using double buffering and synchronizing to the monitor's frame blanks. This register controls full-screen double buffering through the graphics hyperpipeline and, thus, the host does not need to wait for the vertical frame blank. Instead, once the SuspendUntilFrameBlank command register is loaded, the host can continue to load the TVP4020 registers and issue commands. The TVP4020 continues to process these as long as they do not involve writing to the framebuffer. The data field of this register, that is, the base address of the buffer to be displayed, is passed to the internal video timing generator.

4.11.9 Registers

The FBReadMode register layout is shown in Figure 4−35.

Figure 4−35. FBReadMode Register

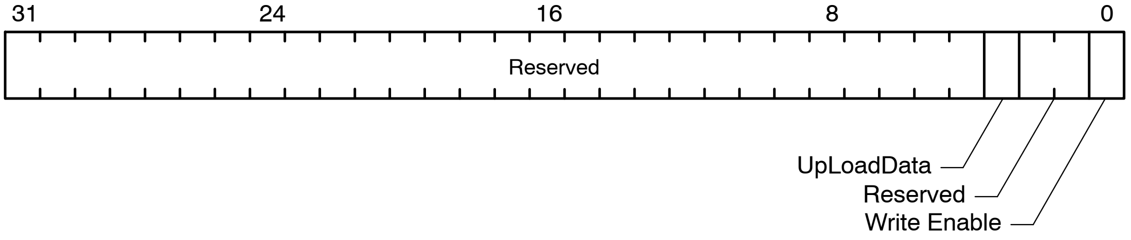


See Appendix B, *Screen Widths Table*, for more information on setting partial product codes.

The FBWindowBase register holds the base address of the framebuffer window, in 24-bit unsigned format. The FBPixelOffset and FBSourceOffset registers hold 24-bit 2s complement offset values for copy operations and multibuffer updates, as described above.

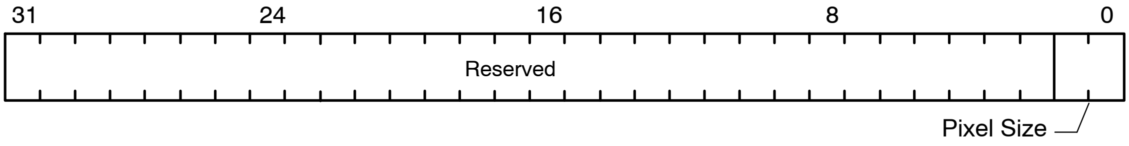
The FBWriteMode register controls the framebuffer write operations as shown in Figure 4−36.

Figure 4−36. FBWriteMode Register



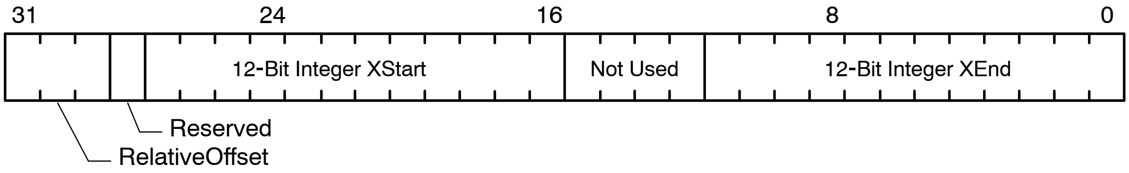
The FBReadPixel sets the pixel size as shown in Figure 4−37.

Figure 4−37. FBReadPixel Register



The PackedDataLimits register controls packed copies as shown in Figure 4−38.

Figure 4−38. PackedDataLimits Register



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

4.11.10 Image Copy Example

In this example, the TVP4020 copies a rectangular region of the framebuffer, without moving any data in the localbuffer. The region extends from the origin (0,0) to (100,100) and is shifted right by 200 pixels. The destination rectangle is scan converted.

```
// First set-up the framebuffer read mode
fbReadMode.ReadSource = TVP4020_ENABLE
fbReadMode.ReadDestination = TVP4020_DISABLE
fbReadMode.DataType = TVP4020_FBDEFAULT
FBReadMode(fbReadMode)        // Update register
// Now enable framebuffer write
fbWriteMode.WriteEnable = TVP4020_ENABLE
FBWriteMode(fbWriteMode)      // Update register
// Offsets. No Pixel offset, source offset of 200
FBPixelOffset (0x0)
FBSourceOffset (−200)
// All the tests which could remove the fragment must
// be disabled (Stipple, Stencil, Depth) except
// the Scissor test which is still needed for screen
// and possibly window clipping.
// If software writemasks are to be used then they are
// set appropriately, and the framebuffer set up to do
// extra read operation
// Disable the Color DDA unit, we do not want to
// associate a color with this fragment.
colorDDAMode.UnitEnable = TVP4020_FALSE
ColorDDAMode(colorDDAMode)
// Define the region we wish to copy from.
StartXDom (200<<16)
StartXSub (300<<16)
dXSub (0)
dXDom (0)
StartY (0)
dY (1<<16)
Count (100)
render.PrimitiveType = TVP4020_TRAPEZOID
Render (render)    // Start the rasterization
```

4.12 Color DDA Unit

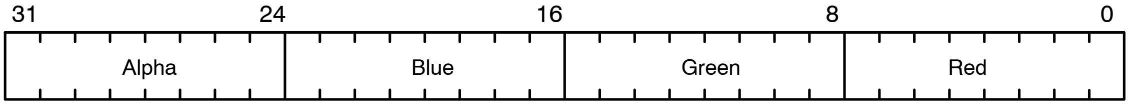
The Color DDA unit associates a color with a fragment produced by the rasterizer. Enable this unit for rendering operations and disable it for pixel rectangle operations (that is, copies, uploads, and downloads).

4.12.1 RGBA and Color Index (CI) Modes

Two color modes are supported by the TVP4020: true color RGBA and color index (CI).

The TVP4020 internal color representation is RGBA with eight bits per component, as shown in Figure 4−39.

Figure 4−39. TVP4020 Color Representation (True Color)



This format is the same for all the supported framebuffer configurations supported. If the number of bits in the framebuffer for a color component is less than eight, then the TVP4020 shifts the color value to the left into the most significant bits of that component field. The TVP4020 sets the unused, least significant bits to zero.

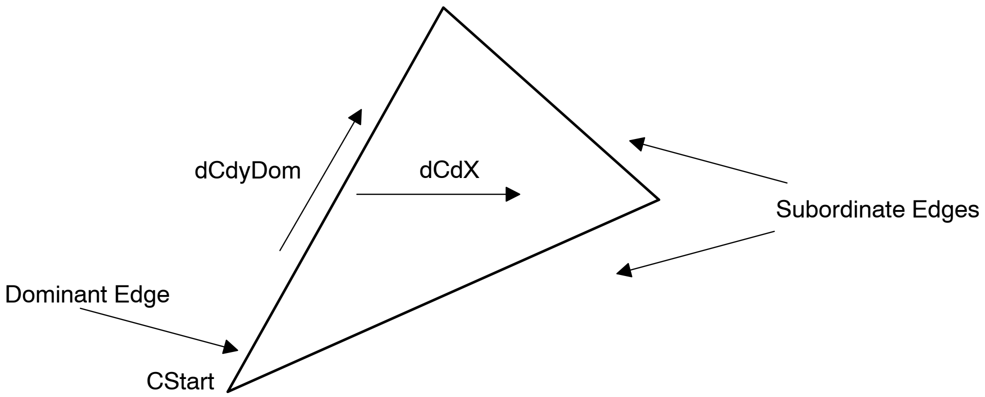
In CI mode, the TVP4020 places the color index in the lower byte of the 32-bit register (that is, the red component).

4.12.2 Gouraud Shading

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

For a Gouraud-shaded trapezoid, the TVP4020 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 4−40, where C represents a color component (red, green, blue, or color index). Alpha is not interpolated and stays at its initial value.

Figure 4−40. Color Interpolation



CStart = the initial color value

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

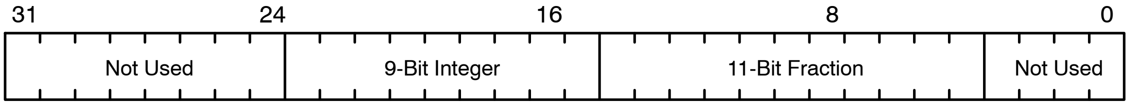
dCdx = color gradient in the X direction

See Section 4.3, *A Gouraud-Shaded Triangle Without Using the Delta Unit*, for the Gouraud-shaded triangle delta values.

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

To allow accurate interpolation, the increment values are specified in a 17-bit fixed-point format. The format is 2s complement with a one-bit sign, a five-bit integer, and an 11-bit fraction, as shown in Figure 4−41.

Figure 4−41. Fixed-Point Color Format



Note:

If you are rendering to multiple buffers and initialize the start and increment values in the Color DDA unit, then any subsequent Render command will reload the start values.

If subpixel correction is enabled for a primitive, then any correction required will apply to the color components.

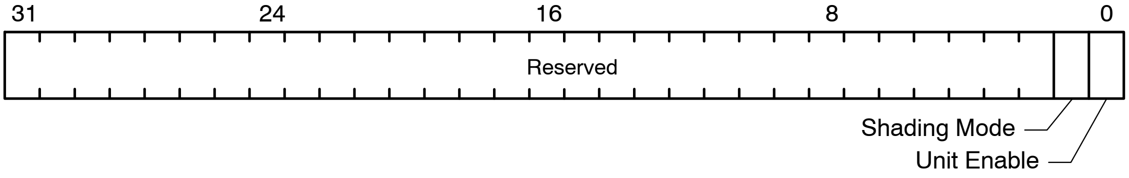
4.12.3 Flat Shading

In flat-shading mode, a constant color is associated with each fragment. This color is loaded into the ConstantColor register in the format previously shown in Figure 4−39.

4.12.4 Registers

The main control register for the Color DDA unit is the ColorDDAMode register, as shown in Figure 4−42.

Figure 4−42. ColorDDAMode Register



The registers to set up Gouraud shading in the Color DDA unit are defined in Table 4−19.

Table 4−19. Color Interpolation Registers

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

4.12.5 Flat Shading Example

The following example shows pseudocode for a flat-shaded primitive:

```
// Set DDA to flat shade mode
colorDDAMode.UnitEnable = TVP4020_ENABLE
colorDDAMode.Shade = TVP4020_FLAT_SHADE_MODE
ColorDDAMode(colorDDAMode)
ConstantColor(0xFFFFFFFF)// Load the flat color
```

4.12.6 Gouraud-Shaded Trapezoid Example

The following example shows pseudocode for a Gouraud-shaded trapezoid. See Section 4.3, *A Gouraud-Shaded Triangle Without Using the Delta Unit*, for details on how to calculate delta values.

```
// Enable unit in Gouraud shading mode
colorDDAMode.UnitEnable = TVP4020_ENABLE
colorDDAMode.Shade = TVP4020_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 ()
```

4.12.7 Gouraud-Shaded Line Example

The following example shows pseudocode for a Gouraud-shaded line. See Section 4.3, *A Gouraud-Shaded Triangle Without Using the Delta Unit*, for details on how to calculate delta values.

```
   // Set DDA for Gouraud shaded mode
   colorDDAMode.UnitEnable = TVP4020_ENABLE
   colorDDAMode.Shade = TVP4020_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
```

4.13 Texture/Fog/Blend

The Texture/Fog/Blend unit applies effects to the interpolated color in the following order: texture, fog, blend.

4.13.1 RGB/Ramp Texture Applications

There are two major types of texture applications: RGB applications and Ramp. RGB is the OpenGL-type application. Ramp applications use RGB textures and framebuffer format but are limited to a white-light source. The Enable bit in the TextureColorMode register and the TextureEnable bit in the Render register must both be enabled before texture is applied.

❑ RGB Texture Application

The RGB texture application is also referred to as the OpenGL type of texture application. It is applied in one of three modes.

- In copy mode, the texture color replaces the current fragment color.
- In decal mode, the texture color blends with the fragment color using the texture alpha value:

  $C_f = C_tA_t + C_f(1-A_t)$
  $A_f = A_f$
  where:

  C_f is the fragment color, C_t is the texture color, A_f is the fragment alpha, and A_t is the texture alpha. If the texture alpha value is one, decal mode becomes the same as copy mode.
- In modulate mode, the color components are multiplied together:

  $C_f = C_tC_f$
  $A_f = A_tA_f$

  where:

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

❑ Ramp Texture Application

The Ramp texture application is also referred to as the Apple type of texture application because of the approach adopted by QuickDraw3D™. This type of texture application is applied in three modes, where each mode can be independently enabled or disabled.

- The first mode is decal, which performs the following operation:

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

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

$C_f = C_t$
$A_f = A_t A_f$

- The next mode is modulate, which performs the following operation:

  $C_f = K_d C_D$
  $A_f = K_d A_D$

  where:

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

- The next mode is highlight, which performs the following operation:

  $C_f = C_M + K_s$
  $A_f = A_M + K_s$

  where:

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

4.13.2 Fog Application

The fog unit combines the incoming fragment color (generated by the Color DDA unit, and potentially modified by the texture unit) with a predefined fog color. Fogging is used for simulating atmospheric fogging and for depth-cueing images.

The fog application has two stages: the first for derivating the fog index for a fragment and the second for applying the fogging effect. The fog index is a value that is interpolated over the primitive using a DDA in the same way color and depth are interpolated. The fogging effect is applied to each fragment using the equation described on the next page under Fog Index Calculation – The Fog DDA.

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

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

The Enable bit in the FogMode register and the FogEnable bit in the Render register must both be enabled before fog is applied.

❑ Fog Index Calculation – The Fog DDA

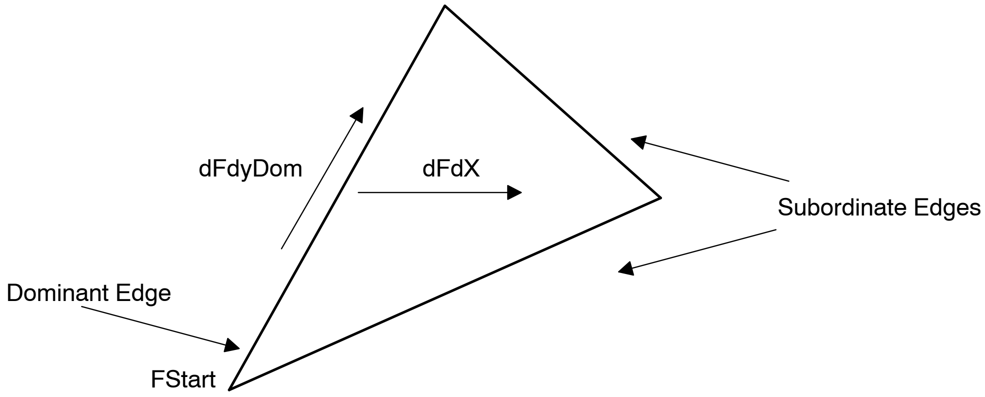
The fog DDA interpolates the fog index (F) across a primitive. For a fogged trapezoid, the TVP4020 interpolates from the dominant edge of a trapezoid to the subordinate edges. This means that two increment values are required, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in Figure 4–43. The rendering direction is bottom to top, as defined below.

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

The *dFdx* value is not required for fogged lines.

The mechanics are similar to those of the other DDA units, as shown in Figure 4–43.

Figure 4–43. Fog Interpolation Over A Triangle



where:

FStart = Initial fog value.

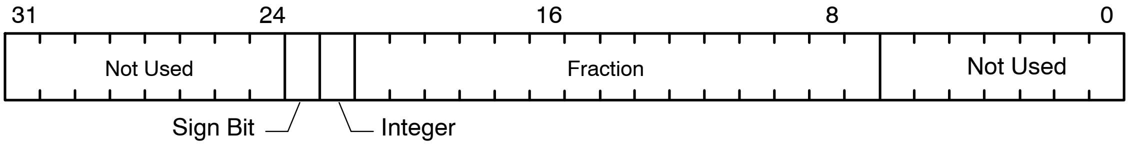
dFdx = Fog gradient in the X direction.

dFdyDom = Fog gradient along the dominant edge of a primitive.

The *dFdx* delta value is not required for fogged lines.

The fog index is specified as an 18-bit fixed-point value. The format is 2s complement with a 2-bit integer and a 16-bit fraction, as shown in Figure 4−44.

Figure 4−44. Fog Interpolant Fixed Point Format



The fog DDA calculates a fog index value which lies clamped in the range of 0.0 to 1.0, before it is used in the following fogging equations.

❑ Fogging Equation

The fogging equation is:

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

where:

C = outgoing fragment color

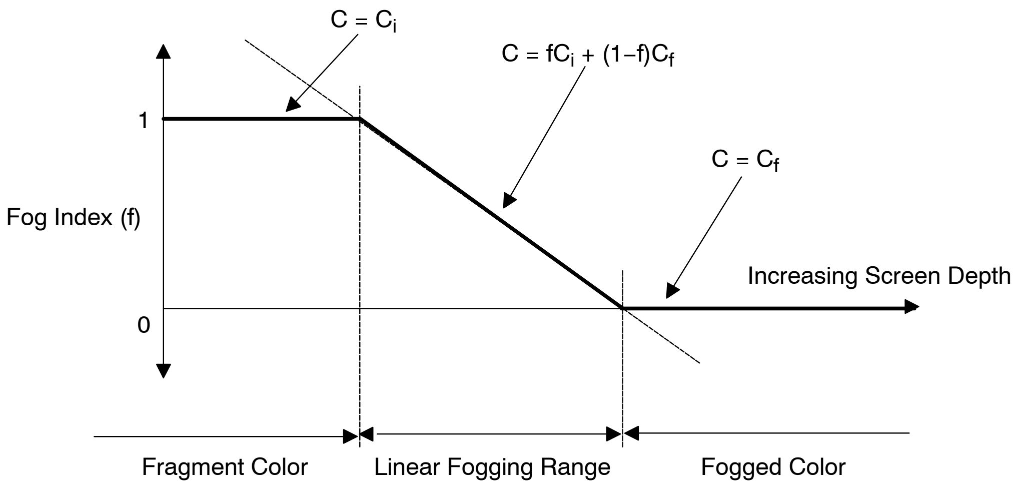
C_f = fog color

C_i = incoming fragment color

f = fog index

The equation is applied to the color components red, green and blue; alpha is not modified. Figure 4−45 shows how the fogging typically affects a scene. Initially no fogging occurs ($f \geq 1.0$), then a region of linear-combined fragment color and fog color occurs ($1.0 < f > 0.0$), followed by a region of constant fog color ($f \leq 0.0$).

Figure 4−45. Fogging



4.13.3 Alpha Blending

Two types of alpha blending are supported: one that is common to RGB and Ramp applications and one that is specific to Ramp applications. Alpha blending combines the fragment color with that stored in the framebuffer, after texture and fog are applied.

Data from the framebuffer is in raw format, so it must be converted to the internal format before blending. To convert, set the ColorFormat and ColorFormat Extension fields in the AlphaBlendMode register.

In some situations, blending is desired when no retained alpha buffer is present. In this case, the alpha value that is read from the framebuffer is set to 1.0 using the NoAlphaBuffer bit in the AlphaBlendMode register.

❑ Common Blend Mode

The common blend operation is defined as:

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

where:

C_o is the output color, C_s is the source color, A_s is the source alpha, and C_d is the destination color read from the framebuffer. To achieve this, set the operation field to Blend in the AlphaBlendMode register.

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

❑ Ramp Blend Mode

The alternative blend mode is called PreMult and performs the following operation:

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

4.13.4 Image Formatting

The Alpha Blend and Color Format units can be used to format image data into any of the supported TVP4020 framebuffer formats.

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

❑ Set the rasterizer as appropriate (see subsection 4.4.10, *Copy/Upload/Download*).

❑ Enable framebuffer read modes.

❑ Disable framebuffer write modes and set the UpLoadData bit in the FBWriteMode register.

❑ Enable the Alpha Blend unit, set the operation to format (assuming no alpha blending is needed), and set the color mode to RGBA 5.5.5.1 by setting the appropriate fields in the AlphaBlendMode register.

❑ Set the Color Format unit to an 8-bit RGB 3:3:2 framebuffer format for incoming fragments.

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

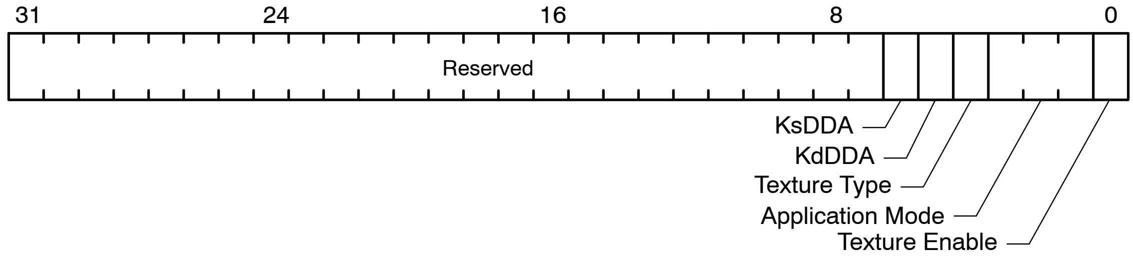
The same technique can be used to download data that is in any supported framebuffer format. In this case, set the rasterizer to synchronize with FBData (rather than Color), enable the framebuffer write modes, and clear the UpLoadData bit.

Usually, internal color and alpha values require scaling if they are less than 8 bits. However, there are situations where the least significant bits should be at zero. This is necessary for multipass rendering to prevent dithering from occurring multiple times. You can independently apply this option to color and alpha values by setting the ColorConversion and/or AlphaConversion bits to shift rather than to scale in the AlphaBlendMode register.

4.13.5 Registers

The TextureColorMode register, as shown in Figure 4–46, enables and disables texturing (qualified by the texture application bit in the Render command register). The KsDDA and KdDDA bits enable the internal DDAs and must be set to modulate or highlight Ramp texture application modes. The Texture Type field differentiates between Ramp and RGB application modes. Combinations of decal, modulate, and highlight are supported with Ramp application mode.

Figure 4−46. TextureColorMode Register



The Texel0 register holds the texture value, which may be loaded automatically by the Texture Read unit, or supplied from the host for a procedural texture. Figure 4−47 and Figure 4−48 show texture values in RGB and YUV formats, respectively. The Texel0 register also holds the background color for the bitmask and stipple tests. If the tests fail, then this color can be used in place of the color from the Color DDA unit.

Figure 4−47. Texel0 Register – RGB Format

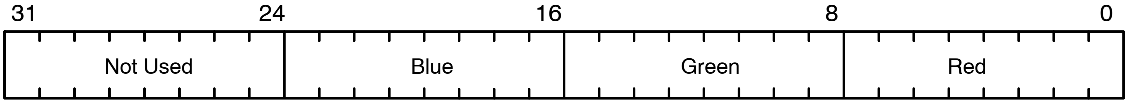
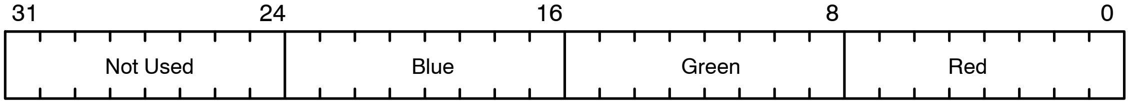


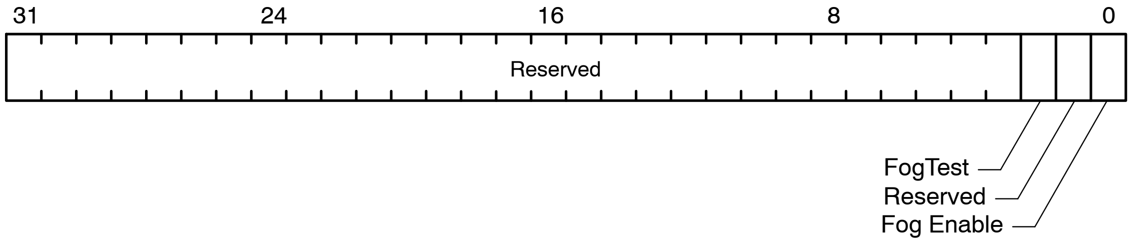
Figure 4−48. Texel0 Register – YUV Format



The six registers: KsStart, dKsdx, dKsdyDom, KdStart, dKddx and dKddyDom hold the *start*, *dx*, and d*yDom* parameters for Ks and Kd. The format is 2s complement 2.16 fixed-point format (1-bit sign, 1-bit integer, 16-bit fraction) with an effective range of ±1.999. The values of Ks and Kd at each vertex are used to calculate the gradient values in much the same way as the *Z* gradients are used to interpolate depth, as described in subsection 4.3.5, *Calculating Depth Gradient Values*.

The FogMode register, as shown in Figure 4−49, enables and disables fogging (qualified by the fog application bit in the Render command register). When set, the Fog Test rejects fragments with negative fog values, as described in subsection 4.13.2, *Fog Application*.

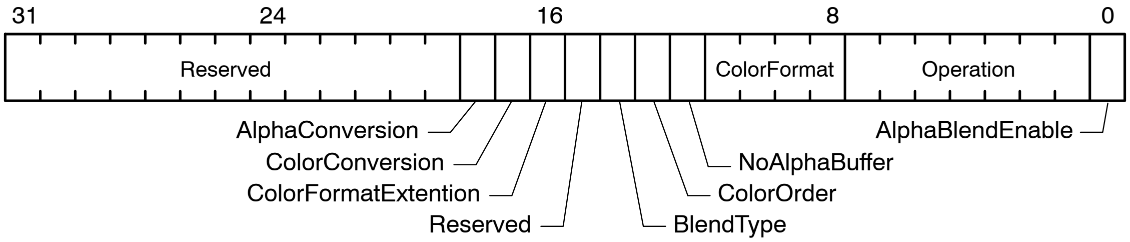
Figure 4−49. FogMode Register



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

Blending is controlled by the AlphaBlendMode register, as shown in Figure 4−50.

Figure 4−50. AlphaBlendMode Register



The color format and order are needed when the destination color is read from the framebuffer and converted into the internal TVP4020 color representation. The color should, therefore, be set, as appropriate, for the framebuffer. The operation can be performed in either format, blend, or PreMult mode.

4.13.6 Texture Application Example

The following pseudocode is an example of a texture-mapped trapezoid:

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

textureColorMode.Enable = TVP4020_TRUE
textureColorMode.ApplicationMode = TVP4020_TEXTURE_MODULATE
TextureColorMode(textureColorMode)
// Render with texture enabled in render command
// render.TextureEnable = TVP4020_TRUE
```

4.13.7 FogExample

The following pseudocode defines how to set a Gouraud-shaded, fogged RGBA trapezoid to white. For details on how to calculate depth delta values, see subsection 4.3.5, *Calculating Depth Gradient Values*. Calculate fog values the same way.

```
// Enable the Color DDA unit in Gouraud shading mode
colorDDAMode.UnitEnable = TVP4020_ENABLE
colorDDAMode.Shade = TVP4020_GOURAUD_SHADE_MODE
ColorDDAMode(colorDDAMode)
// Enable the Fog unit
fogMode.FogEnable = TVP4020_TRUE
FogMode(fogMode)
// Set the fog color to white
FogColor(0xFFFFFFFF)
// Load the color start values and deltas for dominant edge
// and the body of the trapezoid
RStart()     // Set-up the red component start value
dRdx()       // Set-up the red component increments
dRdyDom()
GStart()     // Set-up the green component start value
dGdx()       // Set-up the green component increments
dGdyDom()
BStart()     // Set-up the blue component start value
dBdx ()      // Set-up the blue component increments
dBYDom()
```

```
// Load the start value and delta for dominant edge
// and the body of the trapezoid
// Note that the fog deltas are calculated in the same
// way as the color deltas
FStart()     // Set-up the fog component start value
dFdx()       // Set-up the fog component increments
dFdyDom()
// When issuing a Render command the FogEnable bit
// should be set in addition to the fog unit being
// enabled:
// render.FogEnable = TVP4020_TRUE
```

4.14 Color Format Unit

The Color Format unit converts the TVP4020 internal color representation to a format suitable to be written into the framebuffer. This process may optionally include color value dithering. If the unit is disabled, the color is not modified in any way.

4.14.1 Color Formats

The framebuffer may be configured to RGBA or Color Index (CI) mode. Table 3−1 shows the full list of color modes, or formats, supported by the TVP4020. The R, G, B, and A columns show the width of each color component. The least significant bit position is 0. For the front and back modes, the value repeats in both buffers, and writemasks may be used to update only one buffer. In CI mode, the index repeats in all streams.

4.14.2 Color Dithering

The TVP4020 uses an ordered dither algorithm to implement color dithering. It also has a line dither mode that uses a different algorithm, which generally gives better results for lines because it is independent of orientation. This mode is not available for trapezoids.

If the Color Format unit is disabled, the color component’s RGBA values are not modified and are truncated when placed in the framebuffer. In CI mode, the values are truncated to the nearest integer. In both cases, the results are clamped to a maximum value to prevent overflow.

The TVP4020 supports an 8:8:8:8 RGBA format for 2-D operations only. With this mode selected and dithering enabled, the RGBA quality for each 32-bit pixel is 5:5:5:1. You can use this mode when the window manager needs to be set up for true color at the same time that 3-D windows are required.

In some situations, only screen coordinates are available but window-relative dithering is required. You can resolve this by setting up the optional X and Y offset values that are added to the coordinates before the dither tables are indexed. Each offset value for each coordinate is a 2-bit number. The XOffset and YOffset fields in the DitherMode register control this operation. Set them to zero if you use window-relative coordinates.

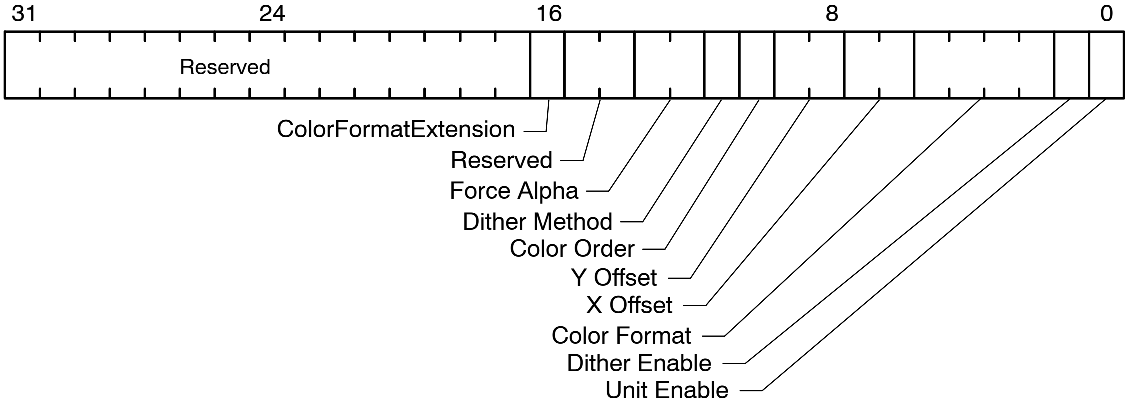
4.14.3 ForceAlpha

The Color Format unit can force the alpha value to be either 0×0, the maximum 0×FF, or it can leave the value unchanged. This value can be used to implement overlays. See Section 6.6, *Overlays*, for a detailed description.

4.14.4 Registers

One register controls the operation of the Color Format unit, the DitherMode register, and the DitherMode layout, as shown in Figure 4−51.

Figure 4−51. DitherMode Register



The X and Y offset fields are for window-relative dithering. Color order specifies RGB or BGR color order. The color format and Color format-extension fields control color depth.

4.14.5 Dither Example

Use the pseudocode below to set the framebuffer format to RGB 3:3:2 and enable dithering:

```
// 332 Dithering
ditherMode.UnitEnable = TVP4020_TRUE
ditherMode.DitherEnable = TVP4020_TRUE
ditherMode.ColorMode = TVP4020_COLOR_FORMAT_RGB_332
DitherMode (ditherMode)  // Load register
```

4.14.6 3:3:2 Color Format Example

Use the pseudocode below to set the framebuffer format to RGB 3:3:2 and disable dithering:

```
// 332 No Dither
ditherMode.UnitEnable = TVP4020_TRUE
ditherMode.DitherEnable = TVP4020_FALSE
ditherMode.ColorMode = TVP4020_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)   // Load register
```

4.14.7 8:8:8:8 Color Format Example

Use the pseudocode below to set the framebuffer to RGBA 8:8:8:8 without dithering:

```
// 8888 Dithered (No effect as 8 bit components are
// not dithered)
ditherMode.UnitEnable = TVP4020_TRUE
ditherMode.DitherEnable = TVP4020_FALSE
ditherMode.ColorMode = TVP4020_COLOR_FORMAT_RGBA_8888
DitherMode(ditherMode)   // Load register
```

4.15 Logical Op Unit

The Logical Op unit controls three functions:

- ❑ Controls logical operations between the fragment color (source color) and a value from the framebuffer (destination color)
- ❑ Controls software writemasking
- ❑ Optionally controls a special TVP4020 mode that allows high-performance, flat-shaded rendering

4.15.1 Logical Operations

The logical operations supported by the TVP4020 are listed in Table 4−20.

Table 4−20. Logical Operations

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

[†] S = source (fragment) color
[‡] D = destination (framebuffer) color

To operate this unit in a mode that accepts the destination color, use the FBReadMode register to configure the TVP4020 to read from the framebuffer. See Section 4.11, *Framebuffer Read and Write Units*, for more details.

The TVP4020 makes no distinction between RGBA and CI modes when performing logical operations. However, logical operations are generally only used in CI mode.

4.15.2 Software Writemasks

Software writemasking is usually implemented only when hardware writemasking is unavailable. It is controlled by the FBSoftwareWriteMask register. The data field has one bit per framebuffer bit that, when set, updates the corresponding framebuffer bit. When reset, it protects the bit from being overwritten. 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 ones. If the mask is not all ones, the ReadDestination bit must be enabled in the FBReadMode register to correctly use software writemasks. See Section 4.11, *Framebuffer Read and Write Units*, for details on how to enable/disable framebuffer read modes.

Note:

The software writemask must be set to all ones, except when software writemasking is explicitly required.

4.15.3 Flat-Shaded Rendering

A special TVP4020 rendering mode renders unshaded images.The method, however, is not highly recommended for the TVP4020. Other methods of flat shading are at least as fast and are simpler to set up. It is included here as part of the legacy TVP4010 software. To use this mode, you must satisfy the following requirements:

❑ Flat-shaded primitive
❑ No dithering
❑ No logical operations
❑ No stencil or depth testing
❑ No alpha blending

The following are available:

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

If you meet all the requirements, you can use this method of rendering by setting the FBWriteData register to hold the framebuffer data (in raw framebuffer format) and by setting the UseConstantFBWriteData bit in the LogicalOpMode register. Be certain to disable all unused units.

This mode is most useful for 2-D applications or for clearing the framebuffer when the memory does not support block write operations.

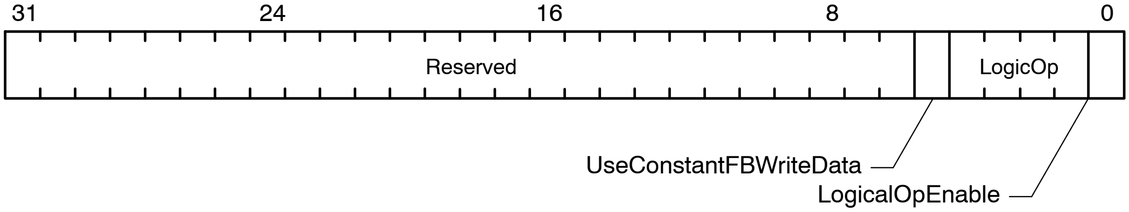
Note:

The FBWriteData register is volatile during context switching.

4.15.4 Registers

The LogicalOpMode register controls the operation of the LogicOp unit, as shown in Figure 4−52.

Figure 4−52. LogicalOpMode Register



4.15.5 XOR Example

Use the following pseudocode to set the logical operation to XOR:

```
// Set framebuffer to allow reads
// Not shown
logicalOpMode.UnitEnable = TVP4020_ENABLE
logicalOpMode.LogicalOp = TVP4020_LOGICOP_XOR
LogicalOpMode(logicalOpMode)// Load register
```

4.15.6 Software Writemask Example

Use the following pseudocode 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 = TVP4020_ENABLE
ditherMode.DitherEnable = TVP4020_ENABLE
ditherMode.ColorMode = TVP4020_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)// Load register
logicalOpMode.UnitEnable = TVP4020_ENABLE
logicalOpMode.LogicalOp = TVP4020_LOGICOP_COPY
LogicalOpMode(logicalOpMode)// Load register
FBSoftwareWriteMask(0xFFFFFFE3)
```

4.16 Host Out Unit

The Host Out unit controls which registers are available at the output FIFO, gathers statistics about rendering operations (picking and extent testing), and controls synchronization of the TVP4020 with the host.

4.16.1 Filtering

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

❑ Depth, Stencil, Color: These are data values associated with a fragment that is read from the localbuffer or framebuffer, or generated using the UpLoadData flag in the Framebuffer Write unit. This category is usually associated with uploading data to the host.

❑ Synchronization: This is a single register, Sync, which synchronizes the TVP4020 and flushes the graphics hyperpipeline.

❑ Statistics: These are registers associated with extent checking and picking.

The filtering is controlled by the FilterMode register, which has two-bit fields for each category. These fields select whether the register tag and/or register data are passed to the output FIFO. The format of the FilterMode register is shown in Table 4−21.

Table 4–21. Filter Modes

Register Category	Tag Control Bit	Data Control Bit	Description
Reserved	0	1	Not used
Reserved	2	3	Not used
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	This ensures that the TVP4020 completed all outstanding actions.
Statistics	12	13	This is the data generated following a command to read back the results of the statistic measurements: PickResult, MaxHitRegion, MinHitRegion.
Reserved	14	15	Not used

Note: The filter unit must be set appropriately before any synchronization can take place.

4.16.2 Statistic Operations

There are two statistic collection modes of operation: picking and extent checking. Picking selects drawn objects or regions of the screen. Extent checking determines the bounds of each drawing so that a smaller area of the framebuffer can subsequently be cleared.

Statistic collection is controlled using the StatisticMode register.

❑ Picking

In picking mode, the active and/or passive fragments have their associated XY coordinates compared against the coordinates specified in the MinRegion and MaxRegion registers. If the result is true, then the PickResult flag is set; otherwise, it holds its previous state. The compare function can be either inside or outside the defined rectangular regions. 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} 3 $X < X_{max}$

Y_{min} 3 $Y < Y_{max}$

where:

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

The following steps are required for picking:

1) Load the ResetPickResult, MinRegion, and MaxRegion registers.
2) Set up FilterMode to allow statistic commands out of the TVP4020.
3) Draw the primitives.
4) Send a PickResult command.
5) Poll the output FIFO waiting for the PickResult to pass through the TVP4020.

The picking operation ignores block fills.

❑ Extent Checking

In extent mode, the XY coordinates of active and/or passive fragments are compared to the MinRegion and MaxRegion registers. If found to be outside the defined rectangular region, the TVP4020 updates the applicable register 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 you set the StatisticMode register to include spans.

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

X_{min} 3 $X < X_{max}$
Y_{min} 3 $Y < Y_{max}$

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 are rendered, the results can be found using the MinHitRegion and MaxHitRegion commands, which write the contents of the MinRegion and MaxRegion registers, respectively, into the output FIFO (under the control of the FilterMode register).

4.16.3 Synchronization

The Sync command register ensures that the TVP4020 completes all outstanding actions, such as localbuffer and framebuffer accesses. Sync is filtered and written to the output FIFO in a manner similar to that for the other registers. The host can either poll for Syncs by reading the output FIFO or await a Sync interrupt.

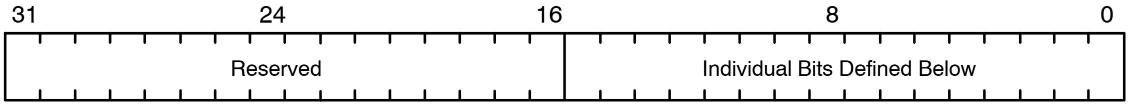
For the TVP4020 to send an interrupt to the host, you must set the most significant bit of the Sync command register and, at least, set up the proper filter to allow the Sync to write to the FIFO. If you set up the FilterMode so that the Sync does not write to the FIFO, then Sync interrupts are not generated. The actual interrupt does not occur until the Sync data or tag passes through the TVP4020 and is on the output of the FIFO. This allows low-level resynchronization between the graphics core and PCI clock domains. The FIFO has an extra bit, in width, to accommodate the interrupt signal. When both the data and tag write to the FIFO, only the first entry in the FIFO causes the interrupt (assuming an interrupt was requested).

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

4.16.4 Registers

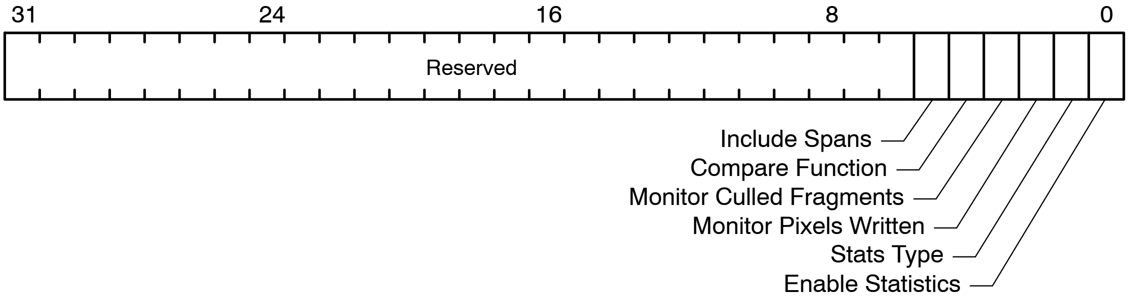
The FilterMode register controls filtering, as shown in Figure 4−53.

Figure 4−53. FilterMode Register



The StatisticMode register controls statistic collection, as shown in Figure 4−54.

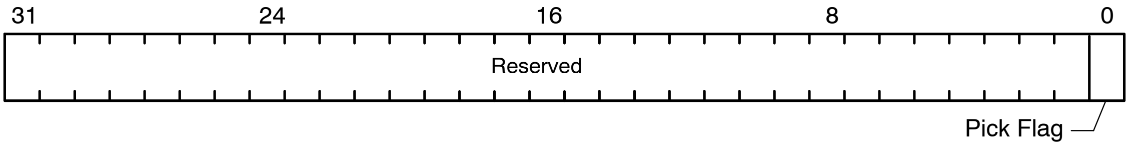
Figure 4−54. StatisticMode Register



The Include-Spans bit controls whether or not block fills are included in the returned information.

The ResetPickResult register clears the pick flag, as shown in Figure 4−55. The data field for this register is unused.

Figure 4−55. ResetPickResult Register



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

Setting the most significant bit of the Sync register requests a Sync interrupt. Bits 0−30 are available for the user.

4.16.5 Filter Mode Example

The following pseudocode implements an example filter mode:

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

4.16.6 Picking Example

The following pseudocode sets the statistic mode to picking, detects any active fragments in the region 0x0 ≤ x < 0x100, 0x0 ≤ y < 0x100, and renders some primitives, then reading 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)
```

4.16.7 Sync Interrupt Example

The following pseudocode generates a synchronization interrupt and encodes user defined data (0×34) 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.
```

Chapter 5

Initialization

This chapter outlines the requirements for initializing the TVP4020.

Topic **Page**

5.1 Initializing the TVP4020

This section describes how to initialize the TVP4020 following a reset and prior to carrying out rendering operations.

There are three areas of initialization; however, in different systems, responsibilities may vary:

- ❏ System initialization covers the setting up of the PCI bus, memory, and video output. Initialize only once following a reset.
- ❏ Window initialization, also referred to as context initialization, covers the setting of the base address and color format of the current rendering window. Initialize at reset and update each time the TVP4020 starts to draw to a new window.
- ❏ Application initialization covers those states that are typically dynamic; enabling and disabling depth testing are examples. Initialize at reset and update frequently, as applicable.

To make use of the full functionality of the TVP4020, consult the relevant sections in Chapter 4. Examples based on pseudocode conventions are in Appendix A, *Pseudocode Definitions*.

In general, the graphics registers (listed in Appendix B, *Screen Widths Table*, as opposed to those documented in the *TVP4020 3D Graphics Processor Data Manual*) are not hardware initialized to specific values at reset. In the examples that follow, the data structures that load these registers are initialized to zero. Thus, bit fields that are not set explicitly default to zero.

5.2 System Initialization

To initialize the system, you must set and configure system components as described in the sections that follow.

5.2.1 PCI

There is a set of PCI related registers that can be interrogated for information about the chip, for example, its revision and device ID numbers. You must set some of these PCI related registers at reset, for instance, to configure the base addresses of the different memory regions of the chip. For more details, refer to the *TVP4020 3D Graphics Processor Data Manual* and the *PCI Local Bus Specification*, Revision 2.1.

5.2.2 Memory Configuration

Program the memory interface control registers to reflect the type and amount of memory fitted. The registers are specified in the *TVP4020 3D Graphics Processor Data Manual*.

5.2.3 SVGA and Internal Video Timing Registers

Details for programming the SVGA registers are in the *TVP4020 3D Graphics Processor Data Manual.*

Program the core video timing generator (VTG) to reflect the timings of the monitor, the screen resolution, and the color depth. Because there is also a SVGA VTG, you must ensure that the correct video timing is enabled at the right time. To change from SVGA to core display mode, two stages are required. First, you must set the core VTG and then load VGAControlReg (the EnableVGADisplay bit set to 0).

Details for programming both VTR registers are in the *TVP4020 3D Graphics Processor Data Manual*.

5.2.4 Screen Width

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

To initialize the screen to be 1024 pixels wide, set the registers as follows:

```
fbReadMode.PP0 = 5
fbReadMode.PP1 = 5
fbReadMode.PP2 = 0
FBReadMode(fbReadMode)
lbReadMode.PP0 = 5
lbReadMode.PP1 = 5
lbReadMode.PP2 = 0
LBReadMode(lbReadMode)
textureMapFormat.PP0 = 5
textureMapFormat.PP1 = 5
textureMapFormat.PP2 = 0
TextureMapFormat(textureMapFormat)
```

The TVP4020 supports a maximum screen resolution of 2048 pixels in width and 2048 pixels in height, although the actual obtainable screen resolution is limited by the RAMDAC. In the case of the integrated RAMDAC, the resolution is 1600 x 1280 pixels, at a screen refresh rate of 85 Hz.

5.2.5 Screen Clipping Region

Set the TVP4020 screen scissor clip at system initialization, and disable the user scissor clip. Assuming that you set the FBWindowBase and LBWindowBase registers, then setting the screen clip prevents writing outside the framebuffer memory (and localbuffer), which could have undesirable results. The following example is appropriate for a pixel resolution of 1024×768:

```
screenSize.X = 1024
screenSize.Y = 768
ScreenSize(ScreenSize)
scissorMode.ScreenScissorEnable =   TVP4020_ENABLE
scissorMode.UserScissorEnable =     TVP4020_DISABLE
ScissorMode(ScissorMode)
```

5.2.6 Localbuffer and Framebuffer Configuration

The TVP4020 supports a unified memory architecture, so you must decide how to partition the memory between framebuffer, localbuffer, and texture. A typical configuration might be to allocate two screen-sized buffers: one for the visible screen and the other for the 3-D back buffer. Then allocate a localbuffer, which is always 16 bits per pixel, and allow the remainder to be used for texture memory. The localbuffer and texture memory have different front and back buffer shapes. For example, suppose that a screen resolution of 800×600 pixels at eight bits per pixel is required. Then the following offset values can be used. Each offset value is a pixel count from start of memory.

```
Front buffer: pixel offset 0

Back buffer: pixel offset 480000 (= 600*800 bytes)

Local buffer: pixel offset 480000 (offset in 16 bit pix-
els)

Texture memory: byte offset 1920000 (= 2*600*800 +
600*800*sizeof (USHORT))
```

The size of the pixel depends on the buffer being considered. Thus, the offset value to the back buffer and the localbuffer appears to be the same but the shape is different because one value is measured in bytes and the other in shorts.

Save these offset values as software copies to use as required. For example, to select the front buffer for rendering, set the FBPixelOffset register to 0; to select the back buffer, set to the back buffer pixel offset value. Add the localbuffer offset value to the window base offset value whenever you update the LBWindowBase register. The value that you load into the TextureBaseAddress register is a count of the number of texels from the start of memory. Thus, modify the byte offset value to a texel count when used. In practice, you will need some sort of texture allocation scheme where you allocate textures starting at the texture memory offset value. The final value that you load into the TextureBaseAddress register is the texture memory offset value, plus the offset value to the required texture, with the final value converted to a texel count from start of memory.

The TVP4020 supports a range of localbuffer configurations. During initialization, set the fields in the LBWriteFormat and LBReadFormat registers to the appropriate values. For example:

```
lbReadFormat.DepthWidth = 3       // 15 bit depth buffer

lbReadFormat.StencilWidth  = 3    // 1 bit stencil

LBReadFormat(lbReadFormat)

lbWriteFormat.DepthWidth = 3      // 15 bit depth buffer

lbWriteFormat.StencilWidth  = 3   // 1 bit stencil

LBWriteMode(lbWriteFormat)
```

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

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

```
fbReadMode.DataType =   TVP4020_FBDATA
FBReadMode(fbReadMode)
lbReadMode.DataType =   TVP4020_LBDEFAULT
LBReadMode(lbReadMode)
```

The following registers are typically only needed for specialized operations. Usually their offset values are zero.

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

5.2.7 Host Out Unit

Under some circumstances, you must use the Sync command to synchronize the host with the TVP4020. Initialize the Host-Out FIFO to output the Sync tag and data (they can be filtered out).

In addition, set the Host Out unit to filter out all other output data. Otherwise, the host software must regularly poll the output FIFO to keep it drained and prevent it from freezing the graphics hyperpipeline. For example:

```
filterMode.Depth =              TVP4020_NULL
filterMode.Stencil =            TVP4020_NULL
filterMode.Color =              TVP4020_NULL
FilterMode.Synchronization =    TVP4020_FILTER_TAG_AND_DATA
                                // Allow Syncs through
filterMode.Statistics =         TVP4020_NULL
filterMode.Remainder =          TVP4020_NULL
FilterMode(filterMode)
```

5.2.8 Disabling Specialized Modes

Disables some operations until they are needed. See Chapter 4, *Graphics Programming*, for more details on operations.

```
window.LBUpdateSource = TVP4020_TRUE
window.ForceLBUpdate =  TVP4020_FALSE
window.DisableLBUpdate = TVP4020_TRUE
Window(window)
```

5.3 Window Initialization

The TVP4020 supports the concept of a window origin and makes it relatively simple to implement systems that allow different color formats to coexist in different windows.

5.3.1 Color Format

Initialize the Color Format unit and the Alpha Blend mode in the Texture/Fog/Blend unit to an appropriate color format at reset. The units support a variety of different formats, as listed in Table 3–1.

For example, to render in 3:3:2, 8-bit color format, use the following pseudocode:

```
ditherMode.ColorFormat = TVP4020_COLOR_FOR-
MAT_RGB_332_FRONT

DitherMode(ditherMode)

alphaBlendMode.ColorFormat = TVP4020_COLOR_FOR-
MAT_RGB_332_FRONT

AlphaBlendMode(alphaBlendMode)
```

To enable dithering use the following:

```
ditherMode.XOffset =          0

ditherMode.YOffset =          0

ditherMode.DitherEnable =     TVP4020_ENABLE

ditherMode.UnitEnable =       TVP4020_ENABLE

DitherMode(ditherMode)
```

The Color Format unit is usually enabled by the user even if dithering itself is not. This is because the unit handles color formatting as well as the dithering operation.

5.3.2 Setting the Window Address and Origin

The TVP4020 supports the concept of a current window origin. The origin of the window can be specified as either the top-left or bottom-left corner. This allows you to pick the most appropriate coordinate system: bottom left for 3-D graphics and top left for window systems. Thus, for OpenGL™ 3-D graphics, set the following:

```
fbReadMode.WindowOrigin =  TVP4020_BOTTOM_LEFT_WINDOW_ORI-
GIN

FBReadMode(fbReadMode)
```

```
lbReadMode.WindowOrigin =  TVP4020_BOTTOM_LEFT_WINDOW_ORIGIN
LBReadMode(lbReadMode)
textureMapFormat.WindowOrigin = TVP4020_BOTTOM_LEFT_WINDOW_ORIGIN
TextureMapFormat(textureMapFormat)
```

The window origin is set in the Scissor unit. This information is usually provided by the window system and requires 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 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. This 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 window pixel is as follows:

```
fbWindowBase = fbBaseAddress +
               (fbWidth * (fbHeight−1−600) + 200)
FBWindowBase(fbWindowBase)
lbWindowBase = lbBaseAddress +
               (lbWidth * (lbHeight−1−600) + 200)
LBWindowBase(lbWindowBase)
```

where:

fbBaseAddress, fbWidth, and fbHeight are the physical base address, width, and height of the framebuffer (in pixels). fbBaseAddress and lbBaseAddress are precomputed as described previously in Section 5.2.6. As for the WindowOrigin data, if the window moves, these registers must be updated.

5.3.3 Writemasks

Typically, hardware (if present) and software writemasks are initially set to make all bitplanes writeable:

```
FBSoftwareWriteMask(TVP4020_ALL_WRITEMASKS_SET)
FBHardwareWriteMask(TVP4020_ALL_WRITEMASKS_SET)
```

5.3.4 Enabling Writing

Which buffers are enabled, at any given time, is window specific and should be considered for performance reasons. Performance improves if you disable unnecessary read and write modes to and from the buffers. For example, if the current rendering does not use depth or stencil testing, then you may disable the read and write modes to the localbuffer. The following example defines how to initialize the buffers to allow depth buffering and alpha blending:

```
fbWriteMode.UnitEnable =           TVP4020_ENABLE
FBWriteMode(fbWriteMode)
lbWriteMode.UnitEnable =           TVP4020_ENABLE
LBWriteMode(lbWriteMode)
lbReadMode.ReadSourceEnable =      TVP4020_DISABLE
lbReadMode.ReadDestinationEnable = TVP4020_ENABLE
LBReadMode(lbReadMode)
fbReadMode.ReadSourceEnable =      TVP4020_DISABLE
fbReadMode.ReadDestinationEnable = TVP4020_ENABLE
FBReadMode(fbReadMode)
```

To use software writemasking, you must set the FBReadMode register ReadDestinationEnable field if the writemask is set to anything other than all ones.

5.3.5 Setting Pixel Size

Set the size of the pixels so that the memory can be accessed correctly. To do this, for example, use the FBReadPixel register as follows:

```
fbReadPixel.PixelSize = TVP4020_16_BIT_PIXEL
FBReadPixel(fbReadPixel)
```

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

5.4 Application Initialization

While an application is running, it may dynamically use features of the TVP4020 such as depth buffering, alpha blending, logical operations, etc. Initially, however, it is recommended that you disable the respective units to ensure that they are in a known state:

```
areaStippleMode.UnitEnable =    TVP4020_DISABLE
AreaStippleMode(areaStippleMode)
depthMode.UnitEnable =          TVP4020_DISABLE
DepthMode(depthMode)
stencilMode.UnitEnable =        TVP4020_DISABLE
StencilMode(stencilMode)
textureAddressMode.UnitEnable = TVP4020_DISABLE
TextureAddressMode(textureAddressMode)
textureReadMode.UnitEnable =    TVP4020_DISABLE
TextureReadMode(textureReadMode)
texelLUTMode.UnitEnable =       TVP4020_DISABLE
TexelLUTMode(texelLUTMode)
yuvMode.UnitEnable =            TVP4020_DISABLE
YUVMode(yuvMode)
colorDDAMode.UnitEnable =       TVP4020_DISABLE
ColorDDAMode(colorDDAMode)
textureColorMode.UnitEnable =   TVP4020_DISABLE
TextureColorMode(textureColorMode)
fogMode.UnitEnable =            TVP4020_DISABLE
FogMode(fogMode)
alphaBlendMode.UnitEnable =     TVP4020_DISABLE
AlphaBlendMode(alphaBlendMode)
logicalOpMode.UnitEnable =      TVP4020_DISABLE
LogicalOpMode(logicalOpMode)
statisticMode.EnableStats =     TVP4020_DISABLE
StatisticMode(statisticMode)
```

5.5 Bypass Initialization

The TVP4020 bypass mechanism gives direct access to memory that the TVP4020 uses to hold the framebuffer, localbuffer, and textures. In some situations, it is useful for an application to have direct access to this memory without going through the graphics processor. By initializing PCI registers, in particular, the BypassWriteMask register, you initialize the bypass mechanism.

The writemask register BypassWriteMask is undefined at boot time and you must set it to −1.

See the *TVP4020 3D Graphics Processor Data Manual* for further details.

Chapter 6

Programming T ips

This chapter describes techniques that make best use of the TVP4020. The topics covered are not, by any means, exhaustive.

Topic **Page**

6.1 PCI Bus Issues

The following sections address PCI bus issues, problems, and solutions.

6.1.1 Improving PCI Bus Bandwidth for Programmed I/O and DMA

The simplest way to program the TVP4020 is by writing data values into the memory-mapped registers, that is, programmed input/output. This is appropriate for primitives that require few setup parameters, such as 2-D 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 advantageous to write directly to the TVP4020 FIFO input. This mechanism makes it possible to use DMA burst transfers. The disadvantage of this method is that you must write both the address of the register and the data value to be loaded to the TVP4020, doubling the amount of data to load.

However, to improve bus bandwidth use, the registers are grouped into blocks that frequently need to be updated together. The TVP4020 register set (see Chapter 8, *Register Tables*) supports an indexed addressing mode that allows a single address to load, followed by the data for a whole set of registers.

The TVP4020 register set supports an additional mode that allows a large number of data values to load to the same register. This is useful for image downloads.

For more details, refer to Section 2.3, *TVP4020 I/O Interfaces*.

6.1.2 PCI Burst Transfers Under Programmed I/O

PCI bus-burst transfers typically allow up to four times the bandwidth of individual transfers. However, burst transfers are only initiated on the PCI bus when you are writing to successive addresses (that is, the byte address is incremented by four). When using burst transfers to perform the programmed I/O that loads the TVP4020 FIFOs, the TVP4020 multiply-maps the FIFO input register throughout the range: 0×00002000 to 0×00002FFF in region 0.

Thus, when data is being loaded into the FIFO, use a software loop that writes the first data item at the lower extreme of this address range, which then works toward the upper range. For additional information, see Section 2.3, *TVP4020 I/O Interfaces*.

6.1.3 Using PCI Disconnect Under Programmed I/O

The PCI bus protocol incorporates a feature known as PCI disconnect, a mode which is supported by the TVP4020. Once the TVP4020 is in this mode, if the host processor attempts to write to the full FIFO, the TVP4020 chip triggers the PCI disconnect mode instead of risking the loss of the write operation. In turn, the host processor retries the write operation until the cycle succeeds.

This PCI disconnect feature allows faster download of data to the TVP4020 because the host need not poll the InFIFOSpace register. Be cautious when using the PCI disconnect mode because when it is enabled, the bus is saturated by the host processor until the TVP4020 frees up an entry in its FIFO.

6.1.4 Using Bus Mastership (DMA)

Most TVP4020 boards support PCI bus mastership. This allows the on-board DMA to copy data from host memory into the TVP4020 FIFO.

The PCI bus mastership has a number of benefits:

❑ PCI bus bandwidth generally improves.

❑ PCI bus bandwidth further improves because the driver software no longer needs to poll the FIFO flags before loading to find how many entries are empty.

❑ Overall system performance benefits through increased parallelism between the TVP4020 and the host because the host prepares the next DMA buffer once it initiates a DMA transfer.

See subsection 2.3.4, *The DMA Interface*, for more details on using DMA.

6.1.5 Improving Performance With DMA

Using DMA interrupts improves performance by freeing up system CPU time that would otherwise be used for polling. Using multiple DMA buffers is typically more advantageous.

The size and number of DMA buffers to use is dependent on operating system issues, such as context switch time.

6.1.6 Improving Texture Mapping Performance

Interrupts can significantly improve the performance of texture mapping operations by downloading textures on demand. During a texture mapping operation, if the required texture map does not exist in local memory, an interrupt is generated so that it can be downloaded. See subsections 4.9.2 to 4.9.6 for further details.

6.1.7 AGP Support

The advanced graphics port extensions to the PCI protocol are supported by the TVP4020. When in an AGP slot, the TVP4020 functions as a 66-MHz PCI device, and also performs single-edge AGP read-master transfers with optional sideband addressing.

6.2 Graphics Hyperpipeline

The graphics hyperpipeline performs the rasterization and framebuffer operations. Section 4.1, *The Graphics Hyperpipeline*, describes the functions supported by this unit.

6.2.1 Disable Unused Units

To maximize pixel throughput in the graphics core, disable any unit that will not be used. Also, make certain that data is not being read from the texture buffer, localbuffer or framebuffer, unless it is needed.

For example, it is possible to set up the Localbuffer Read unit such that the TVP4020 reads per-pixel information, such as Z or stencil-buffer data, which is then discarded. The results are visually the same, but the cost of accessing unnecessary memory is high.

Additionally, disable the LBDisableUpdate bit in the window register if you do not need to use localbuffer write operations.

For optimal performance, use hardware writemasks instead of software writemasks.

6.2.2 Avoid Unnecessary Register Updates

The TVP4020 control registers maintain their state between primitives. Therefore, they do not need updating unless the data changes. For example, the dY register might be set to +1 for a trapezoid; thus, reloading is unnecessary until a line primitive is drawn.

All delta values and start values are maintained across primitives, so if two triangles share a dominant edge, the start and dominant edge values do not need calculating or loading twice.

Similarly, the window clipping function need not reload all the registers for each clip rectangle. For example: you can load those registers ready for a primitive to be drawn, and then enter a loop that repeatedly loads the coordinates for a clip rectangle into the Scissor unit, which then sends the Render command. You can process any number of clip rectangles in this way, but the TVP4020 requires only one setup procedure for each primitive.

6.2.3 Loading Registers in Unit Order

To maximize performance, load the control registers for the next primitive into the TVP4020 FIFO in unit order; that is, load the registers associated with the

Rasterizer unit first, then the Scissor, Stipple, Localbuffer Read and so on, until the last unit to load is the Host Out unit (if necessary). Then, finally, load the relevant command register.

For the order of the units in the hyperpipeline, see Figure 4−1.

6.2.4 Use of Continue Commands

The continue commands provide an efficient method for drawing complex primitives without decomposing them into trapezoids or single lines, but they have some requirements.

For context switching, treat each primitive as atomic, that is, as if it cannot be broken into a smaller operation. For example, if the TVP4020 context switch occurs after the Render command is written for a triangle but before its associated ContinueNewDom command is written, then the second part of the primitive may draw incorrectly. This is because the TVP4020 relies on the internal state that is set up by the Render command, which becomes corrupted by any intervening context.

Additionally, data written by the TVP4020 to the framebuffer or localbuffer before the continue command is written by the host should not be read afterward by the host. This can occur when two lines are drawn, the second joining the end of the first, which are started by the ContinueNewLine command. If these lines are XORed, they read the pixel to which they are about to write. If the second line is at a sharp angle so that it folds back and overwrites some or all of the first line, the XOR operation is incorrect because the pixels from the first line were not written to memory before the second line read them.

If you believe this situation is likely to occur, you must send a Sync command before the ContinueNewLine command. This ensures that all necessary write operations finish before the corresponding read operations. The software does not have to wait for the Sync to be read from the output FIFO; sending Sync is enough to ensure correct operation.

6.3 Area Filling Techniques

This section describes the various techniques to fill an area.

6.3.1 Clearing Buffers Quickly

A block write operation is a feature of SGRAMs (synchronous graphic RAMS). Data written from the TVP4020 to a single address can be applied to several addresses at the same time. This is a very fast way of filling areas of the screen, but there are restrictions on when block writing can be used. These restrictions are covered elsewhere in this manual.

Block writing is most useful for clearing the screen, but because the TVP4020 has a unified memory buffer it is also possible to clear the localbuffer.

The extent checking feature in the Host Out unit can be used to indicate the area of the screen that was written to, thus reducing the number of screen-clear operations.

6.3.2 Avoid Clearing Buffers

Although a block write operation can quickly clear buffers, it is not recommended. If all pixels on the screen are drawn at least once per frame, the framebuffer does not need clearing.

The localbuffer does not need clearing either if you use the following procedure:

❑ For even frames, put the viewer at a depth position of zero and draw objects in the lower half of the depth range with the depth test set to *less than*.

❑ For odd frames, put the viewer at the maximum depth value and draw objects into the upper half of the depth range with the depth test set to *greater than.*

This procedure results in a loss to half of the depth range, but avoids the need to clear the depth buffer as long as every pixel is accessed at least once.

6.3.3 Trapezoid Fills

A block write operation is most useful when clearing the framebuffer, but it can be used to fill any trapezoid.

A block fill operation, however, is limited to the area defined by the rasterizer and cannot be changed by the stipple test. You can, however, use a quick filling technique for these tests by setting the UseConstantFBWriteData bit in the Logic Op unit. After you set this bit, you must load the required color into the FBWriteData register in the format needed by the memory. Then, disable all unrequired units and start the rasterizer. The filling occurs up to twice as quickly using this method, as opposed to using the ConstantColor register method.

Even though the display may be eight bits per pixel, the chip can be programmed to draw at 32 bits per pixel. This plots four pixels at one time, reducing the width of the region that the rasterizer covers by a factor of four. Use the technique described in Section 6.4, *Copies and Downloads*, for packed copies, to get the Framebuffer Write unit to calculate addresses correctly for 32-bit pixels. You can also use the PackedDataLimits register to mask out unwanted pixels on the left and right edges.

6.4 Copies and Downloads

Copying and downloading of the framebuffer or localbuffer are explained in the following paragraphs.

6.4.1 Copies

If the pixel size is 8 or 16 bits per pixel, you can improve the copy speed by moving more than one pixel at a time. To do this, set the PackedCopy bit in the Framebuffer Read unit. This bit enables a 32-bit pixel size and calculates the addresses accordingly. The screen width does not need to change, nor does the base address or source offset value. Program the rasterizer to rasterize a rectangle that is a factor-of-four narrower (8-bit pixels) or a factor-of-two narrower (16-bit pixels) than the normal size.

The groups of four or two copied pixels align to a 32-bit boundary. However, if some of the edge pixels are not needed, the PackedDataLimits register masks them out. If the source and destination pixels have a different alignment, use the RelativeOffset field in the FBReadMode register to specify how to shift the source to line up with the destination.

6.4.2 Downloads

You can also use the same registers described in the previous section to pack data during a download to the framebuffer or localbuffer. If you set the rasterizer to synchronize on FBData, the data sent to the TVP4020 by the host must be in the raw memory format. Four 8-bit pixels can be written by the host at one time to the chip, and you can set the PackedDataLimits register to mask any unwanted pixels at the left and right edges. Use the RelativeOffset field to shift the alignment of the data as it is being stored.

You can use LBData to download to the localbuffer. However, the rasterizer does not support synchronization with LBData, so the data must be explicitly synchronized using the Sync command. Alternatively, you can download stencil and/or depth data through the Framebuffer Write unit, using the WaitForCompletion command or synchronizing with FBData.

6.4.3 Loading Textures

The TVP4020 handles internal synchronization so that, for a given buffer, all necessary write operations finish before read operations. If the same data is treated as two different types, you must explicitly synchronize the chip. When you download a texture, it is written to memory through the Framebuffer Write unit but is read through the Texture Read unit. This means that you must synchronize the chip between loading the texture and reading it; otherwise, you are not guaranteed that the write cyle will finish before the read cycle begins. The Sync command or a WaitForCompletion command that does not require the polling of the output FIFO can do this.

Similarly, if you use the Framebuffer Write unit to clear the localbuffer or use the Texture Read unit in a copy operation, you must synchronize the chip. The chip will synchronize itself between the localbuffer read and write operation, and between the framebuffer read and write operation, but you must synchronize any operations that mix the buffers.

If you download a texture as a normal image, the chip can change the color format and reorganize the data into rectangular patches. If the texture is already in the required format, you can use a fast texture download. To do this, set the TextureDownloadOffset register to point to the start address of the texture (in 32-bit words). Then write the 32-bit texture data to the TextureData register, which will write to memory without changing the format. The TextureDownloadOffset will automatically increment following each write operation. If the texture is eight bits per texel, then supply four texels at a time. This method of texture download avoids the need to set up the rasterizer for an image download and allows the state of the chip to remain unchanged. Even the framebuffer write operations do not have to be enabled.

6.5 Multibuffering

The TVP4020 supports double and triple buffering operations, as described in the sections below.

6.5.1 Fast Double Buffering

TVP4020 board designs support a variety of double-buffering mechanisms that depend on the memory configuration and LUT-DAC (look-up table – digital-to-analog converter) being used, including:

❑ BLT (BITBLT)
❑ Full screen
❑ Bitplane

For further details see Section 3.4, *Double Buffering*, Section 4.13, *Texture/Fog/Blend*, and Section 4.14, *Color Format Unit*.

For optimal functionality, you can mix two or more of the above double-buffering techniques.

Send nonframebuffer-related commands to the TVP4020 following a SuspendUntilFrameBlank command. For example, following this command, send any commands that will clear the depth buffer between frames because these will not affect the framebuffer and will execute while the TVP4020 waits for the VBLANK (vertical blank). This allows better overlap between the host and the TVP4020. In general, any commands that do not render the framebuffer can be queued in the TVP4020 FIFO before waiting on the VBLANK.

6.5.2 Triple Buffering

Most 3-D systems support double buffering, where one frame is displayed while the next frame is being drawn. To avoid display artifacts, the transition between old and new buffers must occur during a vertical frame blank. This, however, imposes a granularity on the frame rate. If a scene takes slightly longer than one frame period to draw, the scene must wait for another frame before it can display, so the frame rate halves.

Using three buffers removes the quantization so the system can continue to draw at maximum rate.

6.6 Overlays

Overlays are only available with the 5:5:5:1 color format in a 32-bit pixel. The 5:5:5:1 color format copies the data into both 16-bit halves of the 32-bit pixel. The writemask writes either the upper or lower half to memory.

You can program the RAMDAC to display a 16-bit pixel from either the upper or lower half of the 32-bit word; bit 31 sets the pixel to display. Bit 31 corresponds to the alpha bit of the 16-bit pixel and you can force it to either a one or zero by the Color Format unit.

When drawing to the underlay (or main image), set the Color Format unit to force the alpha to zero, and set the writemask to enable write operations to the lower half of the word. When drawing to the overlay, set the Color Format unit to force the alpha value to 1, and set the writemask to enable write operations to the upper half of the word.

If you set the RAMDAC to the appropriate mode, the TVP4020 draws the pixels in the overlay half of the word where the alpha bit is a one while it draws pixels from the main overlay image where alpha is zero.

6.7 Memory Organization

The amount of memory available to the TVP4020 depends on the board to which it is fitted. The most efficient way to allocate memory depends on the needs of the system. In general, allocate the display at one end of the SGRAM and the localbuffer at the other end. This leaves a region between the two buffers in which textures can be stored. For optimal performance, store each buffer (front color, back color, texture, and depth) in separate memory banks. Memory is organized as shown in Table 6–1.

Table 6–1. Memory Organization

Memory Size	Banks	Size Per Bank
2 MB	2	1 MB
4 MB	4	1 MB
6 MB	4	1 or 2 MB
8 MB	4	2 MB

With 6 MB of memory, the first two banks will contain 1 MB and the following two, 2 MB.

6.8 Chroma Test

Chroma key testing can be performed without texture mapping. Set the TexelDisableUpdate field in the YUVMode register. The texels are read in and tested, and fragments are rejected, as part of the copy operation, if the colors do not match. Setting the TexelDisableUpdate bit discards the data as soon as the test finishes, which improves performance.

This is described in more detail in subsection 4.10.1, *Chroma Test*.

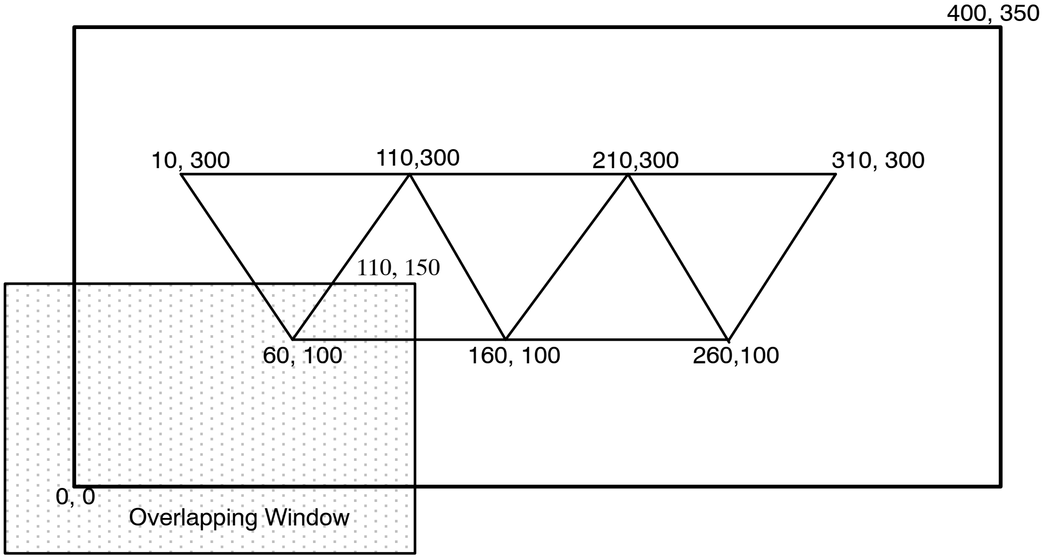
6.9 Configuration for 2-D

You can set specific fields of several registers by writing to a single register, Config. This register groups fields of those registers commonly used in 2-D operations allowing you to configure the TVP4020 with fewer accesses. Reading from this register, however, returns invalid data.

6.10 Delta Programming Examples

This section demonstrates how to render a depth-buffered, Gouraud-shaded triangle mesh using the Delta unit. The window into which the rendering takes place is partially obscured and, thus, clipped by two clip rectangles as shown in Figure 6−1.

Figure 6−1. Geometry of the Mesh and Clip Regions



The three sections that follow cover drawing the mesh as a set of points at the vertices, as connected line segments, and finally as filled triangles. For simplicity, the triangles in these examples are either flat topped or flat bottomed. In practice, triangles are not restricted to these shapes and can have any orientation, size, or shape.

6.10.1 Header File

```
// This is the header file for the Delta Unit example
// code. It only contains the necessary items to support
// the examples.
#ifdef BIG_ENDIAN
// The DeltaMode register fields.
typedef struct {
   unsigned int pad:                          14;
   unsigned int ColorOrder:                   1;
   unsigned int BackfaceCallEnable:           1;
   unsigned int TextureParameterMode:          2;
   unsigned int ClampEnable:                   1;
   unsigned int NoDraw:                        1;
   unsigned int DiamondExit:                   1;
   unsigned int SubPixelCorrectionEnable:      1;
   unsigned int DiffuseTextureEnable:          1;
   unsigned int SpecularTextureEnable:         1;
   unsigned int DepthEnable:                   1;
   unsigned int SmoothShadingEnable:           1;
   unsigned int TextureEnable:                 1;
   unsigned int FogEnable:                     1;
   unsigned int Reserved:                      4;
   unsigned int TargetChip:                    2;
} __DeltaModeFmat;
// The DrawTriangle and DrawLine command fields.
typedef struct {
   unsigned int pad:                           14;
   unsigned int ReuseBitMask:                  1;
   unsigned int SubPixelCorrectionEnable:      1;
   unsigned int Reserved:                      1;
   unsigned int FogEnable:                     1;
   unsigned int TextureEnable:                 1;
   unsigned int SyncOnHostData:                1;
   unsigned int SyncOnBitMask:                 1;
   unsigned int Reserved:                      3;
   unsigned int PrimitiveType:                 2;
   unsigned int Reserved:                      2;
   unsigned int FastFillEnable:                1;
   unsigned int Reserved:                      2;
   unsigned int LineStippleEnable:             1;
} __DeltaRenderFmat;
#else
// The DeltaMode register fields.
typedef struct {
   unsigned int Reserved:                      4;
   unsigned int FogEnable:                     1;
   unsigned int TextureEnable:                 1;
   unsigned int SmoothShadingEnable:           1;
   unsigned int DepthEnable:                     1;
   unsigned int SpecularTextureEnable:           1;
   unsigned int DiffuseTextureEnable:            1;
   unsigned int SubPixelCorrectionEnable:        1;
   unsigned int DiamondExit:                     1;
   unsigned int NoDraw:                          1;
   unsigned int ClampEnable:                     1;
   unsigned int TextureParameterMode:            2;
   unsigned int BackfaceCallEnable:              1;
   unsigned int ColorOrder:                      1;
   unsigned int pad:                             14;
} __DeltaModeFmat;
// The DrawTriangle and DrawLine command fields.
typedef struct {
   unsigned int AreaStippleEnable:               1;
   unsigned int ReservedC:                       2;
   unsigned int FastFillEnable:                  1;
   unsigned int reserved:                        2;
   unsigned int PrimitiveType:                   2;
   unsigned int ReservedB:                       1;
   unsigned int SyncOnBitMask:                   1;
   unsigned int SyncOnHostData:                  1;
   unsigned int TextureEnable:                   1;
   unsigned int FogEnable:                       1;
   unsigned int ReservedA:                       1;
   unsigned int SubPixelCorrectionEnable:        1;
   unsigned int pad:                             14;
   unsigned int ReuseBitMask:                    1;
} __DeltaRenderFmat;
#endif
// The tag values for the registers.
   #define __Delta_V0FloatTag                    0x230
   #define __Delta_V1FloatTag                    0x240
   #define __Delta_V2FloatTag                    0x250
   #define __DeltaTagDeltaMode                   0x260
   #define __DeltaTagDrawTriangle                0x261
   #define __DeltaTagRepeatTriangle              0x262
   #define __DeltaTagDrawLine01                  0x263
   #define __DeltaTagDrawLine10                 0x264
   #define __DeltaTagRepeatLine                 0x265
// Some temp defines to keep things compiling easily.
   #define DrawTriangleTag       __DeltaTagDrawTriangle
   #define DrawLine01Tag         __DeltaTagDrawLine01
   #define DrawLine10Tag         __DeltaTagDrawLine10
   #define RepeatTriangleTag     __DeltaTagRepeatTriangle
   #define RepeatLineTag         __DeltaTagRepeatLine
#include "delta.h"
#include <stdio.h>
extern unsigned long *dmaPtr;
extern DMA *dma;
// Change these macros to what is needed to write the val-
ues to Delta // Unit, or add them to a dma buffer.
#define LD_REG(reg, value)  dmaPtr = dma->Space(2);
*dmaPtr++ = reg;\
                                         *dmaPtr++ = value;
#define LD_PARAM(reg, value) dmaPtr = dma->Space(2);
*dmaPtr++ = reg;\
                   *dmaPtr++ = *((unsigned long *) &value);
// Prototypes
void PointMesh (gal &cx);
void LineMesh (gal &cx);
void TriangleMesh (gal &cx);
// Simple structure to use in the example code
typedef struct { float  x, y, z, r, g, b, a; }  Vertex;
typedef struct { short  x, y; }                 XY;
typedef struct { XY scissorMin, scissorMax; }   ClipRec-
tangle;
// Define some test data.
#define verticesInMesh 7
Vertex  mesh[verticesInMesh] = {
    //   x     y     z     r     g     b     a
     {  10,   300,  0.1,  1.0,  1.0,  1.0,   1.0  },
     {  60,   100,  0.2,  1.0,  1.0,  0.0,   1.0  },
     {  110,  300,  0.3,  1.0,  0.0,  1.0,   1.0  },
     {  160,  100,  0.4,  1.0,  0.0,  0.0,   1.0  },
     {  210,   300,   0.5,   0.0,   1.0,   1.0,    1.0   },
     {  260,   100,   0.6,   0.0,   1.0,   0.0,    1.0   },
     {  310,   300,   0.7,   0.0,   0.0,   1.0,    1.0   }};
#define numberClipRectangles 2


ClipRectangle clipRectangles[numberClipRectangles] = {
            { {110, 0}, {400, 150} },
            { {0, 150}, {400, 350} }};
enum {paramS, paramT, paramQ, paramKs, paramKd, paramR,
paramG, paramB, paramA, paramF, paramX, paramY, paramZ};
```

6.10.2 Drawing Vertices in the Mesh as Points

```
// This function draws the vertices in the mesh as points.
// There is no direct support for points in Delta Unit as
// they do not need any set-up calculations. Delta Unit
// can be used to plot points (maybe because you want to
// always work in floating point) by having Delta Unit do
// the set-up calculations for a line, but tell the
// rendering device to render points.


void PointMesh (gal &cx)
{
     __DeltaModeFmat                  deltaMode;
     __DeltaRenderFmat                drawCmd;
     int                              rect, v;
    // Assume the rendering device is already initialized.
// Note we expect the BiasCoords mode in the
// RasterizerMode register to be set to add a bias of
   // zero. Set-up the DeltaMode register.

 deltaMode.pad                     = 0;
 deltamode ColorOrder              = 0;
 deltamode BackfaceCallEnable      = 0;
 deltaMode.TextureParameterMode    = 1;  // Clamp.
 deltaMode.ClampEnable             = 1;  // Clamp enabled.
 deltaMode.NoDraw                  = 0;  // Do drawing.
 deltaMode.DiamondExit             = 0;  // Not needed for
                                         // this example.
 deltaMode.SubPixelCorrectionEn-   = 0;  // No sub pixel
 able                                    // correction.
 deltaMode.DiffuseTextureEnable    = 0;  // Disable.
 deltaMode.SpecularTextureEnable   = 0;  // Disable.
 deltaMode.DepthEnable             = 1;  // Enable.
 deltaMode.SmoothShadingEnable     = 1;  // Enable.
 deltaMode.TextureEnable           = 0;  // Disabled.
 deltaMode.FogEnable               = 1;  // Enabled, but
                                         // controlled
                                         // from the draw
                                         // command.
 deltaMode.Reserved                = 0;
   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));
   // Set-up the draw command data.
 drawCmd.pad                        = 0;
 drawCmd ReuseBitMask               = 0;
 drawCmd.SubPixelCorrectionEnable   = 0;  // Enable.
 drawCmd.ReservedA                  = 0;
 drawCmd.FogEnable                  = 0;  // Disable.
 drawCmd.TextureEnable              = 0;  // Disable.
 drawCmd.SyncOnHostData             = 0;  // Disable.
 drawCmd.SyncOnBitMask              = 0;  // Disable.
 drawCmd.ReservedB                  = 0;
 drawCmd.AntialiasEnable            = 0;  // Disable.
 drawCmd.PrimitiveType              = 2;  // ** Point **
 drawCmd.reserved                   = 0;
 drawCmd.FastFillEnable             = 0;  // Disable.
 drawCmd.ReservedC                  = 0;
 drawCmd.AreaStippleEnable          = 0;  // Disable.

 // We need to ensure that the end vertex of the line
 // (in V1) can never be the same as the point vertices.
 // Any X (or Y) coordinate which is out of the normal
    // range (0.0 to screen width) will do so in this case
 // an X of -1.0 has been used.
```

```
float   tempEndCoord = -1.0;
   LD_PARAM ((__Delta_V1FloatTag + paramX), tempEndCoord);
   for (v = 0; v < verticesInMesh; v++)
   {
   LD_PARAM((__Delta_V0FloatTag + paramR), mesh[v].r);
   LD_PARAM((__Delta_V0FloatTag + paramG), mesh[v].g);
   LD_PARAM((__Delta_V0FloatTag + paramB), mesh[v].b);
   LD_PARAM((__Delta_V0FloatTag + paramA), mesh[v].a);
   LD_PARAM((__Delta_V0FloatTag + paramX), mesh[v].x);
   LD_PARAM((__Delta_V0FloatTag + paramY), mesh[v].y);
   LD_PARAM((__Delta_V0FloatTag + paramZ), mesh[v].z);

   for (rect = 0; rect < numberClipRectangles; rect++)
   {
      // Load in the scissor rectangle.
      LD_REG(ScissorMinXYTag, (clipRectangles[rect].scis-
sorMin.y
      << 16 |  clipRectangles[rect].scissorMin.x));
      LD_REG(ScissorMaxXYTag, (clipRectangles[rect].scis-
sorMax.y
      << 16 |  clipRectangles[rect].scissorMax.x));
      if (rect == 0)
      {
      LD_REG(DrawLine01Tag, *((long *) &drawCmd));
      }
      else
      {
      LD_REG(RepeatLineTag, 0);// data field not used.
      }
     }
    }
}
// This array holds the order we are going to visit the
// vertices in to draw each line segment.
```

```
Lint lineOrder[12] = {1, 0, 2, 4, 6, 5, 4, 3, 2, 1, 3, 5};
```

6.10.3 Drawing the Mesh as a Series of Lines

```
// This function draws the mesh as a series of lines.  The
// order the lines are drawn in is hardcoded (this is only
// an example!).
void LineMesh (gal &cx)
{
    __DeltaModeFmat      deltaMode;
    __DeltaRenderFmat       drawCmd;
    int            vertexStore, rect, i, v;

// Assume the rendering device is already initialized.
// Note we expect the BiasCoords mode in the
// RasterizerMode register to be set to add a bias of
// zero.

   // Set-up the DeltaMode register.
deltaMode.pad                   = 0;
deltamode.ColorOrder            = 0;
deltamode.BackfaceCallEnable    = 0;
deltaMode.TextureParameterMode  = 2;  // Auto normalize.
deltaMode.ClampEnable           = 1;  // Clamp enabled.
deltaMode.NoDraw                = 0;  // Do drawing.
deltaMode.DiamondExit           = 1;  // Not needed for
                                      // this example.
deltaMode.SubPixelCorrectio-    = 1;  // Enable sub pixel
nEnable                               // correction.
deltaMode.DiffuseTextureEnable  = 0;  // Disable.
deltaMode.SpecularTextureEn-    = 0;  // Disable.
able
deltaMode.DepthEnable           = 1;  // Enable.
deltaMode.SmoothShadingEnable   = 1;  // Enable.
deltaMode.TextureEnable         = 0;  // Disabled.
deltaMode.FogEnable             = 1;  // Enabled, but
                                      // controlled from
                                      // the draw command.
deltaMode.Reserved              = 0;
```

```
   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

      // Set-up the draw command data.
drawCmd.pad                         = 0;
DrawCmd.ReuseBitMask                = 0;
drawCmd.SubPixelCorrectionEnable    = 1;   // Enable.
drawCmd.ReservedA                   = 0;
drawCmd.FogEnable                   = 0;   // Disable.
drawCmd.TextureEnable               = 0;   // Disable.
drawCmd.SyncOnHostData              = 0;   // Disable.
drawCmd.SyncOnBitMask               = 0;   // Disable.
drawCmd.ReservedB                   = 0;
drawCmd.AntialiasingQuality         = 0;   // Not used.
drawCmd.AntialiasEnable             = 0;   // Disable.
drawCmd.PrimitiveType               = 0;   // Line.
drawCmd.reserved                    = 0;
drawCmd.FastFillEnable              = 0;   // Disable.
drawCmd.ReservedC                   = 0;
drawCmd.AreaStippleEnable           = 0;   // Disable.

 for (i = 0; i < 12; i++)
    {
      v = lineOrder[i];
      vertexStore = __Delta_V0FloatTag + 16 * (i % 2);
LD_PARAM((vertexStore + paramR), mesh[v].r);
LD_PARAM((vertexStore + paramG), mesh[v].g);
LD_PARAM((vertexStore + paramB), mesh[v].b);
LD_PARAM((vertexStore + paramA), mesh[v].a);
LD_PARAM((vertexStore + paramX), mesh[v].x);
LD_PARAM((vertexStore + paramY), mesh[v].y);
LD_PARAM((vertexStore + paramZ), mesh[v].z);

if (i >= 1)
{
          // We now have enough vertices to draw a line.
                for (rect = 0; rect < numberClipRec-
tangles; rect++)
          {
      // Load in the scissor rectangle.
      LD_REG(ScissorMinXYTag,
      (clipRectangles[rect].scissorMin.y << 16 |
      clipRectangles[rect].scissorMin.x));
      LD_REG(ScissorMaxXYTag,
      (clipRectangles[rect].scissorMax.y << 16 |
      clipRectangles[rect].scissorMax.x));

      if (rect == 0)
      {
   if (i & 1)
{
    LD_REG(DrawLine01Tag, *((long *) &drawCmd));
      }
      else
      {
          LD_REG(DrawLine10Tag, *((long *) &drawCmd));
        }
      }
      else
{
      LD_REG(RepeatLineTag, 0);// data field unused
      }
     }
    }
   }
}
```

6.10.4 Drawing the Mesh as a Series of Shaded Triangles

```
// This function draws the mesh as a series of shaded
// triangles.
void TriangleMesh (gal &cx)
{
       __DeltaModeFmat      deltaMode;
       __DeltaRenderFmat    drawCmd;
       int                  vertexStore;
       int                  rect, v;

    // Assume the rendering device is already initialized.

    // Note we expect the BiasCoords mode in the

    // RasterizerMode register to be set to add a bias of

    // zero.


// Set-up the DeltaMode register.


deltaMode.pad                      = 0;
deltamode.ColorOrder               = 0;
deltamode.BackfaceCallEnable       = 0;
deltaMode.TextureParameterMode     = 2;  // Auto
                                         // normalize.
deltaMode.ClampEnable              = 1;  // Clamp enabled.
deltaMode.NoDraw                   = 0;  // Do drawing.
deltaMode.DiamondExit              = 1;  // Not needed for

                                         // this example.
deltaMode.SubPixelCorrectionEnable = 1;  // Enable

                                         // sub pixel
                                         // correction.
deltaMode.DiffuseTextureEnable     = 0;  // Disable.
deltaMode.SpecularTextureEnable    = 0;  // Disable.
deltaMode.DepthEnable              = 1;  // Enable.
deltaMode.SmoothShadingEnable      = 1;  // Enable.
deltaMode.TextureEnable            = 0;  // Disabled.
deltaMode.FogEnable                = 1;  // Enabled, but
                                         // controlled
                                         // from the draw
                                            // command.
deltaMode.Reserved                 = 0;


     LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));
    // Set-up the draw command data.
drawCmd.pad                            = 0;
drawCmd.ReuseBitMask                   = 0;
drawCmd.SubPixelCorrectionEnable       = 1;  // Enable.
drawCmd.RaservedA                      = 0;
drawCmd.SyncOnBitMask                  = 0;  // Disable.
drawCmd.FogEnable                      = 0;  // Disable.
drawCmd.TextureEnable                  = 0;  // Disable.
drawCmd.SyncOnHostData                 = 0;  // Disable.
drawCmd.ReservedB                      = 0;
drawCmd.AntialiasEnable                = 0;  // Disable.
drawCmd.PrimitiveType                  = 1;  // Trapezoid.
drawCmd.reserved                       = 0;
drawCmd.FastFillEnable                 = 0;  // Disable.
drawCmd.ReservedC                      = 0;
drawCmd.AreaStippleEnable              = 0;  // Disable.


    for (v = 0; v < verticesInMesh; v++)
    {
       vertexStore = __Delta_V0FloatTag + 16 * (v % 3);
LD_PARAM((vertexStore + paramR), mesh[v].r);
LD_PARAM((vertexStore + paramG), mesh[v].g);
LD_PARAM((vertexStore + paramB), mesh[v].b);
LD_PARAM((vertexStore + paramA), mesh[v].a);
LD_PARAM((vertexStore + paramX), mesh[v].x);
LD_PARAM((vertexStore + paramY), mesh[v].y);
LD_PARAM((vertexStore + paramZ), mesh[v].z);
    if (v >= 2)
    {
// We now have enough vertices to draw a triangle.
for (rect = 0; rect < numberClipRectangles; rect++)
       {
// Load in the scissor rectangle.
LD_REG(ScissorMinXYTag,
       (clipRectangles[rect].scissorMin.y << 16 |
       clipRectangles[rect].scissorMin.x));
       LD_REG(ScissorMaxXYTag,
       (clipRectangles[rect].scissorMax.y << 16
       clipRectangles[rect].scissorMax.x));
```

```
      if (rect == 0)
      {
      LD_REG(DrawTriangleTag,  *((long *) &drawCmd));
      }
      else
      {
      LD_REG(RepeatTriangleTag, 0); // data field not
                       // used.
      }
     }
    }
   }
}
```

Chapter 7

Graphics Register Reference

This chapter details each TVP4020 graphics register format. The registers are listed alphabetically, by name.

7.1 Graphics Register Information Types

Each register format includes the following information:

- ❑ Name: the description of the function.
- ❑ Unit: the unit with which the register and function are associated.
- ❑ Tag: the register offset value from the base address of the region.
- ❑ Reset Value: the value of the register following hardware reset. In general this is undefined for graphics registers.
- ❑ Read/write: indicates that the register can be read and written.
- ❑ Write: indicates that the register can only be written. The value of any read from this address is undefined.

Each diagram format includes some or all of the following information:

- ❑ Reserved: indicates bits that may be used for future members of the TVP4020 family. To ensure upward compatibility, make certain that these bits do not assume values during a read operation and that they are always written as zeros.
- ❑ Not Used: indicates bits that are adjacent to numeric fields. These may be used for future members of the TVP4020 family, but only to extend the dynamic range of these fields. The data returned from a read of these bits is undefined. When a Not Used field resides in the most significant position, sign-extend the numeric value before writing to the register, rather than mask the field to zero. This will ensure compatibility if the dynamic range is increased for future members of the TVP4020 family.
- ❑ For enumeration fields that do not specify the full range of possible values, only use the specified values. An example of an enumeration field is the comparison field in the DepthMode register. Future members of the TVP4020 family may define a meaning for the unused values.

7.2 Graphics Registers

This section contains the alphabetical listings of all TVP4020 graphics registers.

AlphaBlendMode

Name: AlphaBlend Mode

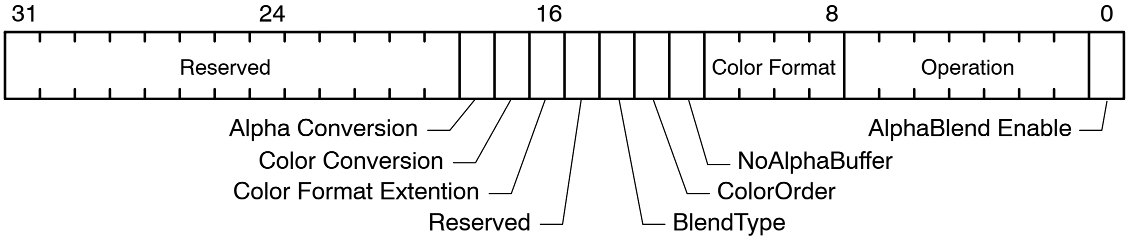
Unit: Texture/Fog/Blend

Region: 0 Offset: 0×0000.8810

Tag: 0×0102

Reset Value: Undefined

Read/write



This register controls alpha blending.

Bit 0 AlphaBend Enable:
0 = Disable
1 = Enable alpha blending or color formatting

Bits 1 – 7 Operation

Bit 17 Color Conversion:
0 = Scale
1 = Shift

Bit 18 Alpha Conversion:
0 = Scale
1 = Shift

Mode	**Operation**	**R**	**G**	**B**	**A**
16	Format	R_d	G_d	B_d	A_d
84	Blend	$R_s * A_s + R_d * (1 - A_s)$	$G_s * A_s + G_d * (1 - A_s)$	$B_s * A_s + B_d * (1 - A_s)$	$A_s * A_s + A_d * (1 - A_s)$
81	PreMult	$R_s + R_d * (1 - A_s)$	$G_s + G_d * (1 - A_s)$	$B_s + B_d * (1 - A_s)$	$A_s + A_d * (1 - A_s)$

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

The results of the different operations are as follows:
$_s$ = source color component, $_d$ = destination color component.

See the following table for a description of the remaining bits.

Bits 8–11 ColorFormat:

Format			**Internal Color Channel**			
(see Note1)	**Color Order**	**Name**	**R**	**G**	**B**	**A**
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0
9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0
17	RGB	5:6:5 Back	5@27	6@21	5@16	0

Note: The format column is also dependant on bit 16. n@m means n bits starting at bit m. Front and back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the 7-bit formatted value. If the format has no alpha bits, the alpha field defaults to 0×F8.

Bit 12	NoAlphaBuffer	
		0 = Alpha buffer present 1 = No alpha buffer present
Bit 13	ColorOrder:	
		0 = BGR 1 = RGB
Bit 14	BlendType:	
		0 = RGB 1 = Ramp
Bit 16	Color Format Extension:	
		The most significant bit extension to the color format is held in bits 8–11.

AlphaMapLowerBound

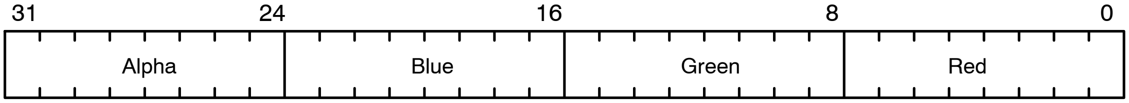
Name:	Alpha Map Color Test Lower Bounds	
Unit:	Texture/Read	
Region: 0	Offset: Tag:	0×0000.8F20 0×01E4
Reset Value:	Undefined	
Read/write		

31–24	23–16	15–8	7–0
Alpha	Blue	Green	Red

This register specifies the lower bounds for the alpha map test.

AlphaMapUpperBound

Name:	Alpha Map Color Test Upper Bounds	
Unit:	Texture/Read	
Region: 0	Offset:	0×0000.8F18
	Tag:	0×01E3
Reset Value:	Undefined	
Read/write		



This register specifies the upper bounds for the alpha map test.

AreaStippleMode

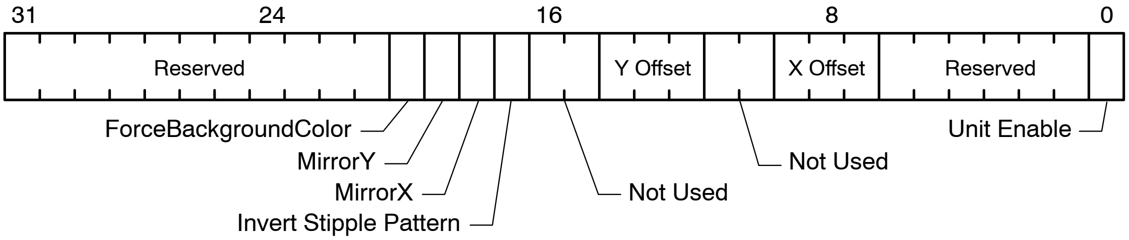
Name: Area Stipple Mode

Unit: Scissor/Stipple

Region: 0: Offset: 0.000.81A0
Tag: 0×0034

Reset Value: Undefined

Read/write



This register controls area stippling. You must set both the AreaStippleEnable bit in the Render command register and the enable bit 0 in the AreaStippleMode register to enable the area stipple test.

Bit 0 Unit Enable:
0 = Disable
1 = Enable

Bits 7–9 X Offset:

Bits 12–14 Y Offset:

Bit 17 Invert Stipple Pattern:
0 = No invert
1 = Invert

Bit 18 MirrorX:
0 = No mirror in X
1 = Mirror stipple pattern in X direction

Bit 19 MirrorY:

0 = No mirror in Y

1 = Mirror stipple pattern in Y direction

Bit 20 ForceBackgroundColor.

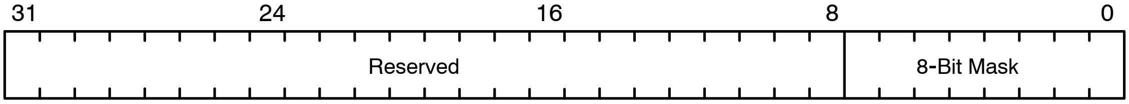
Controls operation of the stipple test. If disabled any fragment failing the test is discarded. If enabled any fragment failing the est is drawn (other tests allowing) but the color is taken from the Texel0 register. Used to support foreground and background colors.

0 = Disable

1 = Enable

AreaStipplePattern(0...7)

Name:	Area Stipple Pattern	
Unit:	Scissor/Stipple	
Region: 0	Offset: Tag:	0×0000.8200...0.×0000.8238 0×0040...0×0047
Reset Value:	Undefined	
Read/write		

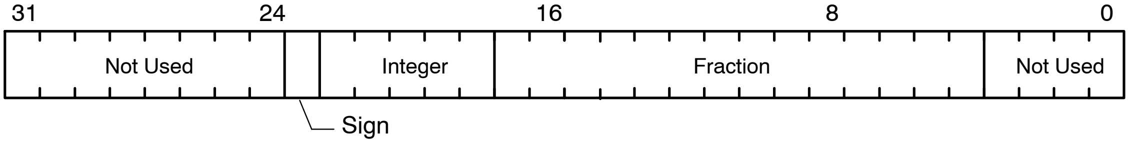


When rasterizing a primitive with area stippling, these eight registers provide the bitmask that enables and disables corresponding fragments for drawing.

You must set both the AreaStippleEnable bit 0 in the Render command register and the enable bit 0 in the AreaStippleMode register to enable the Area Stipple test.

AStart

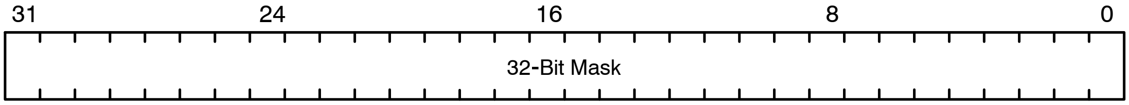
Name:	Initial Alpha Color	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.87C8 0×00F9
Reset Value:	Undefined	
Read/write		



This register sets the initial alpha value for a vertex when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

BitMaskPattern

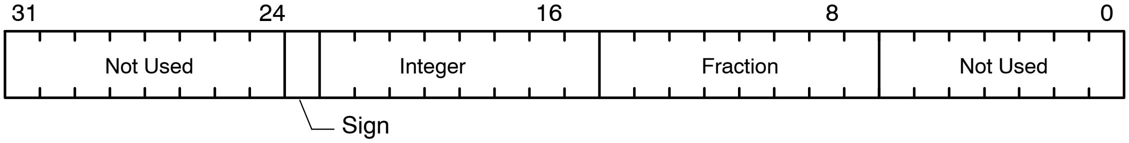
Name:	Bit Mask Pattern	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8068 0×000D
Reset Value:	Undefined	
Write only		



This register sets the value that controls the bit-mask stipple operation (if enabled). Fragments are accepted or rejected based on the current BitMask test modes, as defined by the RasterizerMode register. The SyncOnBitmask bit in the Render command register must also be enabled.

BStart

Name:	Initial Blue Color	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.87B0 0×00F6
Reset Value:	Undefined	
Read/write		



This register sets the initial blue value for a vertex when in Gouraud-shading mode. The value is 2s-complement 9.11 fixed-point format.

ChromaLowerBound,ChromaUpperBound

Name:	Chroma Lower Bound, Chroma Upper Bound	
Unit:	YUV	
Region: 0	Offset: Tag:	0×00008F10...0×0000.8F08 0×01E2, 0×01E1
Reset Value:	Undefined	
Read/write		

31 24	16	8	0
Alpha	Blue	Green	Red

31 24	16	8	0
Alpha	V	U	Y

This register specifies the lower and upper bounds for the chroma test. The test is performed against the contents of the Texel0 register, which holds data in the internal RGB format or the YUV format (before conversion) of 8 bits per component. The test is performed on all 8 bits of each component. All components must be inside the bounds for the test to pass if TestMode is set to one in the YUVMode register. For the test to fail, the TestMode must be set to two in the YUVMode register.

Color

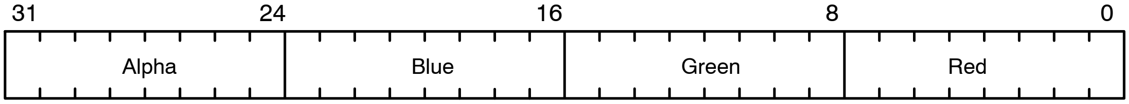
Name: Color

Unit: Color DDA

Region: 0 Offset: 0×0000.87F0
Tag: 0×00FE

Reset Value: Undefined

Write



This register downloads image data to the framebuffer. The format is either the standard color format or the raw framebuffer format if the Color Format unit is disabled.

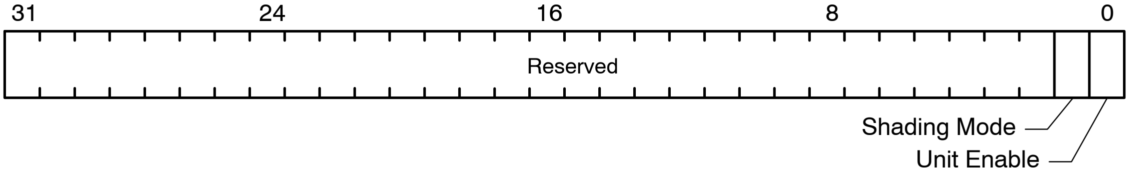
In CI mode, the color index is placed in bits 0–7. If there are fewer than 8 bits in a component, it left justifies the color index and sets the unused bits to zero.

This register cannot be saved and restored as part of a task context switch.

When used, always reload this register at the start of every command, and disable the Color DDA unit prior to loading it.

ColorDDAMode

Name:	Color DDA Mode	
Unit:	Color DDA	
Region: 0	Offset Tag:	0×0000.87E0 0×00FC
Reset Value:	Undefined	
Read/write		



This register controls the mode of operation of the Color DDA unit:

Bit 0 Unit Enable:
0 = Disable
1 = Enable

Bit 1 Shading Mode:
0 = Flat
1 = Gouraud

Config

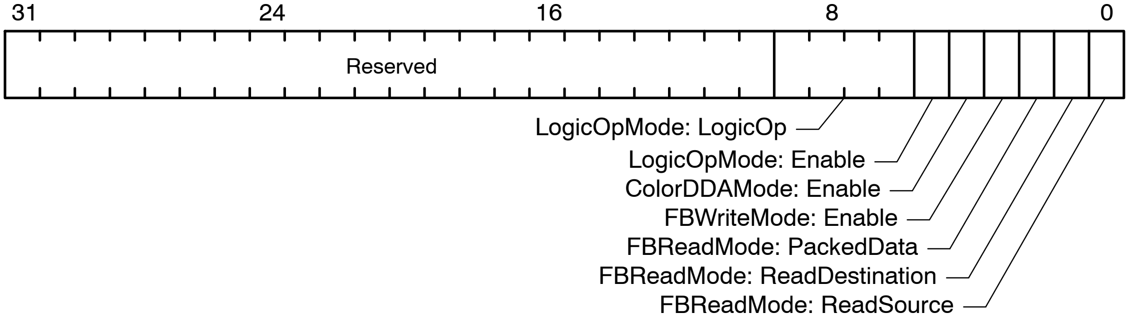
Name: Configuration

Unit:

Region: 0 Offset: 0×0000.8D90
Tag: 0×01B2

Reset Value: Undefined

Read/write



This register sets the specified fields in various registers.

ConstantColor

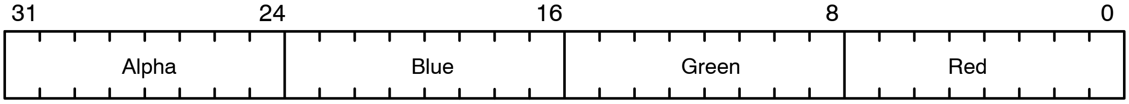
Name: Constant Color

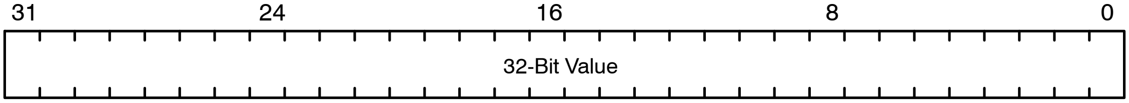
Unit: Color DDA

Region: 0 Offset: 0×0000.87E8
Tag: 0×00FD

Reset Value: Undefined

Read/write



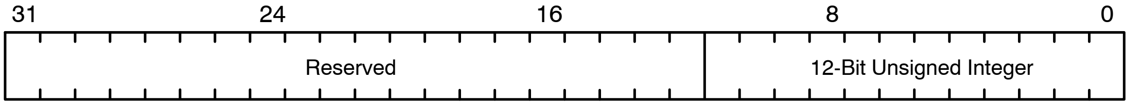


This register holds the constant color in either RGBA or raw framebuffer format. This value is used when the ColorDDAMode register is set to flat-shading mode.

The internal color format interprets the eight-bit fields as either a 5.3 fixed-point format for 3-D operations or an 8-bit integer for 2-D operations. In CI mode, the color index is placed in bits 0–7. If a component has fewer than eight bits, then left justify the color index and set the unused bits to zero.

Continue

Name:	Continue	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8058 0×000B
Reset Value:	Undefined	
Write		

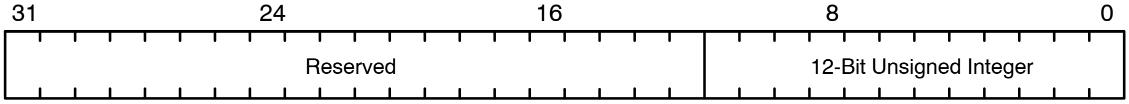


This register continues rasterization after new delta value(s) are loaded, but does not reload either of the trapezoid-edge DDAs.

The data field holds the number of scanlines to fill. This count does not load to the Count register.

ContinueNewDom

Name:	Continue – New Dominant Edge	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8048 0×0009
Reset Value:	Undefined	
Write		



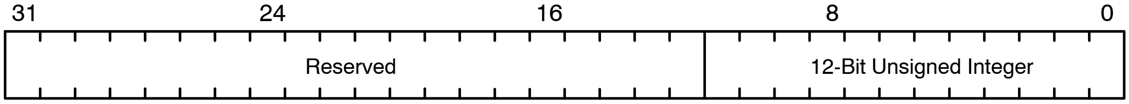
This register continues rasterization with a new dominant edge. The dominant-edge DDA reloads with the new parameters. The subordinate edge carries on from the previous trapezoid. This breaks any convex polygon into a collection of trapezoids and maintains continuity across boundaries.

Because this command only affects the Rasterizer DDA (and not other units), it is not suitable for 3-D operations.

The data field holds the number of scanlines to fill. This count does not load to the Count register.

ContinueNewLine

Name:	Continue – New Line Segment	
Unit:	Rasterizer	
Region: 0	Offset:	0×0000.8040
	Tag:	0×0008
Reset Value:	Undefined	
Write		



This register continues rasterization for the next segment in a polyline. The XY position carries on from the previous line; however, the fraction bits in the DDAs can either be retained or set to zero, one half, or nearly one half under control of the RasterizerMode register.

The data field holds the number of pixels in a line. This count does not load to the Count register.

The use of ContinueNewLine is not recommended for OpenGL because the DDA units start with a slight error, as compared to the value with which they would have been loaded for the second and subsequent segments.

ContinueNewSub

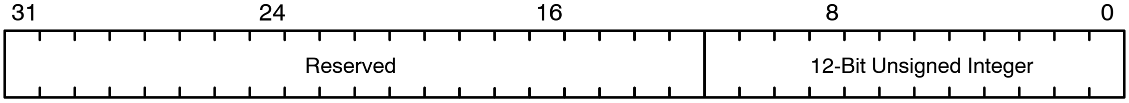
Name: Continue – New SubordinateEdge

Unit: Rasterizer

Region: 0 Offset: 0×0000.8050
Tag: 0×000A

Reset Value: Undefined

Write

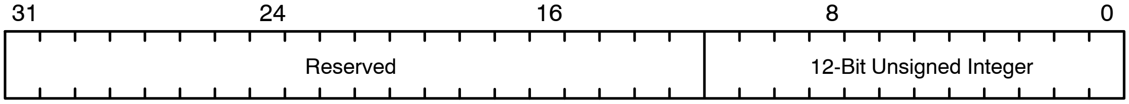


This register continues rasterization with a new subordinate edge. The subordinate DDA reloads with the new parameters. The dominant edge carries on from the previous trapezoid. This is useful when scan-converting triangles with a knee, that is, two subordinate edges.

The data field holds the number of scanlines to fill. This count does not load to the Count register.

Count

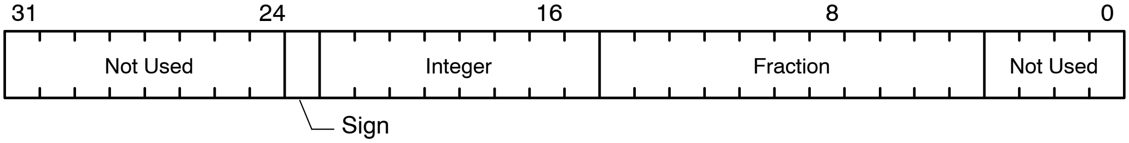
Name:	Count	
Unit:	Rasterizer	
Region: 0	Offset:	0×0000.8030
	Tag:	0×0006
Reset Value:	Undefined	
Read/write		



Interpretation of contents is dependent on the mode set in the Render command; that is, it specifies the number of pixels in a line, or the number of scanlines in a trapezoid.

dBdx

Name:	X Derivative – Blue	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.87B8 0×00F7
Reset Value:	Undefined	
Read/write		



This register sets the blue-value X derivative for the interior of a trapezoid when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

dBdyDom

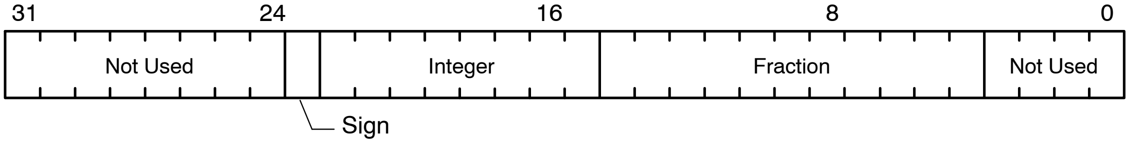
Name: Y Derivative Dominant – Blue

Unit: Color DDA

Region: 0 Offset: 0×0000.87C0
Tag: 0×00F8

Reset Value: Undefined

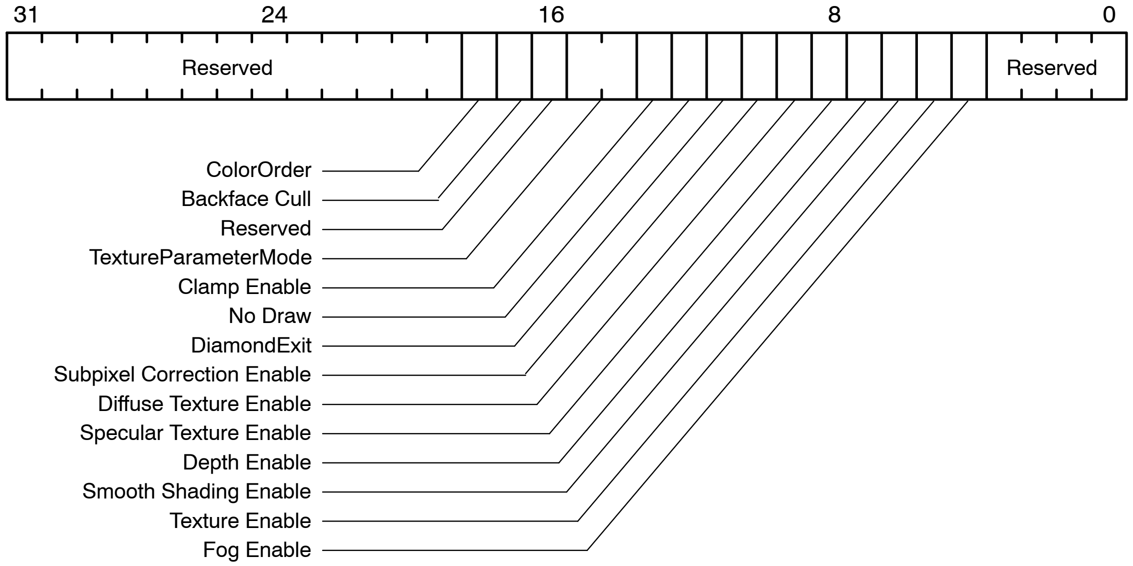
Read/write



This register sets the blue-value Y derivative to dominant along a line, or sets it for the dominant edge of a trapezoid when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

DeltaMode

Name:	Delta Mode	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9300 0×00260
Reset Value:	Undefined	
Read/write		



Bit 4 Fog Enable: This field is qualified by the FogEnable bit in the Draw command.

0 = Disable
1 = Enable

Bit 5 Texture Enable: This field is qualified by the TextureEnable bit in the Draw command.

0 = Disable
1 = Enable

Bit 6 Smooth Shading Enable:

0 = Disable
1 = Enable

Bit 7 Depth Enable:

0 = Disable
1 = Enable

Bit 8 Specular Texture Enable:
0 = Disable
1 = Enable

Bit 9 Diffuse Texture Enable:
0 = Disable
1 = Enable

Bit 10 Subpixel Correction Enable: This field is qualified by the SubpixelCorrectionEnable bit in the Draw command.
0 = Disable
1 = Enable

Bit 11 Diamond Exit:
0 = Disable
1 = Enable

Bit 12 No Draw: When set, prevents a Render command from being sent to the rendering devices. This field only affects the Draw commands. This field allows the host to alter the setup parameters before sending a Render command.
0 = Disable
1 = Enable

Bit 13 Clamp Enable: When set, clamps the input values to a parameter-specific range. The texture parameters are not affected by this field.
0 = Disable
1 = Enable

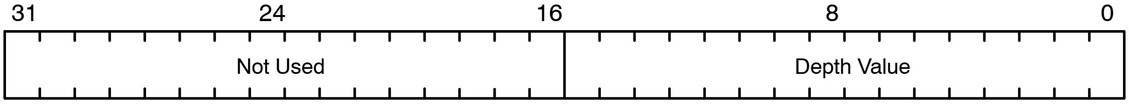
Bits 14, 15 TextureParameterMode:
0:Used as given
1: Clamped to lie in the range −1.0 to 1.0
2: Normalize to lie in the range −1.0 to 1.0

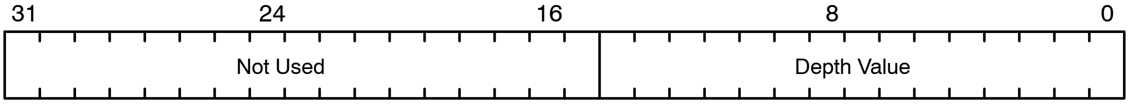
Bit 17 BackFace Cull:
0 = Disable
1 = Enable

Bit 18 ColorOrder: Specifies order of colors in V*PackedColor messages.
Bit 31 Bit 0
0 = Alpha, Blue, Green, Red
1 = Alpha, Red, Green, Blue
Each color component is 8 bits.

Depth

Name:	Depth	
Unit:	Stencil/Depth	
Region: 0	Offset: Tag:	0×0000.89A8 0×0135
Reset Value:	Undefined	
Read/write		





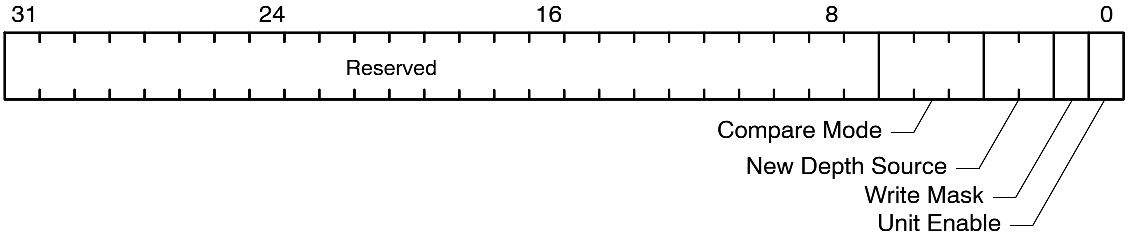
This register holds an externally sourced 16- or 15-bit depth value. Set the unused most significant bits to zero.

The register is used in the draw pixels function when the host supplies the depth values through the Depth register.

Alternatively, it is used when a constant depth value is needed, for example, when clearing the depth buffer or for 2-D rendering where the depth is held constant.

DepthMode

Name:	Depth Mode	
Unit:	Stencil/Depth	
Region: 0	Offset: Tag:	0×0000.89A0 0×0134
Reset Value:	Undefined	
Read/write		



This register compares a fragment's depth value with the updated depth buffer. If the compare function is less and the result is true, then the fragment value is less than the source value.

Bit 0 — Unit Enable:
0 = Disable
1 = Enable

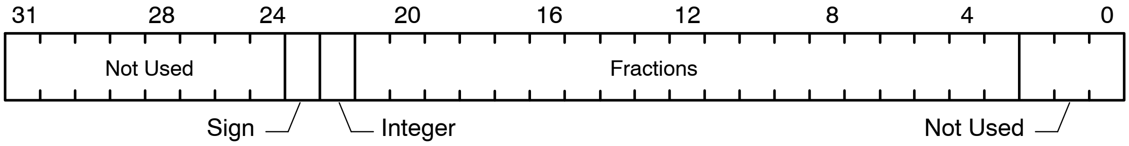
Bit 1 — Write Mask:
0 = Disable write to depth buffer.
1 = Enable write to depth buffer.

Bits 2, 3 — New Depth Source:
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

Bits 4–6	Compare Mode: 0 = Never 1 = Less 2 = Equal 3 = Less or equal 4 = Greater 5 = Not equal 6 = Greater or equal 7 = Always

dFdx

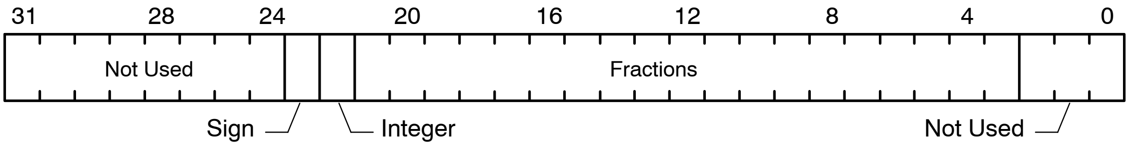
Name:	X Derivative – Fog	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86A8 0×00D5
Reset Value:	Undefined	
Read/write		



This register sets the fog coefficient derivative per unit X for use in rendering trapezoids. The value is in 2s-complement 2.19 fixed-point format.

dFdyDom

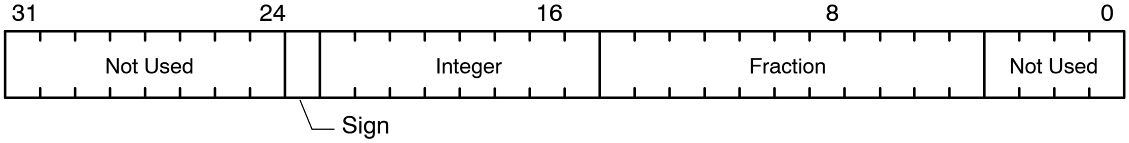
Name:	Y Derivative Dominant – Fog	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86B0 0×00D6
Reset Value:	Undefined	
Read/write		



This register sets the fog coefficient derivative per unit Y along a line, or sets it for the dominant edge of a trapezoid. The value is in 2s-complement 2.19 fixed-point format.

dGdx

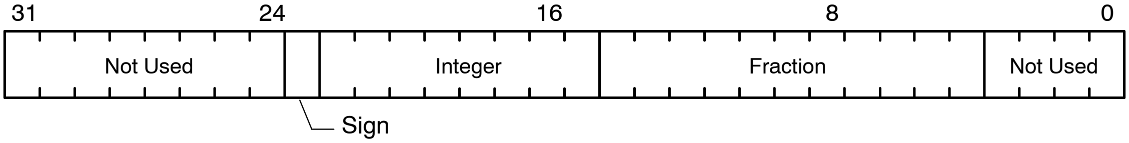
Name:	X Derivative – Green	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.87A0 0×00F4
Reset Value:	Undefined	
Read/write		



This register sets the green-value X derivative for the interior of a trapezoid when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

dGdyDom

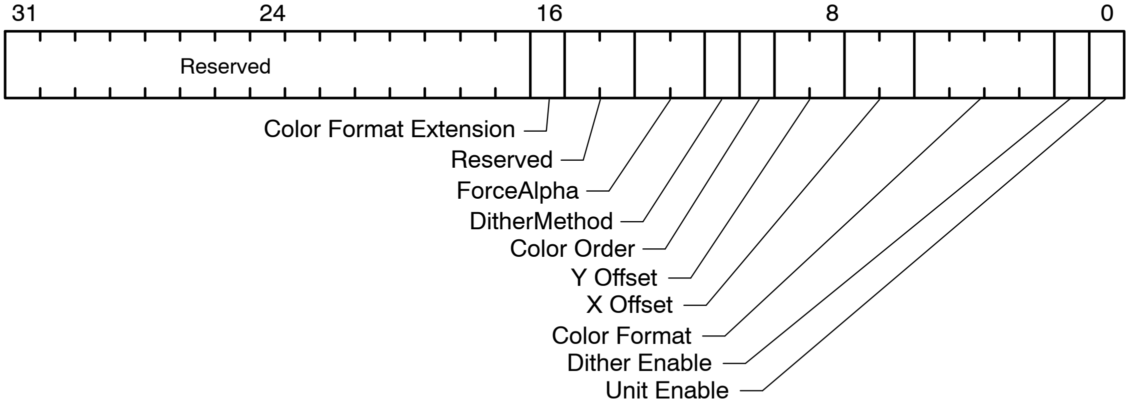
Name:	Y Derivative Dominant – Green	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.87A8 0×00F5
Reset Value:	Undefined	
Read/write		



This register sets the green-value Y derivative to dominant along a line, or sets it for the dominant edge of a trapezoid when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

DitherMode

Name:	Dither Mode	
Unit:	Color Format	
Region: 0	Offset: Tag:	0×0000.8818 0×0103
Reset Value:	Undefined	
Read/write		



This register controls the Color Format unit.

Bit 0 Unit Enable:

0 = Disable
1 = Enable

Bit 1 Dither Enable:

0 = Disable
1 = Enable

See the register description in the following table for the remaining bits.

Bits 2–5 Color Format:

Format			Internal Color Channel			
(see Note 1)	**Color Order**	**Name**	**R**	**G**	**B**	**A**
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0
9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0
17	RGB	5:6:5 Back	5@27	6@21	5@16	0

Note: The format column is also dependant on bit 16. n@m means n bits starting at bit m. Front and back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the 7-bit formatted value. If the format has no alpha bits, the alpha field defaults to 0×F8.

Bits 6, 7 X Offset to enable window relative dithering

Bits 8, 9 Y Offset to enable window relative dithering

Bit 10 Color Order:

0 = BGR
1 = RGB

Bit 11 DitherMethod:

0 = Ordered
1 = Line

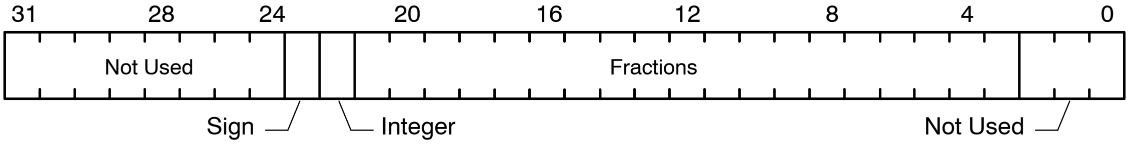
Bits 12, 13 ForceAlpha:

0 = Disable
1 = Force to 0
2 = Force to 0xF8

Bit 16 Color Format Extension: The most significant bit extension to the color format is held in bits 2–5.

dKddx

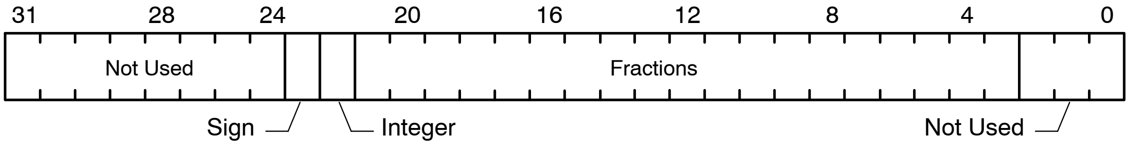
Name:	X Derivative – Kd	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86E8 0×00DD
Reset Value:	Undefined	
Read/write		



This register sets the diffuse light coefficient derivative per unit X for use in rendering texture-mapped trapezoids using the ramp application mode. The value is in 2s-complement 2.19 fixed-point format.

dKddyDom

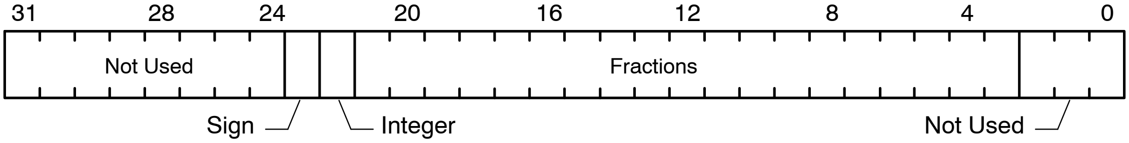
Name:	Y Derivative Dominant – Kd	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0×0000.86F0
	Tag:	0×00DE
Reset Value:	Undefined	
Read/write		



This register sets the diffuse light coefficient derivative per unit Y along a line, or sets it for the dominant edge of a trapezoid, for use with the ramp texture application mode. The value is in 2s-complement 2.19 fixed-point format.

dKsdx

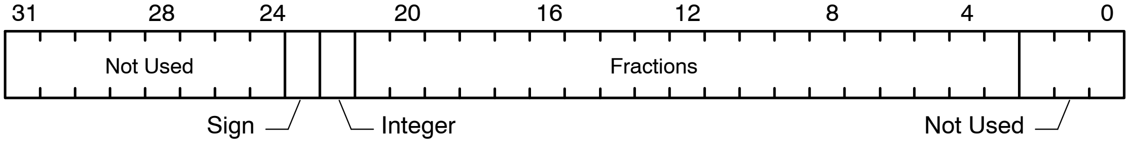
Name:	X Derivative – Ks	
Unit:	Texture/Fog/Blend	
Region: 0	Offset:	0×0000.86D0
	Tag:	0×00DA
Reset Value:	Undefined	
Read/write		



This register sets the specular light coefficient derivative per unit X for use in rendering texture-mapped trapezoids using the ramp application mode. The value is in 2s-complement 2.19 fixed-point format.

dKsdyDom

Name:	Y Derivative Dominant – Ks	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86D8 0×00DB
Reset Value:	Undefined	
Read/write		

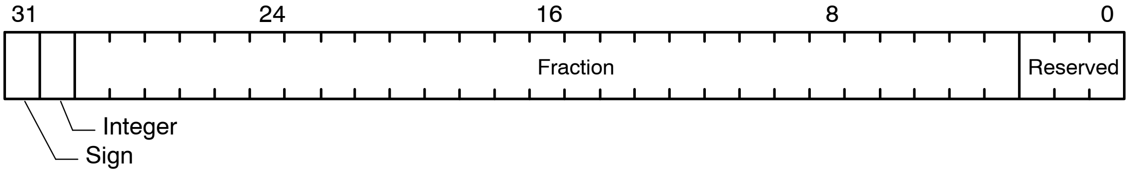


This register sets the specular light coefficient derivative per unit Y along a line, or sets it for the dominant edge of a trapezoid, for use with the ramp texture application mode. The value is in 2s-complement 2.19 fixed-point format.

dQdx

Name:	X Derivative − Homogeneous texture coordinate	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.83C0 0×0078
Reset Value:	Undefined	

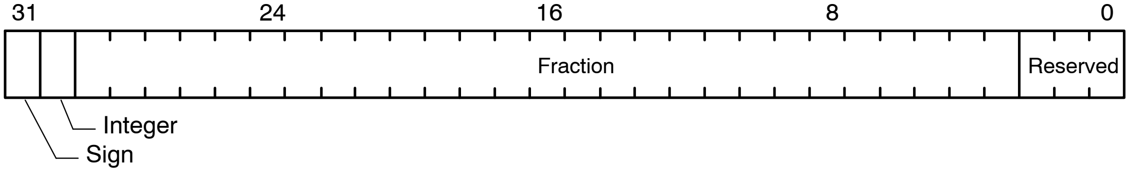
Read/write



This register sets the X derivative for the Q coordinate during texture mapping. The value is in 2s-complement 2.27 fixed-point format.

dQdyDom

Name:	Y Derivative Dominant – Homogeneous texture coordinate	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.83C8 0×0079
Reset Value:	Undefined	
Read/write		



This register sets the Y dominant derivative for the Q coordinate during texture mapping. The value is in 2s-complement 2.27 fixed-point format.

DrawLine01

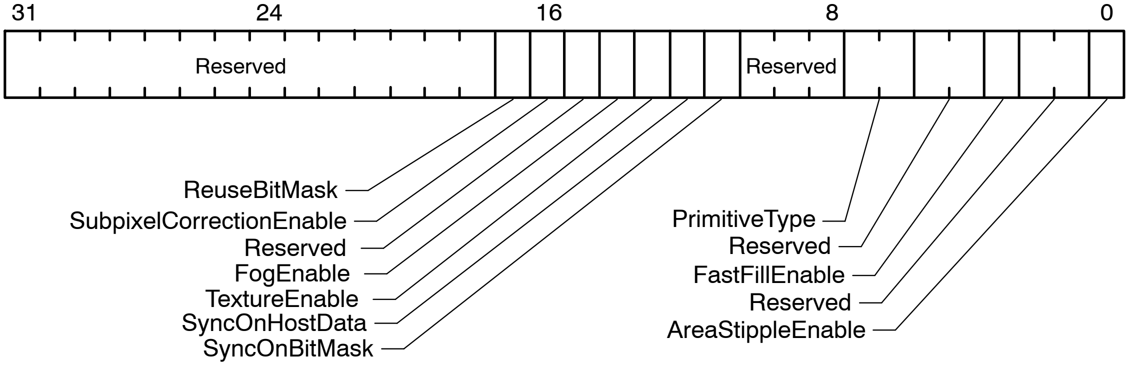
Name: Draw Line

Unit: Delta

Region: 0 Offset: 0×0000.9318
Tag: 0×0263

Reset Value: Undefined

Read/write



This register starts the rendering process from Vertex 0 to Vertex 1 through the Delta unit.

The data field defines the short-term modes required by this primitive.

Bit 0 AreaStippleEnable: Area stipple in the Stipple unit must also be enabled for stippling to occur.
0 = Disable
1 = Enable

Bit 3 FastFillEnable:
0 = Disable block filling
1 = Enable block filling

Bits 6, 7 PrimitiveType: These bits indicate the type of TVP4010 primitive to be drawn. The primitives supported and the corresponding codes are:
0 = Lines
1 = Trapezoids
2 = Points

Bit 11

SyncOnBitMask: This bit enables the bitmask test. Wait for the new bitmask when the current one expires unless SyncOnHostData or Reuse BitMask is enabled.

0 = Disable
1 = Enable

Bit 12

SyncOnHostData: When this bit is set, a fragment is produced only when one of the following registers is written to by the host: Depth, FBData, FBSourceData, Stencil, Color, or Texel0. The BitMaskPattern register is also written to if SyncOnBitMask is set.

0 = Disable
1 = Enable

Bit 13

TextureEnable: The Texture units must also be enabled for any texturing to occur.

0 = Disable
1 = Enable

Bit 14

FogEnable: The Fog Unit must also be enabled for any fogging to occur.

0 = Disable
1 = Enable

Bit 16

SubpixelCorrectionEnable: This bit enables the subpixel correction of color, depth, fog, and texture values at the start of a scanline span.

0 = Disable
1 = Enable

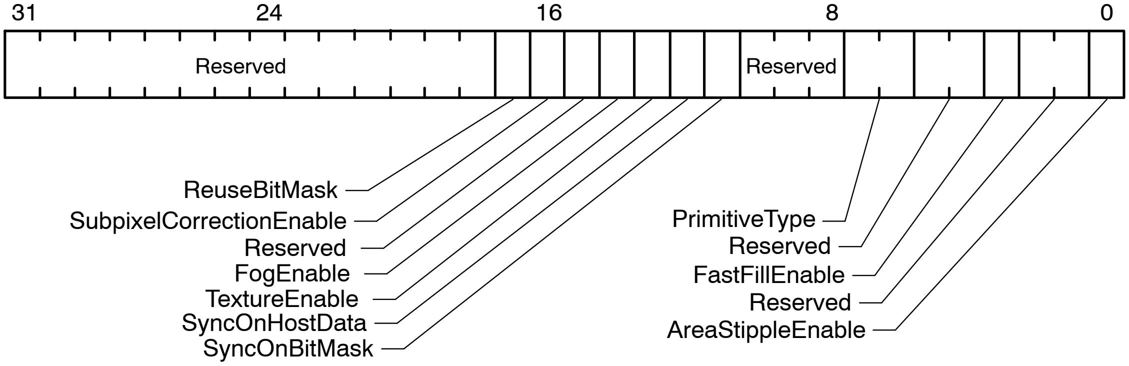
Bit 17

ReuseBitMask: This bit allows the bitmask to be reused when it expires. If enabled, the rasterizer does not wait for a new mask when the current one has been used.

0 = Disable
1 = Enable

DrawLine10

Name:	Draw Line	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9320 0×0264
Reset Value:	Undefined	
Read/write		



This register starts the rendering process from Vertex 1 to Vertex 0 through the Delta unit.

The data field defines the short-term modes required by this primitive.

Bit 0 AreaStippleEnable: Area stipple in the Stipple unit must also be enabled for stippling to occur.
0 = Disable
1 = Enable

Bit 3 FastFillEnable:
0 = Disable block filling
1 = Enable block filling

Bits 6, 7 PrimitiveType: These bits indicate the type of TVP4010 primitive to be drawn. The primitives supported and the corresponding codes are:
0 = Lines
1 = Trapezoids
2 = Points

Bit 11

SyncOnBitMask: This bit enables the bitmask test. Wait for the new bitmask when the current one expires unless SyncOnHostData or Reuse BitMask is enabled.
0 = Disable
1 = Enable

Bit 12

SyncOnHostData: When this bit is set, a fragment is produced only when one of the following registers is written to by the host: Depth, FBData, FBSourceData, Stencil, Color, or Texel0. The BitMaskPattern register is also written to if SyncOnBitMask is set.
0 = Disable
1 = Enable

Bit 13

TextureEnable: The Texture units must also be enabled for any texturing to occur.
0 = Disable
1 = Enable

Bit 14

FogEnable: The Fog Unit must also be enabled for any fogging to occur.
0 = Disable
1 = Enable

Bit 16

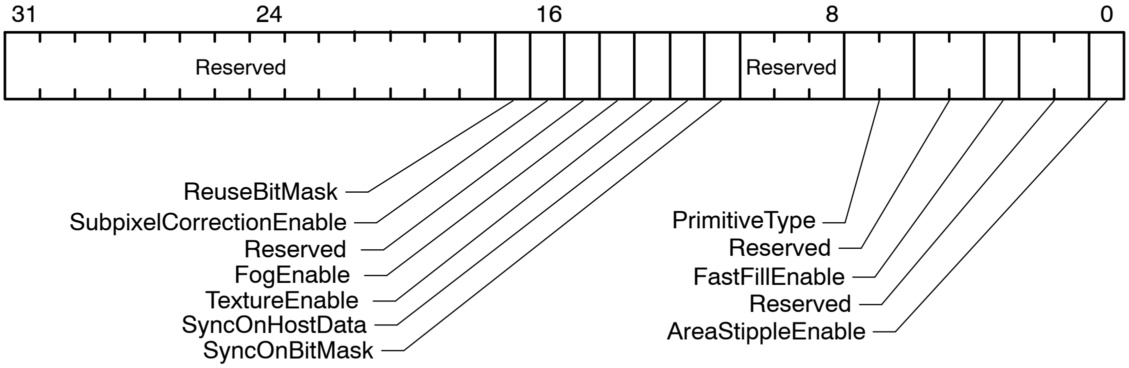
SubpixelCorrectionEnable: This bit enables the subpixel correction of color, depth, fog, and texture values at the start of a scanline span.
0 = Disable
1 = Enable

Bit 17

ReuseBitMask: This bit allows the bitmask to be reused when it expires. If enabled, the rasterizer does not wait for a new mask when the current one has been used.
0 = Disable
1 = Enable

DrawTriangle

Name:	Draw Triangle	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9308 0×0261
Reset Value:	Undefined	
Read/write		



This register starts the rendering process through the Delta unit.

The data field defines the short-term modes required by this primitive.

Bit 0

AreaStippleEnable: Area stipple in the Stipple unit must also be enabled for stippling to occur.

0 = Disable
1 = Enable

Bit 3

FastFillEnable:

0 = Disable block filling
1 = Enable block filling

Bits 6, 7

PrimitiveType: These bits indicate the type of TVP4010 primitive to be drawn. The primitives supported and the corresponding codes are:

0 = Lines
1 = Trapezoids
2 = Points

Bit 11
SyncOnBitMask: This bit enables the bitmask test. Wait for the new bitmask when the current one expires unless SyncOnHostData or Reuse BitMask is enabled.
0 = Disable
1 = Enable

Bit 12
SyncOnHostData: When this bit is set, a fragment is produced only when one of the following registers is written to by the host: Depth, FBData, FBSourceData, Stencil, Color, or Texel0. The BitMaskPattern register is also written to if SyncOnBitMask is set.
0 = Disable
1 = Enable

Bit 13
TextureEnable: The Texture units must also be enabled for any texturing to occur.
0 = Disable
1 = Enable

Bit 14
FogEnable: The Fog Unit must also be enabled for any fogging to occur.
0 = Disable
1 = Enable

Bit 16
SubpixelCorrectionEnable: This bit enables the subpixel correction of color, depth, fog, and texture values at the start of a scanline span.
0 = Disable
1 = Enable

Bit 17
ReuseBitMask: This bit allows the bitmask to be reused when it expires. If enabled, the rasterizer does not wait for a new mask when the current one has been used.
0 = Disable
1 = Enable

dRdx

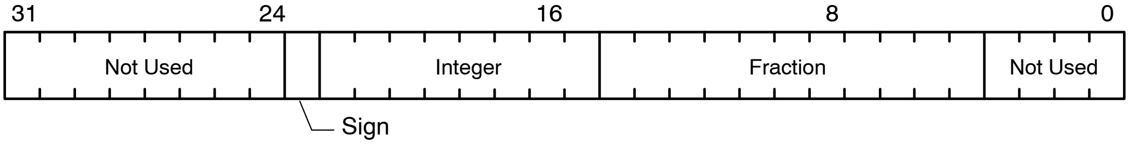
Name: X Derivative – Red

Unit: Color DDA

Region: 0 Offset: 0×0000.8788
Tag: 0×00F1

Reset Value: Undefined

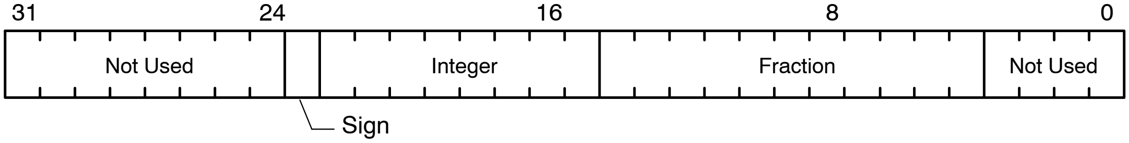
Read/write



This register sets the red-value X derivative for the interior of a trapezoid when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

dRdyDom

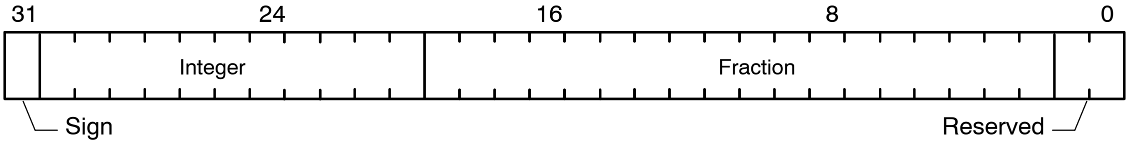
Name:	Y Derivative Dominant – Red	
Unit:	Color DDA	
Region: 0	Offset:	0×0000.8790
	Tag:	0×00F2
Reset Value:	Undefined	
Read/write		



This register sets the red-value Y derivative to dominant along a line, or sets it for the dominant edge of a trapezoid, when in Gouraud-shading mode. The value is in 2s-complement 9.11 fixed-point format.

dSdx

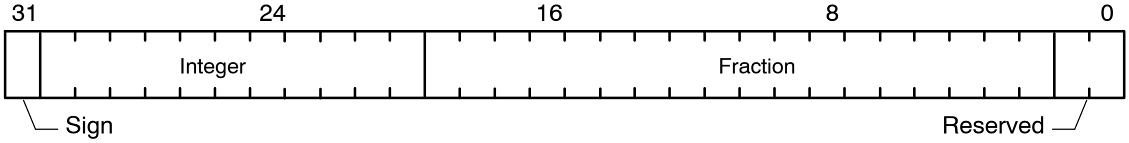
Name:	X Derivative – Texture S coordinate	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.8390 0×0072
Reset Value:	Undefined	
Read/write		



This register sets the X derivative for the S coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

dSdyDom

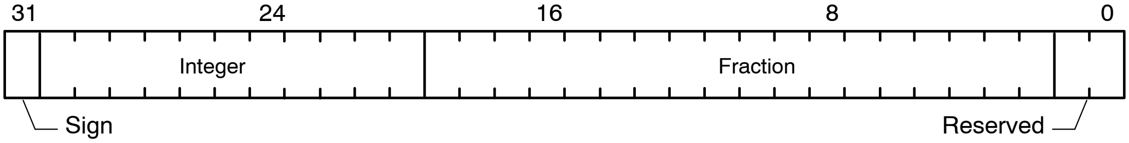
Name:	Y Derivative Dominant – Texture S coordinate	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.8398 0×0073
Reset Value:	Undefined	
Read/write		



This register sets the Y dominant derivative for the S coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

dTdx

Name:	X Derivative – Texture T coordinate	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.83A8 0×0075
Reset Value:	Undefined	
Read/write		



This register sets the X derivative for the T coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

dTdyDom

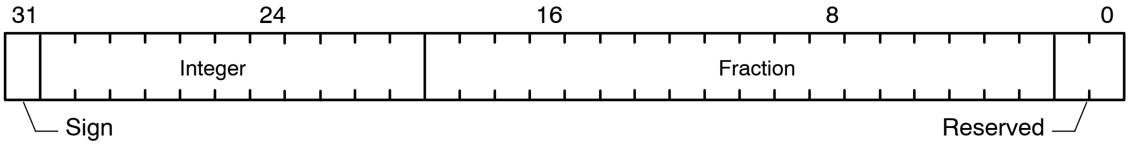
Name: Y Derivative Dominant – Texture T coordinate

Unit: Texture Address

Region: 0 Offset: 0×0000.83B0
Tag: 0×0076

Reset Value: Undefined

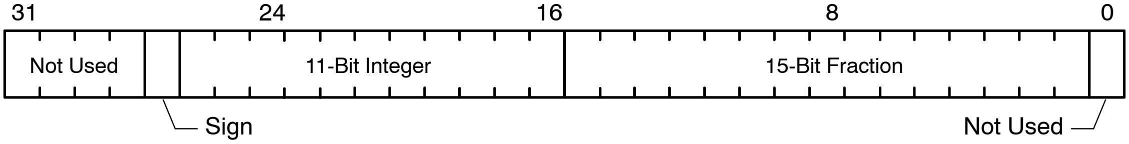
Read/write



This register sets the Y dominant derivative for the T coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

dXDom

Name:	Delta X Dominant	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8008 0×0001
Reset Value:	Undefined	
Read/write		

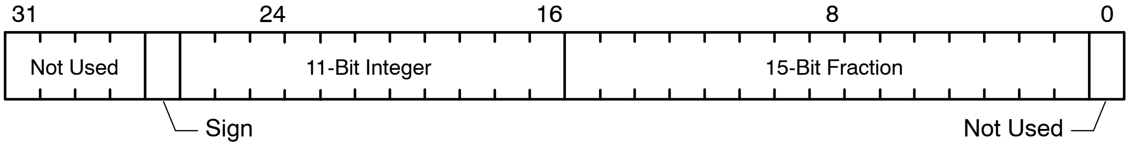


This register determines the delta value of the trapezoid when the dominant edge moves from one scanline to the next during trapezoid filling. The value is in 2s-complement 12.15 fixed-point format.

It also holds the change in X when plotting lines. For Y major lines this will be some fraction (dx/dy); otherwise, it is usually ±1.0, depending on the required scanning direction.

dXSub

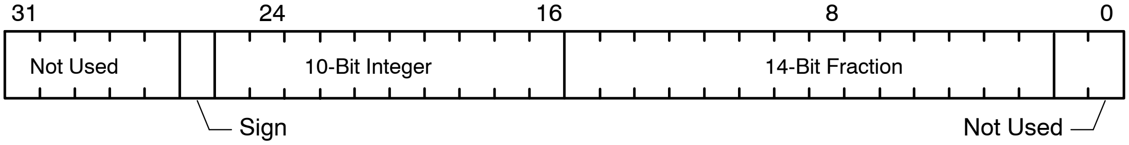
Name:	Delta X Subordinate	
Unit:	Rasterizer	
Region: 0	Offset:	0×0000.8018
	Tag:	0×0003
Reset Value:	Undefined	
Read/write		



This register determines the delta value of the trapezoid when the subordinate edge moves from one scanline to the next during trapezoid filling. The value is in 2s-complement 12.15 fixed-point format.

dY

Name:	Delta Y	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8028 0×0005
Reset Value:	Undefined	
Read/write		



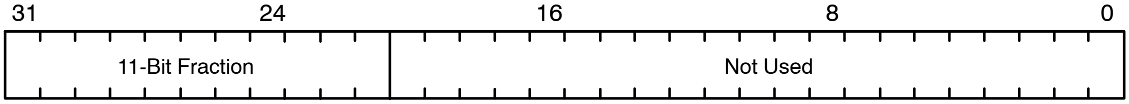
This register determines the delta value added to Y when Y moves from one scanline to the next.

For X major lines this is some fraction (dy/dx); otherwise, it is usually ±1.0, depending on the required scanning direction. The value is in 2s-complement 11.14 fixed-point format.

For trapezoids the value is ±1.0, depending on the scanning direction.

dZdxL

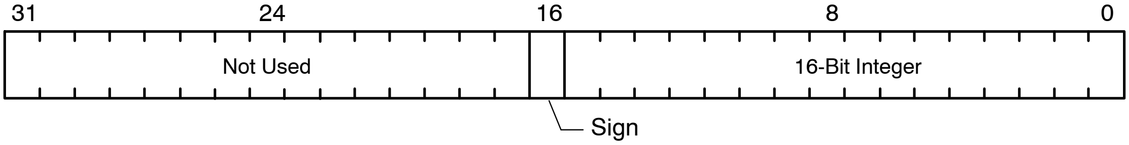
Name:	Depth Derivative X – Lower	
Unit:	Stencil/Depth	
Region: 0	Offset:	0×0000.89C8
	Tag:	0×0139
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in X, used in rendering trapezoids. dZdxU holds the most significant bits, and dZdxL the least significant bits. The combined value is in 2s-complement 17.11 fixed-point format. For trapezoids, the value is ±1 depending on the scanning direction.

dZdxU

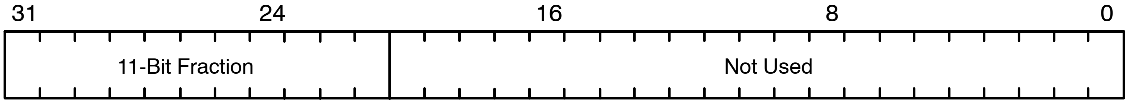
Name:	Depth Derivative X – Upper	
Unit:	Stencil/Depth	
Region: 0	Offset: Tag:	0×0000.89C0 0×0138
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in X used in rendering trapezoids. dZdxU holds the most significant bits, and dZdxL the least significant bits. The value is in 2s-complement 17.11 fixed-point format.

dZdyDomL

Name:	Depth Derivative Y Dominant − Lower	
Unit:	Stencil/Depth	
Region: 0	Offset: Tab:	0×0000.89D8 0×013B
Reset Value:	Undefined	
Read/write		



This register holds part of the depth derivative per unit in Y used for the dominant edge of a trapezoid or along a line. dZdyDomU holds the most significant bits and dZdyDomL the least significant bits. The value is in 2s-complement 17.11 fixed-point format.

dZdyDomU

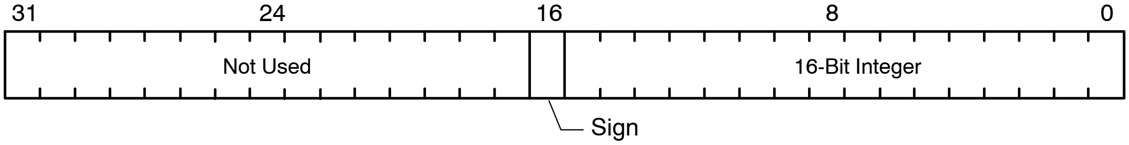
Name: Depth Derivative Y Dominant – Upper

Unit: Stencil/Depth

Region: 0 Offset: 0×0000.89D0
Tag: 0×013A

Reset Value: Undefined

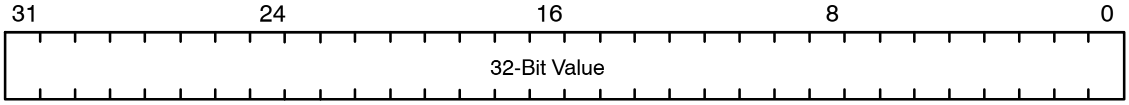
Read/write



This register holds part of the depth derivative per unit in Y used for the dominant edge of a trapezoid or along a line. dZdyDomU holds the most significant bits and dZdyDomL the least significant bits. The value is in 2s-complement 17.11 fixed-point format.

FBBlockColor

Name:	Framebuffer Block Color	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0×0000.8AC8
	Tag:	0×0159
Reset Value:	Undefined	
Read/write		



This register contains the color (and, optionally, the alpha value) to be written to the framebuffer during block write operations. The format is the raw data format of the framebuffer.

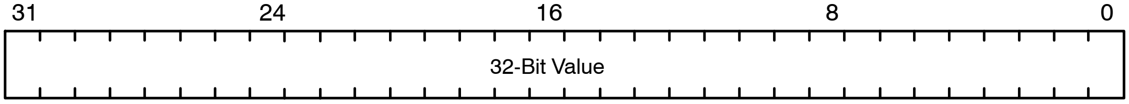
If the framebuffer is used in 8-bit packed mode, repeat the data in all four bytes of the register.

If the framebuffer is in 16-bit packed mode, repeat the data in both halves of the register.

Do not update this register immediately after a Render command that performs a block write operation.

FBBlockColorL

Name:	Framebuffer Block Fill Lower Color	
Unit:	FramebufferWrite	
Region: 0	Offset: Tag:	0×0000.8C70 0×018E
Reset Value:	Undefined	
Read/write		



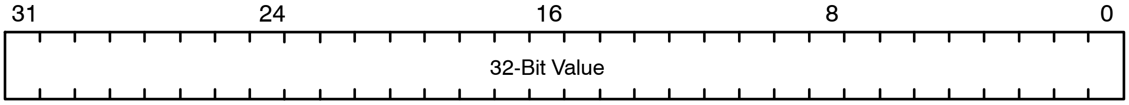
This register contains the color (and, optionally, the alpha value) to be written to the framebuffer during block write operations. It uses the raw data format of the framebuffer. Each block fill operation writes a pattern of 8 bytes that are defined by the FBBlockColorL and FBBlockColorU registers, repeating the same data until 32 pixels are filled.

If the framebuffer is used in 8-bit packed mode, repeat the data in all 4 bytes of the register.

If the framebuffer is used in 16-bit packed mode, repeat the data in both halves of the register.

FBBlockColorU

Name:	Framebuffer Block Fill Upper Color	
Unit:	FramebufferWrite	
Region: 0	Offset:	0×0000.8C68
	Tag:	0×018D
Reset Value:	Undefined	
Read/write		



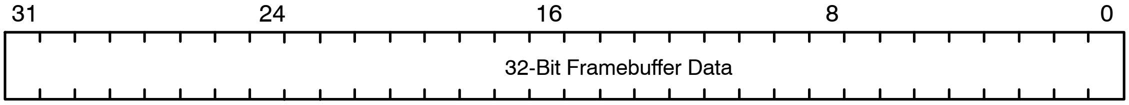
This register contains the color (and, optionally, the alpha value) to be written to the framebuffer during block write operations. It uses the raw data format of the framebuffer. Each block fill operation writes a pattern of 8 bytes that are defined by the FBBlockColorL and FBBlockColorU registers, repeating the same data until 32 pixels are filled.

If the framebuffer is used in 8-bit packed mode, repeat the data in all 4 bytes of the register.

If the framebuffer is used in 16-bit packed mode, repeat the data in both halves of the register.

FBColor

Name:	Framebuffer Color Upload	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8A98 0×0153
Reset Value:	Undefined	
Read/write		

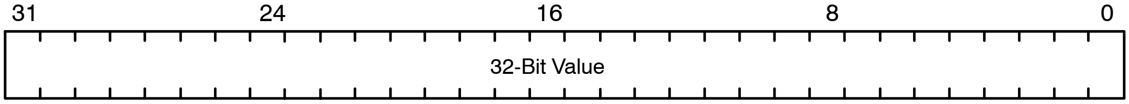


This register is used internally for image uploading. Do not write to this register. It gives the format and tag value of the data that returns through the Host Out FIFO.

The format is dependent on the raw framebuffer organization. Any reformatting is due to the format specified in the DitherMode register.

FBData

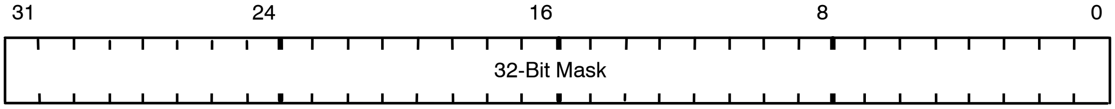
Name:	Framebuffer Data	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8AA0 0×0154
Reset Value:	Undefined	
Write		



This register supplies the data for image downloading when subsequent formatting is required. Use the AlphaBlendMode register to convert to the internal TVP4010 format, and then use the DitherMode register to convert to the required format.

FBHardwareWriteMask

Name:	Hardware Writemask	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8AC0 0×0158
Reset Value:	Undefined	
Read/write		



This register contains the hardware writemask for the framebuffer. If you set a bit to one, then the corresponding bit in the framebuffer is enabled for writing; otherwise, it is disabled. This register is only applicable to configurations where the framebuffer supports a hardware writemask. In cases where it is not supported, do not write to this register.

If you use hardware writemasks, then you must set all the bits in the FBSoftwareWriteMask register to one to disable software writemasking.

If you use the framebuffer in 8-bit packed mode, then you must repeat an 8-bit hardware writemask in all 4 bytes of the FBHardwareWriteMask register.

If you use the framebuffer in 16-bit packed mode, then you must repeat the 16-bit hardware writemask in both halves of the FBHardwareWriteMask register.

FBPixelOffset

Name:	Framebuffer Pixel Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8A90 0×0152
Reset Value:	Undefined	
Read/write		

31 — 24	16 — 8 — 0
Not Used	24-Bit 2s-Complement Integer

This register sets the offset value between buffers when simultaneously operating on multiple buffers in the framebuffer, for example, left/right/top/bottom in some OpenGL implementations. The offset value is signed or unsigned.

FBReadMode

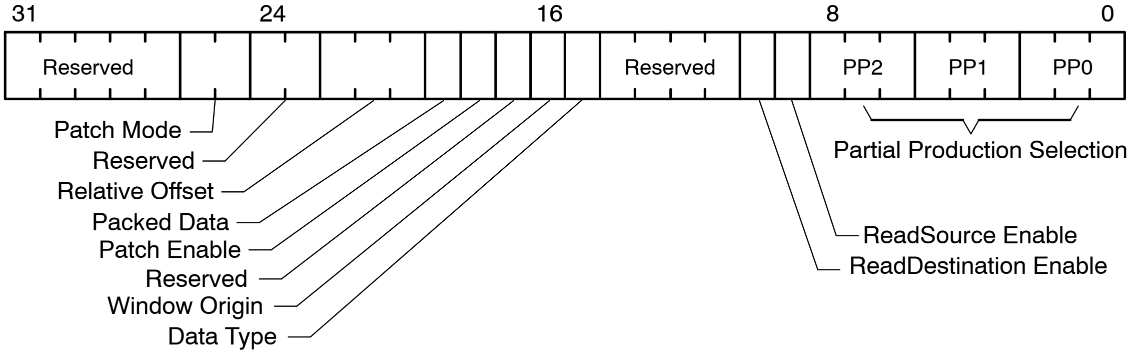
Name: Framebuffer Read Mode

Unit: Framebuffer R/W

Region: 0 Offset: 0×000.8A80
Tag: 0×0150

Reset Value: Undefined

Read/write



This register controls reading from framebuffer memory.

If you enable the read modes immediately after that same data was just written with the read modes disabled, then incorrect data can be read. To avoid this problem, send a WaitForCompletion command after enabling the read modes, but prior to the next primitive.

Bits 0–2 Partial Product 0 – See Appendix B, *Screen Widths Table*, for a table of values.

Bits 3–5 Partial Product 1 – See Appendix B, *Screen Widths Table* for a table of values.

Bits 6–8 Partial Product 2 – See Appendix B, *Screen Widths Table* for a table of values.

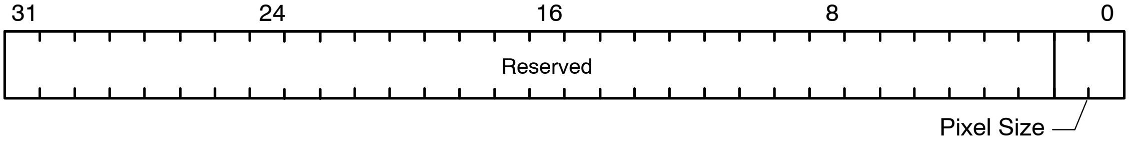
Bit 9 ReadSource Enable:
0 = No read
1 = Enable read

Bit 10 ReadDestination Enable:
0 = No read
1 = Enable read

Bit 15	Data Type:	0 = FBDefault – for data that may be written back to the frame buffer 1 = FBColor – for image upload
Bit 16	Window Origin:	0 = Top left 1 = Bottom left
Bit 18	Patch Enable:	0 = Disable 1 = Enable patched addressing for framebuffer accesses
Bit 19	Packed Data:	0 = Disable. Force TVP4020 to read one pixel at a time. 1 = Enable. Allow TVP4020 to read multiple packed pixels when possible.
Bits 20–22	Relative Offset	Three-bit 2s complement value, which specifies the number of pixels to which the source data must adjust, to align with the destination data. The PackedData Limits register also has this field, and the last of these two registers to be loaded is effected.
Bits 25–26	Patch Mode	0 = Patch (suitable for depth buffer patching) 1 = Subpatch (suitable for texture buffer patching) 2 = SubpatchPack (suitable for packed texture patching)

FBReadPixel

Name:	Framebuffer Read Pixel	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0×0000.8AD0
	Tag:	0×015A
Reset Value:	Undefined	
Read/write		



This register sets the pixel size for reading from the framebuffer.

Bits 0, 1	Pixel Size: 0 = 8 bits 1 = 16 bits 2 = 32 bits 3 = Reserved 4 = 24 bits

FBSoftwareWriteMask

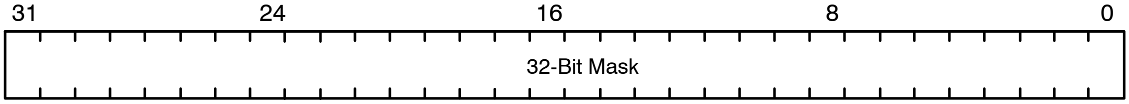
Name: Software Writemask

Unit: Logic Op

Region: 0 Offset: 0×0000.8820
Tag: 0×0104

Reset Value: Undefined

Read/write

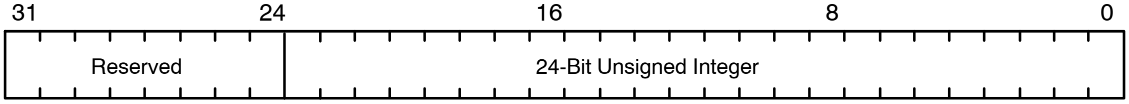


This register contains the software writemask for the framebuffer. If you set a bit to one, then the corresponding bit in the framebuffer is enabled for writing; otherwise, it is disabled. In addition, whenever the writemask is other than all ones, you must enable the framebuffer read modes by setting the ReadSourceEnable bit in the FBReadMode register.

If you use hardware writemasks, then you must set all the bits in the software writemask to one to disable software writemasking.

FBSourceBase

Name:	Base address of source framebuffer data	
Unit:	Framebuffer Read	
Region: 0	Offset:	0×0000.8D80
	Tag:	0×01B0
Reset Value:	Undefined	
Write		

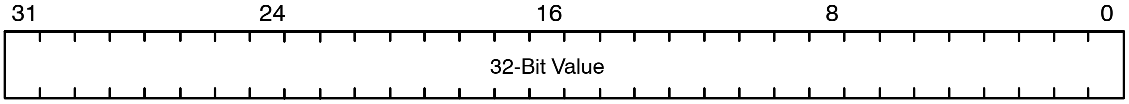


This register contains the base address of source data for framebuffer copies and tracks the value of FBWindowBase. To modify this register, load it after FBWindowBase.

FBSourceData

Name:	Framebuffer Source Data	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8AA8 0×0155
Reset Value:	Undefined	

Write



This register supplies the data for image downloading with logical operations when the data is treated as the source rather than as the destination parameter.

The data supplied should be in raw framebuffer format.

FBSourceDelta

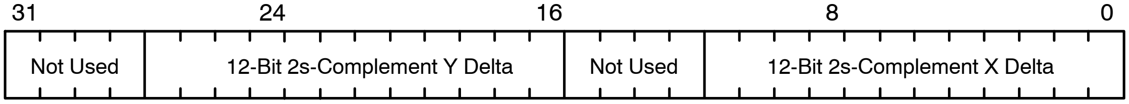
Name: Difference between destination to source data

Unit: Framebuffer Read

Region: 0 Offset: 0×0000.8D88
Tag: 0×01B1

Reset Value: Undefined

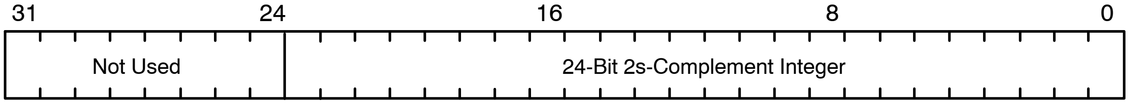
Read/write



This register contains the difference from destination to the source data in the framebuffer. Loading this register calculates and loads an appropriate value into FBSourceOffset.

FBSourceOffset

Name:	Framebuffer Source Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8A88 0×0151
Reset Value:	Undefined	
Read/write		



This register sets the offset value from destination to source for a copy operation in the framebuffer; that is:

```
source offset = destination address – source address
```

FBWindowBase

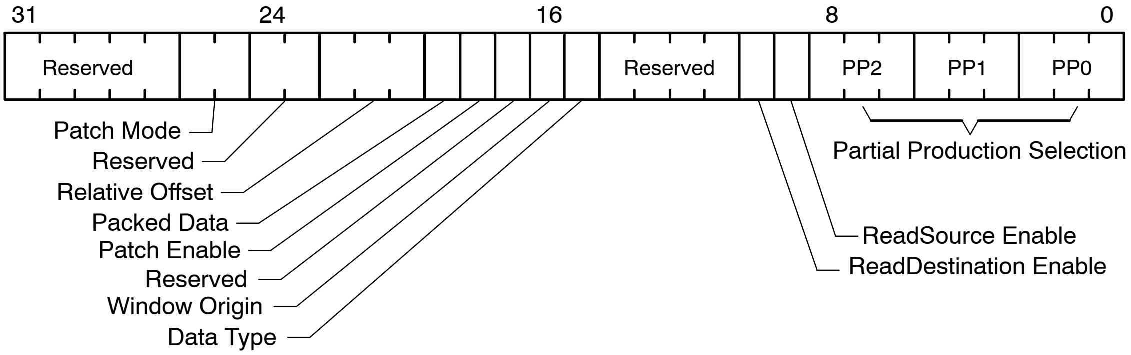
Name:	Framebuffer Window Base	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0×0000.8AB0
	Tag:	0×0156
Reset Value:	Undefined	
Read/write		

31 24	16 8 0
Reserved	24-Bit Unsigned Integer

This register contains the current base address of the window in the framebuffer.

FBWriteConfig

Name:	Framebuffer Write Configuration	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×000.8AE8 0×015D
Reset Value:	Undefined	
Read/write		



This register controls writing to the framebuffer memory and works in conjunction with the FBReadMode register.

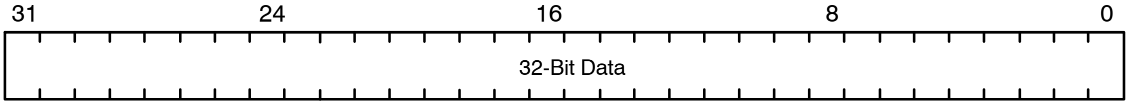
When the FBReadMode register is loaded, this register is also loaded with the same value. This register can hold a different partial product to differentiate between the read surface pitch and the write surface pitch; for example, copying from a section of the framebuffer with a pitch of 256 to the screen that has a pitch of 1024. To use this register correctly, make sure to load it after you load the FBReadMode register. The bit definitions are the same as FBReadMode.

Bits 0 – 2	Partial Product 0 – See Appendix B, *Screen Widths Table*, for a table of values.
Bits 3 – 5	Partial Product 1 – See Appendix B, *Screen Widths Table* for a table of values.
Bits 6 – 8	Partial Product 2 – See Appendix B, *Screen Widths Table* for a table of values.
Bit 9	ReadSource Enable: 0 = No read 1 = Enable read

Bit 10	ReadDestination Enable:	0 = No read 1 = Enable read
Bit 15	Data Type:	0 = FBDefault – for data that may be written back to the frame buffer 1 = FBColor – for image upload
Bit 16	Window Origin:	0 = Top left 1 = Bottom left
Bit 18	Patch Enable:	0 = Disable 1 = Enable patched addressing for framebuffer accesses
Bit 19	Packed Data:	0 = Disable. Force TVP4020 to read one pixel at a time. 1 = Enable. Allow TVP4020 to read multiple packed pixels when possible.
Bits 20–22	Relative Offset	Three-bit 2s complement value, which specifies the number of pixels to which the source data must adjust, to align with the destination data. The PackedData Limits register also has this field, and the last of these two registers to be loaded is effected.
Bits 25–26	Patch Mode	0 = Patch (suitable for depth buffer patching) 1 = Subpatch (suitable for texture buffer patching) 2 = SubpatchPack (suitable for packed texture patching)

FBWriteData

Name:	Framebuffer Write Data	
Unit:	Logic Op	
Region: 0	Offset: Tag:	0×0000.8830 0×106
Reset Value:	Undefined	
Read/write		



This register contains the color value that writes to the framebuffer when you set the UseConstantFBWriteData bit of the LogicalOpMode register to one. You must meet the following conditions to use this mode of rendering:

- □ Flat-shaded aliased primitive
- □ No required dithering
- □ No logical operation involving a destination factor
- □ No stencil or depth test
- □ No texture, fog or alpha blending
- □ No software writemasking

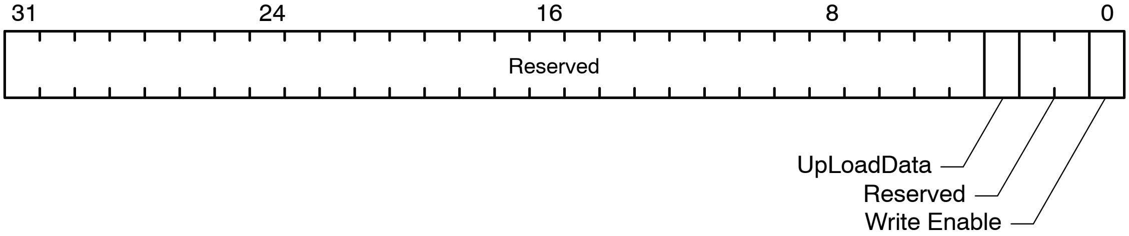
The data is in the raw format of the framebuffer. If the pixel size is 8 bits, then repeat the data in all 4 bytes. If the pixel size is 16 bits, then repeat the data in both halves of the word.

You can use hardware writemasks, if they are available.

It is not recommended that this register be used. It is included here solely to understand legacy TV4020 software.

FBWriteMode

Name:	Framebuffer Write Mode	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8AB8 0×0157
Reset Value:	Undefined	
Read/write		



This register controls writing to the framebuffer.

Bit 0 Write Enable:

0 = Disable
1 = Enable

Bit 3 UpLoadData:

0 = No upload
2 = Upload color to host

FilterMode

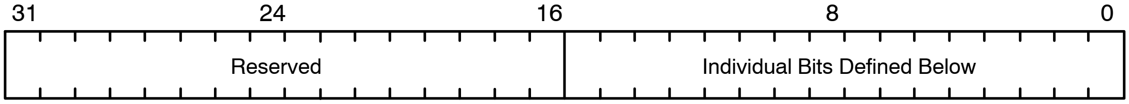
Name: Filter Mode

Unit: Host Out

Region: 0 Offset: 0×0000.8C00
Tag: 0×0180

Reset Value: Undefined

Read/write



This register controls the culling of information from the output FIFO. If both the tag and data are specified, then the tag is always the first word in the FIFO.

Bits 0–3 Reserved for future use: set to zero.

Bit 4 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.

Bit 5 Depth Data Filter: Used in depth-buffer image upload
0 = Cull depth data values from being passed to output FIFO.
1 = Pass depth data values to output FIFO.

Bit 6 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.

Bit 7 Stencil Data Filter: Used in stencil-buffer image upload

0 = Cull stencil data values from being passed out put FIFO.

1 = Pass stencil data values to output FIFO.

Bit 8 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.

Bit 9 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.

Bit 10 Synchronization Tag Filter:

0 = Cull synchronization tags from being passed to output FIFO.

1 = Pass synchronization tags to output FIFO.

Bit 11 Synchronization Data Filter:

0 = Cull synchronization data values from being passed to output FIFO.

1 = Pass synchronization data values to output FIFO.

Bit 12 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.

Bit 13 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.

Bits 14, 15 Reserved for future use: set to zero.

FogColor

Name:	Fog Color	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.8698 0×00D3
Reset Value:	Undefined	
Read/write		

31 24	16	8	0
Alpha	Blue	Green	Red

This register provides the color to blend with the fragment color when you enable fogging.

FogMode

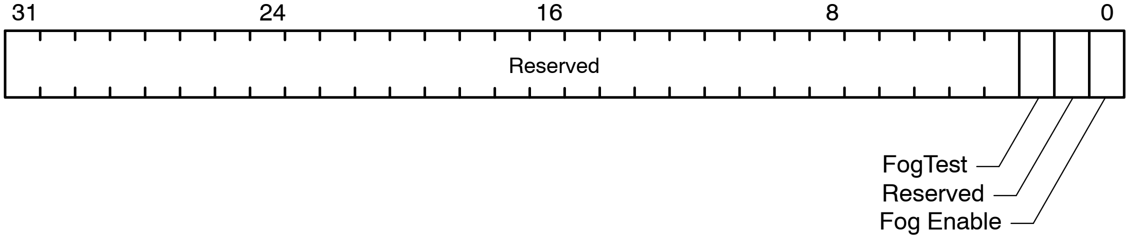
Name: Fog Mode

Unit: Texture/Fog/Blend

Region: 0 Offset: 0×0000.8690
Tag: 0×00D2

Reset Value: Undefined

Read/write



This register controls operation of the Fog unit.

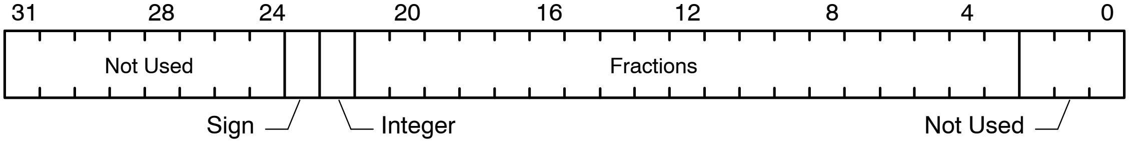
When you enable FogTest, it rejects fragments with negative fog values.

To apply fogging to a primitive, set the FogEnable bit in the Render command register.

Bit 0	Fog Enable:	
		0 = Disable
		1 = Enable
Bit 2	FogTest:	
		0 = Disable
		1 = Enable

FStart

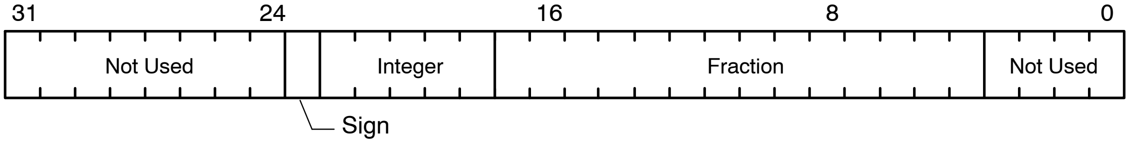
Name:	Initial Fog Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86A0 0×00D4
Reset Value:	Undefined	
Read/write		



This register sets the fog coefficient start value. The interpolation coefficient blends the fragment color with the color in the FogColor register. The value is in 2s-complement 2.19 fixed-point format.

GStart

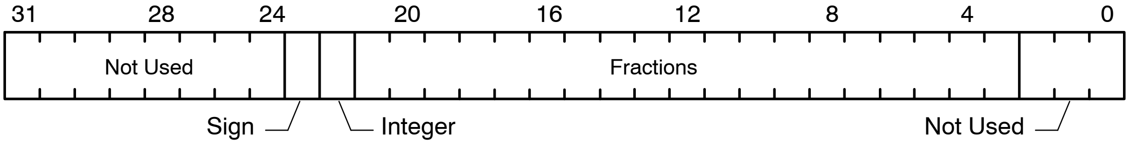
Name:	Initial Green Color	
Unit:	Color DDA	
Region: 0	Offset:	0×0000.8798
	Tag:	0×00F3
Reset Value:	Undefined	
Read/write		



This register sets the initial green value for a vertex when in Gouraud-shading mode. The value is 2s-complement 9.11 fixed-point format.

KdStart

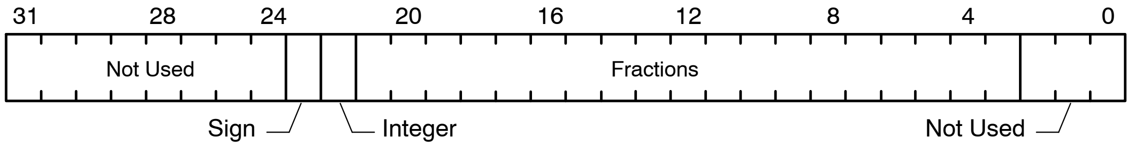
Name:	Initial Kd Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86E0 0×00DC
Reset Value:	Undefined	
Read/write		



This register sets the start value for the diffuse light parameter during texture mapping in ramp application mode. The value is in 2s-complement 2.19 fixed-point format.

KsStart

Name:	Initial Ks Value	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.86C8 0×00D9
Reset Value:	Undefined	
Read/write		

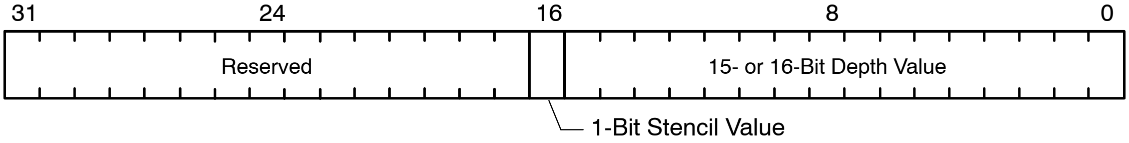


This register sets the start value for the specular light parameter during texture mapping in the ramp application mode. The value is in 2s-complement 2.19 fixed-point format.

LBData

Name:	Localbuffer Data Download	
Unit:	Localbuffer R/W	
Region:	Offset:	0×0000.8898
	Tag:	0×0113
Reset Value:	Undefined	

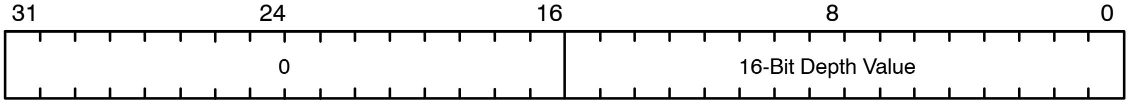
Write



This register downloads depth and/or stencil data to localbuffer memory. Supply the data in the raw localbuffer format.

LBDepth

Name:	Localbuffer Depth Upload	
Unit:	Localbuffer R/W	
Region: 0	Offset:	0×0000.88B0
	Tag:	0×0116
Reset Value:	Undefined	
Read/write		



This register uploads depth data from localbuffer memory. Do not write to this register. It gives the tag value and format of the data when it is read from the Host Out FIFO. If the depth buffer is less than 16 bits, the depth value is right-justified and zero-extended.

LBReadFormat

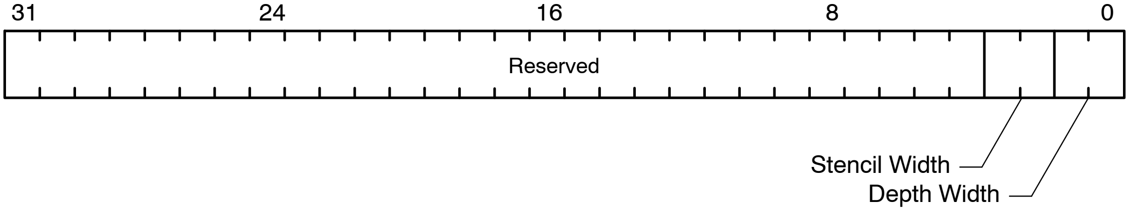
Name: Localbuffer Read Format

Unit: Localbuffer R/W

Region: 0 Offset: 0×0000.8888
Tag: 0×0111

Reset Value: Undefined

Read/write



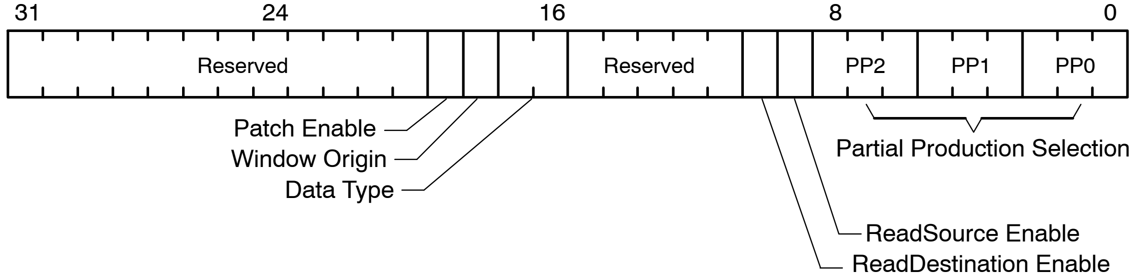
This register specifies the format used when reading from localbuffer memory. The effect of creating a format with overlapping fields is undefined. There is no need to synchronize the TVP4020 before changing this register.

Bits 0, 1 Depth Width:
0 = 16
1 = Reserved
2 = Reserved
3 = 15

Bits 2, 3 Stencil Width:
0 = 0
1 = Reserved
2 = Reserved
3 = 1

LBReadMode

Name:	Localbuffer Read Mode	
Unit:	Localbuffer R/W	
Region: 0	Offset: Tag:	0×0000.8880 0×0110
Reset Value:	Undefined	
Read/write		



This register controls reading from localbuffer memory.

If you enable the read modes after the same data was just written with the read modes disabled, incorrect data can be read. To avoid this problem, send a WaitForCompletion command after enabling the read modes, but prior to the next primitive.

Bits 0–2 Partial Product 0 – See Appendix B, *Screen Widths Table*, for a table of values.

Bits 3–5 Partial Product 1 – See Appendix B, *Screen Widths Table,* for a table of values.

Bits 6–8 Partial Product 2 – See Appendix B, *Screen Widths Table,* for a table of values.

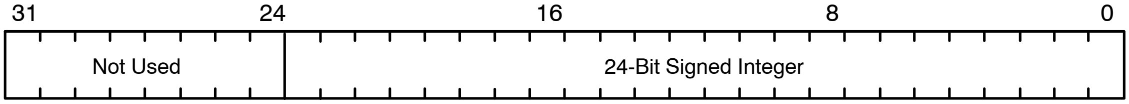
Bit 9 ReadSource Enable:
0 = No read
1 = Do read

Bit 10 ReadDestination Enable:
0 = No read
1 = Do read

Bits 16, 17	Data Type:	0 = Default 1 = Localbuffer stencil 2 = Localbuffer depth
Bit 18	Window Origin:	0 = Top left 1 = Bottom left
Bit 19	Patch Enable	0 = Disable 1 = Enable patched addressing of the localbuffer

LBSourceOffset

Name:	Localbuffer Source Offset	
Unit:	Localbuffer R/W	
Region: 0	Offset: Tag:	0×0000.8890 0×0112
Reset Value:	Undefined	
Read/write		

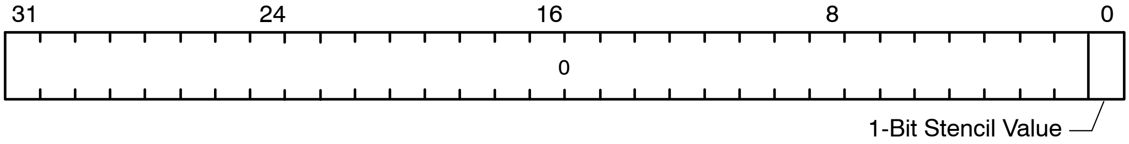


This register sets the offset value from destination to source for a copy operation in the localbuffer; that is:

```
source offset = destination address – source address
```

LBStencil

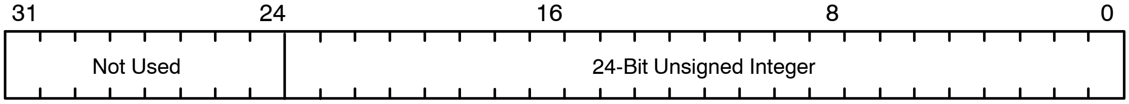
Name:	Localbuffer Stencil Upload	
Unit:	Localbuffer R/W	
Region: 0	Offset: Tag:	0×0000.88A8 0×0115
Reset Value:	Undefined	
Read/Output		



This register uploads stencil data from localbuffer memory. Do not write to this register. It gives the tag value and format of the data when it is read from the Host Out FIFO.

LBWindowBase

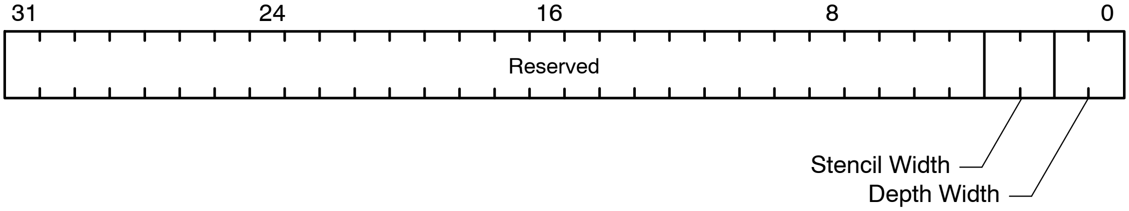
Name:	Localbuffer Window Base	
Unit:	Localbuffer R/W	
Region: 0	Offset: Tag:	0×0000.88B8 0×0117
Reset Value:	Undefined	
Read/write		



This register contains the current base address of the window in the localbuffer.

LBWriteFormat

Name:	Localbuffer Write Format	
Unit:	Localbuffer R/W	
Region: 0	Offset: Tag:	0×0000.88C8 0×0119
Reset Value:	Undefined	
Read/write		



This register specifies the format used when writing to localbuffer memory. The effect of setting a configuration with overlapping fields is undefined.

Bits 0, 1 Depth Width:

0 = 16
1 = Reserved
2 = Reserved
3 = 15

Bits 2, 3 Stencil Width:

0 = 0
1 = Reserved
2 = Reserved
3 = 1

LBWriteMode

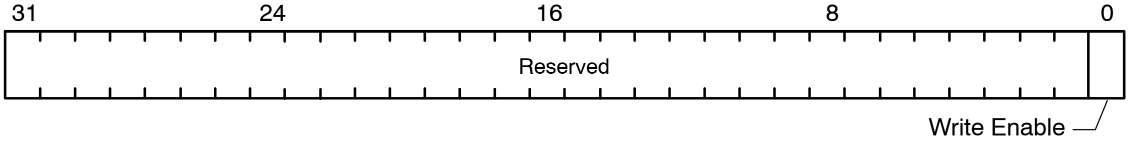
Name: Localbuffer Write Mode

Unit: Localbuffer R/W

Region: 0 Offset: 0×0000.88C0
Tag: 0×0118

Reset Value: Undefined

Read/write



This register controls writing to the localbuffer.

Bit 0 Write Enable:
0 = Disable
1 = Enable

LogicalOpMode

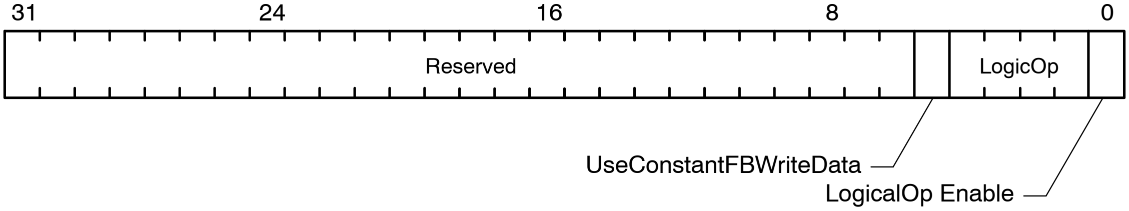
Name: Logic Op Mode

Unit: Logic Op

Region: 0 Offset: 0×0000.8828
Tag: 0×0105

Reset Value: Undefined

Read/write



This register controls logical operations on the framebuffer.

When you set the UseConstantFBWriteData bit to one, the FBWriteData register writes the color value rather than the fragment color to the framebuffer. This results in higher bandwidth in the framebuffer for flat-shaded primitives, but the mode can only be used when you disable logical operations (bit 0 cleared to 0).

Bit 0 LogicalOp Enable:
0 = Disable
1 = Enable

Bits 1−4 LogicOp:

Mode	Name	Operation	Mode	Name	Operation
0	CLEAR	0	8	NOR	−(S | D)
1	AND	S and D	9	EQUIV	−(S ^ D)
2	AND REVERSE	S and −D	10	INVERT	−D
3	COPY	S	11	OR REVERSE	S | −D
4	AND INVERTED	−S and D	12	COPY INVERT	−S
5	NO-OP	D	13	OR INVERT	−S | D
6	XOR	S ^ D	14	NAND	−(S and D)
7	OR	S | D	15	SET	1

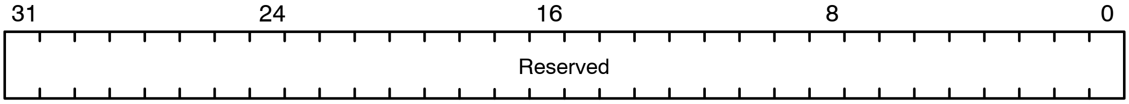
Where: S = Source (fragment) color, D = Destination (framebuffer) color.

Bit 5	UseConstantFBWriteData: 0 = Variable 1 = Constant

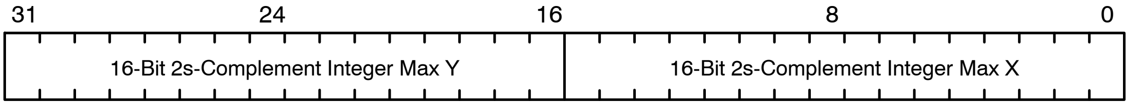
MaxHitRegion

Name:	Max Hit Region	
Unit:	Host Out	
Region: 0	Offset: Tag:	0×0000.8C30 0×0186
Reset Value:	Undefined	
Write		

The format of the data input is:



The format of the data output is:

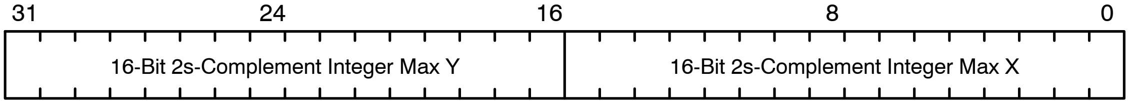

31
24
16
8
0
16-Bit 2s-Complement Integer Max Y
16-Bit 2s-Complement Integer Max X

This register passes the maximum coordinates of the hit region to the Host Out FIFO unless it is culled by the statistics bits in the FilterMode register.

The corresponding tag value output is: 0×186.

MaxRegion

Name:	Max Region	
Unit:	Host Out	
Region: 0	Offset: Tag:	0×0000.8C18 0×0183
Reset Value:	Undefined	
Read/write		


31
24
16
8
0
16-Bit 2s-Complement Integer Max Y
16-Bit 2s-Complement Integer Max X

This register has two uses:

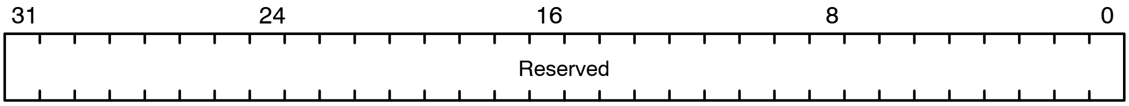
- During picking it contains the maximum (X,Y) value for the pick region.
- During extent collection, it is set to the initial minimum (X,Y) extent, and thereafter is updated whenever an eligible fragment is generated that has a higher X or Y value, using that higher value. Eligible fragments are either those that are written as pixels or those that are rasterized but culled from being drawn, as controlled by the StatisticMode register.

This register is unusual in that its contents are updated by the TVP4020 during rendering so that if read back, the contents are not necessarily the same as when originally stored.

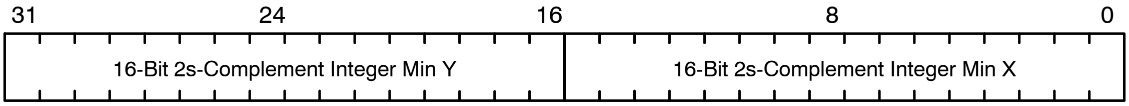
MinHitRegion

Name:	Min Hit Region	
Unit:	Host Out	
Region: 0	Offset:	0×0000.8C28
	Tag:	0×0185
Reset Value:	Undefined	
Write		

The format of the data input is:



The format of the data output is:

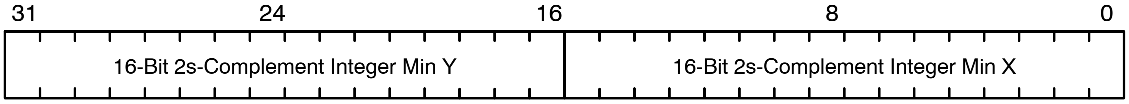

31
24
16
8
0
16-Bit 2s-Complement Integer Min Y
16-Bit 2s-Complement Integer Min X

This register passes the minimum coordinates of the hit region to the Host Out FIFO unless it is culled by the statistics bits in the FilterMode register.

The corresponding tag value output is: 0×185.

MinRegion

Name:	Min Region	
Unit:	Host Out	
Region: 0	Offset: Tag:	0×0000.8C10 0×0182
Reset Value:	Undefined	
Read/write		


31
24
16
8
0
16-Bit 2s-Complement Integer Min Y
16-Bit 2s-Complement Integer Min X

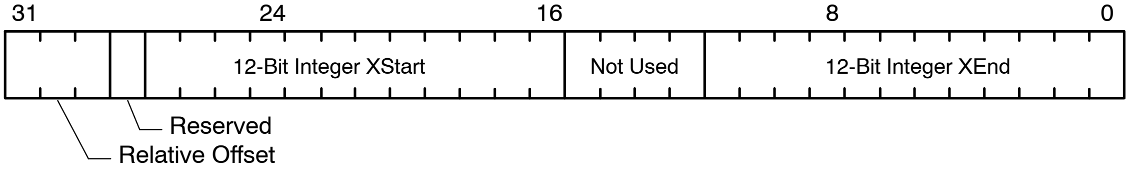
This register has two uses:

- ❑ During picking it contains the minimum (X,Y) value for the pick region.
- ❑ During extent collection, it is set to the initial maximum (X,Y) extent, and thereafter is updated whenever an eligible fragment is generated that has a lower X or Y value, using that lower value. Eligible fragments are either those that are written as pixels or those that are rasterized but culled from being drawn, as controlled by the StatisticMode register.

This register is unusual in that its contents are updated by the TVP4020 during rendering so that if it is read back, the contents are not necessarily the same as when originally stored.

PackedDataLimits

Name:	Packed copy limits	
Units:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.8150 0×002A
Reset Value:	Undefined	
Read/write		



This register sets the start and end limits in X for packed copies. Any pixels lying outside the specified range are not plotted. This register is only active when you enable the PackedData bit in FBReadMode.

Bits 0 – 11	XEnd: 12-bit 2s-complement value
Bits 16 – 27	XStart: 12-bit 2s-complement value
Bits 29–31	Relative Offset: 3-bit 2s-complement value, which specifies the number of pixels to which the source data must adjust, to align with the destination data. The FBReadMode register also has this field, and the last of these two registers to be loaded is effected.

PickResult

Name: Pick Result

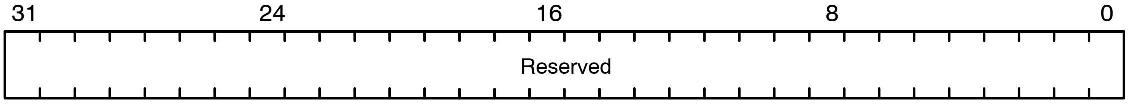
Unit: Host Out

Region: 0 Offset: 0×0000.8C38
Tag: 0×0187

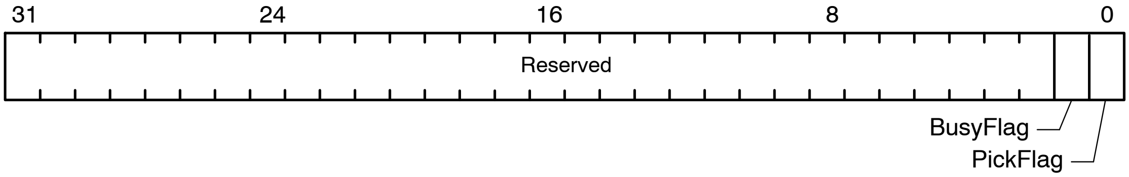
Reset Value: Undefined

Write

The format of the data input is:



The format of the data output is:



This register passes the current status of the picking result to the Host Out FIFO, unless it is culled by the statistics bits in the FilterMode register.

The corresponding tag value output is: 0×187

Bit 0	PickFlag:	
		0 = Miss 1 = Hit has occurred
Bit 1	BusyFlag:	
		0 = Idle 1 = Busy − used to validate the Pick Flag bit if this register is polled directly

QStart

Name: Initial Texture Q Value

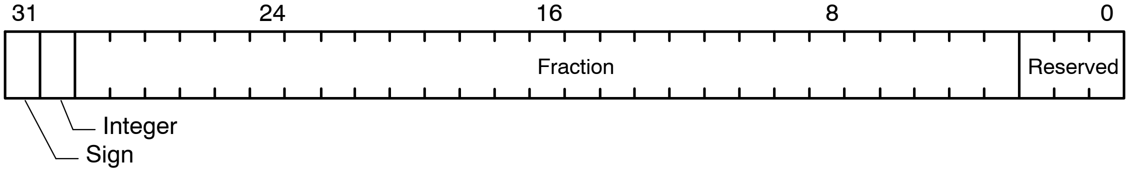
Unit: Texture Address

Region: 0 Offset: 0×0000.83B8
Tag: 0×0077

Reset Value: Undefined

Write

The format of the data input is:



This register sets the initial value for the Q coordinate during texture mapping. The value is in 2s-complement 2.27 fixed-point format.

RasterizerMode

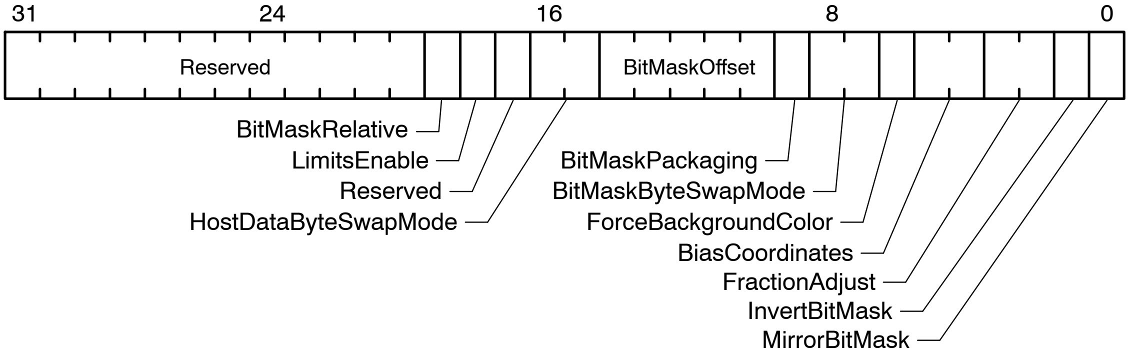
Name: Rasterizer Mode

Unit: Rasterizer

Region: 0 Offset: 0×0000.80A0
Tag: 0×0014

Reset Value: Undefined

Read/write



This register defines the long-term mode of operation of the rasterizer.

Bit 0 MirrorBitMask:
0 = Use bit mask from least to most significant bit
1 = Use bit mask from most to least significant bit

Bit 1 InvertBitMask:
0 = Test against bitmask
1 = Test against inverted bitmask

Bits 2, 3 FractionAdjust: These bits are for the Continue NewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted.
0 = No adjustment is made.
1 = Set the fraction bits to zero.
2 = Set the fraction bits to half.
3 = Set the fraction to nearly half; that is, 0×7FFF.

Bits 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 is added; that is, 0×7FFF.

Bit 6

ForceBackgroundColor: This bit controls operation of the bitmask test. If disabled, any fragment failing the test is discarded. If enabled, any fragment failing the test is drawn (other tests allowing), but the color is taken from the Texel0 register. It is used to support foreground/background colors.

0 = Disabled
1 = Enabled

Bits 7, 8

BitMaskByteSwapMode: These bits control byte swapping for the bitmask. Input ABCD.

0 = ABCD
1 = BADC
2 = CDAB
3 = DCBA

Bit 9

BitMaskPacking:

0 = Bitmask packed
1 = New data every scanline

Bits 10–14

BitMaskOffset: These bits set the position of the first bit to test in the bitmask.

Bits 15–16

HostDataByteSwapMode: These bits control byte swapping for host data. Input ABCD.

0 = ABCD
1 = BADC
2 = CDAB
3 = DCBA

Bit 18

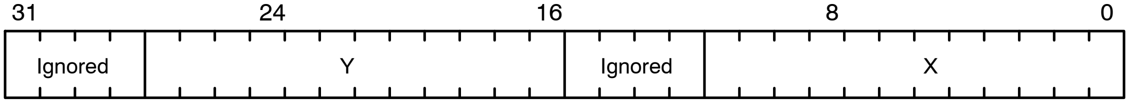
LimitsEnable: This bit enables X and Y limits checking.

0 = Disabled
1 = Enabled

Bit 19	BitMaskRelative: 0 = Bitmask indexed by counter 1 = Bitmask indexed by X position

RectangleOrigin

Name:	Rectangle Origin	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.80D0 0×001A
Reset Value:	Undefined	
Read/write		



This register defines the rectangle origin.

Bits 0−15	X origin of the rectangle to be drawn
Bits 16−31	Y origin of the rectangle to be drawn
Name:	Rectangle origin

RectangleSize

Name:	Rectangle Size	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.80D8 0×001B
Reset Value:	Undefined	
Read/write		

31 24 16 8 0

Ignored	Height	Ignored	Width

This register defines the rectangle size.

Bit 0−15	Width of the rectangle to be drawn
Bits 16−31	Height of the rectangle to be drawn

Render

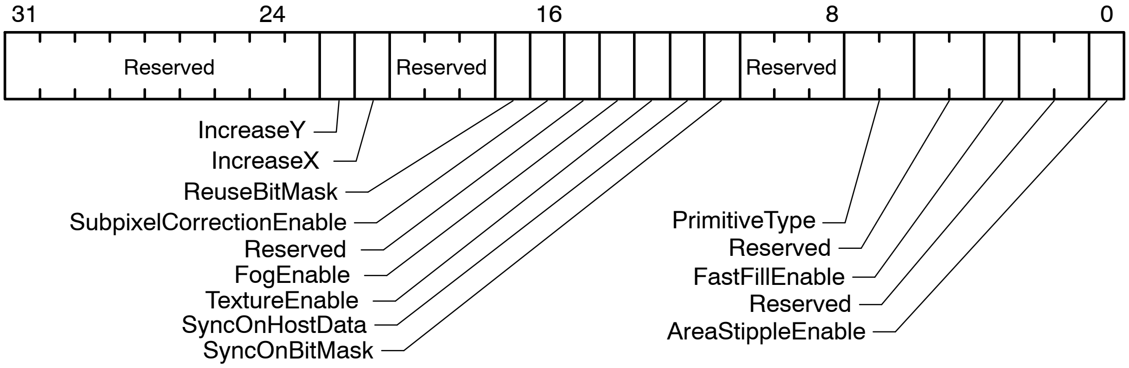
Name: Render

Unit: Rasterizer

Region: 0 Offset: 0×0000.8038
Tag: 0×0007

Reset Value: Undefined

Write



This register starts the rendering process.

The data field defines the short-term modes required by this primitive.

Bit 0 — AreaStippleEnable: Area stipple in the Stipple unit must also be enabled for stippling to occur.
0 = Disable
1 = Enable

Bit 3 — FastFillEnable:
0 = Disable block filling
1 = Enable block filling

Bits 6, 7 — PrimitiveType: These bits indicate the type of TVP4010 primitive to be drawn. The primitives supported and the corresponding codes are:
0 = Lines
1 = Trapezoids
2 = Points

Bit 11

SyncOnBitMask: This bit enables the bitmask test. Wait for the new bitmask when the current one expires unless SyncOnHostData or Reuse BitMask is enabled.

0 = Disable
1 = Enable

Bit 12

SyncOnHostData: When this bit is set, a fragment is produced only when one of the following registers is written to by the host: Depth, FBData, FBSourceData, Stencil, Color, or Texel0. The BitMaskPattern register is also written to if SyncOnBitMask is set.

0 = Disable
1 = Enable

Bit 13

TextureEnable: The Texture units must also be enabled for any texturing to occur.

0 = Disable
1 = Enable

Bit 14

FogEnable: The Fog Unit must also be enabled for any fogging to occur.

0 = Disable
1 = Enable

Bit 16

SubpixelCorrectionEnable: This bit enables the subpixel correction of color, depth, fog, and texture values at the start of a scanline span.

0 = Disable
1 = Enable

Bit 17

ReuseBitMask: This bit allows the bitmask to be reused when it expires. If enabled, the rasterizer does not wait for a new mask when the current one has been used.

0 = Disable
1 = Enable

Bits 18–20

Reserved

Bit 21

IncreaseX: This bit specifies that the rectangle primitive should be filled in the direction of increasing X.

0 = Disable
1 = Enable

Bit 22

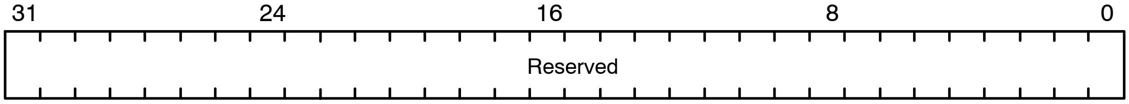
IncreaseY: This bit specifies that the rectangle primitive should be filled in the direction of increasing Y.

0 = Disable
1 = Enable

RepeatLine

Name:	Repeat Line	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9328 0×0265
Reset Value:	Undefined	

Write



This register is not used.

RepeatTriangle

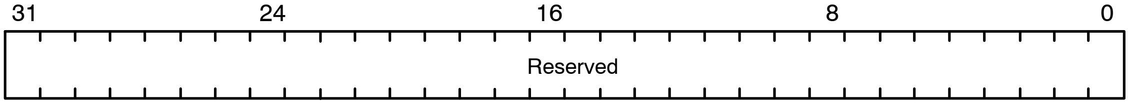
Name:	Repeat Triangle	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9310 0×0262
Reset Value:	Undefined	
Write		

31	24	16	8	0
Reserved				

This register is not used.

ResetPickResult

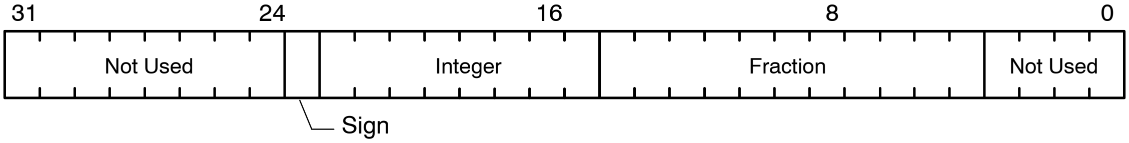
Name:	Reset Pick Result	
Units:	Host Out	
Region: 0	Offset: Tag:	0×0000.8C20 0×0184
Reset Value:	Undefined	
Write		



This register resets the current value of the picking result to zero. The data field is not used.

RStart

Name:	Initial Red Color	
Unit:	Color DDA	
Region: 0	Offset: Tag:	0×0000.8780 0×00F0
Reset Value:	Undefined	
Read/write		



This register determines the initial red value for a vertex when Gouraud-shading mode is set. The value is 2s-complement 9.11 fixed-point format.

ScissorMaxXY

Name:	Scissor Rectangle – Maximum XY
Unit:	Scissor/Stipple
Region: 0	Offset: 0×0000.8190 Tag: 0×0032
Reset Value:	Undefined

Read/write

31	24 16		8 0
Not Used	12-Bit 2s-Complement Max Y	Not Used	12-Bit 2s-Complement Max X

This register specifies the user scissor-rectangle corner farthest from the screen origin.

ScissorMinXY

Name:	Scissor Rectangle – Minimum XY	
Unit:	Scissor/Stipple	
Region: 0	Offset: Tag:	0×0000.8188 0×0031
Reset Value:	Undefined	
Read/write		

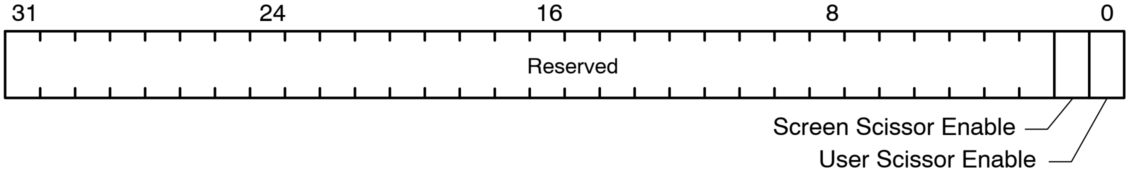
31 24 16 8 0

Not Used	12-Bit 2s-Complement Min Y	Not Used	12-Bit 2s-Complement Min X

This register specifies the user scissor-rectangle corner closest to the screen origin.

ScissorMode

Name:	Scissor Mode	
Unit:	Scissor/Stipple	
Region: 0	Offset: Tag:	0×0000.8180 0×0030
Reset Value:	Undefined	
Read/write		



This register controls enabling of the screen and user scissor tests.

Bit 0 User Scissor Enable:
0 = Disable
1 = Enable

Bit 1 Screen Scissor Enable:
0 = Disable
1 = Enable

ScreenSize

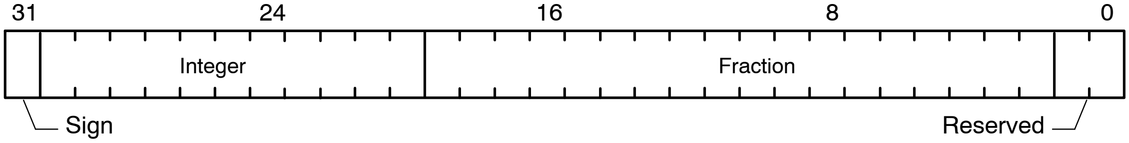
Name:	Screen Size
Unit:	Scissor/Stipple
Region: 0	Offset: 0×0000.8198 Tag: 0×0033
Reset Value:	Undefined
Read/write	

31	24	16	8 0
Not Used	11-Bit Unsigned Integer Height	Not Used	11-Bit Unsigned Integer Width

This register sets screen dimensions for screen-scissor clipping. The screen boundaries are (0, 0) to (width – 1, height – 1), inclusive.

SStart

Name:	Initial Texture S Value	
Unit:	Texture Address	
Region: 0	Offset:	0×0000.8388
	Tag:	0×0071
Reset Value:	Undefined	
Read/write		



This register sets the initial value for the S coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

StartXDom

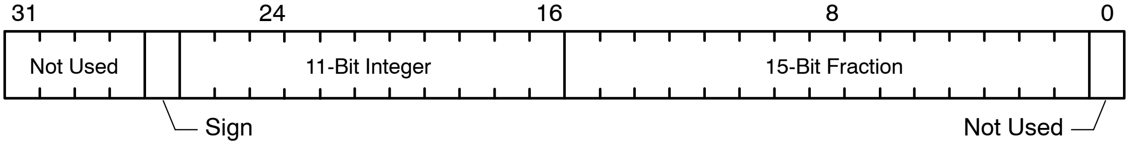
Name: Start X Value – Dominant Edge

Unit: Rasterizer

Region: 0 Offset: 0×0000.8000
Tag: 0×0000

Reset Value: Undefined

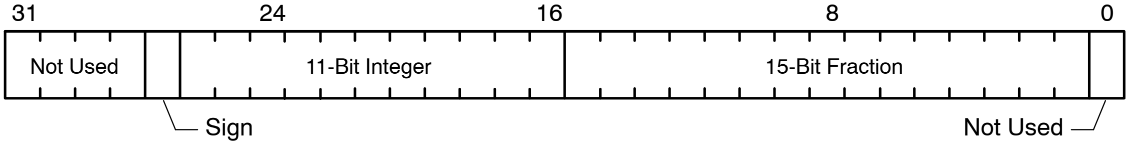
Read/write



This register sets the initial X value for the dominant edge in trapezoid filling or the initial X value in line drawing. The value is in 2s-complement 12.15 fixed-point format.

StartXSub

Name:	Start X Value – Subordinate Edge	
Unit:	Rasterizer	
Region: 0	Offset: Tag:	0×0000.8010 0×0002
Reset Value:	Undefined	
Read/write		



This register sets the initial X value for the subordinate edge in trapezoid filling. The value is in 2s-complement 12.15 fixed-point format.

StartY

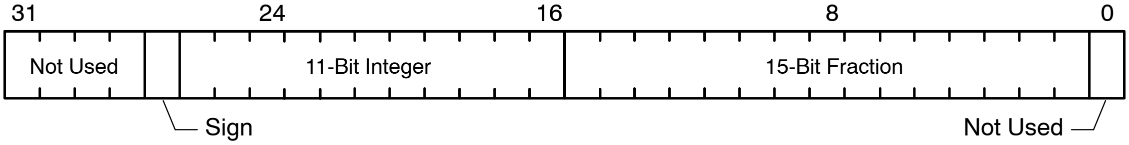
Name: Start Y Value

Unit: Rasterizer

Region: 0 Offset: 0×0000.8020
Tag: 0×0004

Reset Value: Undefined

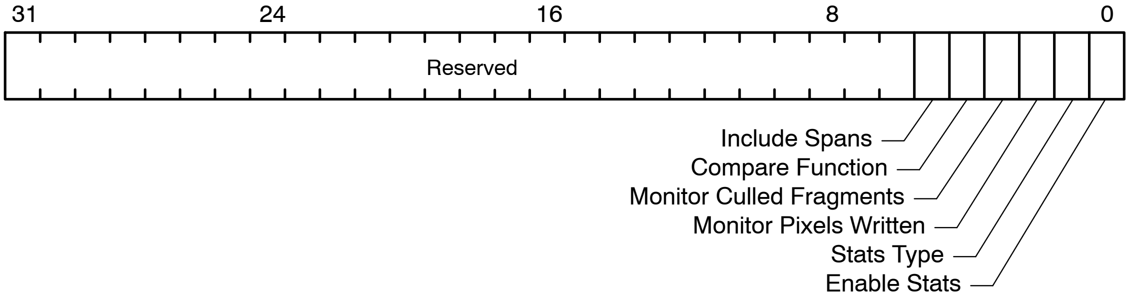
Read/write



This register sets the initial scanline in trapezoid filling or the initial Y position for line drawing. The value is in 2s-complement 12.15 fixed-point format.

StatisticMode

Name:	Statistic Mode	
Unit:	Host Out	
Region: 0	Offset: Tag:	0×0000.8C08 0×0181
Reset Value:	Undefined	
Read/write		



This register controls the mode of statistics collection.

Bit 0 — Enable Stats:
0 = Disable statistics collection
1 = Enable statistics collection

Bit 1 — Stats Type:
0 = Picking mode
1 = Extent collection

Bit 2 — Monitor Pixels Written (active steps):
0 = Excludes pixels that were drawn
1 = Includes pixels that were drawn

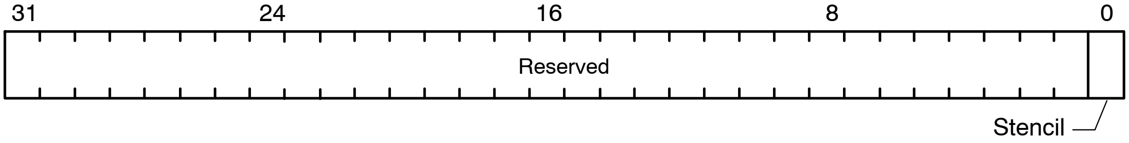
Bit 3 — Monitor Culled Fragments (passive steps):
0 = Excludes fragments that were culled from being drawn
1 = Includes fragments that were culled from being drawn

Bit 4 — Compare Function:
0 = Inside region
1 = Outside region

Bit 5	Include Spans:	
		0 = Exclude block filled spans
		1 = Include block filled spans

Stencil

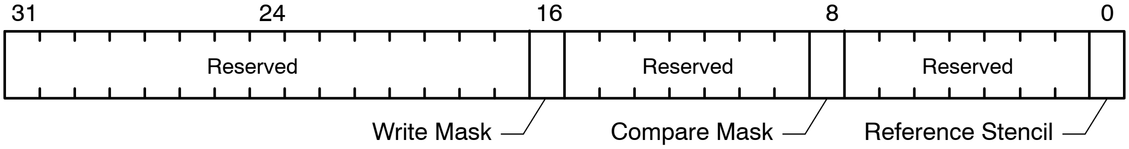
Name:	Stencil	
Unit:	Stencil/Depth	
Region: 0	Offset Tag:	0×0000.8998 0×0133
Reset Value:	Undefined	
Read/write		



This register sets the stencil value used to clear the stencil buffer or draw a primitive where the host supplies the stencil value.

StencilData

Name:	Stencil Data	
Unit:	Stencil/Depth	
Region: 0	Offset: Tag:	0×0000.8990 0×0132
Reset Value:	Undefined	
Read/write		



This register holds data used in the stencil test.

The stencil writemask controls which stencil planes are updated as a result of the test.

Bit 0 Reference Stencil: This bit is the reference value for the stencil test.

Bit 8 Compare Mask: This bit is the mask used to determine which bits are significant in the comparison.

Bit 16 Write Mask: This bit is the mask used to determine which bits in the localbuffer are updated.

StencilMode

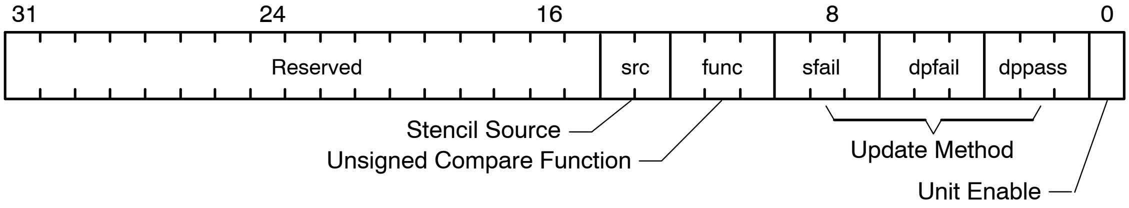
Name: Stencil Mode

Unit: Stencil/Depth

Region: 0 Offset: 0×0000.8988
Tag: 0×0131

Reset Value: Undefined

Read/write



This register 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 value is less and the result is true, then the fragment value is less than the source value.

Bit 0 Unit Enable:
0 = Disable
1 = Enable

Bits 1 − 3 These bits set the update method if the depth test and stencil test pass.

Bits 4 − 6 These bits set the update method if the depth test fails and stencil test passes.

Bits 7 − 9 These bits set the update method if the 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

Bits 10 – 12 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

Bits 13 – 14 Stencil Source:

0 = Test Logic
1 = Stencil Register
2 = LBData
3 = LBSourceData

SuspendUntilFrameblank

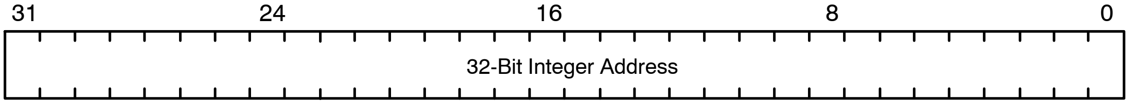
Name: Suspend until frameblank

Unit: Framebuffer R/W

Region: 0 Offset: 0×0000.8C78
Tag: 0×018F

Reset Value: Undefined

Write

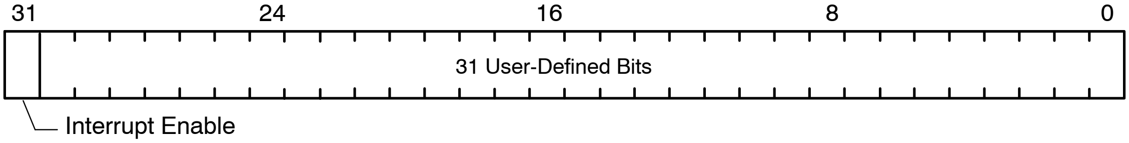


This register flushes all outstanding framebuffer write operations and then suspends framebuffer accesses until the next frameblank period. The data field is the start address of the next frame to be displayed. This address is used from the next frameblank until a new address is supplied.

Bits 0 – 31 Integer Address

Sync

Name:	Synchronization	
Unit:	Host Out	
Region: 0	Offset:	0×0000.8C40
	Tag:	0×0188
Reset Value:	Undefined	
Write		



This command register synchronizes the TVP4010 with the host. It flushes outstanding TVP4010 operations such as pending memory accesses. It additionally passes the current status of the picking result to the Host Out FIFO, unless it is culled by the statistics bits in the FilterMode register.

Bits 0–30	User Defined
Bit 31	Interrupt Enable: 0 = Disable interrupt for this command. 1 = Enable interrupt for this command.

The data output is the value written to the register by the sync command. If interrupts are enabled, then the interrupt does not occur until the tag and/or data is written to the output FIFO.

The corresponding tag value output is: 0×188.

Texel0

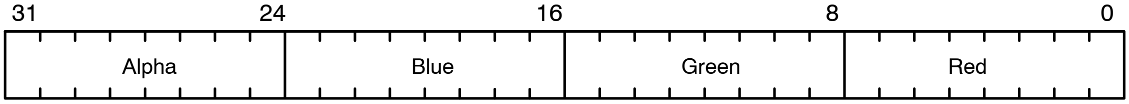
Name: Texel Value

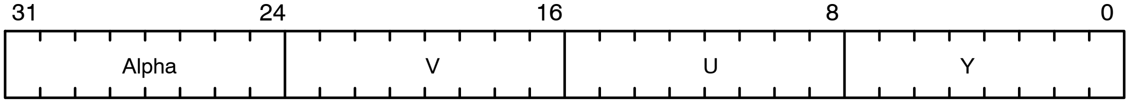
Unit: Texture/Fog/Blend

Region: 0 Offset: 0×0000.8600
Tag: 0×00C0

Reset Value: Undefined

Read/write



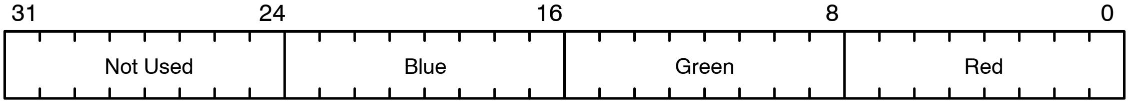


This register sets the texel value to be loaded using the rasterizer SyncOnHostData mode. It is used for direct application of procedural textures. It is also used when downloading YUV data that needs to be converted to RGB; the YUV conversion is done on the contents of this register.

This register also supplies background color if ForceBackgroundColor is enabled in either the RasterizerMode or the AreaStippleMode register.

TexelLUT(0..15)

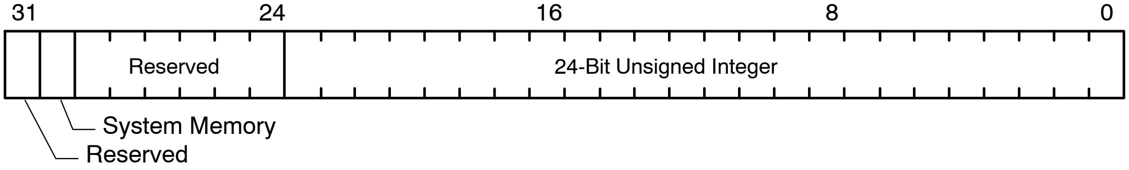
Name:	Texel LUT entries 0 to 15
Unit:	Texture Read
Region: 0	Offset: 0×0000.8E80...0×0000.8EF8 Tag: 0×01D0...0×1DF
Reset Value:	Undefined
Read/write	



This register sets the value to load into the specified texel look-up table entry.

TexelLUTAddress

Name:	Address of LUT in memory	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.84D0 0×009A
Reset Value:	Undefined	
Read/write		



This register stores the address of 32-bit units of memorized data to load into the texture look-up table. If bit 30 is set, the LUT resides in system memory rather than in the localbuffer and should be loaded across the PCI bus. Bit 31 is ignored if this register is loaded either directly or indirectly by the TEXELLUTID register.

TexelLUTData

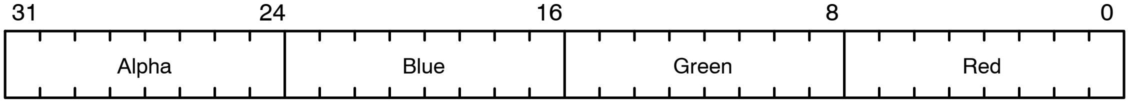
Name: Data for texture LUT

Unit: Texture Read

Region: 0 Offset: 0×0000.84C8
Tag: 0×0099

Reset Value: Undefined

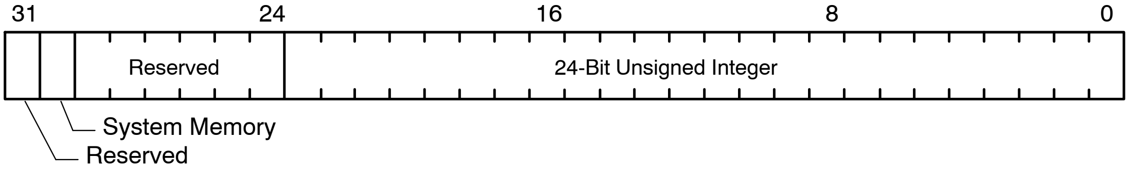
Read/write



This register contains the data to be loaded into the texture look-up table.

TexelLUTID

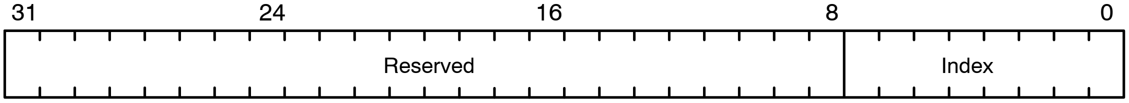
Name:	Indirect handle for texture LUT	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8F78 0×001EF
Reset Value:	Undefined	
Read/write		



This register contains the 24-bit field that holds the address of the data to be loaded into the TexelLUTAddress register. If bit 30 is set, then this data resides in system memory and should be fetched through the PCI bus.

TexelLUTIndex

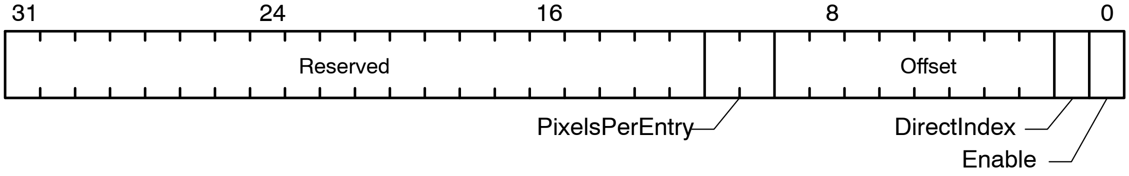
Name:	Index data for LUT	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.84C0 0×00098
Reset Value:	Undefined	
Read/write		



This register indexes texel LUT data. The index is in the lower 8 bits of the register and auto-increments after every write transaction to the TexelLUTData register. Reading from the TexelLUTIndex register returns the auto-incremented value if there was a write transaction to the TexelLUTData register. A side effect of reading the TexelLUTIndex register is that the internal counter, which generates the LUT index when reading from the TexelLUTData register, resets.

TexelLUTMode

Name:	Texel LUT Mode	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8678 0×00CF
Reset Value:	Undefined	
Read/write		

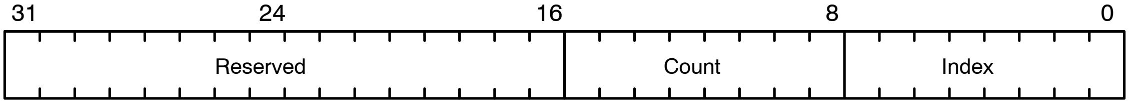


This register specifies the organization of the texture map in memory.

Bit 0	Enable:	
		0 = No 1 = Lookup
Bit 1	DirectIndex:	
		0 = Index from texture data 1 = Index from fragment XY values
Bits 2−9	Offset:	0×0000: Offset to add index in DirectIndex mode.
Bits 10−11	PixelsPerEntry: number of pixels per entry in LUT	
		0 = 1 pixel 1 = 2 pixels 2 = 4 pixels

TexelLUTTransfer

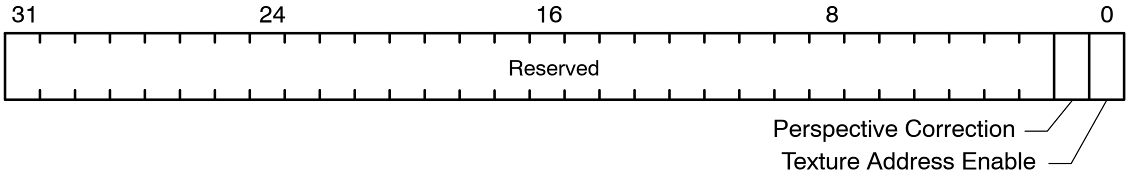
Name:	Initiates loading of LUT data from memory	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.84D8 0×0009B
Reset Value:	Undefined	
Read/write		



This register contains an index field that specifies the first entry to load in the LUT and a count field that specifies the number of entries to load.

TextureAddressMode

Name:	Texture Address Mode	
Unit:	Texture Address	
Region: 0	Offset: Tag:	0×0000.8380 0×0070
Reset Value:	Undefined	
Read/write		



This register controls the calculation of texture addresses.

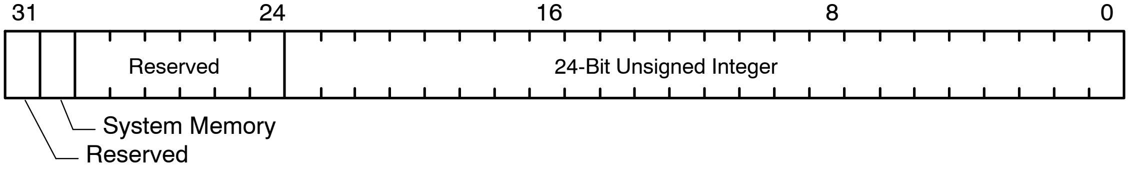
If you set bit 1, the TVP4020 performs accurate perspective correction.

Bit 0 Texture Address Enable:
0 = Disable
1 = Enable

Bit 1 Perspective Correction:
0 = Disable
1 = Enable

TextureBaseAddress

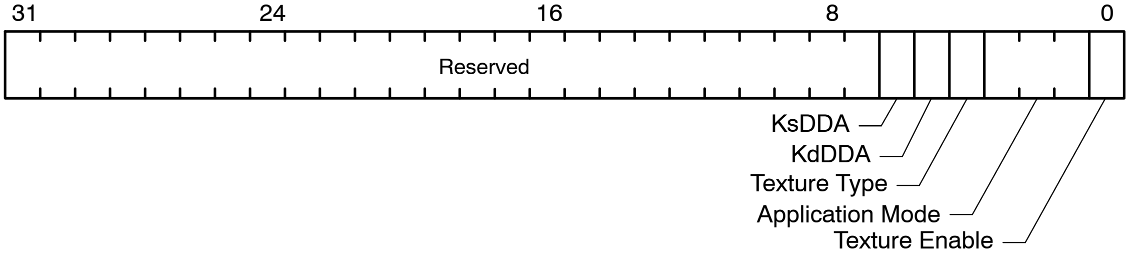
Name:	Texture Base Address	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8580 0×00B0
Reset Value:	Undefined	
Read/write		



This register determines the base address of the texture map. The address is specified in texels from the base of the memory. If bit 30 is set, then the texture resides in the system memory rather than in the localbuffer and should be fetched through the PCI bus. Bit 31 is ignored if the register is loaded directly. If it is loaded indirectly by the TextureID register, bit 31 indicates that the address is invalid and should not be used.

TextureColorMode

Name:	Texture Color Mode	
Unit:	Texture/Fog/Blend	
Region: 0	Offset: Tag:	0×0000.8680 0×00D0
Reset Value:	Undefined	
Read/write		



This register controls the application of texture. The KsDDA and KdDDA bits enable the internal DDAs. Set them for modulate or highlight ramp texture application modes. The texture type field differentiates between ramp (Apple) and RGB (OpenGL™) application modes. With ramp application mode, you can apply various modes simulataneously, for example, decal with highlight.

To texture-map a primitive, you must set the TextureEnable bit in the Render command register.

Bit 0 Texture Enable:

0 = Disable
1 = Enable texture application

Bits 1 – 3 Application Mode:

RGB	Ramp
0 = Modulate	Bit 1 = Decal
1 = Decal	Bit 2 = Modulate
2 = Reserved	Bit 3 = Highlight
3 = Copy	
4 = Modulate + Highlight	
5 = Decal + Highlight	
6 = Reserved	
7 = Copy + Highlight	

Bit 4 Texture Type:

0 = RGB
1 = Ramp

Bit 5 KdDDA:

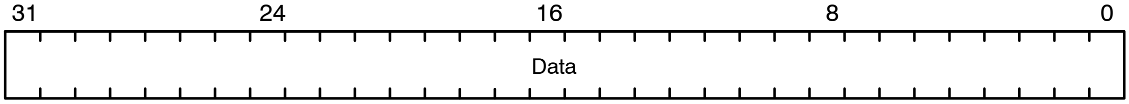
0 = Disable
1 = Enable

Bit 6 KsDDA:

0 = Disable
1 = Enable

TextureData

Name:	Texture Data	
Unit:	Framebuffer R/W	
Region: 0	Offset:	0×0000.88E8
	Tag:	0×011D
Reset Value:	Undefined	
Write		

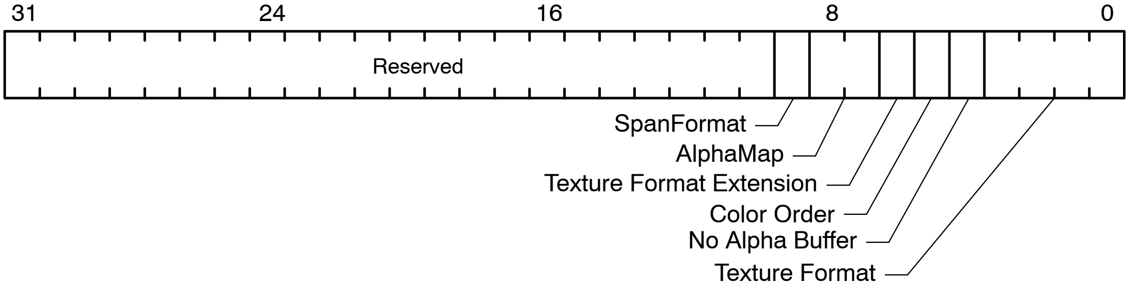


This register is used with TextureDownloadOffset to load raw texture data into memory. This may include multiple texels, depending on the texel size.

Bits 0 – 31 Data

TextureDataFormat

Name:	Texture Data Format	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8590 0×00B2
Reset Value:	Undefined	
Read/write		



This register specifies the color format of the texture map in memory. See the following register descriptions for the bit fields.

Bits 0 – 3 Texture Format:

Format			**Internal Color Channel**			
(see Note)	**Color Order**	**Name**	**R/Y**	**G/U**	**B/V**	**A**
0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
1	BGR	5:5:5:1 Front	5@0	5@5	5@10	1@15
2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
5	BGR	3:3:2 Front	3@0	3@3	2@6	0
6	BGR	3:3:2 Back	3@8	3@11	2@14	0
9	BGR	2:3:2:1 Front	2@0	3@2	2@5	1@7
10	BGR	2:3:2:1 Back	2@8	3@10	2@13	1@15
11	BGR	2:3:2 FrontOff	2@0	3@2	2@5	0
12	BGR	2:3:2 BackOff	2@8	3@10	2@13	0
13	BGR	5:5:5:1 Back	5@16	5@21	5@26	1@31
14	BGR	CI8	8@0	0	0	0
15	BGR	CI4	4@0	0	0	0
16	BGR	5:6:5 Front	5@0	6@5	5@11	0
17	BGR	5:6:5 Back	5@16	6@21	5@27	0
18	BGR	YUV411	8@0	8@8	8@16	8@24
19	BGR	YUV422	8@0	8@8	8@8	0
0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
1	RGB	5:5:5:1 Front	5@10	5@5	5@0	1@15
2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
5	RGB	3:3:2 Front	3@5	3@2	2@0	0
6	RGB	3:3:2 Back	3@13	3@10	2@8	0
9	RGB	2:3:2:1 Front	2@5	3@2	2@0	1@7
10	RGB	2:3:2:1 Back	2@13	3@10	2@8	1@15
11	RGB	2:3:2 FrontOff	2@5	3@2	2@0	0
12	RGB	2:3:2 BackOff	2@13	3@10	2@8	0
13	RGB	5:5:5:1 Back	5@26	5@21	5@16	1@31
14	RGB	CI8	8@0	0	0	0
15	RGB	CI4	4@0	0	0	0
16	RGB	5:6:5 Front	5@11	6@5	5@0	0

Bits 0 – 3 Texture Format:

Format			**Internal Color Channel**			
(see Note)	**Color Order**	**Name**	**R/Y**	**G/U**	**B/V**	**A**
17	RGB	5:6:5 Back	5@27	6@21	5@16	0
18	RGB	YUV411	8@16	8@8	8@0	8@24
19	RGB	YUV422	8@8	8@0	8@0	0

Note: The format column is also dependant on bit16. n@m means n bits starting at bit m. Front and back modes replicate the color value to assist with double buffering. CI values are replicated into each byte to assist with double buffering. Offset modes have 64 added to the seven-bit formatted value. If the format has no alpha bits, the alpha field defaults to 0×F8.

Bit 4 No Alpha Buffer:

0 = Alpha buffer present
1 = Alpha buffer not present

Bit 5 Color Order:

0 = BGR
1 = RGB

Bit 6 Texture Format Extension: The most significant bit extension to texture format is held in bits 0 – 3.

Bits 7, 8 AlphaMap:

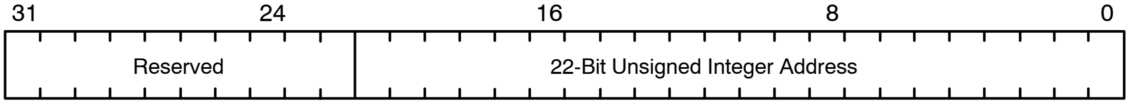
0 = None
1 = Include: Pass texels that lie within the AlphaMap bounds
2 = Exclude: Fail texels that lie within the AlphaMap bounds

Bit 9 SpanFormat: Used to control the data format of a texture map holding block fill masks

0 = Normal
1 = Flip: Mirror the bits within each byte

TextureDownloadOffset

Name:	Texture Download Offset	
Unit:	Framebuffer R/W	
Region: 0	Offset: Tag:	0×0000.88F0 0×011E
Reset Value:	Undefined	
Write/Read		

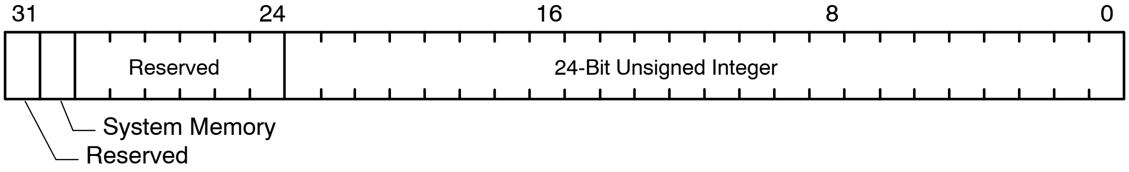


This register contains the 32-bit aligned address at which the texture load will start. Each write to TextureData increments this value by one after storage takes place. If this register is read back, it does not necessarily contain the same value as the written value.

Bits 0 – 21 Unsigned Integer Address

TextureID

Name:	Indirect handle for texture map	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8F70 0×001EE
Reset Value:	Undefined	
Write/Read		



This register contains a 24-bit field that holds the address of the data to be loaded into the TextureBaseAddress register. If bit 30 is set, then the data resides in system memory and should be fetched through the PCI bus.

TextureMapFormat

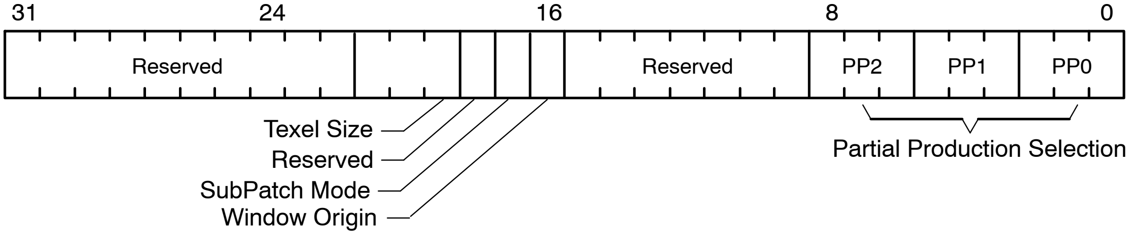
Name: Texture Map Format

Unit: Texture Read

Region: 0 Offset: 0×0000.8588
Tag: 0×00B1

Reset Value: Undefined

Read/write



This register specifies the organization of the texture map in memory.

The enabling of subpatch addressing improves the performance of texture mapping in typical situations.

Bits 0 – 2 Partial Product 0: See Appendix B, *Screen Widths Table*, for a table of values.

Bits 3 – 5 Partial Product 1: See Appendix B, *Screen Widths Table* for a table of values.

Bits 6 – 8 Partial Product 2: See Appendix B, *Screen Widths Table* for a table of values.

Bit 16 Window Origin:
0 = Top
1 = Bottom left

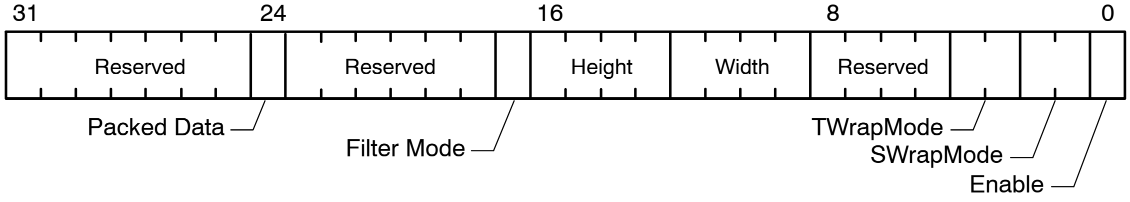
Bit 17 SubPatch Mode:
0 = Disable
1 = Enable

Bits 19–20 Texel Size:

0 = 8 bits
1 = 16 bits
2 = 32 bits
3 = 4 bits
4 = 24 bits

TextureReadMode

Name:	Texture Read Mode	
Unit:	Texture Read	
Region: 0	Offset: Tag:	0×0000.8670 0×00CE
Reset Value:	Undefined	
Read/write		



This register controls texture read operations. When you set filter mode, bilinear texture mapping is performed; otherwise, nearest-neighbor texture mapping occurs. The SWrapMode and TWrapMode specify the action to take when the S and T coordinates fall outside the required range. The clamp action clamps a single image onto an object during texture mapping. The repeat action repeats the texture pattern, whereas the mirror action alternatively reverses the texture pattern. The Packed Data bit defines how texels are read from memory. If you clear this bit, each texel is read one at a time; if you set it, several texels can be read simultaneously, improving efficiency. The actual number of texels read in this case is dependent on the texel size.

Bit 0	Enable:	0 = Disable texture reads. 1 = Enable texture reads.
Bits 1 , 2	SWrapMode:	0 = Clamp 1 = Repeat 2 = Mirror
Bits 3 , 4	TWrapMode:	0 = Clamp 1 = Repeat 2 = Mirror
Bits 9 – 12	Width: log2 texture map width	

Bits 13 – 16	Height: log2 texture map height
Bit 17	Filter Mode: 0 = Disable bilinear texture filtering. 1 = Enable bilinear texture filtering.
Bit 24	Packed Data: 0 = Off 1 = On

TStart

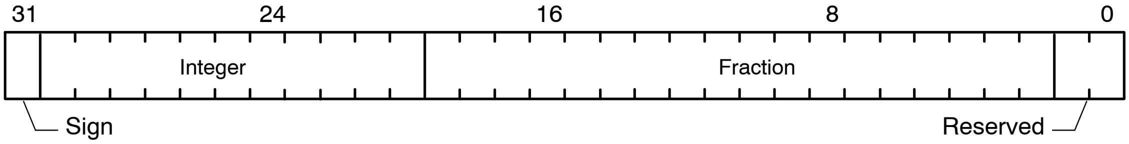
Name: Initial Texture T Value

Unit: Texture Address

Region: 0 Offset: 0×0000.83A0
Tag: 0×0074

Reset Value: Undefined

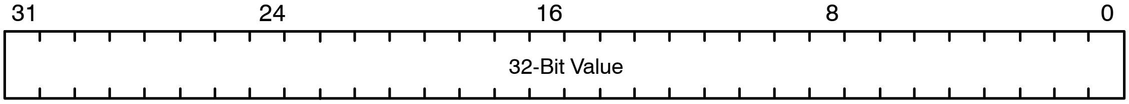
Read/write



This register sets the initial value for the T coordinate during texture mapping. The value is in 2s-complement 12.18 fixed-point format.

V0Fixed[0..15]

Name:	Vertex 0 data	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9000...9078 0×00200...0x0020F
Reset Value:	Undefined	
Read/write		



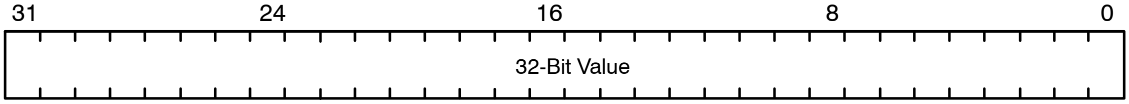
This register contains the data for vertex 0. The valid entries are listed below.

Offset	Category	Parameter	Fixed-Point Format
0	Texture	s	2.30 s (see Note)
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	
7		blue	
8		alpha	
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Note: This is the range where normalize is not used. When normalize is enabled, the fixed-point format can be anything, providing it is the same for the s, t, and q parameters. The numbers are interpreted as if they had the 2.30 format for conversion to floating point. If the fixed-point format (2.30) is different from what the user had in mind, then the input values are prescaled by a fixed amount (that is, the difference in binary point positions) prior to conversion.

V1Fixed[0..15]

Name:	Vertex 1 data	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9080...90F8 0×00210...0x0021F
Reset Value:	Undefined	
Read/write		



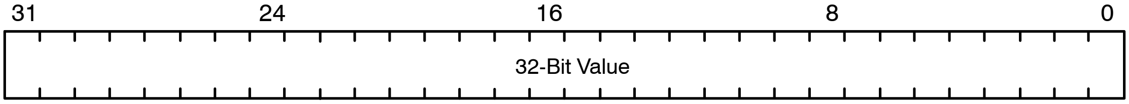
This register contains the data for vertex 1. The valid entries are listed below.

Offset	Category	Parameter	Fixed-Point Format
0	Texture	s	2.30 s (see Note)
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	
7		blue	
8		alpha	
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Note: This is the range where normalize is not used. When normalize is enabled, the fixed-point format can be anything, providing it is the same for the s, t, and q parameters. The numbers are interpreted as if they had the 2.30 format for conversion to floating point. If the fixed-point format (2.30) is different from what the user had in mind, then the input values are prescaled by a fixed amount (that is, the difference in binary point positions) prior to conversion.

V2Fixed[0..15]

Name:	Vertex 2 data	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9100...9178 0×00220...0x0022F
Reset Value:	Undefined	
Read/write		



This register contains the data for vertex 2. The valid entries are listed below.

Offset	Category	Parameter	Fixed-Point Format
0	Texture	s	2.30 s (see Note)
1		t	2.30 s
2		q	2.30 s
3		Ks	2.22 us
4		Kd	2.22 us
5	Color	red	1.30 us
6		green	
7		blue	
8		alpha	
9	Fog	f	10.22 us
10	Coordinate	x	16.16 s
11		y	16.16 s
12		z	1.30 us
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Note: This is the range where normalize is not used. When normalize is enabled, the fixed-point format can be anything, providing it is the same for the s, t, and q parameters. The numbers are interpreted as if they had the 2.30 format for conversion to floating point. If the fixed-point format (2.30) is different from what the user had in mind, then the input values are prescaled by a fixed amount (that is, the difference in binary point positions) prior to conversion.

V0Float[0..15]

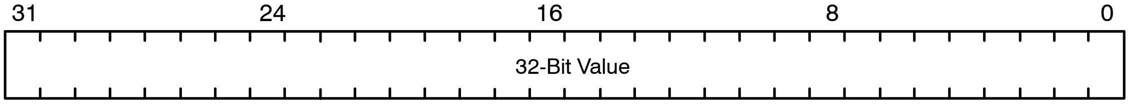
Name: Vertex 0 data

Unit: Delta

Region: 0 Offset: 0×0000.9180, 0×0000.91F8
Tag: 0×00230, 0x0023F

Reset Value: Undefined

Read/write



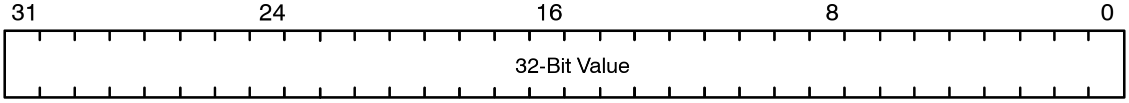
This register contains the data for vertex 0. The valid entries are listed below.

Offset	Category	Parameter	IEEE Single Precision Floating-Point Range
0	Texture	s	−1.0 . . . 1.0
1		t	−1.0 . . . 1.0
2		q	−1.0 . . . 1.0
3		Ks	0.0 . . . 2.0
4		Kd	0.0 . . . 1.0
5	Color	red	0.0 . . . 1.0
6		green	
7		blue	
8		alpha	
9	Fog	f	−512.0 . . . 512.0
10	Coordinate	x	−32K . . . +32K (see Notes 1 and 2)
11		y	−32K . . . +32K (see Notes 1 and 2)
12		z	0.0 . . . 1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Notes: 1) The normal range here is limited by the size of the screen.
2) K = 1024

V1Float[0..15]

Name:	Vertex 1 data	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9200, 0×0000.8278 0×00240, 0×0024F
Reset Value:	Undefined	
Read/write		



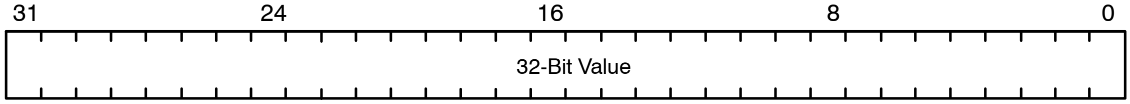
This register contains the data for vertex 1. The valid entries are listed below.

Offset	**Category**	**Parameter**	**IEEE Single Precision Floating-Point Range**
0	Texture	s	−1.0 . . . 1.0
1		t	−1.0 . . . 1.0
2		q	−1.0 . . . 1.0
3		Ks	0.0 . . . 2.0
4		Kd	0.0 . . . 1.0
5	Color	red	0.0 . . . 1.0
6		green	
7		blue	
8		alpha	
9	Fog	f	−512.0 . . . 512.0
10	Coordinate	x	−32K . . . +32K (see Notes 1 and 2)
11		y	−32K . . . +32K (see Notes 1 and 2)
12		z	0.0 . . . 1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Notes: 1) The normal range here is limited by the size of the screen.
2) K = 1024

V2Float[0..15]

Name:	Vertex 2 data	
Unit:	Delta	
Region: 0	Offset: Tag:	0×0000.9280, 0×0000.92F8 0×00250, 0×0025F
Reset Value:	Undefined	
Read/write		



This register contains the data for vertex 2. The valid entries are listed below.

Offset	Category	Parameter	IEEE Single Precision Floating-Point Range
0	Texture	s	−1.0 . . . 1.0
1		t	−1.0 . . . 1.0
2		q	−1.0 . . . 1.0
3		Ks	0.0 . . . 2.0
4		Kd	0.0 . . . 1.0
5	Color	red	0.0 . . . 1.0
6		green	
7		blue	
8		alpha	
9	Fog	f	−512.0 . . . 512.0
10	Coordinate	x	−32K . . . +32K (see Notes 1 and 2)
11		y	−32K . . . +32K (see Notes 1 and 2)
12		z	0.0 . . . 1.0
13		Reserved	Reserved
14	PackedColor	PackedColor	8888

Notes: 1) The normal range here is limited by the size of the screen.
2) K = 1024

WaitForCompletion

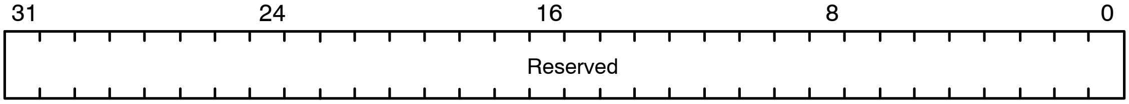
Name: Wait for Completion

Unit: Rasterizer

Region: 0 Offset: 0×0000.8088
Tag: 0×0017

Reset Value: Undefined

Write

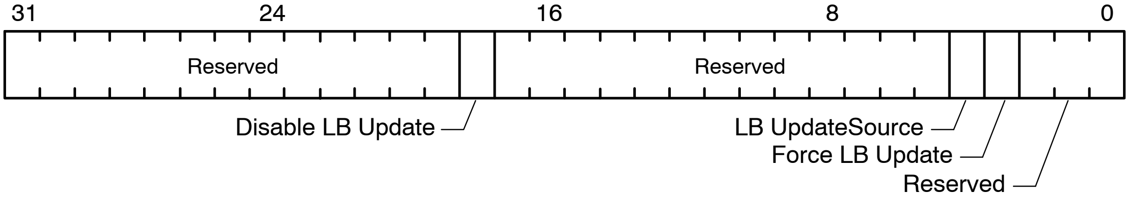


This command register suspends the TVP4020 operation until all framebuffer writes operations are complete. It is useful when separating, for example, a texture download from subsequent primitives.

Bits 0–31 Reserved

Window

Name:	Window	
Unit:	Stencil/Depth	
Region: 0	Offset:	0×0000.8980
	Tag:	0×0130
Reset Value:	Undefined	
Read/write		



This register sets the Force LB Update bit to override the stencil and depth tests. Enabling this bit forces the update of the localbuffer. You must still enable the write modes in the LBWriteMode register. When this bit clears, any update is conditional on the outcome of the stencil and depth tests.

If you set the Disable LB Update bit, the results of the stencil and depth tests are overridden and the localbuffer is not updated, even if you enabled localbuffer write modes. When you disable write modes in LBWriteMode, there may be a performance advantage in also setting Disable LB Update.

Bit 3	Force LB Update: 0 = Not Forced 1 = Forced
Bit 4	LB UpdateSource: 0 = LBSourceData 1 = Registers
Bit 18	Disable LB Update: 0 = Enabled 1 = Disabled

WindowOrigin

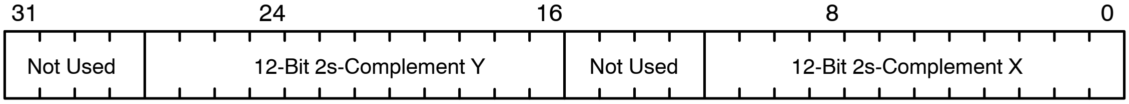
Name: Window Origin

Unit: Scissor/Stipple

Region: 0 Offset: 0×0000.81C8
Tag: 0×0039

Reset Value: Undefined

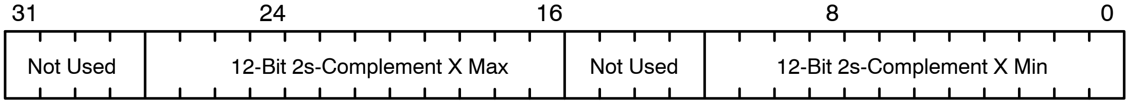
Read/write



This register adds the origin of each fragment generated by the Rasterizer unit to the coordinates of the fragment to generate its screen coordinates. This occurs prior to the screen-scissor test.

XLimits

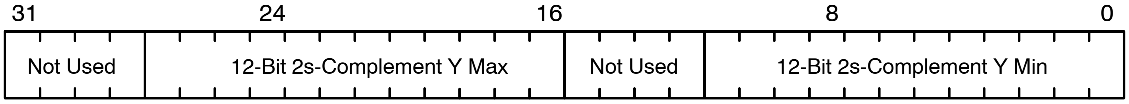
Name:	X Extent for Rasterizing	
Unit:	Rasterizer	
Region: 0	Offset:	0×0000.80C8
	Tag:	0×0019
Reset Value:	Undefined	
Read/write		



This register defines the X extent for the rasterizer to fill.

YLimits

Name:	Y Extent for Rasterizing	
Unit:	Rasterizer	
Region: 0	Offset:	0×0000.80A8
	Tag:	0×0015
Reset Value:	Undefined	
Read/write		



This register defines the Y extent for the rasterizer to fill.

YUVMode

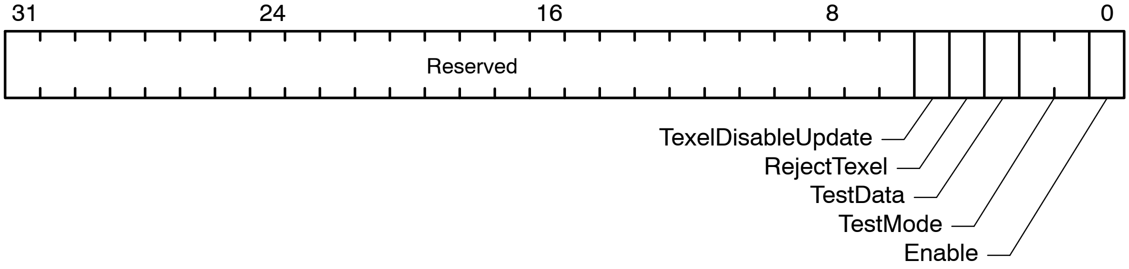
Name: YUV Mode

Unit: YUV

Region: 0 Offset: 0×0000.8F00
Tag: 0×01E0

Reset Value: Undefined

Read/write



This register controls YUV to RGB conversions and/or chroma tests.

Bit 0 Enable:

0 = YUV to RGB color space conversion disabled
1 = YUV to RGB color space conversion enabled

Bits 1 , 2 TestMode:

0 = No chroma test
1 = Pass if within chroma bounds
2 = Fail if within chroma bounds

Bit 3 TestData:

0 = Apply chroma test on input data (before color space conversion if enabled).
1 = Apply chroma test on output data (after color space conversion if enabled).

Bit 4 RejectTexel:

0 = Do not plot pixel if chroma test fails.

1 = Do not texture pixel if chroma test fails.

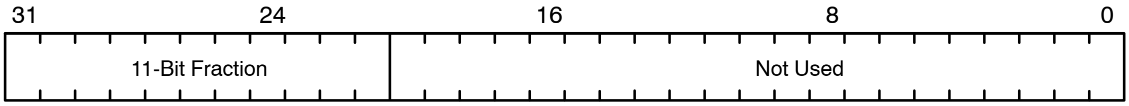
Bit 5 TexelDisableUpdate:

0 = Pass on texel data.

1 = Reject texel data immediately after chroma test.

ZStartL

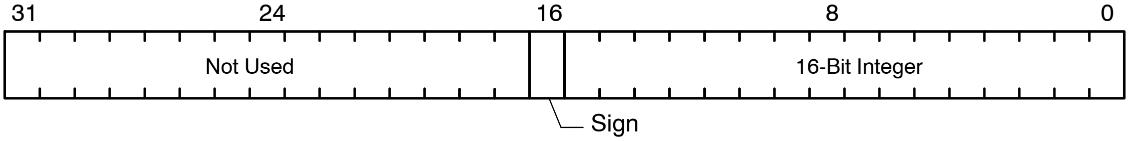
Name:	Depth Start Value – Lower	
Unit:	Stencil/Depth	
Region: 0	Offset:	0×0000.89B8
	Tag:	0×0137
Reset Value:	Undefined	
Read/write		



This register holds part of the start value for depth interpolation. ZStartU holds the most significant bits, and ZStartL holds the least significant bits. The combined value is in 2s-complement 17.11 fixed-point format.

ZStartU

Name:	Depth Start Value – Upper	
Unit:	Stencil/Depth	
Region: 0	Offset:	0×0000.89B0
	Tag:	0×0136
Reset Value:	Undefined	
Read/write		



This register holds part of the start value for depth interpolation. ZStartU holds the most significant bits, and ZStartL holds the least significant bits. The combined value is in 2s-complement 17.11 fixed-point format.

Chapter 8

Register T ables

This chapter lists registers by unit type, name, and register address. The tables include the register tag values and indicate the register type. Use the register group information to improve data transfer rates to the TVP4020 during direct memory access (DMA).

Topic **Page**

8.1 Overview

The following types of registers are distinguished in this chapter:

❏ Control: This register sets the control bits to a ready state for use in drawing a primitive. This is the default register and it is indicated by a blank entry in the Type column.

❏ Command: This register initiates operations, for example, the drawing of a primitive.

❏ Mixed: This register functions as a control register, and may also be used to supply successive data values during download operations.

❏ Output: This register functions as an internal register that cannot be read or written to. Its contents are passed to the Host Out FIFO using specific commands.

The tables indicate whether the register can be read back.

8.2 Unit Type

This section lists the graphics registers by unit type.

Table 8−1. Registers by Unit

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
Delta	V0Fixed[14]	20	0−D		Yes
	V1Fixed[14]	21	0−D		Yes
	V2Fixed[14]	22	0−D		Yes
	V0Float[14]	23	0−D		Yes
	V1Float[14]	24	0−D		Yes
	V2Float[14]	25	0−D		Yes
	DeltaMode	26	0		Yes
	DrawTriangle	26	1		No
	RepeatTriangle	26	2		No
	DrawLine01	26	3		No
	DrawLine10	26	4		No
	RepeatLine	26	5		No
Rasterizer	StartXDom	00	0		Yes
	dXDom	00	1		Yes
	StartXSub	00	2		Yes
	dXSub	00	3		Yes
	StartY	00	4		Yes
	dY	00	5		Yes
	Count	00	6		Yes
	Render	00	7	Command	No
	ContinueNewLine	00	8	Command	No
	ContinueNewDom	00	9	Command	No
	ContinueNewSub	00	A	Command	No
	Continue	00	B	Command	No
	BitMaskPattern	00	D	Mixed	No
	RectangleOrigin	01	A		No
	RectangleSize	01	B		No
	RasterizerMode	01	4		Yes
	YLimits	01	5		Yes
	WaitForCompletion	01	7	Command	No

Table 8−1. Registers by Unit (Continued)

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
	XLimits	01	9		Yes
	PackedDataLimits	02	A		Yes
Scissor/Stipple	ScissorMode	03	0		Yes
	ScissorMinXY	03	1		Yes
	ScissorMaxXY	03	2		Yes
	ScreenSize	03	3		Yes
	AreaStippleMode	03	4		Yes
	WindowOrigin	03	9		Yes
	AreaStipplePattern (0−7)	04	0−7		Yes
LBRead/Write	LBReadMode	11	0		Yes
	LBReadFormat	11	1		Yes
	LBSourceOffset	11	2		Yes
	LBData	11	3		No
	LBStencil	11	5	Output	No
	LBDepth	11	6	Output	No
	LBWindowBase	11	7		Yes
	LBWrite Mode		8		Yes
Stencil/Depth	Window	13	0		Yes
	StencilMode	13	1		Yes
	StencilData	13	2		Yes
	Stencil	13	3	Mixed	Yes
	DepthMode	13	4		Yes
	Depth	13	5	Mixed	Yes
	ZStartU	13	6		Yes
	ZStartL	13	7		Yes
	dZdxU	13	8		Yes
	dZdxL	13	9		Yes
	dZdyDomU	13	A		Yes
	dZdyDomL	13	B		Yes
Texture Address	TextureAddressMode	07	0		Yes
	SStart	07	1		Yes
	dSdx	07	2		Yes
	dSdyDom	07	3		Yes

Table 8−1. Registers by Unit (Continued)

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
	TStart	07	4		Yes
	dTdx	07	5		Yes
	dTdyDom	07	6		Yes
	QStart	07	7		Yes
	dQdx	07	8		Yes
	dQdyDom	07	9		Yes
Texture Read	TextureBaseAddress	0B	0		Yes
	TextureMapFormat	0B	1		Yes
	TextureDataFormat	0B	2		Yes
	Texel0	0C	0		Yes
	TextureReadMode	0C	E		Yes
	TexelLUTMode	0C	F		Yes
	TexelLUT (0 – 15)	1D	0 – F		Yes
	AlphaMapUpperBound	1E	3		Yes
	AlphaMapLowerBound	1E	4		Yes
	TexelLUTIndex	09	8		Yes
	TexelLUTData	09	9		Yes
	TexelLUTAddress	09	A		Yes
	TexelLUTTransfer	09	B		Yes
	TextureID	1E	E	Command	Yes
	TexelLUTID	1E	F	Command	Yes
YUV	YUVMode	1E	0		Yes
	ChromaUpperBound	1E	1		Yes
	ChromaLowerBound	1E	2		Yes
FBRead/Write	FBReadMode	15	0		Yes
	FBSourceOffset	15	1		Yes
	FBPixelOffset	15	2		Yes
	FBColor	15	3	Output	No
	FBData	15	4	Mixed	No
	FBSourceData	15	5	Mixed	No
	FBWindowBase	15	6		Yes
	FBWriteMode	15	7		Yes

Table 8−1. Registers by Unit (Continued)

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
	FBHardwareWriteMask	15	8		Yes
	FBBlockColor	15	9		Yes
	FBReadPixel	15	A		Yes
	TextureData	11	D		No
	TextureDownloadOffset	11	E		Yes
	SuspendUntilFrameBlank	18	F	Command	No
	FBBlockColorU	18	D		Yes
	FBBlockColorL	18	E		Yes
	FBSourceBase	1B	0		Yes
	FBSourceDelta	1B	1	Command	Yes
Color DDA	RStart	0F	0		Yes
	dRdx	0F	1		Yes
	dRdyDom	0F	2		Yes
	GStart	0F	3		Yes
	dGdx	0F	4		Yes
	dGdyDom	0F	5		Yes
	BStart	0F	6		Yes
	dBdx	0F	7		Yes
	dBdyDom	0F	8		Yes
	AStart	0F	9		Yes
	ColorDDAMode	0F	C		Yes
	ConstantColor	0F	D		Yes
	Color	0F	E	Mixed	No
Texture/Fog/Blend	TextureColorMode	0D	0		Yes
	FogMode	0D	2		Yes
	FogColor	0D	3		Yes
	FStart	0D	4		Yes
	dFdx	0D	5		Yes
	dFdyDom	0D	6		Yes
	KsStart	0D	9		Yes
	dKsdx	0D	A		Yes
	dKsdyDom	0D	B		Yes

Table 8–1. Registers by Unit (Continued)

Unit	Register	Major Group (hex)	Offset (hex)	Type	Readable
	KdStart	0D	C		Yes
	dKddx	0D	D		Yes
	dKddyDom	0D	E		Yes
	AlphaBlendMode	10	2		Yes
Color Format	DitherMode	10	3		Yes
Logic Ops	FBSoftwareWriteMask	10	4		Yes
	LogicalOpMode	10	5		Yes
Host Out	FilterMode	18	0		Yes
	StatisticMode	18	1		Yes
	MinRegion	18	2		Yes
	MaxRegion	18	3		Yes
	ResetPickResult	18	4	Command	No
	MinHitRegion	18	5	Command	No
	MaxHitRegion	18	6	Command	No
	PickResult	18	7	Command	Yes
	Sync	18	8	Command	No
Multiple	Config	1B	2		No

8.3 Name

Table 8−2. Registers by Name

Register	Major Group (hex)	Offset (hex)	Type	Readable
AlphaBlendMode	10	2		Yes
AlphaMapLowerBound	1E	4		Yes
AlphaMapUpperBound	1E	3		Yes
AreaStippleMode	03	4		Yes
AreaStipplePattern (0 – 7)	04	0 – 7		Yes
AStart	0F	9		Yes
BitMaskPattern	00	D	Mixed	No
BStart	0F	6		Yes
ChromaLowerBound	1E	2		Yes
ChromaUpperBound	1E	1		Yes
Color	0F	E	Mixed	No
ColorDDAMode	0F	C		Yes
Config	1B	2		No
ConstantColor	0F	D		Yes
Continue	00	B	Command	No
ContinueNewDom	00	9	Command	No
ContinueNewLine	00	8	Command	No
ContinueNewSub	00	A	Command	No
Count	00	6		Yes
dBdx	0F	7		Yes
dBdyDom	0F	8		Yes
DeltaMode	26	0		Yes
Depth	13	5	Mixed	Yes
DepthMode	13	4		Yes
dFdx	0D	5		Yes
dFdyDom	0D	6		Yes
dGdx	0F	4		Yes
dGdyDom	0F	5		Yes
DitherMode	10	3		Yes
dKddx	0D	D		Yes

Table 8–2. Registers by Name (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
dKddyDom	0D	E		Yes
dKsdx	0D	A		Yes
dKsdyDom	0D	B		Yes
dQdx	07	8		Yes
dQdyDom	07	9		Yes
DrawLine01	26	3		No
DrawLine10	26	4		No
DrawTriangle	26	1		No
dRdx	0F	1		Yes
dRdyDom	0F	2		Yes
dSdx	07	2		Yes
dSdyDom	07	3		Yes
dTdx	07	5		Yes
dTdyDom	07	6		Yes
dXDom	00	1		Yes
dXSub	00	3		Yes
dY	00	5		Yes
dZdxL	13	9		Yes
dZdxU	13	8		Yes
dZdyDomL	13	B		Yes
dZdyDomU	13	A		Yes
FBBlockColor	15	9		Yes
FBBlockColorL	18	E		Yes
FBBlockColorU	18	D		Yes
FBColor	15	3	Output	No
FBData	15	4	Mixed	No
FBHardwareWriteMask	15	8		Yes
FBPixelOffset	15	2		Yes
FBReadMode	15	0		Yes
FBReadPixel	15	A		Yes
FBSoftwareWriteMask	10	4		Yes
FBSourceBase	1B	0		Yes
FBSourceData	15	5	Mixed	No

Table 8−2. Registers by Name (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
FBSourceDelta	1B	1	Command	Yes
FBSourceOffset	15	1		Yes
FBWindowBase	15	6		Yes
FBWriteData	10	6		Yes
FBWriteMode	15	7		Yes
FilterMode	18	0		Yes
FogColor	0D	3		Yes
FogMode	0D	2		Yes
FStart	0D	4		Yes
GStart	0F	3		Yes
KdStart	0D	C		Yes
KsStart	0D	9		Yes
LBData	11	3		No
LBDepth	11	6	Output	No
LBReadFormat	11	1		Yes
LBReadMode	11	0		Yes
LBSourceOffset	11	2		Yes
LBStencil	11	5	Output	No
LBWindowBase	11	7		Yes
LBWriteFormat	11	9		Yes
LBWriteMode	11	8		Yes
LogicalOpMode	10	5		Yes
MaxHitRegion	18	6	Command	No
MaxRegion	18	3		Yes
MinHitRegion	18	5	Command	No
MinRegion	18	2		Yes
PackedDataLimits	02	A		Yes
PickResult	18	7	Command	Yes
QStart	07	7		Yes
RasterizerMode	01	4		Yes
RectangleOrigin	01	A		No
RectangleSize	01	B		No
Render	00	7	Command	No

Table 8–2. Registers by Name (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
RepeatLine	26	2		No
RepeatTriangle	26	5		No
ResetPickResult	18	4	Command	No
RStart	0F	0		Yes
ScissorMaxXY	03	2		Yes
ScissorMinXY	03	1		Yes
ScissorMode	03	0		Yes
ScreenSize	03	3		Yes
SStart	07	1		Yes
StartXDom	00	0		Yes
StartXSub	00	2		Yes
StartY	00	4		Yes
StatisticMode	18	1		Yes
Stencil	13	3	Mixed	Yes
StencilData	13	2		Yes
StencilMode	13	1		Yes
SuspendUntilFrameBlank	18	F	Command	No
Sync	18	8	Command	No
Texel0	0C	0		Yes
TexelLUT (0–15)	1D	0–F		Yes
TexelLUTAddress	09	A		Yes
TexelLUTData	09	9		Yes
TexelLUTID	1E	F	Command	Yes
TexelLUTIndex	09	8		Yes
TexelLUTMode	0C	F		Yes
TexelLUTTransfer	09	B		Yes
TextureAddressMode	07	0		Yes
TextureBaseAddress	0B	0		Yes
TextureColorMode	0D	0		Yes
TextureData	11	D		No
TextureDataFormat	0B	2		Yes
TextureDownloadOffset	11	E		Yes
TextureID	1E	E	Command	Yes

Table 8−2. Registers by Name (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
TextureMapFormat	0B	1		Yes
TextureReadMode	0C	E		Yes
TStart	07	4		Yes
V0Fixed[14]	20	0−D		Yes
V0Float[14]	23	0−D		Yes
V1Fixed[14]	21	0−D		Yes
V1Float[14]	24	0−D		Yes
V2Fixed[14]	22	0−D		Yes
V2Float[14]	25	0−D		Yes
WaitForCompletion	01	7	Command	No
Window	13	0		Yes
WindowOrigin	03	9		Yes
XLimits	01	9		Yes
YLimits	01	5		Yes
YUVMode	1E	0		Yes
ZStartL	13	7		Yes
ZStartU	13	6		Yes

8.4 Address

Table 8–3. Registers by Address

Register	Major Group (hex)	Offset (hex)	Type	Readable
StartXDom	00	0		Yes
dXDom	00	1		Yes
StartXSub	00	2		Yes
dXSub	00	3		Yes
StartY	00	4		Yes
dY	00	5		Yes
Count	00	6		Yes
Render	00	7	Command	No
ContinueNewLine	00	8	Command	No
ContinueNewDom	00	9	Command	No
ContinueNewSub	00	A	Command	No
Continue	00	B	Command	No
BitMaskPattern	00	D	Mixed	No
RectangleOrigin	01	A		No
RectangleSize	01	B		No
RasterizerMode	01	4		Yes
YLimits	01	5		Yes
WaitForCompletion	01	7	Command	No
XLimits	01	9		Yes
PackedDataLimits	02	A		Yes
ScissorMode	03	0		Yes
ScissorMinXY	03	1		Yes
ScissorMaxXY	03	2		Yes
ScreenSize	03	3		Yes
AreaStippleMode	03	4		Yes
WindowOrigin	03	9		Yes
AreaStipplePattern (0 – 7)	04	0 – 7		Yes
TextureAddressMode	07	0		Yes
SStart	07	1		Yes
dSdx	07	2		Yes
dSdyDom	07	3		Yes

Table 8–3. Registers by Address (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
TStart	07	4		Yes
dTdx	07	5		Yes
dTdyDom	07	6		Yes
QStart	07	7		Yes
dQdx	07	8		Yes
dQdyDom	07	9		Yes
TextureLUTIndex	09	8		Yes
TextureBaseAddress	0B	0		Yes
TextureMapFormat	0B	1		Yes
TextureDataFormat	0B	2		Yes
Texel0	0C	0		Yes
TextureReadMode	0C	E		Yes
TexelLUTMode	0C	F		Yes
TextureColorMode	0D	0		Yes
FogMode	0D	2		Yes
FogColor	0D	3		Yes
FStart	0D	4		Yes
dFdx	0D	5		Yes
dFdyDom	0D	6		Yes
KsStart	0D	9		Yes
dKsdx	0D	A		Yes
dKsdyDom	0D	B		Yes
KdStart	0D	C		Yes
dKddx	0D	D		Yes
dKddyDom	0D	E		Yes
RStart	0F	0		Yes
dRdx	0F	1		Yes
dRdyDom	0F	2		Yes
GStart	0F	3		Yes
dGdx	0F	4		Yes
dGdyDom	0F	5		Yes
BStart	0F	6		Yes

Table 8–3. Registers by Address (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
dBdx	0F	7		Yes
dBdyDom	0F	8		Yes
AStart	0F	9		Yes
ColorDDAMode	0F	C		Yes
ConstantColor	0F	D		Yes
Color	0F	E	Mixed	No
AlphaBlendMode	10	2		Yes
TexelLUTData	09	9		Yes
TexelLUTAddress	09	A		Yes
TexelLUTTransfer	09	B		Yes
DitherMode	10	3		Yes
FBSoftwareWriteMask	10	4		Yes
LogicalOpMode	10	5		Yes
LBReadMode	11	0		Yes
LBReadFormat	11	1		Yes
LBSourceOffset	11	2		Yes
LBData	11	3		No
LBStencil	11	5	Output	No
LBDepth	11	6	Output	No
LBWindowBase	11	7		Yes
LBWriteMode	11	8		Yes
LBWriteFormat	11	9		Yes
TextureData	11	D		No
TextureDownloadOffset	11	E		Yes
Window	13	0		Yes
StencilMode	13	1		Yes
StencilData	13	2		Yes
Stencil	13	3	Mixed	Yes
DepthMode	13	4		Yes
Depth	13	5	Mixed	Yes
ZStartU	13	6		Yes
ZStartL	13	7		Yes
dZdxU	13	8		Yes

Table 8–3. Registers by Address (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
dZdxL	13	9		Yes
dZdyDomU	13	A		Yes
dZdyDomL	13	B		Yes
FBReadMode	15	0		Yes
FBSourceOffset	15	1		Yes
FBPixelOffset	15	2		Yes
FBColor	15	3	Output	No
FBData	15	4	Mixed	No
FBSourceData	15	5	Mixed	No
FBWindowBase	15	6		Yes
FBWriteMode	15	7		Yes
FBHardwareWriteMask	15	8		Yes
FBBlockColor	15	9		Yes
FBReadPixel	15	A		Yes
FilterMode	18	0		Yes
StatisticMode	18	1		Yes
MinRegion	18	2		Yes
MaxRegion	18	3		Yes
ResetPickResult	18	4	Command	No
MinHitRegion	18	5	Command	No
MaxHitRegion	18	6	Command	No
PickResult	18	7	Command	Yes
Sync	18	8	Command	No
FBBlockColorU	18	D		Yes
FBBlockColorL	18	E		No
SuspendUntilFrameBlank	18	F	Command	No
TextureID	1E	E	Command	Yes
V0Fixed[14]	20	0–D		Yes
V1Fixed[14]	21	0–D		Yes
V2Fixed[14]	22	0–D		Yes
V0Float[14]	23	0–D		Yes
V1Float[14]	24	0–D		Yes
V2Float[14]	25	0–D		Yes

Table 8–3. Registers by Address (Continued)

Register	Major Group (hex)	Offset (hex)	Type	Readable
DeltaMode	26	0		Yes
DrawTriangle	26	1		No
RepeatTriangle	26	2		No
DrawLine01	26	3		No
DrawLine10	26	4		No
RepeatLine	26	5		No
FBSourceBase	1B	0		Yes
FBSourceDelta	1B	1		Yes
Config	1B	2		Yes
TexelLUT(0 – 15)	1D	0 – F		Yes
AlphaMapUpperBound	1E	3		Yes
AlphaMapLowerBound	1E	4		Yes
TextureID	1E	E	Command	Yes
TexelLUTID	1E	F	Command	Yes
YUVMode	1E	0		Yes
ChromaUpperBound	1E	1		Yes
ChromaLowerBound	1E	2		Yes

Appendix A

Pseudocode Definitions

This appendix provides pseudocode definitions and examples for the fragmentary loading of registers.

Topic **Page**

A.1 Pseudocode Overview

Fragments of pseudocode are provided in many areas of this document to describe the loading of registers. These fragments are based on a C interface to the TVP4020 in which each 32-bit register is represented as a C structure potentially split into a series of bit fields. For example, where only a subset of the bit fields in a register is set, either a software copy of the register is being modified, or the current contents of the register are first read back to the host.

The constant and register bit field definitions are those used in the C example programs.

Note:

The order for loading control registers into the graphics hyperpipeline is chosen for clarity rather than efficiency. The optimal order is documented in subsection 6.2.3, *Loading Registers in Unit Order*.

Loading of a TVP4020 register is expressed as:

```
register–name(value)
```

When writing directly to the register file, that is, to a FIFO, write the word *value* to the mapped-in address of the register called register-name.

A.2 Fragmentary Examples

Fragmentary examples are not in strict C syntax. A typical example is as follows:

```
// Sample code to rasterize a 10x10 rectangle at the
// framebuffer origin.
StartXDom (0)         // Start dominant edge
StartXSub (1<<16)     // Start of subordinate
dXDom (0x0)
dXSub (0x0)
Count (0xA)
YStart(0)
dY (1<<16)
// Set up to render a trapezoid.
render.AreaStippleEnable = TVP4020_DISABLE
render.PrimitiveType = TVP4020_TRAPEZOID
render.FastFillEnable = TVP4020_DISABLE
render.FogEnable = TVP4020_DISABLE
render.TextureEnable = TVP4020_DISABLE
render.ReuseBitMask = TVP4020_DISABLE
render.SyncOnBitMask = TVP4020_FALSE
render.SyncOnHostData = TVP4020_FALSE
Render (render)       // Render the rectangle
```

Code is shown in Courier style, and comments are in C++ style "//" to indicate that the rest of the line is a comment. Any statement that ends in parenthesis is a register update; other statements are generally assignments. A variable, for example, render, is of a type associated with the register being modified. This is usually clear by the context and, thus, is not generally declared. All the type definitions are in the header files. The values assigned to a register are either a variable as described above; a macro, that is, TVP4020_TRUE as found in the headers; or an immediate constant in C style format, that is, 0×45. In registers that have several fields that are not relevant to a particular example, the field can be ignored. In some registers, values for fields that need to be set but are not readily available are typically set as appropriate.

In some fragments, a list of simple commands is provided; for example:

```
// 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 list provides a brief description of the registers involved in a particular operation in which a detailed treatment is not warranted.

For 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 it increments the pointer:

```
*buf++ = StartXDom
```

To test the value of a register, the register name is dereferenced using the C “*” operator, as in this example, which tests for the completion of a DMA operation:

```
while( *DMACount != 0 ) ;
```

Appendix B

Screen Widths Table

Screen width is the sum of selected partial products (PP). A full multiplication operation is not necessary. The partial products are selected by the fields PP0, PP1, and PP2 in the FBReadMode register, the LBReadMode register, and the TextureMapFormat register. The range of widths supported by the selected partial products is tabulated in Table B–1, together with the values for each of the partial products fields.

The TVP4020 supports a maximum screen resolution of 2048 pixels in width by 2048 pixels in height.

Table B–1. Partial Products

Screen Width	PP2	PP1	PP0
0	0	0	0
32	0	0	1
64	0	1	1
96	1	1	1
128	1	1	2
160	1	2	2
192	2	2	2
224	1	2	3
256	2	2	3
288	1	3	3
320	2	3	3
384	3	3	3
416	1	3	4
448	2	3	4
512	3	3	4
544	1	4	4
576	2	4	4
640	3	4	4
768	4	4	4
800	1	4	5
832	2	4	5
896	3	4	5
1024	4	4	5
1056	1	5	5
1088	2	5	5
1152	3	5	5
1280	4	5	5
1536	5	5	5
1568	6	5	1
1600	6	5	2
1664	6	5	3
1792	6	5	4
2048	6	5	5

Appendix C

TVP4010 and TVP4020 Differences

This appendix provides a summary of the differences between the TVP4010 and TVP4020 graphics processors and the features of the TVP4020. Most differences are a result of the newer TVP4020 functions and features.

Topic **Page**

C.1 New Units

The following sections describe the features and functions of the new units associated with the TVP4020 graphics processor.

C.1.1 Video Streams

The TVP4020 supports independent input and output of digital video. The input stream complies with the Video Electronics Standard Association (VESA) Video Module Interface (VMI) specification. You can scale and filter input data before it is written to local memory. The output stream is based on the VMI specification as well as work with common programmable array logic/National Television Standards Committee (PAL/NTSC) encoders. Both streams are independent of the video output to the monitor.

The interface is configurable to meet different needs. Table C−1 shows the modes that are supported.

Table C−1. Video Modes

Input Width	Output Width	Description
8	8	Simultaneous video input/output
16	0	Input only zoom video port
0	16	Output only zoom video port
8	0	Input data with random access parallel bus

You can scale and filter input data to reduce memory requirements. You can have the output stream gamma corrected and converted from RGB to YUV. The output video functions as a slave and supplies data on demand from the external encoder chip. Both streams support automatic hardware triple buffering.

You can separately control vertical blank interval (VBI) data such as closed caption, teletext, or intercast. You can insert VBI data into the output stream or extract it from the input stream, as required.

The interface supports two separate buses for programming devices connected to the video streams. The I^{2}C bus is a two-wire serial bus that is commonly used to control chips that supply or receive data on the video ports. The general-purpose bus is a parallel bus that supports a higher bandwidth and uses an 8-bit data path with a 4-bit address. If you use the parallel bus, only input video is available.

In the TVP4020, the external ROM stores the VGA BIOS and the power-up configuration information (removing most of the configuration resistors needed for a TVP4010 design). Access to the ROM is by the general-purpose bus during which time both video streams are disabled.

C.1.2 RAMDAC

The TVP4020 incorporates a high-performance 230-MHz RAMDAC. It supports resolutions of up to 1600×1280 pixels at 85 Hz, with a wide variety of pixel formats and a hardware cursor of 64×64×2 pixels. Integrated phase-locked loops generate all clocks required by the TVP4020.

The TVP4020 directly supports DDC1 and DDC2 monitor configurations and Apple Macintosh™ monitor sensing. The DDC2 serial bus is independent of the serial bus in the VMI interface.

C.1.3 Delta

The delta chip integrates the 100-MFLOP geometry pipeline processor with the TVP4020. This enhanced chip supports backface culling, which is enabled in the DeltaMode register. The rejection of positive or negative area triangles (that is, front or back faces) is controlled by the Render command.

The delta vertex interface includes a packed color format to load all four color components into a single 32-bit word. To offset address 14 of the vertex store, write the data in a packed 8888 format; the DeltaMode register controls the order of the color components within the word.

C.2 PCI Differences

The following sections describe the differences between the TVP4010 and TVP4020 peripheral component interconnections and the new TVP4020 features.

C.2.1 AGP Support

The TVP4020 supports advanced graphics port (AGP) extensions to the PCI protocol. When used in an AGP slot, the TVP4020 functions as a 66-MHz PCI device, and also performs single-edge AGP read-master transfers with optional sideband addressing.

C.2.2 Bypass DMA Engine

The TVP4020 includes a DMA engine that allows high-speed data transfers through the bypass, from system memory to local memory. As data is transferred, it can be formatted to match the patching organization used by the graphics core texture units. Data can also be converted from YUV420 to YUV422 formats.

C.2.3 Host-Out DMA Engine

The PCI interface includes a DMA engine that allows high-speed data transfers from the graphics core output FIFO to system memory.

C.2.4 Extra Interrupts

The TVP4020 includes the following new interrupts:

- Invalid texture
- Bypass DMA complete
- Video stream A interrupt
- Video stream B interrupt
- Video streams external interrupt
- DDC interrupt

C.3 Video Unit Differences

The following sections describe the differences between the TVP4010 and TVP4020 video units and the new TVP4020 features.

C.3.1 FIFO Threshold Control

The TVP4020 video FIFO includes programmable high and low watermarks to allow optimum bursting of video data for different screen resolutions.

C.3.2 Stereo Control

The TVP4020 includes additional support for left and right eye screens that are alternately displayed. An external pin signals which eye is being displayed and can drive LCD shutter glasses.

C.3.3 Frameblank Control

The TVP4020 has additional control over behavior at frameblank. It continues to support automatic synchronization to frameblank where the new screen base address is only accepted during the vertical blank interval. In addition, it supports a free running mode where the base address is updated immediately, without having to wait for the blanking period.

The TVP4020 also supports a sync-to-frame rate mode that updates the base address in frameblank while the frame rate keeps up with the monitor refresh rate. If the frame rate drops below the refresh rate, the base address is updated immediately.

C.4 Core Differences

The following sections describe the core differences between the TVP4010 and TVP4020 and the new TVP4020 features.

C.4.1 Maximum Screen Size

The TVP4020 maximum screen size is 2048×2048 pixels, an increase over the TVP4010.

C.4.2 Rectangle Primitive

The TVP4020 supports a new primitive for drawing rectangles. It is restricted to integer pixel positions only. You can continue to use the trapezoid primitive for rectangles that require subpixel positioning. The rectangle is defined with two new registers: RectangleOrigin defines the X and Y start point, and Rectangle-Size defines the width and height. You can control the direction in which the rectangle is filled using the Render command. To make the primitive suitable for copy operations, control the fill direction in X and Y separately.

C.4.3 Texture Mapping

The TVP4020 does not support the TextureAddressMode option for accurate or fast perspective divide. All divides are now faster than the TVP4010 fast speed, with improved accuracy.

You can store textures in system memory or local memory but do not split an individual texture map across system and local memory.

You can indirectly access the base address of the texture map through a table instead of through the TextureBaseAddress register. If the TextureID register is loaded, the TVP4020 accesses memory to fetch the actual base address of the texture. Bit 31 of this address is a validity flag and if set to invalid, the graphics pipeline halts and an interrupt is generated. The host can then load the texture through the bypass and restart the graphics core. This mechanism allows efficient texture caching by decoupling the memory management of the textures from the graphics core.

C.4.4 Look Up Tables

The TVP4020 has a 256 entry texture LUT and each entry is 32 bits wide. The LUT can be used as 16 smaller LUTs of 16 entries each. Load the contents of the LUT through the graphics pipeline or from memory (local or system). If the LUT is being held in memory, load its address indirectly using the same mechanism as you would use for texture caching.

You can use the LUT to index 4- or 8-bit textures, in which case the single index generates all four color components. If the texture type has separate color components (that is, it is not an index), each component is indexed independently through the LUT. This allows color remapping operations such as gamma correction.

You can also access the LUT directly from the XY position of the pixel being drawn to hold block-fill colors.

C.4.5 Block Filling

The block-fill color register now extends to 64 bits to allow greater flexibility. Two new registers, BlockColorUpper and BlockColorLower, set the upper and lower 32 bits of the color, respectively. If you use the TVP4010 BlockColor register, its contents provide backward compatibility to both upper and lower halves of the block color.

Texture mapping now holds block-fill masks. You can store a byte packed font, designed specifically for font caching, in local memory. You can use it with a block-fill operation to control the pixels to be drawn.

You can stipple any block-fill pattern using the normal stipple pattern table.

The texture LUT, designed for pattern filling, now holds data that updates the block-fill color on each scanline.

C.4.6 Sprite Control

The TVP4020 extends chroma key testing to improve the quality of cutouts that are bilinearly filtered and to smooth the edges of sprites. Two additional registers, AlphaMapUpperBound and AlphaMapLowerBound, define that range of colors whose alpha values must be mapped to zero. The TVP4010 chroma key registers reject those pixels with alpha values not equal to one. Texels that fail the alpha map test are not included in filtering, so edge effects often seen with filtered cutouts are removed.

The chroma key test filters the alpha values of the edge pixels so that they form a range from one, within the area to be drawn, to zero, within the area not be be drawn. In the region close to the edge of the area to be drawn, the alpha values are filtered to lie between zero and one. The range of alpha values rejected by the chroma key test can be adjusted to allow fine control over the exact size of the cutout. If blending is enabled, then the varying alpha values smooth the transition of the edge of the sprite to the background.

C.4.7 Alpha Blending

The optimized TVP4020 reduces memory bandwidth when you enable blending. If you use an alpha value for blending that is derived exclusively from a texture map, you can set the FBReadMode register to disable reading of the framebuffer for any pixel whose corresponding texel has an alpha value of one. If the alpha value is one, the final color does not include any of the previous framebuffer color. As a result, the color does not need to be read.

The TVP4020 adds extra control over formatting of the framebuffer color when blending. The AlphaBlendMode register controls the way that framebuffer data is mapped to the internal color format. This prevents visual artifacts when blending with a dithered framebuffer.

C.4.8 Color Formats

The TVP4020 performs all internal color calculations at true color accuracy, where TVP4010 performed 3-D calculations at 5 bits per color component.

The TVP4020 includes an additional 24-bit pixel and texel size that has 8 bits of red, green, and blue but no alpha channel. All pixel operations are available at this size except block filling, which is restricted to colors that have all bytes the same value (that is, shades of gray). This restriction is due to the operation of the memory devices.

C.4.9 Miscellaneous

The TVP4020 calculates the value of FBSourceOffset from an XY delta value, removing multiplication from a copy operation setup.

The relative offset field in the FBReadMode register is also in the PackedDataLimits register. The one to be used is the last one to be set before drawing a primitive. This reduces the number of written registers for a copy operation.

A register is included in the TVP4020 PCI interface that determines when the chip is idle. If the idle status is set, then the TVP4020 processes the next command without a delay. This does not mean, however, that all previous operations were completed or that data was written to memory.

An additional TVP4020 register configures a number of controls that are normally set by separate registers. The Config register controls parts of the FBReadMode, FBWriteMode, LogicalOpMode, and ColorDDAMode registers.

Appendix D

Glossary

A

accumlation buffer: A color buffer of higher resolution than the displayed buffer (typically 16-bits per component for an eight-bit-per-component display). It is typically used to sum the rendering result of several frames from slightly different viewpoints, to achieve motion blur effects or eliminate aliasing effects.

active fragment: A fragment that passes all the various culling tests, such as scissor, depth(Z), alpha, etc., is written to or combined with the corresponding pixel in the framebuffer. See also *fragment* and *passive fragment*.

aliasing: A phenomena resulting from a rendering style that ignores the fact that a pixel may not be wholly covered by a primitive, leading to jagged edges on primitives.

alpha blending: The ability to combine supplied red, green, and blue color values with those that exist in the framebuffer according to the supplied alpha value. Alpha blending forms the basis for techniques such as transparency and painting.

alpha buffer: A memory buffer containing the fourth component of a pixel's color, in addition to red, green, and blue. This component is not displayed, but may be used to control color blending.

area stipple: A two-dimensional binary pattern that is used to cull fragments from being drawn.

B

BITBLT: A bit-aligned block transfer that copies a rectangular array of bitmap pixels from one location to another.

BITBLT double buffering: A technique to provide independent, windowed, double buffering by causing a bit-aligned block transfer to copy an area from one buffer to the other.

bitplane double buffering: A technique that provides fast, independent, windowed, double buffering using a single bitplane bit.

block write: A feature, provided in some memory devices, such as video RAM (VRAM) or synchronous graphics RAM (SGRAM), which allows multiple pixels to be set to a given value by a single write operation. *Fast fill* is an alternative name for this feature.

C

chroma keying: Also known as bluescreening, this is the practice of excluding color from an image to allow an underlying image to show through.

chroma test: The means by which chroma keying can be achieved.

color index: The mode in which the color information is stored for each pixel as a single number, that is, as the color index, rather than as separate red, green, blue, and optional alpha values (RGBA mode). Each color index references an entry in a color look-up table that contains a particular set of red, green, and blue values.

command register: A register, which when loaded, triggers activity in the TVP4010. For example, a loaded Render command register prompts the TVP4010 to render the specified primitive using the most current parameters in the control registers.

context: The state information associated with a particular task. Typically, in a system, more than one task uses the TVP4010 to render primitives. The host software must save the current contents of the TVP4010 control registers when it suspends one task to allow another to run, and it must restore the state when that task is next scheduled to run.

control register: A register that contains a state that dictates how the TVP4010 will execute a command.

culling: The process of eliminating a fragment, object face, or primitive, so that it is not drawn.

D

DDA: The digital differential analyzer is an algorithm used to determine the pixels to draw along a line or polygon edge. It is also used to interpolate varying linear values such as color and depth.

delta: A gradient of color, fog, depth, etc., in the X or Y directions for a primitive.

depth (Z) buffer: A memory buffer that contains the depth component of a pixel. It is used to eliminate hidden surfaces.

depth-cueing: A technique that determines the color of a pixel based on its depth. It is used, for example, to fade far-away objects into the background. Depth-cueing is also known as fogging.

dithering: A rendering style that increases the perceived range of displayed colors at the cost of spatial resolution. The technique is similar to that of using stippled patterns of black and white pixels to achieve shades of gray on a black and white display.

dominant edge: The side of a primitive, such as a triangle, which has the greatest range of Y values.

double buffering: A technique for achieving smooth animation by rendering only to an undisplayed back buffer, and then swapping the back buffer to the front once drawing is complete.

E

extent checking: A technique that determines the rectangular bounds of the area that was rendered.

F

fast fill: A feature, provided in some memory devices such as VRAM and SGRAM, which allows multiple pixels to be set to a given value by a single write operation. *Block write* is an alternative name for this feature.

flat shading: The constant color shading or area filling of a primitive.

fogging: A technique that determines the color of a pixel based on its depth. It is used, for example, to fade far-away objects into the background. Also known as depth-cueing.

fragment: An object generated as a result of the rasterization of a primitive. It corresponds to and contains all the components of a single pixel. If a fragment passes all the various culling tests, such as scissor, depth(Z), stencil, etc., it is written to or 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.

G

Gouraud-shading: The technique of variable color shading or area filling of a primitive using interpolation to gradually vary the color between vertices. Often known as smooth shading.

H

hardware writemask: A bitmask, implemented in memory devices such as video RAM (VRAM) and synchronous graphics RAM (SGRAM), to enable or inhibit the writing of the corresponding bits of a fragment's color into the framebuffer.

host: The processor or CPU that controls the TVP4010.

L

localbuffer: An area of memory that may be used to store textures and/or nondisplayable depth(Z) and/or stencil pixel information. This memory is typically separate from the framebuffer.

logical operations (logic ops): The technique of applying logical operations such as OR, XOR, or AND to the fragment color values and/or those in the framebuffer.

LUT: A look-up table. This table usually contains color values to allow mapping from an index value to the desired red, green, and blue values.

O

overlays: A technique which ensures that certain drawn objects always remain foremost in view and are not obscured by others. This is one method that has, historically, provided a cursor using extra bit planes.

P

packed data: The arrangement of data in a buffer that allows multiple pixels to be read or written in a single access.

passive fragment: A fragment that fails one or more of the various culling tests, such as scissor, depth(Z), stencil, etc., which is not written to or combined with the corresponding pixel in the framebuffer. See also *fragment* and *active fragment*.

patched addressing: A technique in which data is organized in memory so that there is improved access performance to adjacent buffer scanlines. For the TVP4010, this addressing is available for accessing depth and/or stencil buffers. For textures, a special form called subpatch addressing is provided.

picking: A means of selecting drawn objects or primitives.

preMult: A method of alpha blending, also known as Ramp blend mode, used by QuickDraw3D™.

pixel: A picture element that 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 TVP4010 primitives are points, lines, trapezoids (including triangles as a subset), and bitmaps.

R

Ramp blend mode: A method of alpha blending, also known as preMult, used by QuickDraw3D.

rasterization: The act of converting a point, line, polygon, or bitmap in device coordinates, into fragments.

rendering: The conversion of primitives in object coordinates into an image.

S

scissor test: A means of culling fragments that lie outside the defined scissor rectangle. The scissor rectangle is defined in device coordinates.

software writemasking: A means of simulating hardware writemasking by performing a read-modify-write operation on framebuffer data.

stencil buffer: A buffer, used to store information about a pixel, which controls how subsequent stenciled fragments at the same location may be combined with the current value. Typically, it is used to mask complex two-dimensional shapes.

stipple: A one- or two-dimensional binary pattern that is used to cull fragments from being drawn.

subordinate edge: The sides of a primitive, such as a triangle, which do not have the greatest range of Y values.

subpatch addressing: A technique in which data is organized in memory so that there is improved access to adjacent buffer scanlines. For the TVP4010, this particular form of patched addressing is available for accessing texture maps. See also *patched addressing*.

subpixel correction: A means of ensuring correct sampling of all interpolated parameters (color, depth, fog, texture) at a fragment's center. This is required, for example, to ensure correct color shading of objects comprised of many primitives.

T

tag: The data item that uniquely identifies a Graphics Core register.

task: A process, or thread on the host, which uses the TVP4010 coprocessor. Tasks generally have sole use of the TVP4010. They rely on a device driver to save and restore their TVP4010 context when they are swapped out.

texel: An element of an image stored in texture memory, called a texture element, which represents the color of the texture to be applied (fully or in part) to a corresponding fragment.

texture: An image used to modify the color of fragments during processing. Often used 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 example, to apply or wrap a wood-grain effect to a table.

W

writemask: A bit pattern that enables or inhibits the writing of the corresponding bits of a fragment's color into the framebuffer. See also *software writemasking* and *hardware writemask*.

Y

YUV: An alternative color format to RGB, also known as YCbCr. The color format used by Motion Pictures Experts Group (MPEG).

Z

Z buffer: An alternative name for the depth buffer.

Index

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265