DECchip 21030
PCI Graphics Accelerator

Reference Manual

Order Number: EC–N0683–72

Revision/Update Information: This is a new manual.

Digital Equipment Corporation
Maynard, Massachusetts

September 1994

Contents

4 Register Descriptions

7 Programming Guide

8 Hardware Interface

A DECchip 21030 step A Differences

B Pin Summary

C Register Summary

D Technical Support, Ordering, and Associated Literature

Glossary

Index

Figures

Tables

Preface

This manual describes the architecture, internal design, and external interface of the DECchip 21030 PCI Graphics Accelerator. It includes register descriptions, an overview of the chip's microarchitecture, descriptions of the the graphics processing algorithms, and programming information.

Note

The DECchip 21030 step A and DECchip 21030 step B are nearly identical. Differences between the two are noted as they occur in this manual, and the description ''defaults'' to functionality that is specific to the 21030 step B; the equivalent 21030 step A functionality is described in Appendix A.

Audience

This manual is for system designers, software developers, and hardware engineers who use the DECchip 21030 PCI Graphics Accelerator.

Manual Organization

This manual contains the following chapters and appendices as well as a glossary and an index.

- Chapter 1, Introduction
- Chapter 2, Memory Space
- Chapter 3, Internal Architecture
- Chapter 4, Register Descriptions
- Chapter 5, PCI Operations
- Chapter 6, Graphics Operations
- Chapter 7, Programming Guide

- Chapter 8, Hardware Interface
- Appendix A, DECchip 21030 step A Differences
- Appendix B, Pin Summary
- Appendix C, Register Summary
- Appendix D, Technical Support, Ordering, and Associated Literature
- Glossary
- Index

Conventions

The following conventions are used throughout this manual.

Abbreviations

- **bpp**

  The terms ''bits per pixel'' and ''bits/pixel'' are abbreviated as bpp.

- **Binary Multiples**

  The abbreviations K, M, and G (Kilo, Mega, and Giga) represent binary multiples and have the following values.

K =	2^{10} (1024)
M =	2^{20} (1,048,576)
G =	2^{30} (1,073,741,824)

  For example:

2KB =	2 Kilobytes =	2×2^{10} bytes
4MB =	4 Megabytes =	4×2^{20} bytes
8GB =	8 Gigabytes =	8×2^{30} bytes
2K pixels =	2 Kilopixels =	2×2^{10} pixels
4M pixels =	4 Megapixels =	4×2^{20} pixels

- **Register Access**

  The abbreviations used to indicate the type of access to register fields and bits have the following definitions:

  IGN — Ignore

  Bits and fields specified as IGN are ignored when written.

  RAZ — Read As Zero

  Bits and fields specified as RAZ return a zero when read.

RES — Reserved

Bits and fields specified as RES are reserved by Digital and should not be used.

RO — Read Only

Bits and fields specified as RO can be read and are ignored (not written) on writes.

RW — Read/Write

Bits and fields specified as RW can be read and written.

R/W1C — Read/Write One to Clear

Bits and fields specified as R/W1C can be read. Writing a one clears the bits.

WO — Write Only

Bits and fields specified as WO can be written but not read.

Aligned and Unaligned

The terms *aligned* and *naturally aligned* are interchangeable and refer to data objects that are powers of two in size. An aligned datum of size 2^n is stored in memory at a byte address that is a multiple of 2^n; that is, one that has *n* low-order zeros. For example, an aligned 64-byte stack frame has a memory address that is a multiple of 64.

A datum of size 2^n is *unaligned* if it is stored in a byte address that is not a multiple of 2^n.

Bit Notation

Multiple bit fields are shown as extents (see Ranges and Extents, below).

Caution

Cautions indicate potential damage to equipment or loss of data.

Core, Core Space, and Core Registers

This manual frequently refers to the 21030 core, core space, and core registers.

- The 21030 core is all of the chip functions except the PCI interface and PCI registers.
- The 21030 memory space consists of one to eight copies of core space. Each copy of core space maps the same frame buffer, external EEPROM, and core registers (see Chapter 2).

- The core registers are all the registers physically implemented in the 21030 except the PCI configuration registers. Note that the block color and plane mask registers are addressed in core space but are physically implemented in the VRAMs.

Data Units

The following data unit terminology is used throughout this manual.

Term	Words	Bytes	Bits	Other
Nibble	¼	½	4	
Byte	½	1	8	
Tribyte	1½	3	24	
Word	1	2	16	
Dword	2	4	32	Longword
Quadword	4	8	64	2 Dwords
Octaword	8	16	128	4 Dwords
Hexaword	16	32	256	8 Dwords

External

Unless otherwise stated, throughout this manual the term external means not contained in the DECchip 21030.

Note

Notes emphasize particularly important information.

Numbering

All numbers are decimal or hexadecimal unless otherwise indicated. In cases of ambiguity, a subscript indicates the radix of nondecimal numbers. For example, 19 is decimal, but 19_{16} and 19A are hexadecimal.

Ranges and Extents

Ranges are specified by a pair of numbers separated by two periods (..) and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets (<>) separated by a colon (:) and are inclusive. For example, bits <7:3> specifies an extent including bits 7, 6, 5, 4, and 3.

Signal Names

Signal names are printed in lowercase, bold-faced type. The names of low-asserted signals carry the suffix **_l**. The names of high-asserted signals have no suffix. For example, **pll_clk_in** is a high-asserted signal, and **pll_clk_in_l** is a low-asserted signal.

1
Introduction

This chapter briefly describes the DECchip 21030 PCI Graphics Accelerator and its hardware features, programming model, and supported frame buffer configurations.

1.1 Overview

The 21030 is an application-specific integrated circuit (ASIC) for high-performance 2D and 3D graphics acceleration in systems that use the peripheral component interconnect (PCI). The 21030 accelerates 2D and 3D applications, and enhances X and Win32 windowing system graphics performance. It includes an on-chip glueless PCI interface and is particularly suited for use in high-performance PCs and entry-level workstations.

The 21030 takes advantage of the speed and power of today's microprocessors to provide best-in-class acceleration without the expense and complexity of a separate graphics coprocessor. As a PCI peripheral in systems based on Digital's Alpha AXP architecture or other current microprocessors (see Table 1–1), the 21030 gives desktop products, such as PCs and windows terminals, the graphics performance of a workstation at PC prices.

The 21030 is the latest implementation of Digital's proven graphics architecture, the same architecture that is designed into Digital's Alpha AXP workstations. With the on-chip inclusion of the PCI interface, the 21030 provides a multigenerational plug-and-play graphics system for a variety of current and emerging PCI-based systems. Its high-level of integration provides low-cost and reliable acceleration with the minimum number of additional chips.

The 21030 provides leadership graphics performance for PCI-based desktop workstations and PCs driven by high-performance microprocessors and industry-standard operating systems, graphics user's interfaces (GUIs), and applications programming interfaces (APIs). The 21030 architecture is carefully tuned to work in conjunction with a high-performance CPU, especially an Alpha AXP processor. The 21030 supports the industry-standard

2D graphics APIs Win32 and X and the industry-standard 3D API OpenGL, ported to Windows NT.

Figure 1–1 shows the 21030 in a typical system based on an Alpha AXP microprocessor. The 21030-based graphics subsystem can reside on the motherboard or on a PCI add-in card, possibly opposite the CPU through a PCI-to-PCI bridge.

Figure 1–1 Typical System Application

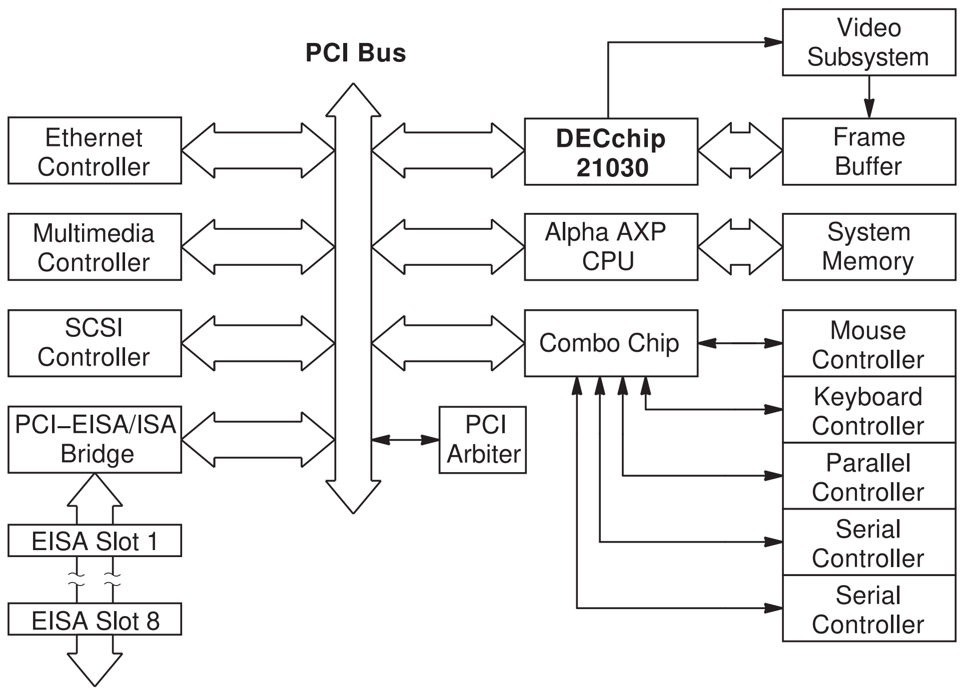


Table 1–1 lists some of the applicable platforms for the 21030.

Table 1–1 Typical Applicable Systems

Processor	Operating Systems	APIs
DECchip 21064	Windows NT	Win32, X, OpenGL
DECchip 21066	Windows NT	Win32, X, OpenGL
DECchip 21068	Windows NT	Win32, X, OpenGL

1.2 Features

The following is a summary of the 21030 hardware features.

- 8-bpp and 32-bpp frame buffers

  The 32-bpp frame buffer supports 8-bpp pseudo color, 12-bpp direct color, and 24-bpp true color.

- 64-byte copy buffer

  The copy buffer supports high-bandwidth local frame buffer bit-block transfers (BitBlts).

- Bresenham line drawing engine and setup hardware

  The on-chip Bresenham line drawing engine and setup hardware performs Bresenham per-pixel line stepping and most of the Bresenham-term setup.

- Faster and simpler line drawing

  In many systems, software is responsible for all of the cumbersome line setup calculations, including generating the Bresenham error and address increments for the major and minor axis steps as well as the initial error term.

  In the 21030's streamlined interface, software needs to write only the absolute *dx* and absolute *dy* values of the line segment to one of eight slope registers, implicitly specifying the drawing octant. The 21030 then automatically generates all of the Bresenham terms, initializes the Bresenham registers, and draws up to 16 pixels. This interface allows software to process more line endpoints, and, combined with 4-way access to the frame buffer, yields 8-bpp line rates in excess of 2,000,000 lines per second.

- Color expansion

  The 21030 expands monochrome bitmaps to various pixel depths, for drawing text or filling regions with solid or bitonal brushes.

- Video RAM (VRAM) block-write support

  VRAM block-write greatly improves the performance of solid fills and common brushed fills.

- Direct memory access (DMA) engine for image data

  The 21030 has DMA-read copy and DMA-write copy modes for fast host-to-screen BitBlts. These modes allow large, contiguous regions to be directly transferred between main memory and the frame buffer. The on-chip PCI interface allows main memory, other PCI graphics devices, or PCI video devices to be the external source or destination.

- High-performance memory controller

  The 64-bit memory interface implements Digital's RapiDraw technology (patent pending) to maximize page mode accesses and support 4-way independent addressing to greatly enhance line drawing performance.
  The memory controller also supports multiple access modes to the frame buffer, which gives 1MB and 2MB frame buffers the same 8-bpp drawing performance as 4MB and 8MB frame buffers.

- 3D support

  The 21030 includes hardware to perform color-interpolation for each of three channels (red, green, and blue), Z-buffering, and stencil processing compatible with OpenGL specifications. These functions accelerate typical 3D primitives such as Gouraud-shaded, Z-buffered polygons and depth-cued lines.

- Proprietary dithering

  The 21030 implements Digital's AccuLook dithering algorithm (patent-pending) to support rendering to 8-bpp and 12-bpp bitmaps. The quality of dithered 8-bpp pseudo-color images surpasses standard 16-bpp direct-color image quality. The quality of the dithered 12-bpp direct-color images is comparable to 24-bpp true-color image quality.

- Support for multiple visual types

  Within the context of a 32-bpp frame buffer, the 21030 supports drawing to 8-bpp and 12-bpp bitmaps as well as typical 24-bpp bitmaps. This feature supports independent bitmap depths per window (that is, per application) and high-speed animation.

- $64 \times 64 \times 2$ on-chip cursor

  The 21030 incorporates on-chip cursor control. Optionally, it can retain a cursor image in its off-screen frame buffer memory and pass its control to the RAMDAC.

- High-performance CRT control (CRTC)

  The 21030 provides monitor timing and VRAM serial-port control signals to refresh screens up to 1600 × 1280 pixels, 76 Hz, and 16M colors.

- Glueless interface for external RAMDAC and flash ROM

  The 21030 can be connected directly to several Brooktree RAMDAC MPU-style interfaces as well as a 256K × 8 flash ROM.

- Stereo support

  The 21030 supports 2 × reduced-resolution vertical refresh with explicit frame indicator output (that is, *stereo goggle control*).

- PCI-compliant interface

  The 21030 is fully electrically compliant with the *PCI Local Bus Specification, Revision 2.0.* The 21030 and its external RAMDAC, EEPROM, and clock generator present only one PCI load.

Functions Not Supported

By design, the 21030 generally provides a lower-level hardware interface than typical GUI accelerators. The 21030 does not support the following common GUI accelerator functions, which have little effect on performance.

- The complete Windows set of 256 Boolean raster operations

  The Windows manager and most applications typically use only three or four of the 256 Boolean raster operations (ROPs). The most commonly used ROPs are included in the 16 functions supported by the 21030 hardware. For the infrequent cases when the 21030 Windows NT display driver encounters an unsupported ROP, it defaults to the graphics device interface (GDI). This does not affect performance for Windows or the majority of applications (including Windows benchmarks) and has only a negligible effect on the performance of the remaining minority of applications.

- On-chip VGA

  Windows NT on an Alpha AXP platform does not require VGA compatibility. The 21030 supports external VGA (that is, *palette snooping*) but does not include VGA register-level compatibility. This saves approximately 12,000 unnecessary gates for applications that do not need VGA.

- Traditional bit-ordering in Windows monochrome bitmaps

  In the 21030 stippling (color expansion) modes, bytes and bits-within-bytes increase from left-to-right across the screen. Traditionally, in Windows monochrome device-independent bitmaps (DIBs), bytes are also mapped from left-to-right, but bits-within-bytes increase from right-to-left with respect to the screen. Although the 21030 hardware does not support such ordering, the 21030 display driver can use the *lazy realization* paradigm of Windows brush-management to handle this format with negligible degradation in performance.

Under Windows, the driver *realizes* a brush only when the brush is ready to be used. Once realized, the brush can be used indefinitely without being realized again. Therefore, the display driver can reverse the bit order once during the realization process and never again for the life of that brush. Because the most commonly used brushes are stock Windows brushes that can be cached after they are realized, they can be used by multiple applications without being realized multiple times. Consequently, one-time brush preprocessing has little effect on performance.

- Advanced rendering

  Functions such as antialiasing and texture mapping must be implemented in software, although the 21030 accelerated modes can be used to assist more advanced rendering algorithms; for example, simple-Z mode can be used for texture mapping.

1.3 Basic Programming Model

In the basic 21030 programming model, the processor writes directly to addresses within the 21030's frame buffer address space. The data is interpreted according to the current graphics mode to perform the desired operation. Exceptions to this paradigm are described below.

There are four primary 21030 operating modes: simple, stipple, line, and copy. Each primary mode of operation has an associated plane mask, Boolean raster operation, and destination bitmap depth. The plane mask determines which bits in a pixel can be modified during a write. The raster operation provides one of sixteen 2-operand Boolean function of source (or pattern) and destination, and automatically performs a read-modify-write cycle when necessary. The bitmap depth (and associated byte selection for 8-bpp bitmaps in a 32-bpp frame buffer) specify how pixel data maps to frame buffer Dwords.

- Simple mode

  In simple mode, writes to the frame buffer are similar to writes to main memory, except for the optional effects of the pixel mask, plane mask, raster operation, bitmap depth, and bitmap rotation. In this mode, the PCI byte mask and the pixel mask determine which pixels are written.

- Stipple mode

In stipple mode (color expansion mode), data written to the frame buffer is interpreted as a monochrome pattern, in which the following occurs:

  – Ones are expanded into foreground pixels.
  – Zeros are either expanded into background pixels (opaque stipple mode) or not expanded (transparent stipple mode).

  In opaque stipple mode, the pixel mask can be programmed to write fewer than 32 pixels.

- Line mode

  In line mode, the processor sets up registers for the Bresenham engine and then writes into the frame buffer at the starting address of the line. The data written by the processor is interpreted as a monochrome pattern, in which the following occurs:

  – Ones are expanded into foreground pixels.
  – Zeros are either expanded into background pixels (opaque line mode) or have no effect (transparent line mode).

  In the 3D line modes, ones are expanded into a shaded, and optionally dithered, color.

- Copy mode

  In copy mode, the processor writes alternately to the source and destination address within the frame buffer. The data written by the processor is interpreted as a bit mask that specifies which pixels are to be read (source) or written (destination).

Extensions to the Basic Programming Model

Several extensions to the basic programming model are available.

- Stipple-fill and block-fill modes

  In stipple-fill and block-fill modes, each write causes the 21030 to fill as many as 2K pixels on a scan line, using the 32-bit data as a 32-bit monochrome pattern. The block-fill modes use VRAM block-write cycles for high-bandwidth fills.

- Line mode

  In line mode, the eight slope registers (one per octant) allow the processor to offload some of the traditional line setup computations. The processor writes the absolute values of the line rise and run to one of the slope registers, implicitly specifying a drawing octant, and causes the 21030 to generate the Bresenham address and error terms and draw up to 16 pixels at one time. Consequently, the processor can specify a short, connected line with one 32-bit write.

- Copy mode

  In copy mode, the copy-64 source and copy-64 destination registers allow the processor to read 64 unmasked bytes from the source and write 64 unmasked bytes to the destination with one write to each register. This makes full use of the 64-byte copy buffer for large area copies of 8-bit pixels.

  In DMA copy modes, the processor can specify the addresses of the source (DMA-read copy mode) or destination (DMA-write copy mode) in PCI memory space. One write to the frame buffer then causes the 21030 to begin reading from the frame buffer (or PCI) and writing to the PCI (or frame buffer), up to 2K transfers at a time.

Chapter 6 describes all of the 21030 modes and processing algorithms that are implemented in hardware.

1.4 Frame Buffer Configurations

While the design-center configurations for the 21030 are 8-bpp and 32-bpp frame buffers, the 21030 supports a variety of 8-, 16-, 24-, and 32-bpp configurations. The supported configurations range from a minimal 8-bpp, 1M-pixel option to a 32-bpp, 2M-pixel option with a full-screen Z-buffer. The 32-bpp frame buffers support 8-bpp, 12-bpp, and 24-bpp bitmap formats. The 21030 supports VRAM-mapped display buffers (that is, the entire visible screen) between 1MB and 16MB. Additional memory for Z-buffers, double-buffers (*back buffers*), or arbitrary off-screen cache and storage can be mapped into DRAM, VRAM, or graphics DRAM (GRAM).

The 21030 normally uses the persistent plane mask and block-write features of advanced VRAMs and GRAMs. However, if these features are not used for drawing off-screen and some performance degradation is acceptable, off-screen memory can be populated with standard DRAMs.

Table 1–2 lists some of the possible frame buffer configurations.

Table 1–2 Supported Frame Buffer Configurations

Size	Resolution @75 Hz NI	Depth	Z-Buffer and Stencil Buffer	Double Buffer
1MB	1024 × 768	8	NA	NA
	800 × 600	16	NA	NA
	640 × 480	24	NA	NA
2MB	1600 × 1280	8	NA	NA
	1024 × 768	16	NA	NA
	800 × 600	24	NA	NA
	640 × 480	32	NA	NA
4MB	1600 × 1280	8	NA	NA
	1600 × 1280	16	NA	NA
	1280 × 1024	24	NA	NA
	1024 × 768	32	Partial	8/8/8, 12/12 *
8MB	1600 × 1280	8	NA	NA
	1600 × 1280	16	NA	NA
	1280 × 1024	24	NA	NA
	1280 × 1024	32	Partial	8/8/8, 12/12 *
	1024 × 768	32	Full	8/8/8, 12/12 *
16MB	1600 × 1280	32	Full	8/8/8, 12/12 *
	1280 × 1024	32	Full	8/8/8, 12/12, full *
	1024 × 768	32	Full	Full *

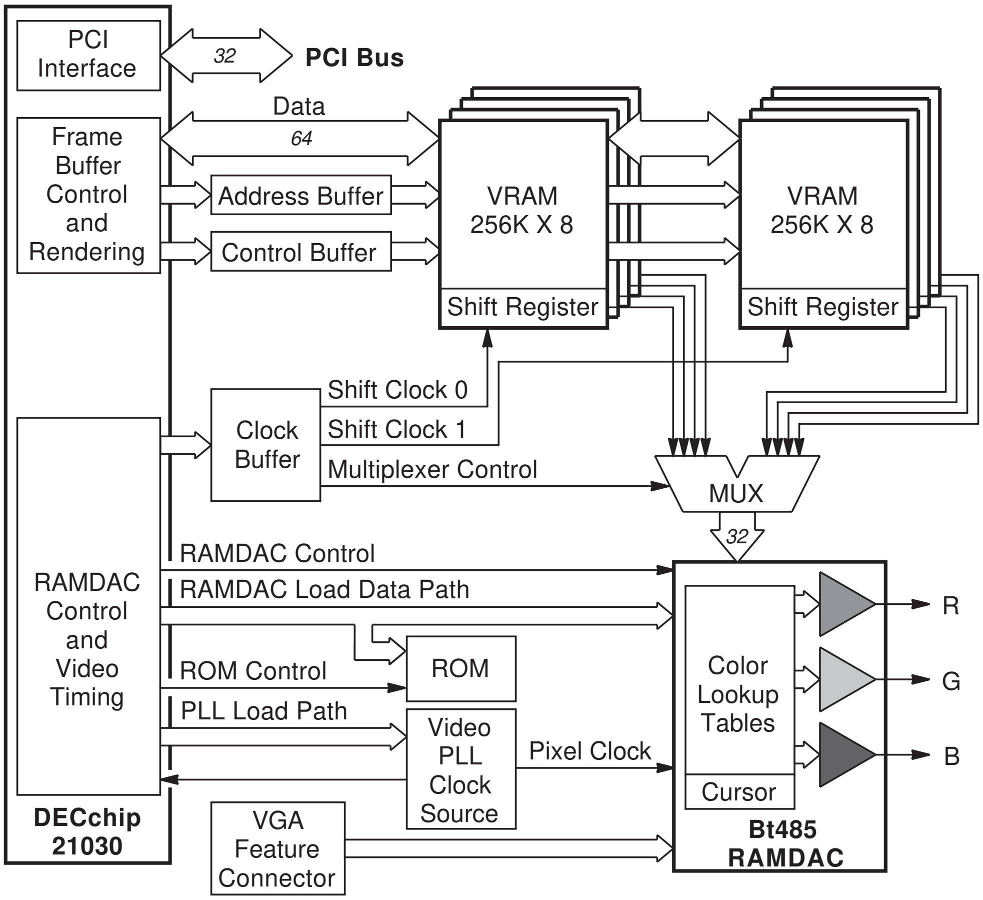
NI — Noninterlaced
NA — Not applicable
* Recommended 3D configurations

Only the 4MB, 8MB, and 16MB 32-bpp configurations provide Z-buffering with stencil support. However, the 21030 can support Z-buffering without a dedicated back buffer to store a full screen of Z data. Any available off-screen space in a 32-bpp frame buffer can be used to perform Z-buffering on a portion of the visible screen (for example, the window for an animation application).

While the 8-bpp frame buffer supports double-buffering only by copy, the 32-bpp frame buffer can support two different types of double-buffering, depending on the configuration of the video back end. A second bank of DRAM can be added as a back buffer to store a full screen of full-depth (24-bit) color. In addition, the 21030 supports multiple formats for 12-bpp and 8-bpp bitmaps within the context of the 32-bpp frame buffer. That is, the 21030 can draw to two 12-bpp bitmaps and three 8-bpp bitmaps at the same pixel locations within 32-bpp buffers. Consequently, in-place double-buffering is possible by coupling the 21030 with either a RAMDAC that can handle images that reside in different planes on a per-window basis (such as the Bt463 RAMDAC) or external multiplexing hardware. By drawing to one 8-bpp or 12-bpp bitmap while displaying from another bitmap, animation can be performed without copying from off-screen to on-screen memory. In-place double-buffering is possible only on the 32-bpp options at reduced pixel depth.

Figure 1–2 shows a typical 8-bpp frame buffer.

Figure 1–2 Typical 8-bpp Configuration



2
Memory Space

The 21030 address space consists of three discrete spaces:

- Memory space, described in this chapter
- Configuration space, which includes all of the PCI configuration registers described in Chapter 4
- Expansion ROM space, a 256KB byte-readable address space described in Section 4.2.6

2.1 Overview

The 21030 memory space includes the following:

- All 21030 registers except the PCI configuration registers
- The 21030 frame buffer
- Alternate ROM space, an alternate Dword-readable 1MB window into the expansion ROM

The size of the 21030 step B memory space is 128MB. It is mapped into the PCI memory space at the address specified in the PCI device base address register (PDBR). The memory space consists of 4 to 32 copies of core space. (See Appendix A for 21030 step A differences.)

2.2 Core Space

Each copy of core space is identical and maps the same frame buffer (frame buffer space), external EEPROM (alternate ROM space), and core registers (register space). Core space ranges between 4MB and 16MB for an 8-bpp frame buffer and between 16MB and 32MB for a 32-bpp frame buffer, depending on the application (Figure 2–1). The organization and size of each core space is a function of the address-mask and deep fields in the deep register (GDER). These fields are programmed with the values needed to organize the core according to the physical size and depth of the frame buffer in a particular configuration.

Table 2–1 shows the core space size and the deep and address mask field values for some of the supported 21030 frame buffer configurations.

Table 2–1 Frame Buffer Configuration and Core Space

Configuration	Physical Memory Size	Core Space Size	Deep* Field	Address* Mask
1MB 8-bpp	1MB	4MB	0	000
2MB 8-bpp	2MB	4MB	0	000
4MB 8-bpp	4MB	8MB	0	100
4MB 8-bpp	8MB	16MB	0	110
4MB 24-bpp	4MB	8MB	1	110
8MB 24-bpp	8MB	16MB	1	110
16MB 24-bpp	16MB	32MB	1	111

*Fields in the GDER (Section 4.4.28)

See Table 1–2 and Section 8.1 for the capabilities and physical layout of each configuration.

Figure 2–1 shows memory space mapping as a function of core space size.

2.2.1 Frame Buffer Space

The frame buffer space consists of one or more buffers, each identified as a *display buffer* or *back buffer*. A display buffer is populated by VRAM and maps to the screen (at least). A back buffer is populated by VRAM, GRAM, or DRAM and maps to such things as a second screen buffer, tiles or brush patterns, and Z and stencil buffers (4MB or 8MB frame buffers only). The block-fill mode, block-stipple mode, and plane-mask feature cannot be used with back buffers populated by DRAMs. The frame buffer space can be accessed through any of the drawing modes described in Chapter 6.

Figures 2–2 and 2–3 show the core space mapping for 8-bpp and 32-bpp frame buffers. Note that some of the reserved regions are aliases for the display and back buffers.

Figure 2–1 21030 step B Memory Space Organization

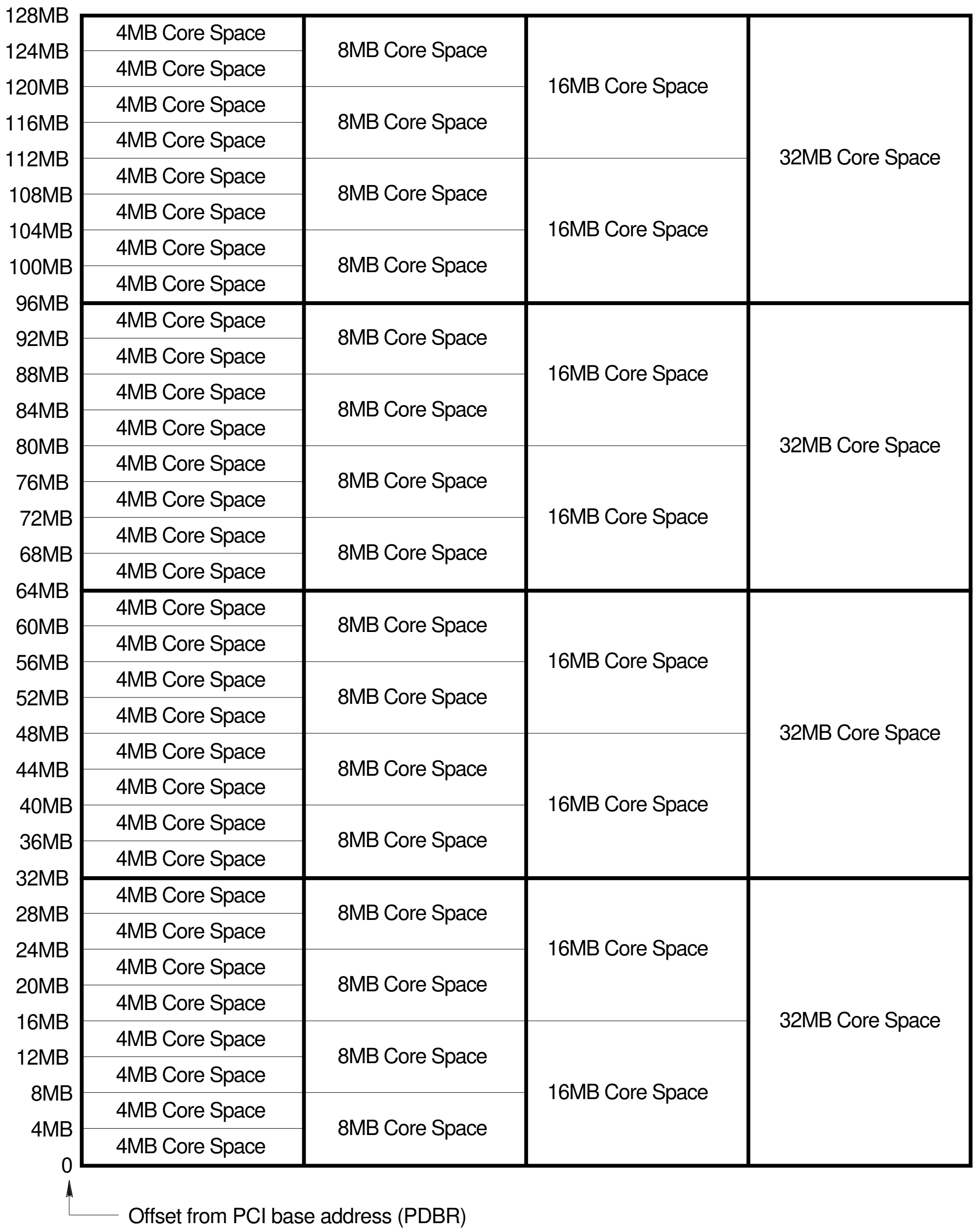


Figure 2–2 Core Space Map for 8-bpp Frame Buffers

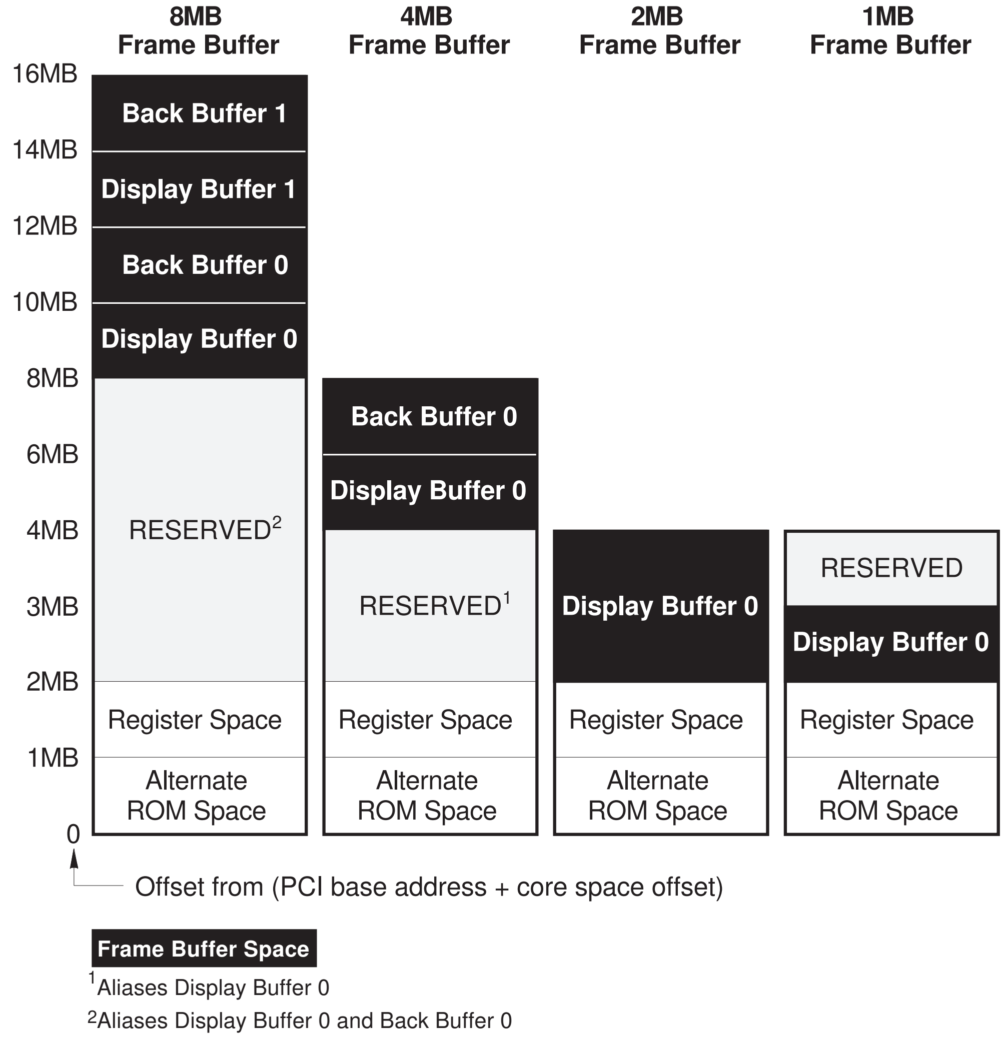
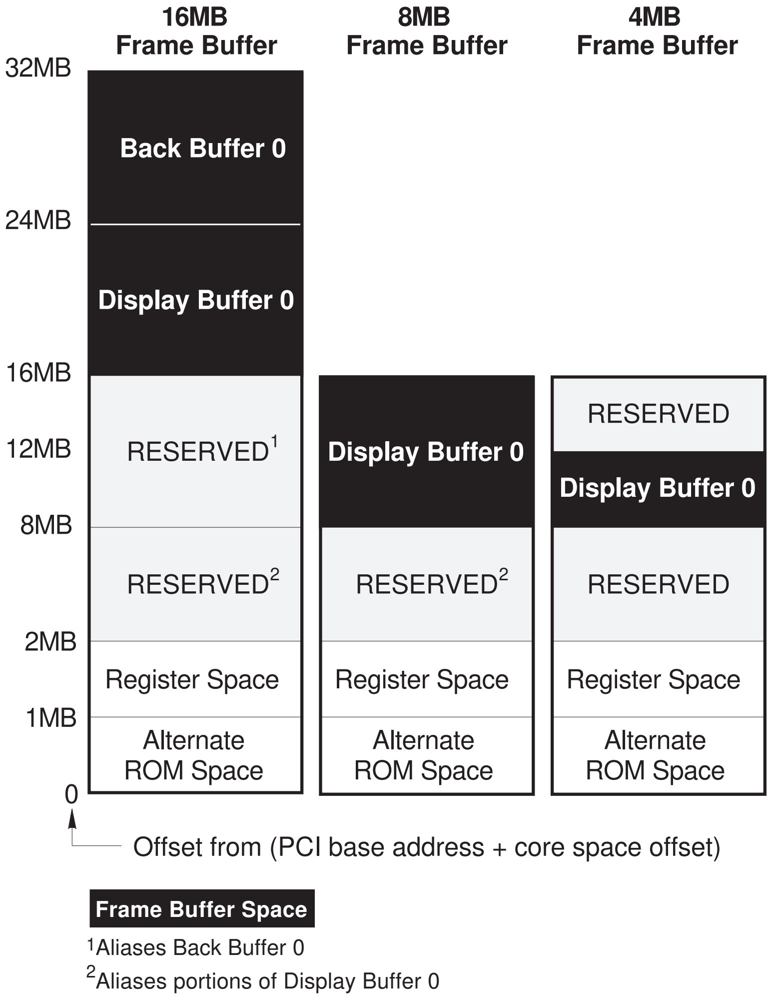


Figure 2–3 Core Space Map for 32-bpp Frame Buffers



2.2.2 Register Space

The register space contains the 21030 core registers; that is, the graphics command, graphics control, video timing, cursor, status, and external device registers. It also maps the block-color and plane-mask registers (physically implemented in VRAM). It does not contain the PCI configuration registers. The 1MB register space is divided into 2K 512-byte regions of register space core (Figure 2–4). Each of the 512-byte cores are identical aliases and maps the registers according to offsets into the 512-byte space (Table 2–2). The register space supports only Dword accesses; individual PCI byte enables are ignored.

Note

To avoid a significant performance degradation, drivers can be programmed to use the register aliases rather than memory barrier (MB) instructions with DECchip 21064, 21066, and 21068 processors (Section 7.3). Digital recommends that the drivers be programmed to use aliases separated by 1KB rather than 512 bytes.

Figure 2–4 shows the register space mapping.

Figure 2–4 Register Space Organization

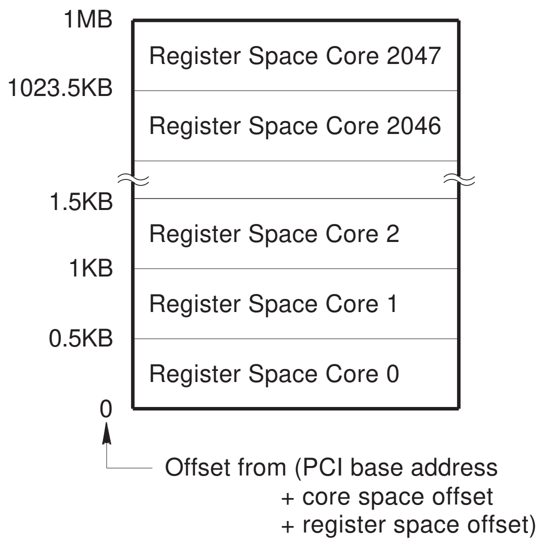


Table 2–2 lists the core registers in descending order-of-offset into register space core. See Appendix C for an alphabetical list of the registers.

Table 2–2 Core Registers

Offset*	Register	Mnemonic	Access
1FC	Reserved	—	—
1F8	Command status register‡	SCSR	RW
1F4	Reserved	—	—
1F0	Palette and DAC data register‡	EPDR	RW
1EC	Reserved	—	—
1E8	Clock register	ECGR	WO
1E4	Reserved	—	—
1E0	EEPROM write register	ERWR	WO
1DC..180	Reserved	—	—
17C	Copy 64 destination register†	GCDR	WO
178	Copy 64 source register†	GCSR	WO
174	Copy 64 destination register†	GCDR	WO
170	Copy 64 source register†	GCSR	WO
16C	Copy 64 destination register†	GCDR	WO
168	Copy 64 source register†	GCSR	WO
164	Copy 64 destination register	GCDR	WO
160	Copy 64 source register	GCSR	WO
15C	Block color register 7	GBCR7	WO
158	Block color register 6	GBCR6	WO
154	Block color register 5	GBCR5	WO
150	Block color register 4	GBCR4	WO
14C	Block color register 3	GBCR3	WO
148	Block color register 2	GBCR2	WO
144	Block color register 1	GBCR1	WO
140	Block color register 0	GBCR0	WO
13C	Slope register 7	GSLR7	WO
138	Slope register 6	GSLR6	WO
134	Slope register 5	GSLR5	WO
130	Slope register 4	GSLR4	WO
12C	Slope register 3	GSLR3	WO
128	Slope register 2	GSLR2	WO
124	Slope register 1	GSLR1	WO
120	Slope register 0	GSLR0	WO
11C	Slope-no-go register 7	GSNR7	WO
118	Slope-no-go register 6	GSNR6	WO
114	Slope-no-go register 5	GSNR5	WO
110	Slope-no-go register 4	GSNR4	WO

*Offset from (PCI base address + core space offset + register space offset)

†Register alias

‡Registers that do not read exactly as written

(continued on next page)

Table 2–2 (Cont.) Core Registers

Offset*	Register	Mnemonic	Access
10C	Slope-no-go register 3	GSNR3	WO
108	Slope-no-go register 2	GSNR2	WO
104	Slope-no-go register 1	GSNR1	WO
100	Slope-no-go register 0	GSNR0	WO
0FC..0C4	Reserved	—	—
0C0	Palette and DAC setup register	EPSR	WO
0BC	Span width register‡	GSWR	WO
0B8	Blue value register	GBVR	RW
0B4	Green value register‡	GGVR	RW
0B0	Red value register‡	GRVR	RW
0AC	Address register†	GADR	WO
0A8	Z-base address register	GZBR	RW
0A4	Z-value high register‡	GZVR-H	RW
0A0	Z-value low register	GZVR-L	RW
09C	Bresenham width register	GBWR	RW
098	DMA base address register	GDBR	RW
094	Z-increment high register	GZIR-H	RW
090	Z-increment low register	GZIR-L	RW
08C	Blue increment register	GBIR	RW
088	Green increment register	GGIR	RW
084	Red increment register	GRIR	RW
080	Data register	GDAR	RW
07C	Interrupt status register	SISR	RW
078	Video shift address register‡	VSAR	RW
074	Cursor XY register‡	CXYR	RW
070	Video valid register	VVVR	RW
06C	Video base address register	VVBR	WO
068	Vertical control register	VVCR	RW
064	Horizontal control register	VHCR	RW
060	Cursor base address register	CCBR	RW
05C	Pixel mask (persistent) register	GPXR	WO
058	Stencil mode register	GSMR	RW
054	Reserved	—	—
050	Deep register	GDER	RW
04C	Continue register‡	GCTR	RW
048	Bresenham-3 register‡	GB3R	RW
044	Bresenham-2 register	GB2R	RW

*Offset from (PCI base address + core space offset + register space offset)

†Register alias

‡Registers that do not read exactly as written

(continued on next page)

Table 2–2 (Cont.) Core Registers

Offset*	Register	Mnemonic	Access
040	Bresenham-1 register	GB1R	RW
03C	Address register	GADR	RW
038	Pixel shift register	GPSR	RW
034	Raster operation register	GOPR	RW
030	Mode register‡	GMOR	RW
02C	Pixel mask (one-shot) register	GPXR	RW
028	Plane mask register	GPMR	WO
024	Background register	GBGR	RW
020	Foreground register	GFGR	RW
01C	Copy buffer register 7	GCBR7	RW
018	Copy buffer register 6	GCBR6	RW
014	Copy buffer register 5	GCBR5	RW
010	Copy buffer register 4	GCBR4	RW
00C	Copy buffer register 3	GCBR3	RW
008	Copy buffer register 2	GCBR2	RW
004	Copy buffer register 1	GCBR1	RW
000	Copy buffer register 0	GCBR0	RW

*Offset from (PCI base address + core space offset + register space offset)

‡Registers that do not read exactly as written

2.2.3 Alternate ROM Space

Note

The 21030 supports one optional, external ROM (normally an EEPROM). It is accessible either in the standard PCI expansion ROM space or in the alternate ROM space. The EEPROM is typically a flash ROM.

The 1MB alternate ROM space is embedded in core space. It provides an alternate, read-only map of the EEPROM in addition to the standard PCI expansion ROM space. The PCI expansion ROM space is an independent, 256KB, byte-readable address space. Its location in PCI memory space is defined by the PCI expansion ROM base address register (PRBR, Section 4.2.6).

Unlike the PCI expansion ROM space, alternate ROM space is not byte-contiguous. Each Dword read returns 1 byte of ROM data. Alternate ROM space, in which 3 null bytes exist between consecutive valid ROM bytes, is effectively a sparse version of the PCI expansion ROM space.

Figure 2–5 shows the format (on the PCI bus) of the Dword read from alternate ROM space, and Table 2–3 describes the fields.

Figure 2–5 Alternate ROM Space Read Format

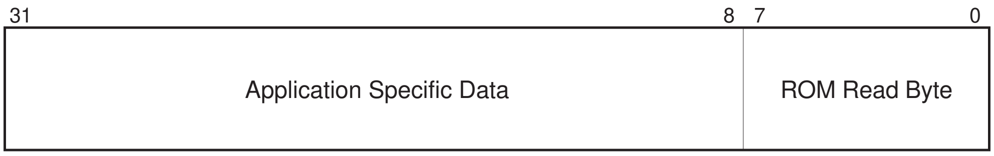


Table 2–3 Alternate ROM Space Read Field Description

Bits	Field	Description
31:8	Application specific data	Arbitrary data that can be read from an external source along with the EEPROM byte
7:0	ROM Read Byte	The desired byte read from external EEPROM at the alternate ROM space Dword address

Because of the sparse layout of alternate ROM space, software must effectively multiply the desired byte offset by 4 (left shift 2 bits) to get the correct alternate ROM space address. The alternate ROM space Dword address is determined as follows:

$$\begin{aligned} \textit{alternate ROM space address} = \textit{PCI base address} &+ \textit{core space offset} \\ &+ \textit{alternate ROM space offset} \\ &+ (\textit{desired byte address} \times 4) \end{aligned}$$

2.2.3.1 Reading Alternate ROM Space

The EEPROM is read through the LSBs the 64-bit memory port. To avoid bus contention, the 21030 disables its drivers and the RAM data bus drivers before reading the EEPROM. Externally, the byte-wide EEPROM must be located on the low byte (**data<7:0>**) of the RAM data bus. When the 21030 reads the low byte, it also latches the next 3 bytes, (**data<31:8>**). This connectivity allows reading arbitrary, application-specific data, such as configuration information (frame buffer size, monitor type, and so on), in the 3 upper bytes of the Dword (Figure 2–5). For example, in Digital options, the option ID is hardwired into the Dword MSBs. The EEPROM data and optional application-specific data must not be driven on **data<31:0>** unless the EEPROM chip enable (**romce_**

l) and output enable (**romoe_l**) pins are active. (See Section 8.3 for more information about the hardware interface to the external EEPROM.)

2.2.3.2 Writing Alternate ROM Space

Writes to alternate ROM space address either the continue register (GCTR) or address register (GADR) according to the specific write address, as follows:

If *alternate ROM space base address* ≤ *(address MOD 8 = 0)* < *alternate ROM space + 512K* then the write addresses the GADR.

If *alternate ROM space base address* ≤ *(address MOD 8 = 4)* < *alternate ROM space + 512K* then the write addresses the GCTR.

In other words, when writing to the first 512KB of alternate ROM space, writes to even locations address the GADR, and all other writes address the GCTR. Sequential access to the GADR and GCTR is useful for 21030 graphics processing on Alpha AXP platforms (Section 7.3.2). The alternate ROM space supports only Dword write accesses; individual PCI byte enables are ignored.

Note

The EEPROM cannot be written through alternate ROM space; the EEPROM write register (ERWR, Section 4.8.1) must be used to write the EEPROM.

3
Internal Architecture

This chapter describes the 21030 microarchitecture. Figure 3–1 is a block diagram of the chip showing its major functional areas. The functions are described in the remainder of this chapter.

3.1 PCI Interface

The PCI interface connects the 21030 core to the PCI bus. The primary function of the PCI interface is to keep the command FIFO filled with writes and commands issued over the PCI to the 21030 registers and frame buffer.

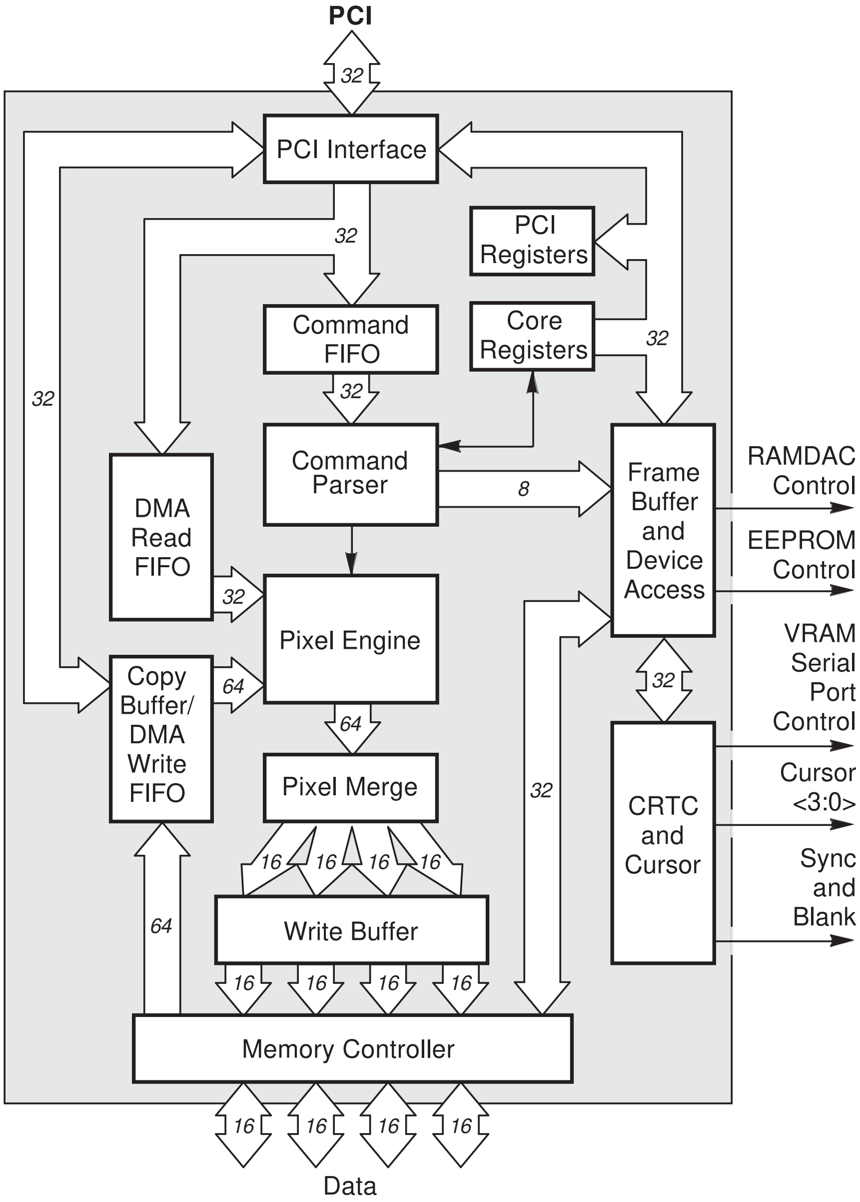
The PCI interface provides read access to all 21030 core registers and external devices such as the EEPROM, RAMDAC, and clock generator. It also provides exclusive read and write access to the 21030 PCI configuration registers.

The PCI interface supports most of the PCI cycles as a target. It also allows the 21030 to be a PCI master for direct memory access (DMA) operations, transferring pixel data between the 21030 frame buffer and memory that can be accessed from the PCI. DMA read data is taken from the PCI and passed to the DMA read FIFO; DMA write data is taken from the DMA write FIFO and burst over the PCI. As a target or master, the PCI interface initiates and responds to different types of termination sequences. (See Chapter 5 for more information about supported PCI transactions and terminations.)

As a target, the PCI interface decodes addresses to the following 21030 address spaces.

- PCI configuration register space
- PCI expansion ROM space
- 21030 core space
- VGA color register I/O space

Figure 3–1 DECchip 21030 Block Diagram



3.1.1 PCI Configuration Reads and Writes

When the PCI interface detects a PCI configuration write or read operation while the **idsel_l** signal is asserted and **ad<1:0>** = 00, it latches data into or fetches data from the PCI configuration register indexed by the **ad<7:2>** signals. The PCI interface controls all access to the PCI configuration registers. (See Chapter 4 for more information about the PCI configuration registers.)

3.1.2 Memory Reads and Writes

The PCI interface decodes all memory read and write transactions and detects accesses to either the 21030 expansion ROM space or core space. These address spaces are specified by the PCI device base address register (PDBR, Section 4.2.2) and PCI expansion ROM register (PRBR, Section 4.2.6). Core space maps the entire frame buffer, the alternate ROM space, and all of the 21030 registers except the PCI configuration registers. (See Chapter 2 for more information about 21030 address space mapping.)

3.1.2.1 Memory Write to Core Space

On a memory write to core space, the PCI interface loads the write address and data into the command FIFO. On a memory write burst, the interface loads the starting address and successive Dwords of data into the command FIFO. If the command FIFO becomes full at any data phase, the interface waits for 8 PCI clock cycles for entries to become free. If a command FIFO entry is still not free, the PCI interface terminates the transaction.

3.1.2.2 Memory Read of Core Space

On a memory read of core space, the PCI interface fetches data from one of the following:

- A core register
- The frame buffer, through the frame buffer and device access (FBDA) function
- The alternate ROM space, through the FBDA function

The interface drives the read data on the PCI and immediately terminates the transaction. The 21030 does not support burst read cycles, and terminates such transactions as soon as the second data phase begins.

3.1.2.3 Read Interlock

The PCI interface provides a read interlock for the 21030 core registers, frame buffer, and all external devices. A read of these objects is not complete until the 21030 is idle; that is, the busy bit is clear in the command status register (SCSR <0>, Section 4.7.1). The PCI interface waits 8 PCI clock cycles for the chip to become idle. If the chip is still not idle, the PCI interface retries the read operation.

Note that the PCI configuration registers and SCSR are exceptions; read data is returned from these registers whether the busy bit is set or clear.

3.1.2.4 Memory Read of Expansion ROM Space

On a memory read of expansion ROM space, the PCI interface shifts the address left 2 bits, to map to the alternate ROM space, and then forwards the request to the FBDA function. To complete the transaction, the interface shifts the bytes that are read to align the data to the PCI byte masks.

3.1.3 DMA Transfers

The command parser can request a DMA read or write transfer over the PCI. While a DMA operation is in progress, the PCI interface retries all target accesses except those to the SCSR or PCI configuration space.

3.1.3.1 DMA Read Transfer

If the command parser requests a DMA read transfer, the PCI interface requests the PCI bus. When the bus is granted, the PCI interface attempts to read from the specified address until the request is completed and as long as the DMA read FIFO is not full. If the DMA read FIFO becomes full, the interface immediately terminates the transaction by deasserting **frame_l** according to the PCI protocol.

3.1.3.2 DMA Write Transfer

If the command parser requests a DMA write transfer, the PCI interface requests the PCI bus and the memory controller returns source pixel data to the DMA write FIFO. When the bus is granted, the PCI interface fetches consecutive Dwords from the FIFO and drives them over the bus, starting at the specified address. If the target terminates the transaction, the PCI interface immediately terminates the transaction by deasserting **frame_l** according to the PCI protocol.

3.2 DMA Read FIFO

The DMA read FIFO contains 8 Dword entries. It is loaded by the PCI interface during a DMA-read copy operation and unloaded by the pixel engine. The DMA read FIFO contains only pixel data.

The DMA read FIFO is a boundary between chip clocking domains. The input runs at the PCI clock rate and the output runs at the 21030 core clock rate.

3.3 Copy Buffer and DMA Write FIFO

The copy buffer contains 8 quadword (64-bit) entries. It is used when transferring data from a frame buffer source to a destination in either the frame buffer or PCI-accessible memory.

The memory controller returns source data to the copy buffer. In copy mode, the pixel engine forwards the data, tagged with a destination address, down the pixel processing pipeline to the memory controller. In DMA-write copy mode, the copy buffer acts as a DMA write FIFO. The PCI interface fetches the data for transfer over the PCI.

3.4 Command FIFO

The command FIFO contains 16 Dword entries. It buffers writes to the frame buffer and core registers for processing by the 21030 core. The PCI interface loads the command FIFO with an address followed by an arbitrary number of data entries. The command parser unloads the entries and initiates processing.

The command FIFO contains only core-space write data, such as writes to the 21030 core registers, alternate ROM space, and frame buffer space. Because the PCI interface accepts burst memory writes as a PCI target, the command FIFO can independently store an address or data in each of its 16 entries. In other words, the command FIFO can hold any combination of addresses and data, from one address and 15 entries of burst data to eight pairs of address and data entries. If the command parser detects a sequence of one address and multiple data entries, it generates and matches the correct address to each data entry when it unloads the command FIFO.

The command FIFO is a boundary between chip clocking domains. The input runs at the PCI clock rate and the output runs at the 21030 core clock rate.

3.5 Command Parser

The command parser processes graphics commands and register write accesses. It unloads graphics commands (in the form of address and data) from the command FIFO and performs initial processing before passing commands to the pixel engine. If the command parser detects a sequence of one address and multiple data entries, it generates an address for each data entry.

The command parser runs at the 21030 core clock rate.

3.5.1 Pixel-Processing Pipeline Coherence

The pixel-processing pipeline consists of the pixel engine, pixel-merge function, write buffer, and memory controller. The command parser imposes hardware register interlocks to ensure coherent processing through the pipeline. The interlocks allow the pipeline to operate concurrently with register updates; that is, updates to graphics operation parameters.

Most of the parameter registers are double-buffered. The command parser schedules buffered-register loading and swapping, and, in certain cases, delays command processing to maintain parameter coherence through the pipeline. In the case of writes to the command status register (SCSR), raster operation register (GOPR), and mode register (GMOR), the interlock mechanism waits until the pipeline has been flushed before resuming processing. (See Section 6.1.4 for information about register accesses that are not managed by hardware interlock and require software scheduling.)

3.5.2 External Device and Register Writes

The command parser detects all writes to the external RAMDAC through the RAMDAC setup and data registers (EPSR and EPDR) and to the external EEPROM through the EEPROM write register (ERWR). It forwards the accesses directly to the FBDA function for processing.

The command parser forwards writes to the VRAM-resident block color registers (GBCR<7:0>) and plane mask registers (GPMRs) through the pixel-processing pipeline.

The command parser writes directly to all other registers.

3.5.3 Frame Buffer Writes

The command parser detects all writes to the frame buffer and begins processing the graphics command specified by the current graphics mode. The command parser does not perform any pixel address or data calculations, but forwards *predigested* commands to the pixel engine for processing.

For all fill mode drawing (opaque fill, transparent fill, block fill, DMA-read copy, and DMA-write copy), the command parser breaks large-span fill commands into 32-pixel span commands which the pixel engine can accept and process. The pixel engine can process individual pixels, 16-pixel lines, and 32-pixel spans.

3.5.4 Bresenham Setup Hardware

The command parser incorporates the Bresenham setup hardware. When the command parser receives a write to the slope registers (GSLR<7:0>), span width register (GSWR), or slope-no-go registers (GSNR<7:0>), it calculates the Bresenham terms: length, initial error, error increments 1 and 2, and address increments 1 and 2. When the write is to a slope register or the span width register, the command parser also forwards the line command to the pixel engine. The command parser forwards all other line, span, and pixel mode drawing commands directly to the pixel engine.

3.6 Pixel Engine

The pixel engine does all of the pixel address and value calculations. It receives single-pixel, 16-pixel line, and 32-pixel span commands from the command parser and reduces them into individual 16-bit-aligned or 64-bit-aligned frame buffer address and data pairs destined for the memory controller. The pixel engine contains all of the following pixel processing hardware to generate pixel addresses and data.

- Stipple logic

  The stipple logic expands a monochrome bitmap (and optional bitmap mask) into foreground or background color (or neither), on a per-pixel basis over a 16-pixel line or 32-pixel span.

- Bresenham engine

  The Bresenham engine steps through the pixels of a line (up to 16 pixels at a time), generating a pixel address and, optionally, a Z address for each step.

- Color interpolators

  The color interpolators generate linearly interpolated 8.12 (integer.fraction) format values for each channel (red, green and blue) at each Bresenham engine step. The color interpolators include three independent adders running in lock-step with the Bresenham engine as it steps along the line. The color interpolators can also generate a single, interpolated, 8-bit grey-scale value in a sequential-interpolation drawing mode.

- Z-interpolator

  In Z-buffered modes, the Z-interpolator generates a linearly interpolated, 24.12 format Z value at each Bresenham engine step across a line or span. It also compares each Z value read from the frame buffer with the calculated value for each step. The results of the comparison determine whether the calculated Z value and the calculated pixel value are written.

- Dither logic

  The dither logic implements Digital's AccuLook dithering algorithm. The algorithm maps 8 bits per channel RGB (24-bpp) color to 4 bits per channel (12-bpp) and 3:3:2 (R:G:B) 8-bpp pseudo-color. The source of the 24-bpp RGB can be selected from the color interpolators or the DMA-read copy 24-bit pixel stream.

After the pixel engine reduces spans into pixels and calculates the mode-dependent pixel data, it translates pixel addresses into frame buffer addresses as a function of the frame buffer depth and target bitmap. The pixel engine forwards each memory access to the pixel-merge function.

The pixel engine also processes register-write requests for the VRAM-resident block color (GBCR<7:0>) and plane mask (GPMR) registers. It aligns the write data to the appropriate 32 bits of the frame buffer data path and forwards the write to the pixel-merge function.

The pixel engine passes a control tag with each address and data pair that it passes to the pixel-merge function. The tag indicates:

- VRAM cycle type—color register cycle, plane mask cycle, or standard read or write cycle
- Whether the access is a read or write
- Whether a write is in block mode

The pixel engine also passes a channel-synchronization code to the pixel-merge function. Processing-channel synchronization is necessary for a read access or when accessing a 32-bpp frame buffer in an unpacked 8-bpp bitmap format.

The pixel engine receives data directly from the memory controller for the following classes of operation.

- Z-buffered spans and lines

  The pixel engine sends a read request at the current Z-address (generated by the Bresenham engine) to the pixel-merge function. Eventually, the memory controller receives the read request and returns the reference-Z value to the pixel engine. The pixel engine then compares the reference-Z value with the calculated-Z value from the Z-interpolator.

- Copy mode

  In copy mode, the pixel engine first forwards a series of read requests, tagged with the source address, to the pixel-merge function. Eventually, the memory controller returns source pixel data (64 bits at a time) to the copy buffer/DMA write FIFO. Then, when instructed by the command parser, the pixel engine unloads the copy buffer and forwards that data back to the pixel-merge function as a write tagged with the destination address.

- DMA-write copy mode

  In DMA-write copy mode, the memory controller returns source pixel data to the copy buffer/DMA write FIFO. The pixel engine unloads the FIFO and transfers the data to the PCI interface for transfer over the PCI.

The pixel engine runs at the core clock rate.

3.7 Pixel Merge

The pixel-merge function merges byte-writes to eliminate consecutive writes to different bytes at the same 16-bit address. (The Bresenham engine often generates such sequences.) By eliminating redundant writes to the same address, the pixel-merge function greatly improves line drawing rates.

The pixel-merge function data path is divided into 16-bit channels. Merging occurs only within channels. Each channel receives frame buffer write requests from the pixel engine, temporarily stores the most recent request, and tags the byte to be written. If the next write request is to a different byte at the same channel address as the previous request, the pixel-merge function merges the two requests into one write. In all other cases, the pixel-merge function forwards each separate request downstream to the write buffer, maintaining the order of the requests. The pixel-merge function does not collapse consecutive writes to the same byte address.

The pixel-merge function runs at the core clock rate.

3.8 Write Buffer

The write buffer is an eight-entry FIFO. It is divided into four independent 16-bit channels in the same way as the pixel-merge function. The FIFO buffers post-merged frame buffer requests from the pixel engine. The requests are unloaded by the memory controller and processed sequentially for VRAM access.

The pixel engine runs at twice the speed of the memory controller's maximum page-mode access time. Therefore, depending on the operation and address sequence, the pixel engine can generate address and data pairs much slower or much faster than the memory controller can process the requests. The write buffer helps to smooth the access rate at the VRAM interface and optimize throughput over time.

The write buffer runs at the core clock rate.

3.9 Memory Controller

The memory controller provides the interface to the VRAM frame buffer. It is actually four independent memory controllers. Each controller addresses a 16-bit channel of the 64-bit memory bus and can independently address, read, and write its channel.

The memory controller responds to requests from two sources: the pixel engine and the FBDA function. It responds to requests from the pixel engine (through the pixel-merge function and write buffer) for accelerated drawing operations in the frame buffer. It also responds to requests from the FBDA function. The FBDA function makes occasional asynchronous requests for the following:

- Direct host reads of the frame buffer
- RAMDAC reads and writes
- External EEPROM reads and writes
- Cursor data fetches
- VRAM read-transfer cycles to support screen refresh

To conserve 21030 pins, the RAMDAC data lines and the external EEPROM address and data lines are tied to a subset of the memory controller address and data lines. Therefore, to read or write the RAMDAC or external EEPROM, the memory controller must interrupt processing of write buffer address, data, and tag triplets.

The memory controller can process requests for all of the following VRAM cycle types:

- Page-mode read and write cycles, which are used for most graphics operations.
- Block-write cycles, which are used for block-fill and block-stipple mode graphics operations. Each address and data pair unloaded from the write buffer is tagged to indicate whether the write is a block write or standard page-mode write.
- CAS-before-RAS dynamic memory refresh cycles.
- Standard read-transfer cycles.
- Split read-transfer cycles.
- Block-color register read and write cycles.
- Plane-mask register read and write cycles.

As long as the write buffer contains valid entries, each memory controller continues to unload address, data, and tag triplets from its corresponding channel. The tag determines how the memory controller processes the associated address and data pair. The tag contains the following information:

- Read or write—Specifies the type of operation.
- Block-write enable—On a write access, instructs the memory controller to execute a block-mode write cycle.
- Color register write—Specifies that the write is a VRAM block-color register write.
- Plane mask register write—Specifies that the write is a VRAM plane-mask register write.
- Synchronization code—Specifies whether an individual memory controller should synchronize with the other controllers before proceeding with an access. If the tag does not specify a synchronization code, the memory controllers are free to work independently, asynchronously processing entries from the write buffer.

The memory controller processes the address and data pair from each write-buffer entry as specified by the tag. Independent of the tag, the memory controller performs a Boolean ROP function on each write (except block write), as specified by the raster operation register (GOPR). If the ROP is a function of the destination, the memory controller automatically performs the necessary read-modify-write operation.

The memory controller returns requested read data to the pixel engine through the copy buffer.

When the memory controller detects an interrupt from the FBDA function, it suspends write-buffer entry processing within a maximum latency and services the interrupt. The FBDA function specifies the type of access and passes address and data as required. The memory controller services the interrupts as follows:

- RAMDAC access—The memory controller drives (writes) or latches (reads) a byte of data.
- External EEPROM access—The memory controller drives an address and either drives (write) or latches (read) a byte of data.
- Cursor data fetch—The memory controller performs a frame buffer read at the specified address and returns two quadwords to the FBDA function.
- Standard or split read-transfer cycle request—The memory controller executes the cycle with the internally stored, current frame address. After each transfer cycle is completed, the memory controller updates its frame pointer to select the next address from which to transfer. In addition, the CRTC and cursor function signals the memory controller when to update its current frame pointer to top-of-frame.

After an asynchronous access has been serviced, the memory controller resumes processing write-buffer address, data, and tag triplets.

The memory controller also issues CAS-before-RAS refresh cycles frequently enough to keep the dynamic memory refreshed.

The memory controller runs at the core clock rate, and its CAS cycle time is twice the core clock rate.

3.10 CRTC and Cursor

The CRTC and cursor function provides monitor timing, schedules screen refresh, and provides a 2-bpp cursor during refresh. It directly drives the external cursor, sync, blank, and VRAM serial port control signals.

3.10.1 Monitor Timing

The CRTC and cursor function provides digital, composite sync and blank signals to drive a noninterlaced monitor. The signal edges are specified by the parameters in the horizontal and vertical control registers (VHCR and VVCR). The CRTC and cursor function also provides stereo control. It drives stereo, sync, and blank control signals through pins **blank_l**, **vsync_l**, and **hysnc_l**.

3.10.2 Video Refresh

The CRTC and cursor function monitors how often the 21030 must issue VRAM read-transfer cycles to keep the serial-access memories (SAMs) replenished during active scan time. At regular intervals, the CRTC and cursor function requests the memory controller (through the FBDA function) to perform a standard or split read-transfer cycle. The CRTC and cursor function also generates the signals that control the VRAM serial port clock and external video multiplexers, driving the control signals through the **toggle** and **hold_l** pins.

3.10.3 Cursor Generation

The CRTC and cursor function monitors which scan line is currently being refreshed. During the horizontal blank time preceding a scan line that intersects the cursor, the CRTC and cursor function generates a request (through the FBDA function) for the memory controller to read a scan line's worth of 2-bpp cursor. Then, at the proper position during the horizontal scan, the CRTC and cursor function drives up to 64 consecutive 2-bit cursor values on the **cursor<1:0>** pins synchronously with the video stream and monitor timing.

The CRTC and cursor function runs at one-fourth of the monitor dot clock rate.

3.11 Frame Buffer and Device Access

The frame buffer and device access (FBDA) function collects requests for access to the frame buffer and external devices (RAMDAC, EEPROM, and clock generator) from several sources. It prioritizes and forwards the requests to the memory controller. The memory controller processes the requests as interrupts to write-buffer processing. The following requests are routed to the FBDA function.

- Direct frame buffer read—From the host through the PCI interface.
- Cursor frame buffer read—CRTC and cursor function reads of the cursor array (stored in the off-screen frame buffer).
- VRAM read transfers—CRTC and cursor function requests for the memory controller to execute VRAM standard and split read-transfer cycles.
- RAMDAC read and write—Palette and DAC data register (EPDR) write requests detected by the command parser and read requests detected by the PCI interface.

The FBDA function provides the signals required to control up to two RAMDACs on pins **dacc<2:0>**, **dacrw**, and **dacce<1:0>**. The FBDA function directly controls the signals to the RAMDAC and external EEPROM. Two of the RAMDAC interface signals are also used to control the external clock generator.

- External EEPROM read and write—Alternate ROM space or PCI expansion ROM space read requests detected and routed by the PCI interface, and EEPROM write register (ERWR) write requests detected and routed by the command parser.

  The FBDA function provides the signals required to write one 8-bit EEPROM and read up to 32 bits of EEPROM (and system-specific) data on pins **romce_l**, **romoe_l**, and **romwe_l**.

- Clock generator write—Clock generator register (ECGR) write requests detected and routed by the command parser.

  The FBDA function provides the clock generator data and hold signals on RAMDAC interface pins **dacc1** (hold) and **dacc0** (data).

The maximum latencies for requests serviced by the FBDA function (and processed by the memory controller) are as follows:

- Cursor data is returned some minimum period before the end of horizontal blank time.
- VRAM read-transfer cycles are completed before the last valid pixels from the previous read-transfer cycle have been shifted out of the SAMs.

3.12 PCI Registers

The PCI registers reside in the 21030 PCI configuration space and include the device-independent registers required for all PCI devices as well as the PCI device registers specific to the 21030. (The PCI registers are described in Chapter 4.)

3.13 Core Registers

The core registers are all the registers physically implemented in the 21030 except the PCI configuration registers. (Note that the block color and plane mask registers are addressed in core space but are physically implemented in the VRAMs.) Many of the core registers are double-buffered to allow pipelined graphics processing to overlap register updates. The command parser controls the core register read access, write access, and double-buffering. (The core registers are described in Chapter 4.)

4 Register Descriptions

This chapter describes all of the 21030 registers.

4.1 Overview

With a few exceptions, all of the 21030 registers can be read and written. Reserved fields return zero when read and are ignored on writes. Most of the registers are cleared at power-up and while chip reset is asserted. Exceptions are noted in the appropriate register descriptions.

On reads, the command status register (SCSR) and all PCI configuration registers are immediately accessible. Reads to other registers do not complete until the command buffer is flushed, because writes might be in progress.

The registers are divided into two classes and several subclasses, as follows:

- PCI registers

  The PCI registers control PCI configuration for the 21030 device.

  – Device-independent registers are required in all PCI devices to implement generic PCI functions.

  – Device-specific registers implement PCI functions specific to the device.

- Core registers

  The core registers implement the core functions.

  – Graphics command registers initiate graphics operations.

  – Graphics control registers provide the parameters for graphics operations.

  – Video timing registers control the timing of monitor sync, blank, and screen refresh signals.

  – Cursor control registers define the location and display of the chip's $64 \times 64 \times 2$ cursor.

– Status registers indicate the current status of chip processing and pending interrupts, enable interrupts, and provide a mechanism for scheduling commands.

– External device registers provide access to the external RAMDAC, EEPROM, and clock generator.

Note

The abbreviations in the type column of the register field description tables indicate field access behavior. The abbreviations are defined in the Conventions section of the Preface.

4.2 PCI Registers

The PCI registers control the PCI configuration. They identify the device and vendor, soft-map the device in I/O or memory space, and specify the allowed modes of operation for the device as a PCI master and PCI target. All PCI devices must implement the PCI configuration registers.

The PCI registers populate the 21030 configuration space as shown in Table 4–1. All other configuration space addresses are reserved. Configuration space occupies 256 bytes, as follows:

- Device-independent, configuration-space header block—64 bytes

  The device-independent registers are implemented in the configuration-space header block.

- Device-specific register set—192 bytes

  The device-specific registers implement VGA pass-through video mode and extend PCI addressing for systems based on Alpha AXP microprocessors. They are implemented in the device-dependent configuration space.

Table 4–1 shows how the 256-byte PCI configuration space is mapped and lists the PCI registers in descending address order.

Table 4–1 21030 PCI Configuration Space

PCI Registers	Mnemonic	Byte Address Range
Device-Dependent Configuration Space		
Reserved	—	FF..48
PCI Address extension register *	PAER	47..44
PCI VGA redirect register	PVRR	43..40
Configuration Space Header Block		
PCI line interrupt register	PLIR	3F..3C
Reserved	—	3B..34
PCI expansion ROM base address register	PRBR	33..30
Reserved	—	2F..14
PCI device base address register	PDBR	13..10
PCI latency timer register	PLTR	0F..0C
PCI class and revision register	PCRR	0B..08
PCI command and status register	PCSR	07..04
PCI identification register	PIDR	03..00

*Active only in the 21030 step A. See Appendix A for more information.

For more information about PCI configuration space organization, see the *PCI Local Bus Specification, Revision 2.0*.

4.2.1 PCI Command and Status Register

Figure 4–1 shows the PCI command and status register (PCSR) format, and Table 4–2 describes its fields.

Figure 4–1 PCSR Format

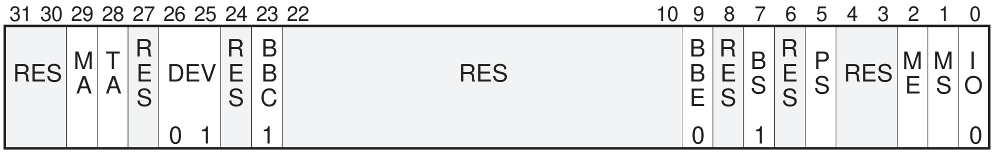


Table 4–2 PCSR Field Description

Bits	Field	Type	Description
31:30	RES	RAZ/IGN	Reserved.
29	MA	R/W1C	Master abort—Set when the 21030 issues a master-abort termination; otherwise, clear.
28	TA	R/W1C	Target abort—Set when the 21030 detects a target-abort termination; otherwise, clear.
27	RES	RAZ/IGN	Reserved.
26:25	DEV	RO	Device select timing—Indicates the 21030 has a medium response time to PCI device select. The code in this field is 01.
24	RES	RAZ/IGN	Reserved.
23	BBC	RO	Back-to-back capable—Indicates the 21030 can handle fast back-to-back PCI transactions as a target. The value of this bit is 1.
22:10	RES	RAZ/IGN	Reserved.
9	BBE	RO	Back-to-back enable—Enables the 21030, as a master, to perform fast back-to-back PCI transactions. The value of this bit is 0.
8	RES	RAZ/IGN	Reserved.
7	BS	RO	Bus stepping—Indicates the 21030 drives the **ad<31:0>** PCI signals over two PCI clock cycles. The value of this field is 1.

(continued on next page)

Table 4–2 (Cont.) PCSR Field Description

Bits	Field	Type	Description
6	RES	RAZ/IGN	Reserved.
5	PS	RW	VGA palette snoop—When set, the 21030 snoops writes to VGA color register space. When clear, the 21030 responds normally to writes to VGA color register space. This bit is enabled by PVRR <31>.
4:3	RES	RAZ/IGN	Reserved.
2	ME	RW	Master enable—When set, enables the 21030 to become bus master; when clear, the 21030 cannot become bus master.
1	MS	RW	Memory space enable—When set, enables response to memory space accesses; when clear, response to memory space accesses is disabled.
0	IO	RO	I/O space enable—When clear, I/O space is disabled. The value of this bit is 0, and the 21030 does not respond to I/O space accesses.

The PCSR specifies the 21030 configuration as a PCI device. It also indicates whether the 21030 detected a target abort or issued a master abort.

The master abort and target abort bits (<29:28>) are set when the 21030 issues or detects the respective transaction terminations. Each of these bits remains set until software explicitly clears it by writing a one to the bit (writing a zero is ignored).

The back-to-back capable bit (<23>) enables the 21030, as a target, to respond to fast back-to-back PCI transactions. The back-to-back enable bit (<9>), enables the 21030, as a master, to perform fast back-to-back cycles.

The VGA palette snoop bit (<5>) determines how the 21030 responds to VGA color register (palette) writes. When the bit is set, the 21030 *snoops;* that is, it transparently accepts write data but does not explicitly respond to write transactions. When <5> is clear, the 21030 responds normally to VGA color register writes; that is, it responds as it does to any other write to its address space. VGA palette snoop is active only when VGA enable is set in the VGA redirect register (PVRR <31>, Section 4.2.8). If VGA enable is not set, the 21030 does not respond to VGA palette writes.

The master enable bit (<2>) must be set in order to invoke any 21030 DMA graphics operation. (The DMA modes are described in Sections 6.2.10 through 6.2.10.2.)

The 21030 address space can be mapped only into PCI memory space. The 21030 responds to PCI memory accesses within its address space when the memory space enable bit (<1>) is set. (Address mapping is described in Chapter 2.)

At power-up and reset, the value of the PCSR is $028000A0_{16}$ — the VGA palette snoop bit is set and bits <26:25,23,9,7,0> return their respective hard-wired values.

4.2.2 PCI Device Base Address Register

Note

This section describes 21030 step B functionality. See Section A.2.1 for a description of the equivalent 21030 step A functionality.

Figure 4–2 shows the PCI device base address register (PDBR) format, and Table 4–3 describes its fields.

Figure 4–2 PDBR Format

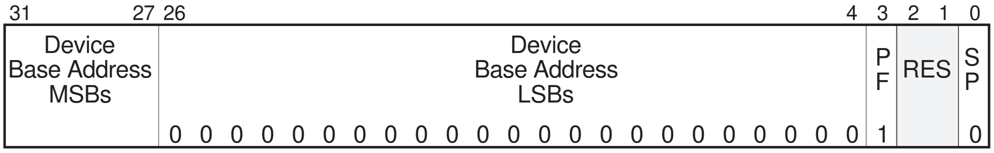


Table 4–3 PDBR Field Description

Bits	Field	Type	Description
31:27	Device Base Address MSBs	RW	The most significant bits of the 21030 address-space base address.
26:4	Device Base Address LSBs	RO	The value of this field is 000000_{16}. It indicates that the base address must be aligned to 128MB or greater.
3	PF	RO	Prefetchable—Indicates that prefetched reads and merged writes to the 21030 address space are allowed. The value of this bit is 1.
2:1	RES	RAZ/IGN	Reserved.
0	SP	RO	Space—Specifies that the 21030 address space must be mapped into PCI memory space. The value of this bit is 0.

The device address space is mapped to the location specified in the PDBR.

The value of the space bit (<0>) is zero, indicating that the 21030 can be mapped only into memory space. The value of the 23 least significant address bits (<26:4>) is zero. This value indicates to configuration firmware that the 21030 and its associated memory requires 128MB of address space. Therefore, configuration firmware can map the 21030 address space into any naturally aligned, contiguous 128MB (or larger) region.

The prefetchable bit (<3>) indicates that there are no side effects on reads to the 21030 address space. The 21030 returns all bytes on reads regardless of the byte enables, and host bridges can merge writes into this region without causing errors.

The value of the PDBR is 00000008_{16} at reset.

4.2.3 PCI Identification Register

Figure 4–3 shows the PCI identification register (PIDR) format, and Table 4–4 describes its fields.

Figure 4–3 PIDR Format

31 … 16	15 … 0
Vendor ID	Device ID
0 0 0 1 | 0 0 0 0 | 0 0 0 1 | 0 0 0 1	0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 1 0 0

Table 4–4 PIDR Field Description

Bits	Field	Type	Description
31:16	Vendor ID	RO	Identifies Digital as the device vendor. The value of this field is 1011_{16}.
15:0	Device ID	RO	Identifies the 21030 as the device. The value of this field is 0004_{16}.

The read-only PIDR identifies the vendor and device to system software. Writes to this register are ignored.

The value of the PIDR is 10110004_{16} at reset.

4.2.4 PCI Class and Revision Register

Figure 4–4 shows the PCI class and revision register (PCRR) format, and Table 4–5 describes its fields.

Figure 4–4 PCRR Format

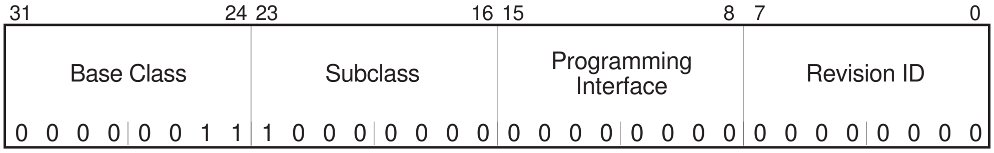


Table 4–5 PCRR Field Description

Bits	Field	Type	Description
31:24	Base Class	RO	Indicates that the 21030 is a display controller. The value of this field is 03_{16}.
23:16	Subclass	RO	Indicates that the 21030 is not VGA or XGA compatible; that is, its display controller subclass is ‘‘other.’’ The value of this field is 80_{16}.
15:8	Programming Interface	RO	Indicates that the 21030 does not support a particular standard programming interface. The value of this field is 00_{16}.
7:0	Revision ID	RO	The 21030 revision number. The value of this field is revision specific, as follows: 01_{16} for 21030 step A 02_{16} for 21030 step B

The PCRR identifies the 21030 revision number, device base class and subclass, and any compatible register-level programming interfaces.

The PCI power-on self-test (POST) code reads the device class information to determine whether the 21030 is suitable as a boot display device. The programming interface code (<15:8>) indicates that the 21030 provides no special support for any register-level programming standard.

The value of the PCRR at reset is 03800001_{16} for the 21030 step A and 03800002_{16} for the 21030 step B.

4.2.5 PCI Latency Timer Register

Figure 4–5 shows the PCI latency timer register (PLTR) format, and Table 4–6 describes its fields.

Figure 4–5 PLTR Format

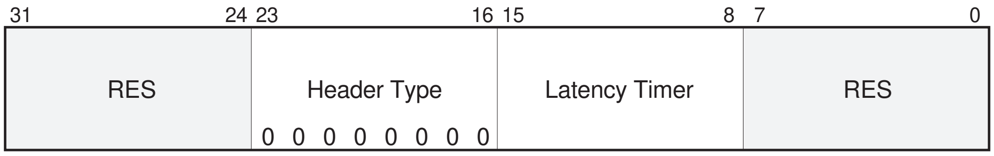


Table 4–6 PLTR Field Description

Bits	Field	Type	Description
31:24	RES	RAZ/IGN	Reserved.
23:16	Header Type	RO	Indicates that the 21030 configuration space header block conforms to standard PCI configuration space. The value of this field is 00_{16}.
15:8	Latency Timer	RW	The length of time that the 21030 owns the PCI bus is limited to the number of PCI clocks specified in this field.
7:0	RES	RAZ/IGN	Reserved.

The PLTR specifies the length of time that the 21030 retains bus ownership in the presence of other bus requests. It also indicates whether the configuration header space is standard.

The value of the PCRR is 00000000_{16} at reset.

4.2.6 PCI Expansion ROM Base Address Register

Figure 4–6 shows the PCI expansion ROM base address register (PRBR) format, and Table 4–7 describes its fields.

Figure 4–6 PRBR Format

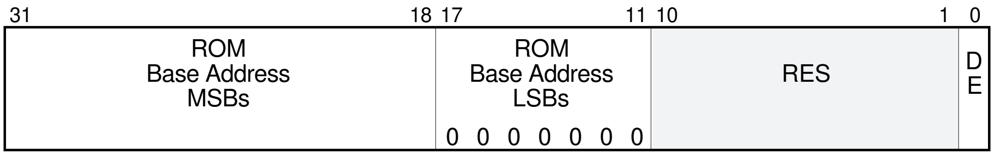


Table 4–7 PRBR Field Description

Bits	Field	Type	Description
31:18	ROM Base Address MSBs	RW	The most significant bits of the 21030 expansion-ROM base address.
17:11	ROM Base Address LSBs	RO	The value of this field is 00_{16}. It indicates that the base address must be aligned to 256KB or greater.
10:1	RES	RAZ/IGN	Reserved.
0	DE	RW	Decode enable—When this bit and PCSR <1> (Section 4.2.1) are set, the 21030 responds to ROM space accesses. When this bit is clear, ROM access decoding is disabled.

The 21030 expansion ROM (EEPROM) is mapped to the memory space location specified by the PRBR. The 21030 supports PCI-compliant EEPROM sizes up to 256KB. The EEPROM must be mapped on naturally aligned 256KB boundaries. The 21030 responds to all accesses in the 256KB ROM space if the decode enable bit (<0> in this register) and the memory space enable bit (PCSR <1>, Section 4.2.1) are both set.

The value of the PRBR is 00000000_{16} at reset.

4.2.7 PCI Line Interrupt Register

Figure 4–7 shows the PCI line interrupt register (PLIR) format, and Table 4–8 describes its fields.

Figure 4–7 PLIR Format

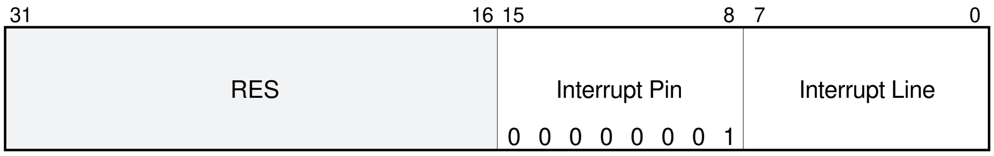


Table 4–8 PLIR Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.
15:8	Interrupt Pin	RO	Indicates that the 21030 signals interrupts on the **inta_l** pin. The value of this field is 01_{16}.
7:0	Interrupt Line	RW	The 21030 **inta_l** pin is tied to the system interrupt controller input identified by this field. POST firmware initializes this field.

The PLIR provides hardware support for POST firmware interrupt configuration and identification.

The value of the PLIR is 00000100_{16} at reset.

4.2.8 PCI VGA Redirect Register

Figure 4–8 shows the PCI VGA redirect register (PVRR) format, and Table 4–9 describes its fields.

Figure 4–8 PVRR Format

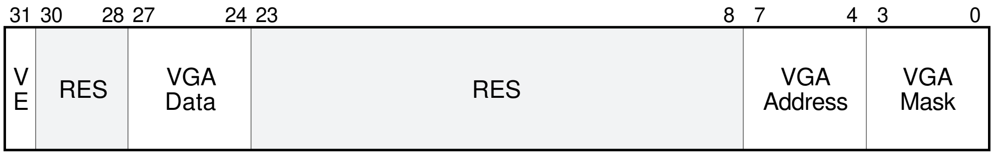


Table 4–9 PVRR Field Description

Bits	Field	Type	Description
31	VE	RW	VGA enable—When set, enables VGA color register snooping. When clear, disables VGA color register snooping. This bit enables PCSR <5> and is set at reset.
30:28	RES	RAZ/IGN	Reserved.
27:24	VGA Data	RW	The redirected address for the VGA pixel data register (3C9). Initialized to 1_{16} at reset.
23:8	RES	RAZ/IGN	Reserved.
7:4	VGA Address	RW	The redirected address for the VGA pixel address register (3C8). Initialized to 0_{16} at reset.
3:0	VGA Mask	RW	The redirected address for the VGA pixel mask register (3C6). Initialized to 2_{16} at reset.

The PVRR enables the 21030 to respond to PCI I/O writes to the VGA color registers. It also controls the destination of snooped VGA color register (palette) data transfers.

The VGA enable bit (<31>) determines whether the 21030 responds to VGA palette writes. If the bit is set, then VGA palette snoop (PCSR <5>, Section 4.2.1) determines whether the 21030 will snoop the write or respond normally. The VGA enable bit is set at power-up and reset to place the 21030 in VGA pass-through mode without explicit initialization.

The 21030 can respond to three VGA color register addresses: pixel data register, pixel address register, and pixel mask register. For each of these register addresses, the PVRR contains a redirected destination address field: VGA data (<27:24>), VGA address (<7:4>), and VGA mask (<3:0>). When the 21030 detects a write to one of the snooped locations, it redirects the write to the location pointed to by the corresponding field.

At reset, the VGA data, VGA address, and VGA mask fields are initialized to be compatible with the Bt485 RAMDAC and similar DAC interfaces that support VGA multiplexing.

The 21030 supports other types of devices in pass-through mode only if initialization code can be run at boot time, before a display is required. If that is the case, the initialization code can rewrite the PVRR to redirect the accesses. The device will be initialized at boot time to redirect each VGA palette write to the appropriate register in the 21030 graphics subsystem RAMDAC.

The value of the PVRR is 81000002_{16} at reset.

4.3 Graphics Command Registers

The 21030 accelerated graphics operations are selected by specifying a mode in the mode register (GMOR, Section 4.4.1) and initiated by a write to either of the following:

- The frame buffer address space (standard)

  The chip is set to a specific mode and the frame buffer is written directly. The address and data are interpreted according to the mode.

- Any graphics command register (alternate)

  The graphics software initiates a drawing operation by writing to a graphics command register.

The graphics command registers are the only 21030 registers that initiate a drawing action when written. They provide a faster and simpler mechanism to draw, extend, and link lines and spans, copy large spans, and allow software to indirectly address the frame buffer. The graphics command registers are also the only mechanism for invoking the 3D line and span drawing operations.

4.3.1 Slope Registers

Figure 4–9 shows the slope registers (GSLR<7:0>) format, and Table 4–10 describes the fields.

Figure 4–9 GSLR<7:0> Format

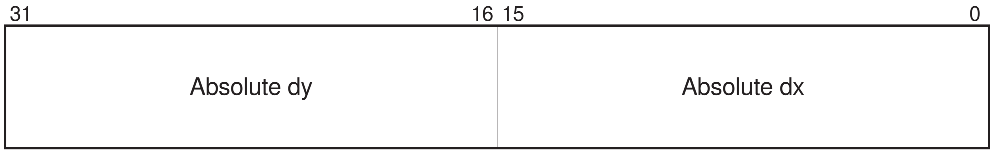


Table 4–10 GSLR<7:0> Field Description

Bits	Field	Type	Description
31:6	Absolute dy	RW	An unsigned integer equal to the absolute value of the difference in *y* of the two line endpoints.
15:0	Absolute dx	RW	An unsigned integer equal to the absolute value of the difference in *x* of the two line endpoints.

The GSLRs initialize the internal Bresenham engine for line drawing. On a write to a GSLR, the following Bresenham terms are automatically calculated as a function of absolute *dx* and absolute *dy*.

- Initial error

  The 16-bit signed initial value stored in the Bresenham engine error accumulator.

- Length

  A 4-bit value specifying the number of pixels to be drawn in this line segment.

- Error Increment 1

  The positive value added to the error term when the Bresenham error term is < 0 (a major axis step).

- Address Increment 1

  The signed value added to the current address when the Bresenham error term is < 0 (a major axis step).

- Error Increment 2

  The positive value subtracted from the error term when the Bresenham error term is ≥ 0 (a step along the major and minor axes).

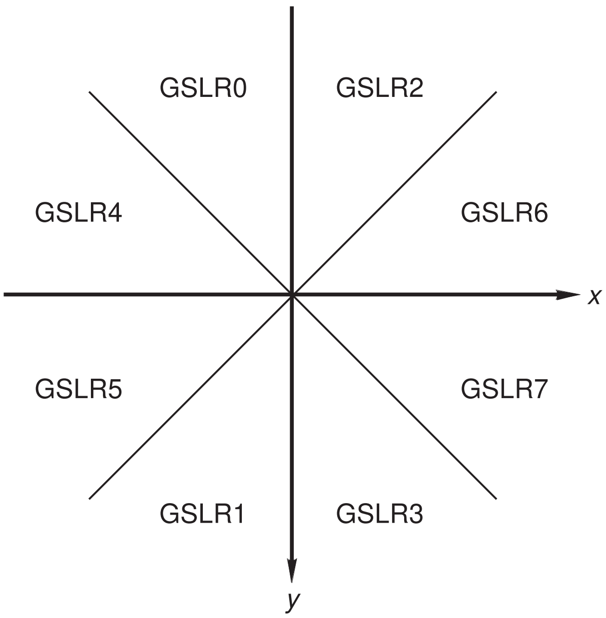
- Address Increment 2

  The signed value added to the current address when the Bresenham error term is ≥ 0 (a step along the major and minor axes).

Each GSLR is associated with one of the drawing octants, and each specifies a slope in terms of the absolute values of the rise in *y* (absolute *dy*) and the run in *x* (absolute *dx*). Results are undefined if both absolute *dy* and absolute *dx* are zero. Software must filter out zero-length lines. (Section 6.2.12.1 includes the algorithm for calculating the Bresenham terms.)

Figure 4–10 shows the slope register associated with each of the drawing octants.

Figure 4–10 Slope Registers and Drawing Octants



On a write to a GSLR, the pixel length of the line segment is initialized to the major axis length MOD 16 (GB3R <3:0>, Section 4.4.12). This means that the 21030 is initialized to draw up to 16 pixels when the GSLR is written. For example, if the major axis length (the greater of absolute *dx* and absolute *dy*) is 19, the GSLR initializes the pixel length to 3. When used with the continue register (GCTR, Section 4.3.3), this feature allows software to draw lines of arbitrary length without monitoring the length of each segment. If the line to be drawn is not an exact multiple of 16 pixels, the shorter line (length MOD 16) is drawn first, and the line is completed with successive writes to the GCTR (which always draws 16 pixels).

Depending on the graphics environment (GMOR <13>, Section 4.4.1), writing a GSLR sets up the Bresenham terms correctly for all X-compliant lines and most lines that comply with Windows NT. The GSLRs create the correct initial terms for lines drawn under Windows NT *only* if the following criteria are met:

- The endpoint coordinates of the line are integers.
- The length of the line, as measured by the run of the line along the major axis, is limited to 64K -1 pixels.

In general, lines that have subpixel endpoints and clipped lines cannot be drawn with the GSLRs; the slope-no-go registers (GSNR<7:0>) and GCTR can be used to draw such lines.

(See Section 6.2.12.1 for more information about using the GSLRs to draw lines.)

Note

The Bresenham width register (GBWR, Section 4.4.13) must be written before writing a GSLR.

The GSLRs are cleared at reset.

4.3.2 Span Width Register

The function of the span width register (GSWR) depends on whether it is being written or read.

4.3.2.1 Write

On a write, the GSWR is an alias for slope register 7 (GSLR7), with the same format and field descriptions (Section 4.3.1). Typically, this register is used to draw shaded, Z-buffered, or dithered spans, and absolute *dy* is zero. (See Section 6.2.14 for more information about using the GSWR to draw 3D spans.)

4.3.2.2 Read

Figure 4–11 shows the GSWR read format, and Table 4–11 describes its fields.

Figure 4–11 GSWR Read Format

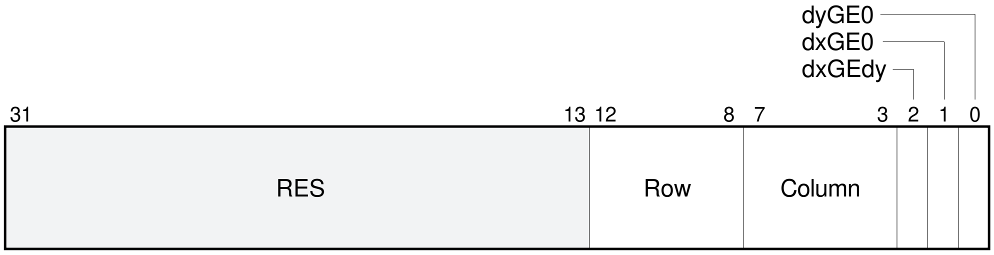


Table 4–11 GSWR Read-Format Field Description

Bits	Field	Type	Description
31:13	RES	RAZ/IGN	Reserved.
12:8	Row	RO	The current internal value of the dither row.
7:3	Column	RO	The current internal value of the dither column.
2	dxGEdy	RO	Set when the absolute value of the run parameter (*dx*) is greater than or equal to the absolute value of the rise parameter (*dy*); otherwise, clear (*dx* is less than *dy*).

(continued on next page)

Table 4–11 (Cont.) GSWR Read-Format Field Description

Bits	Field	Type	Description
1	dxGE0	RO	Set when the run (*dx*) of the slope is greater than or equal to 0 (*dx* is positive); otherwise, clear (*dx* is negative).
0	dyGE0	RO	Set when the rise (*dy*) of the slope is greater than or equal to 0 (*dy* is positive); otherwise, clear (*dy* is negative).

When the GSWR is read, the parameters indicate the state of the internal Bresenham engine and dithering hardware. The slope parameters are generated by the Bresenham engine on the most recent write to the GSWR, the slope registers, or the slope-no-go registers. The dither row (GRVR, Section 4.4.22) and dither column (GGVR, Section 4.4.24) values are initialized at the start of a dithered 3D line drawing operation and are then updated by hardware on a per-pixel basis. (See Section 6.2.12 for more information about the algorithm that generates the slope parameters.)

The GSWR is cleared at reset.

4.3.3 Continue Register

The function of the continue register (GCTR) depends on whether it is being written or read.

4.3.3.1 Write in Any Mode

Figure 4–12 shows the GCTR write format, and Table 4–12 describes its field.

Figure 4–12 GCTR Write Format

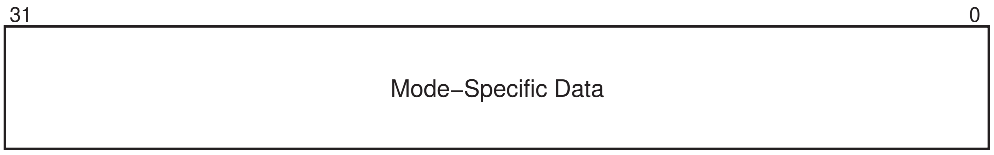


Table 4–12 GCTR Write-Format Field Description

Bits	Field	Type	Description
31:0	Mode-Specific Data	WO	The same format as the mode-specific PCI write data format described in Sections 6.2.1 through 6.2.14.

On a write, the two primary functions of the GCTR are to indirectly address the frame buffer and continue a line or span for an additional 16 pixels without recomputing and reloading parameters.

A PCI write to the 21030 frame buffer space usually initiates a drawing action. The address used for the operation is the frame buffer address of the write, and the PCI write data is interpreted according to the drawing mode. Alternatively, software can initiate mode-dependent operations by writing the GCTR, and indirectly specify the frame buffer address. Writes to the GCTR are interpreted exactly the same as writes to the frame buffer.

Indirect Frame Buffer Addressing

If the address register (GADR, Section 4.4.2) was written since the previous operation, the GCTR will take the frame buffer address from the GADR and initiate a graphics operation. The GCTR mode-specific data (<31:0>) has the mode-dependent format of the frame buffer PCI write-data (Sections 6.2.1 through 6.2.14). For example, when the GCTR is written in transparent-stipple mode, the mode-specific data includes the stipple mask; but when the GCTR is written in DMA-write copy mode, the mode-specific data includes a read count and the edge masks. Line mode mode-dependent data is different because it includes the low-order address bits; the GADR fully contains these bits, so they are not part of the GCTR mode-specific data (Section 4.3.3.2).

Line or Span Continuation

If the GADR is purposely not written before initiating a line mode operation, the GCTR can be written to effectively extend, or continue, the line drawn immediately prior to the current operation, using the address in the 21030 internal addressing hardware.

At the completion of a line or span drawing operation, the 21030 leaves its internal line-drawing hardware in a state that allows a subsequent line-mode operation to continue where the preceding line-mode operation stopped. That state includes at least the frame buffer address, and can also include variables such as the Bresenham error terms and color values, depending on the specific line mode (for example, color-interpolated lines or opaque lines). Therefore, the GCTR can quickly and easily extend the previous line-mode operation. For example, a write to a slope register will set up and draw 16 pixels along a line. After the initial write to the slope register, software can simply write the GCTR twice to extend the line or span to a length of 48 pixels.

4.3.3.2 Write in Line Mode

Figure 4–13 shows the GCTR line-mode write format, and Table 4–13 describes the fields.

Figure 4–13 GCTR Line-Mode Write Format

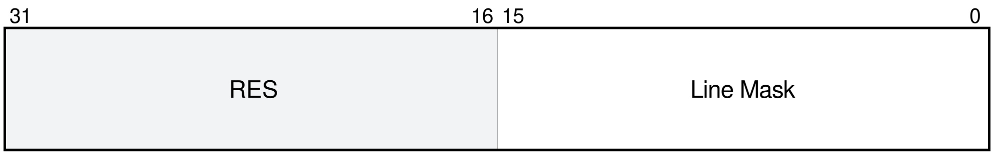


Table 4–13 GCTR Line-Mode Write-Format Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.

(continued on next page)

Table 4–13 (Cont.) GCTR Line-Mode Write-Format Field Description

Bits	Field	Type	Description
15:0	Line Mask	WO	The mask or stipple for the next 16-pixel line segment.

The format of GCTR mode-specific data in line mode matches the frame buffer PCI write-data format in opaque-line mode (Section 6.2.12), except that the address LSBs (AD<1:0>) are unnecessary because the address is fully contained in the GADR. (See Sections 6.2.1 through 6.2.14 for more information about using the GCTR to extend lines.)

4.3.3.3 Read

Figure 4–14 shows the GCTR read format, and Table 4–14 describes its fields.

Figure 4–14 GCTR Read Format

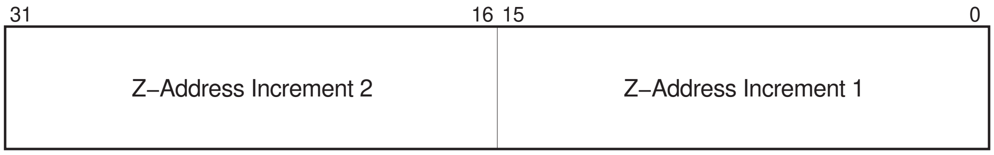


Table 4–14 GCTR Read-Format Field Description

Bits	Field	Type	Description
31:16	Z-Address Increment 2	RO	The signed value of the Z-address increment used when stepping along the minor and major axes in a Z-buffered line mode.
15:0	Z-Address Increment 1	RO	The signed value of the Z-address increment used when stepping along the major axis in a Z-buffered line mode.

On a read in any mode, the GCTR returns the value of the Z-address increment 1 and Z-address increment 2 stored in the Bresenham engine.

When a write to a slope, slope-no-go, or the span width register sets up a line drawing operation, the internal Bresenham engine calculates the address increments to be added to the current Z-buffer address as the engine takes a major axis step or a major and minor axes step. The Z-buffer width parameter in the Bresenham width register (GBWR, Section 4.4.13) is used in the calculation of one or both increments. (See Section 6.2.14 for more information about the generation and use of Z-address increment 1 and Z-address increment 2.)

4.3.3.4 Writes to Alternate ROM Space

The GCTR and GADR are mapped sequentially on writes to the alternate ROM space (Section 2.2.3). Basically, software can alternately write the GCTR and GADR by writing sequential locations in the otherwise read-only alternate ROM space. This method of sequential access can help make effective use of the write buffer in an Alpha AXP CPU. (See Sections 7.3 through 7.3.2 for more information about this method of access and alternate ROM space mapping.)

The GCTR is cleared at reset.

4.3.4 Copy-64 Source and Destination Registers

Figure 4–15 shows the copy-64 source register (GCSR) and copy-64 destination register (GCDR) formats, and Table 4–15 describes their fields.

Figure 4–15 GCSR and GCDR Formats

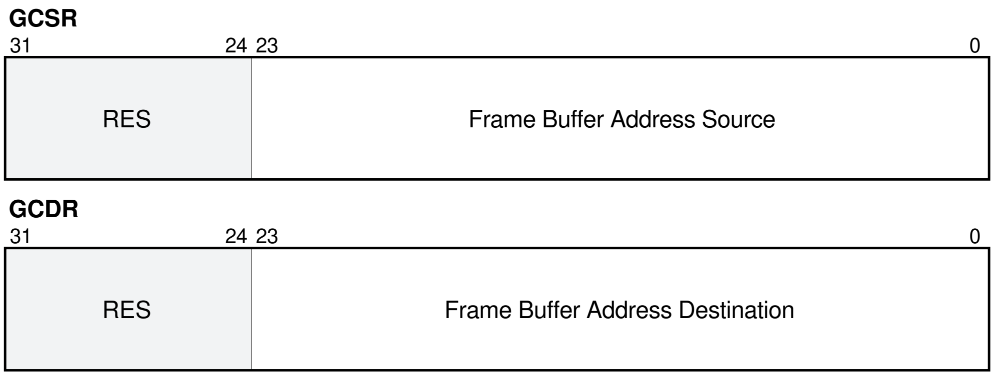


Table 4–15 GCSR and GCDR Field Description

Bits	Field	Type	Description
GCSR			
31:24	RES	RAZ/IGN	Reserved.
23:0	Frame Buffer Address Source	WO	Frame buffer address of the source—The 8-byte-aligned base address of the 64-byte span to be loaded into the 21030 copy buffer.
GCDR			
31:24	RES	RAZ/IGN	Reserved.
23:0	Frame Buffer Address Destination	WO	Frame buffer address of the destination—The 64-byte span will be copied from the 21030 copy buffer to the destination starting at this 8-byte-aligned address.

The GCSR and the GCDR are used together to perform fast, simple copies of aligned, unmasked, 64-byte spans. Both registers are write-only.

A write to the GCSR initiates a fill from the frame buffer to the on-chip 64-byte copy buffer, beginning at the frame buffer address source (GCSR <23:0>). A subsequent write to the GCDR unloads the contents of the copy buffer into the frame buffer, beginning at the frame buffer address destination (GCDR <23:0>).

The frame buffer source and destination addresses must be aligned to 8 bytes except when copying unpacked 8-bpp bitmaps, in which case they must be aligned to 32 bytes.

Writing the frame buffer address of the source span to the GCSR and then writing the frame buffer address of the destination span to the GCDR effectively copies a 64-byte span from an 8-byte-aligned source to an 8-byte-aligned destination.

Copying 8-bpp bitmaps with the GCSR and GCDR, which copies 64 pixels at a time, is faster than copying with writes to the frame buffer, which copies only 32 pixels at a time. However, the GCSR and GCDR can be used to copy only unmasked spans in which the source and destination are aligned to 8 bytes. Therefore, the GCSR and GCDR are used primarily to copy the interiors of large spans. Given an arbitrary source and destination, addresses are not likely to be aligned to 8 bytes. In such cases, the edges of the span must be copied with writes to the frame buffer in standard copy mode. The GCSR and GCDR can then be used to quickly fill the remaining 8-byte-aligned interior of the span.

Although the 21030 does not support masking when using the GCSR and GCDR, it does shift pixel data to support copies in which the source and destination are unaligned. Pixel data is shifted as specified in the pixel shift register (GPSR, Section 4.4.5).

The GCSR and GCDR are cleared at reset.

4.4 Graphics Control Registers

The graphics control registers control 21030 graphics processing and are the largest part of the register set. Reading and writing the graphics control registers does *not* initiate any drawing activity. The register parameters characterize the operations that are initiated by writing to the graphics command registers or frame buffer.

The graphics control registers need not be written for every drawing operation. The number of graphics control registers needed to perform a graphics operation depends on the mode the chip is in and whether drawing is initiated by a write to the frame buffer or to a graphics command register. Additionally, register fields that contain configuration-specific information, such as frame buffer depth or the type of RAM used, are written only at initialization time. (Chapter 6 describes the graphics operations and the registers that are required, optional, or ignored for each type of operation.)

All the graphics control registers can be read and written; however, as noted in the register descriptions, some registers do not read exactly as written.

4.4.1 Mode Register

The function of the mode register (GMOR) depends on whether it is being written or read.

4.4.1.1 Write

Figure 4–16 shows the GMOR write format, and Table 4–16 describes its fields.

Figure 4–16 GMOR Write Format

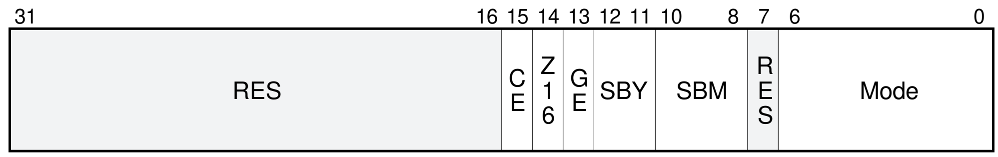


Table 4–16 GMOR Write-Format Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.
15	CE	RW	Cap ends—When set, last pixel write is enabled; when clear, last pixel write is disabled.
14	Z16	RW	When set, Z-values are 16 bits; when clear, Z values are 24 bits.
13	GE	RW	Graphics environment—When set, the 21030 is operating in a Win32 graphics environment; when clear, the 21030 is operating in an X graphics environment.
12:11	SBY	RW	Source byte—Selects the source byte, as follows: 0 0 byte 0 0 1 byte 1 1 0 byte 2 1 1 byte 3

(continued on next page)

Table 4–16 (Cont.) GMOR Write-Format Field Description

Bits	Field	Type	Description
10:8	SBM	RW	Source bitmap—Identifies the type of source bitmap, as follows: 000 8-bpp packed 001 8-bpp unpacked 010 12-bpp (low) 110 12-bpp (high) 011 24-bpp Unused codes are reserved
7	RES	RAZ/IGN	Reserved.
6:0	Mode	RW	Determines the 21030 graphics mode (Table 4–17).

Table 4–17 lists the 21030 graphics modes.

Table 4–17 Graphics Modes

Code*	Graphics Mode
0000000	Simple
0010000	Simple Z
0000001	Opaque stipple
0100001	Opaque fill
0000101	Transparent stipple
0100101	Transparent fill
0001101	Block stipple
0101101	Block fill
0000010	Opaque line
0000110	Transparent line
0001110	Color-interpolated, transparent, non-dithered line
0101110	Color-interpolated, transparent, dithered line
1001110	Sequential-interpolated, transparent line
0010010	Z-buffered, opaque line
0010110	Z-buffered, transparent line
0011010	Z-buffered, opaque, color-interpolated, non-dithered line
0111010	Z-buffered, opaque, color-interpolated, dithered line
1011010	Z-buffered, opaque, sequential-interpolated line
0011110	Z-buffered, transparent, color-interpolated, non-dithered line
0111110	Z-buffered, transparent, color-interpolated, dithered line

*Code in GMOR <6:0>. Unused codes are reserved.

(continued on next page)

Table 4–17 (Cont.) Graphics Modes

Code*	Graphics Mode
1011110	Z-buffered, transparent, sequential-interpolated line
0000111	Copy
0010111	DMA-read copy, non-dithered
0110111	DMA-read copy, dithered
0011111	DMA-write copy

*Code in GMOR <6:0>. Unused codes are reserved.

The mode register determines how the 21030 interprets the address and data on a write to the frame buffer, and how it interprets the data on a write to the graphics command registers. (See Sections 6.1 through 6.2.14.4 for more information about the graphics modes.)

The cap ends bit (<15>) determines whether the last pixel in a line is drawn. It affects only lines drawn by writing to the slope registers; it has no effect when writing to the frame buffer in a line mode. The host must adjust the line length when lines are drawn by writing to the frame buffer.

The Z16 bit (<14>) selects the Z-buffer depth and format when drawing in the Z-buffering modes. (See Section 6.1.5 for more information about the Z-buffer formats, and Section 6.2.14 for more information about Z-buffer manipulation in the 3D line modes.)

The graphics environment bit (<13>) specifies whether graphics processing must conform to Win32 or X specifications. (Currently, this bit controls how lines are drawn, because Win32 requires a style incompatible with existing X-server drawing code.)

A 32-bpp frame buffer can support a variety of 8-bpp, 12-bpp, and full 24-bpp bitmap formats. The source byte and source bitmap fields (<12:11,10:8>) specify the source bitmap for 32-bpp frame buffers. (Both fields must be zero on all 8-bpp frame buffer systems.) In several graphics modes, these two fields provide all the control necessary for the 21030 to extract source data from less than 32-bpp bitmaps. These fields complement the destination bitmap and destination byte fields in the raster operation register (GOPR, Section 4.4.3).

Only the source bitmap field selects a 12-bpp or 24-bpp source bitmap; the source byte field must be set to zero. Together, both fields specify one of the possible 8-bpp source bitmaps. (See Section 6.1.5 for more information about the bitmap formats and the source bitmap and byte fields.)

4.4.1.2 Read

Figure 4–17 shows the GMOR read format, and Table 4–18 describes its fields.

Figure 4–17 GMOR Read Format

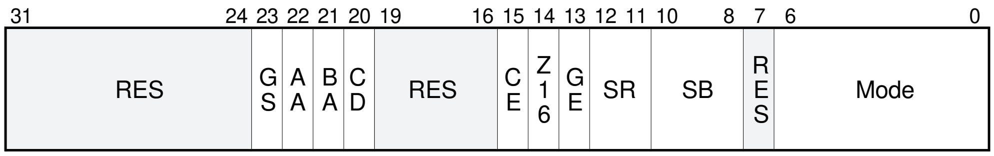


Table 4–18 GMOR Read-Format Field Description

Bits	Field	Type	Description
31:24	Same as write format. See Table 4–16.		
23	GS	RO	GPXR state—When set, indicates that the pixel mask register (GPXR) is in the persistent state; when clear, indicates that the GPXR is in the one-shot state.
22	AA	RO	Address age—When set, indicates that the GADR value is newer than the previous frame buffer operation; when clear, indicates that the GADR value is older than the previous frame buffer operation.
21	BA	RO	Bresenham age—When set, indicates that the GB3R value is newer than the previous frame buffer operation; when clear, indicates that the GB3R value is older than the previous frame buffer operation.
20	CD	RO	Copy direction—When set, indicates that the next copy mode operation will drain the copy buffer; when clear, indicates that the next copy mode operation will fill the copy buffer.
19:0	Same as write format. See Table 4–16.		

On a read, 4 GMOR read-only bits (<23:20>) return the current internal 21030 processing status. Software can read these bits to accurately restore 21030 context. The bits can be probed when saving state and then used to update the appropriate registers when restoring state.

The GPXR state bit (23) returns the current state of the GPXR. It indicates whether the GPXR retains its value from operation to operation (*persistent*) or if its value will be used only for the next operation (*one-shot*).

At the start of a line mode operation (initiated by a write to a slope register, the continue register, or the frame buffer), the 21030 conditionally uses the values stored in the Bresenham 3 register (GB3R, Section 4.4.12) and the address register (GADR, Section 4.4.2) for the operation. Only new GB3R and GADR values are used; that is, only values written after the previous line operation was initiated. The address age and Bresenham age bits (<22:21>) indicate whether the corresponding register values are new. The conditional use of these values makes it easier to draw long or linked lines. (See Section 6.2.12.3 for more information.)

In copy mode, the 21030 interprets alternate writes to the frame buffer as read-source and write-destination operations. Read-source operations fill and write-destination operations drain the copy buffer, starting at the specified address. The copy direction flag (<20>) indicates the current direction of the copy operation. (See Section 6.2.9 for more information about copy mode operations.)

The GMOR is cleared at reset.

4.4.2 Address Register

Figure 4–18 shows the address register (GADR) format, and Table 4–19 describes its field.

Figure 4–18 GADR Format

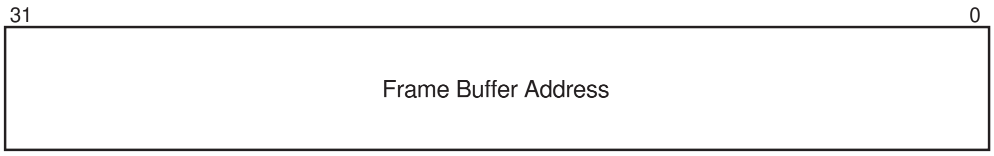


Table 4–19 GADR Field Description

Bits	Field	Type	Description
31:0	Frame Buffer Address	RW	The starting address for the next operation initiated by a write to a graphics command register.

The frame buffer address (<31:0>) is defined as the offset into the 21030 frame buffer space. A value of 00000000_{16} in this field corresponds to the first memory location in frame buffer space. The GADR can be used to access only frame buffer memory; it cannot be used to access the 21030 registers or external ROM address spaces, because they are separate spaces. (See Chapter 2 for more information about address mapping.)

In typical operations initiated by a write to the frame buffer, the write address is the starting address of the operation. On the other hand, when an operation is initiated by a write to certain graphics command registers, the starting address is in the GADR. It specifies the address of the first pixel for a drawing operation.

The GADR is used only for operations initiated by writes to the slope registers (GSLR<7:0>, Section 4.3.1), span width register (GSWR, Section 4.3.2), or continue register (GCTR, Section 4.3.3). For example, when writing GSLR0 to draw a line, the address in the GADR is the starting address of the drawable. Writing to the GADR does not initiate a drawing operation; it is used on the next write to a GSLR or the GCTR.

When the GADR is written in any of the line modes, the frame buffer address (<31:0>) is used only for the line operation immediately following. If the GADR was not written since the last line operation, the 21030 uses the final address of the most recent operation rather than the address in GADR <31:0>. This feature helps accelerate the drawing of long or linked lines (Section 6.2.14). A write to the GADR also sets the address age bit in the mode register (GMOR <22>, Section 4.4.1.2) to indicate that the address was written since the last operation.

The GADR can also be used with the GCTR to indirectly address the frame buffer. If a particular system cannot map all of the 21030 address space, all drawing can be done exclusively through the graphics command registers, with no direct writes to the frame buffer. Furthermore, because all drawing can be done by accessing registers, the frame buffer need not be mapped in PCI memory space. In any case, indirect addressing using the GADR with the GCTR is independent of whether the frame buffer is mapped in PCI memory space.

The GADR can also be written through the even Dword locations of the first 512KB of alternate ROM space (Section 2.2).

The GADR is cleared at reset.

4.4.3 Raster Operation Register

Figure 4–19 shows the raster operation register (GOPR) format, and Table 4–20 describes its fields.

Figure 4–19 GOPR Format

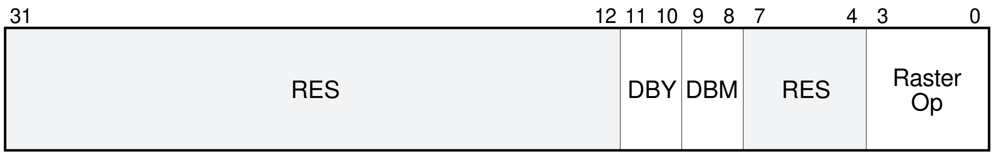


Table 4–20 GOPR Field Description

Bits	Field	Type	Description
31:12	RES	RAZ/IGN	Reserved.
11:10	DBY	RW	Destination byte—Selects the destination byte, as follows: 0 0 byte 0 0 1 byte 1 1 0 byte 2 1 1 byte 3
9:8	DBM	RW	Destination bitmap—Identifies the type of destination bitmap, as follows: 0 0 8-bpp packed 0 1 8-bpp unpacked 1 0 12-bpp 1 1 24-bpp
7:4	RES	RAZ/IGN	Reserved.
3:0	Raster Op	RW	Raster operation—Specifies how the source (src) pixel data and destination (dest) pixel data are logically combined on a write to the destination in most of the graphics modes (Table 4–21).

Table 4–21 lists the Boolean raster operations.

Table 4–21 Boolean Raster Operations

Code*	Operation	X	OpenGL	Win32
0000	dest ⇐ 0	GXclear	lo_zero	blackness
0001	dest ⇐ src AND dest	GXand	lo_and	srcand/mergecopy
0010	dest ⇐ src AND NOT dest	GXandReverse	lo_andr	srcerase
0011	dest ⇐ src	GXcopy	lo_src	srccopy/patcopy
0100	dest ⇐ (NOT src) AND dest	GXandInverted	lo_andi	(22_{16})
0101	dest ⇐ dest	GXnoop	lo_dst	(AA_{16})
0110	dest ⇐ src XOR dest	GXxor	lo_xor	srcinvert/patinvert
0111	dest ⇐ src OR dest	GXor	lo_or	srcpaint
1000	dest ⇐ (NOT src) AND NOT dest	GXnor	lo_nor	notsrcerase
1001	dest ⇐ (NOT src) XOR dest	GXequiv	lo_xnor	(99_{16})
1010	dest ⇐ NOT dest	GXinvert	lo_ndst	dstinvert
1011	dest ⇐ src OR (NOT dest)	GXorReverse	lo_orr	(DD_{16})
1100	dest ⇐ NOT src	GXcopyInverted	lo_nsrc	notsrccopy
1101	dest ⇐ (NOT src) OR dest	GXorInverted	lo_ori	mergepaint
1110	dest ⇐ (NOT src) OR NOT dest	GXnand	lo_nand	(77_{16})
1111	dest ⇐ 1	GXset	lo_one	whiteness

*GOPR raster operation field (<3:0>)

The 21030 uses the GOPR to support all of the Boolean operations specified under X and OpenGL, and a subset of 2-operand operations specified under Windows (Table 4–21). The source (src) can be used as the source or the pattern to implement the Windows 2-operand raster operations. To update the pixel value, most of these operations require the 21030 to perform read-modify-write cycles to display memory. (The 21030 does not directly support Windows 3-operand operations; for information about handling such operations, see Section 7.2.1.)

The raster operation field (<3:0>) defines the Boolean operation that is performed on the source and destination pixel data when writing to the destination bitmap in any graphics mode except block fill, block stipple, or DMA-write copy. In these modes, regardless of the code in the raster operation field, the Boolean operation performed is *dest* ⇐ *src* (0011_2).

In all cases except DMA-write copy, the destination operand is the pixel value stored in a destination bitmap residing in the frame buffer. The source operand can either be specified as a frame buffer bitmap or provided explicitly by software. (See Section 6.1.6 for more information about specifying source and destination operands.)

A 32-bpp frame buffer can support a variety of 8-bpp, 12-bpp, and 24-bpp bitmap formats. The destination byte and destination bitmap fields (<11:10,9:8>) specify the destination bitmap for 32-bpp frame buffers. (Both fields must be zero in all 8-bpp frame buffer systems.) In several graphics modes, these two fields provide all the control necessary for the 21030 to write pixel data to destination bitmaps other than 32-bpp bitmaps. These fields complement the source bitmap and source byte fields in the mode register (GMOR, Section 4.4.1).

Only the destination bitmap field selects a 12-bpp or 24-bpp destination bitmap; the destination byte field must be set to zero. Together, both fields specify one of the 8-bpp destination bitmaps. (See Section 6.1.5 for more information about the various bitmap formats and using the destination bitmap and destination byte fields.)

On a write to the GOPR, the 21030 flushes the write buffer before the write takes effect.

The GOPR is initialized to 00000003_{16} at reset.

4.4.4 Block-Color Registers

Note

The block-color registers are physically resident in the VRAMs rather than the 21030, and are undefined at reset. Because most standard DRAMs do not support block write cycles, block mode operations should not be performed to back buffers populated by such DRAMs.

Figure 4–20 shows the block color registers (GBCR<7:0>) format, and Table 4–22 describes the field.

Figure 4–20 GBCR<7:0> Format

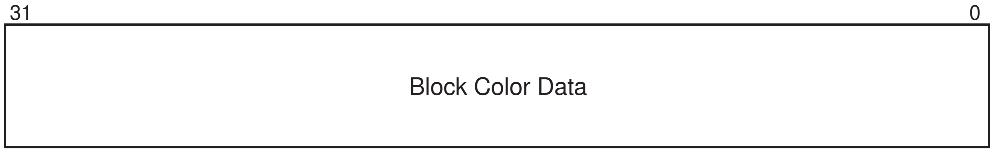


Table 4–22 GBCR<7:0> Field Description

Bits	Field	Type	Description
31:0	Block Color Data	RW	The pixel data written to the destination in block-mode operations. Figure 4–21 shows the pixel data formats.

The block-color registers define the block-color pattern. The pattern is 8 pixels wide and is written to display memory during block-stipple and block-fill operations. The block-color pattern is defined as follows:

block-color pattern = block color 7 ||block color 6 . . . block color 1||block color 0

In the equation, || denotes concatenate.

In either block mode, the 21030 uses VRAM block-write cycles to draw up to four copies of the block-color pattern in one memory write cycle. As a result, the 21030 draws as many as 32 pixels per memory CAS cycle, independent of the bitmap depth. In other words, whether drawing to an 8-bpp frame buffer, or to 8-bpp, 12-bpp, or 24-bpp bitmaps in a 32-bpp frame buffer, block-mode operations fill an 8-pixel block-color pattern four times.

A 32-bpp 8-pixel pattern is equivalent to 32 bytes of data; therefore, all eight GBCRs are used to specify the block-color pattern for 12-bpp and 24-bpp bitmaps in 32-bpp frame buffers (see Figure 4–21). However, 8-bpp bitmaps in either 8-bpp or 32-bpp frame buffers need only 8 bytes of data, and only GBCR1 and GBCR0 are used to specify the block color pattern.

The GBCRs support all destination bitmap formats except packed 8-bpp bitmaps in a 32-bpp frame buffer. (See Section 6.1.5 for more information about bitmap and buffer formats.)

Before the GBCRs are written when drawing to unpacked 8-bpp bitmaps in a 32-bpp frame buffer, the appropriate subset of GBCRs must be correctly initialized by setting the destination bitmap field in the GOPR (<9:8>, Section 4.4.3) according to the type of block-mode drawing to be done with the GBCRs.

The format of block color in a 32-bpp frame buffer depends on the format of the destination bitmap. As Figure 4–21 shows, in 24-bpp bitmaps, each block color contains one RGB triplet. For 12-bpp, the blue, green and red fields must be replicated across nibbles. For 8-bpp destination bitmaps, the color index must be replicated across the bytes of block color.

To draw to individual bits within each byte when drawing in a block mode (as in nonblock-mode writes), the plane mask registers (GPMR, Section 4.4.20) can be used with the GBCRs. (See Sections 6.2.5 and 6.2.6 for more information about block mode operations.)

The GBCRs and block-mode drawing are useful for fast, solid-color fills (for example, when clearing the screen) and for fast tiling, or brushing, a pattern across a fill region. (See Section 7.2.1 for more information about using block-mode operations to implement standard 2D graphics operations.)

Figure 4–21 shows the GBCR color pattern formats.

Figure 4–21 GBCR Color Pattern Formats

8–bpp Destination Bitmap
(8–bpp or 32–bpp Frame Buffer)

	31–24	23–16	15–8	7–0
GBCR1	Block Color 7	Block Color 6	Block Color 5	Block Color 4
GBCR0	Block Color 3	Block Color 2	Block Color 1	Block Color 0

12– or 24–bpp Destination Bitmap
(32–bpp Frame Buffer)

	31–0
GBCR7	Block Color 7
GBCR5	Block Color 6
GBCR3	Block Color 5
GBCR1	Block Color 4
GBCR6	Block Color 3
GBCR4	Block Color 2
GBCR2	Block Color 1
GBCR0	Block Color 0

Where GBCR*n* in a 32–bpp frame buffer is a function of the destination bitmap format:

32–bpp Frame Buffer

	31–24	23–16		15–8		7–0	
12–bpp Bitmap	tag	red	red	green	green	blue	blue
24–bpp Bitmap	tag	red		green		blue	

The GBCRs are undefined at reset.

4.4.5 Pixel-Shift Register

Figure 4–22 shows the pixel-shift register (GPSR) format, and Table 4–23 describes its fields.

Figure 4–22 GPSR Format

Table 4–23 GPSR Field Description

Bits	Field	Type	Description
31:4	RES	RAZ/IGN	Reserved.
3:0	Pixel Shift	RW	A signed value indicating the number of bytes to shift source data on a write into the copy buffer.

The GPSR specifies the number of bytes to shift source data in the copy, DMA-read copy, and DMA-write copy modes. The GPSR is ignored in all other modes. In the copy and DMA-read copy modes, the shift takes place before data is loaded into the copy buffer. In the DMA-write copy mode, the shift takes place before the data is driven onto the PCI bus. The range of the signed pixel-shift value is as follows:

Mode	Range
Copy	$-8 \leq$ pixel-shift value $\leq +7$
DMA-read copy	$0 \leq$ pixel-shift value $\leq +3$
DMA-write copy	$0 \leq$ pixel-shift value $\leq +7$

This allows arbitrary source and destination byte-alignment and copy direction when copying spans.

Writing the GPSR also resets the copy direction flag (GMOR <20>, Section 4.4.1) to select read-source on the next frame buffer write in copy mode. In copy mode, the flag determines whether the current frame buffer write should read the source into the copy buffer or write the copy buffer into the destination. The flag switches between the read-source and write-destination states on every frame buffer write in copy mode.

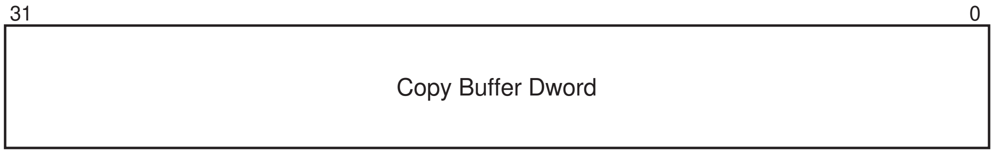
(See sections 6.2.9 through 6.2.9.8 for more information about using the GPSR and the copy direction flag.)

The GPSR is cleared at reset.

4.4.6 Copy-Buffer Registers

Figure 4–23 shows the copy-buffer registers (GCBR<7:0>) format.

Figure 4–23 GCBR<7:0> Format



The GCBRs provide read and write access into the internal, 64-byte copy buffer. A read or write to each GCBR returns one Dword from or stores one Dword into the copy buffer.

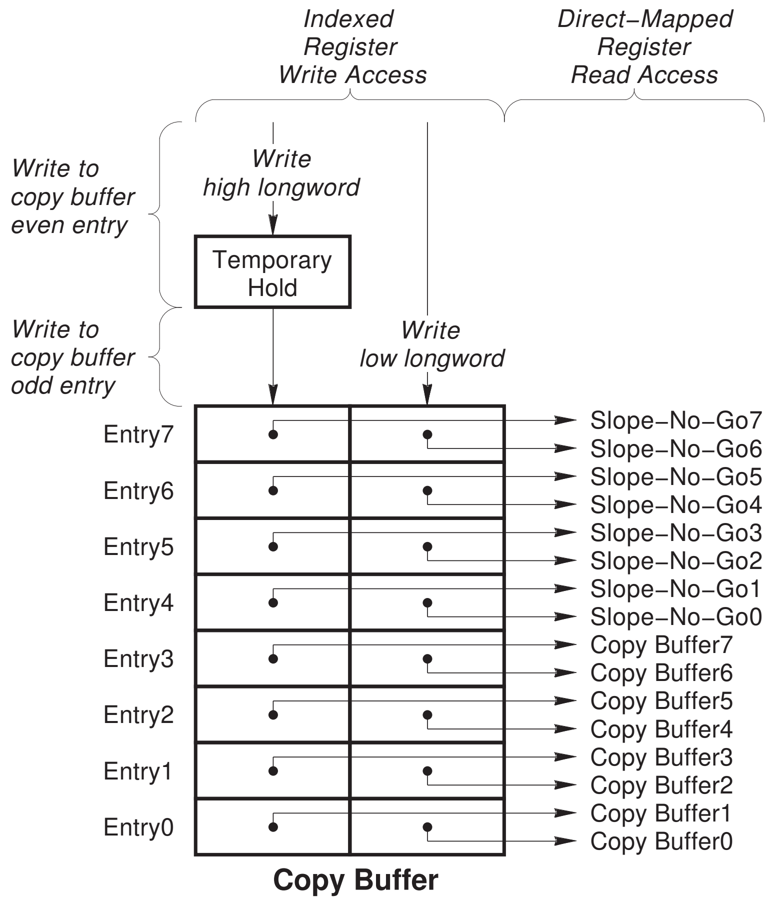
The copy buffer comprises 8 quadword (64-bit) entries (entry<7:0>). When reading a source bitmap in copy mode, the 21030 fills the copy buffer 1 quadword at a time, from entry0 to entry7. When writing a destination bitmap in copy mode, the 21030 unloads the copy buffer 1 quadword at a time (with mask) in the same sequence.

Software can also write the copy buffer in the same order (entry0 to entry7), by alternately writing even-numbered and odd-numbered GCBRs. A write to an even-numbered GCBR specifies, but does not load, the low Dword of the next empty copy buffer entry. A subsequent write to an odd-numbered GCBR loads that Dword into the high Dword of the next entry and loads the previously specified Dword into the low Dword of that entry. The results of a write to a full copy buffer are undefined.

On read, software directly and randomly accesses individual Dwords of each quadword entry (Figure 4–24). The 8 GCBRs are directly mapped to copy-buffer entries<3:0> and the slope-no-go registers (GSNR<7:0>, Section 4.4.9) are directly mapped to copy-buffer entries<7:4>. The GSNRs are mapped to the copy buffer only in read mode.

Figure 4–24 shows how the GCBRs and GSNRs are mapped to the copy buffer entries.

Figure 4–24 Copy Buffer Layout



The GCBRs are cleared at reset.

4.4.7 DMA Base-Address Register

Figure 4–25 shows the DMA base-address register (GDBR) format, and Table 4–24 describes its field.

Figure 4–25 GDBR Format

Table 4–24 GDBR Field Description

Bits	Field	Type	Description
31:0	DMA Address	RW	Dword-aligned PCI address (<1:0> = 00), pointing to base address of a drawable bitmap.

The GDBR specifies the 32-bit PCI memory address of the bitmap used as the source or destination in DMA-read copy or DMA-write copy mode operations. On a write to the frame buffer in one of these modes, the 21030 begins to read or write pixels at the DMA address (<31:0>).

To the 21030, the DMA address is a physical address. The 21030 has no indication of how the CPU maps system addresses into physical PCI memory addresses, how virtual addresses are translated to physical addresses, or how some systems support scatter/gather mapping from the PCI into main memory. Software must translate these levels of address indirection before writing the GDBR.

Note

Writes to the GDBR alter the contents of the Z-base-address register (GZBR, Section 4.4.15), and writes to the GZBR alter the contents of the GDBR.

The GDBR is cleared at reset.

4.4.8 Data Register

The data register (GDAR) specifies a mask for some line-mode operations and all block-fill and DMA-write copy operations. The GDAR format depends on the enabled mode.

4.4.8.1 Line Mode

Figure 4–26 shows the GDAR line-mode format, and Table 4–25 describes its fields.

Figure 4–26 GDAR Line-Mode Format

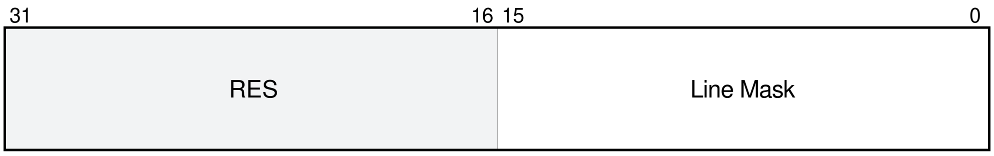


Table 4–25 GDAR Line-Mode Format Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.
15:0	Line Mask	RW	The mask for a 16-pixel line. In transparent-line mode, the foreground color is written to any pixel in the line that corresponds to a set bit in this field. No color is written to any pixel in the line that corresponds to a clear bit in this field. In opaque-line mode, the foreground color is written to any pixel in the line that corresponds to a set bit in this field and the background color is written to any pixel in the line that corresponds to a clear bit in this field.

In any line-mode operation initiated by a write to a slope register (GSLR <7:0>), the write data is the slope data, and the GDAR specifies the mask for the 16 pixels of that line segment.

When drawing line segments by writing to either the frame buffer in a line mode or the continue register (GCTR), the GDAR is not used because the write data specifies the line mask.

The line mask passed in the GDAR is expanded, on a per-pixel basis, into foreground colors in transparent-line mode, and into background or foreground colors (as specified in the foreground and background registers) in opaque-line mode. In color-interpolated line modes, the interpolated color is written to pixels that correspond to set bits in the line mask, and no color is written to pixels that correspond to clear bits.

The GDAR must be written before the slope register, because the write to the GSLR starts the drawing operation.

4.4.8.2 Block-Fill, Opaque-Fill, and Transparent-Fill Modes

Figure 4–27 shows the GDAR fill-mode format, and Table 4–26 describes its field.

Figure 4–27 GDAR Fill-Mode Format

Table 4–26 GDAR Fill-Mode Format Field Description

Bits	Field	Type	Description
31:0	Fill Mask	RW	The mask for each aligned 32-pixel span.

In any of the fill modes, the GDAR defines a repeating mask, aligned to 4 pixels. The 32-bit fill mask enables writes, on a per-pixel basis, for each aligned, 32-pixel span drawn in a block-fill operation. Only one mask is specified, regardless of the span length (that is, regardless of the pixel count passed in block-fill mode). The mask is repeated, or *tiled*, across the span at 32-pixel intervals. (See Chapter 6 for more information about the fill modes.)

In transparent-fill mode, the foreground color is written to each pixel of the span that corresponds to a set bit in the fill mask; no color is written to pixels that correspond to clear mask bits.

In opaque-fill mode, the foreground color is written to each pixel of the span that corresponds to a set bit in the fill mask; the background color is written to pixels that correspond to clear mask bits.

In block-fill mode, the block color is written to each pixel of the span that corresponds to a set bit in the fill mask; no color is written to pixels that correspond to clear mask bits.

In the block-, opaque-, and transparent-fill modes, the GDAR must be written before the frame buffer, because the write to the frame buffer starts the fill operation.

4.4.8.3 DMA-Write Copy Mode

Figure 4–28 shows the GDAR DMA-write copy mode format, and Table 4–27 describes its field.

Figure 4–28 GDAR DMA-Write Copy Mode Format

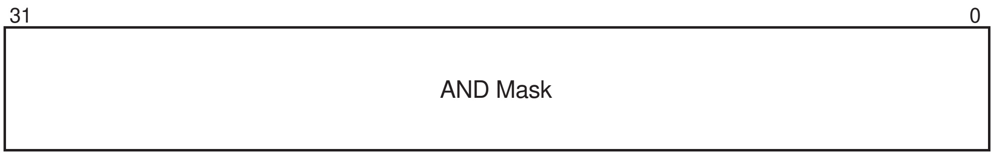


Table 4–27 GDAR DMA-Write Copy Mode Format Field Description

Bits	Field	Type	Description
31:0	AND Mask	RW	The bit mask to be ANDed with each Dword written across the PCI bus.

In DMA-write copy mode, the GDAR data is logically ANDed with write data from the PCI bus. Before the GDAR can be written in this mode, the chip must be idle; that is, the busy bit in the command and status register (SCSR <0>, Section 4.7.1) must be clear. This software interlock can be enforced by writing the SCSR immediately before the GDAR, because a write to the SCSR is not completed until the chip is idle.

The GDAR is set to FFFFFFFF at reset.

4.4.9 Slope-No-Go Registers

The function of the slope-no-go registers (GSNR<7:0>) depends on whether the registers are being written or read.

4.4.9.1 Write

Figure 4–29 shows the GSNR write format, and Table 4–28 describes the fields (identical to the slope registers).

Figure 4–29 GSNR<7:0> Write Format

31 … 16	15 … 0
Absolute dy	Absolute dx

Table 4–28 GSNR<7:0> Write-Format Field Description

Bits	Field	Type	Description
31:6	Absolute dy	RW	Unsigned integer equal to the absolute value of the difference in *y* of the two line endpoints.
15:0	Absolute dx	RW	Unsigned integer equal to the absolute value of the difference in *x* of the two line endpoints.

On a write, the GSNRs mimic the behavior of the slope registers (GSLR<7:0>, Section 4.3.1), but they do not initiate drawing. That is, they initialize the internal Bresenham engine for line drawing, but do not start the Bresenham pixel stepping or draw any pixels.

The GSNRs are primarily used to simplify the drawing of clipped lines and, potentially, to assist in drawing lines with subpixel endpoints. (See Section 7.2.3.2 for more information about drawing clipped lines.)

Note

The Bresenham width register (GBWR, Section 4.4.13) must be written before writing a GSNR.

4.4.9.2 Read

Figure 4–30 shows the GSNR read format, and Table 4–29 describes the contents.

Figure 4–30 GSNR<7:0> Read Format

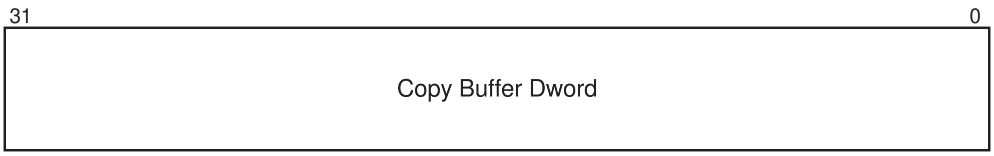


Table 4–29 GSNR<7:0> Read Format Contents

Register	Contents
GSNR7	Copy buffer entry 7 <63:32>
GSNR6	Copy buffer entry 7 <31:0>
GSNR5	Copy buffer entry 6 <63:32>
GSNR4	Copy buffer entry 6 <31:0>
GSNR3	Copy buffer entry 5 <63:32>
GSNR2	Copy buffer entry 5 <31:0>
GSNR1	Copy buffer entry 4 <63:32>
GSNR0	Copy buffer entry 4 <31:0>

On a read, each GSNR returns one Dword of the copy buffer entries <7:4> (Figure 4–24).

(See Sections 4.4.6 and 6.2.9 for more information about programmed I/O access to the copy buffer.)

The GSNRs are cleared at reset.

4.4.10 Bresenham 1 Register

Note

The Bresenham 1 and 2 register (GB1R and GB2R) descriptions are included for continuity. They are explicitly written only when line drawing is initiated using the standard frame buffer write mechanism. However, the alternate slope register write mechanism (Section 4.3.1) more efficiently provides the equivalent functionality. There is no practical reason for explicitly using the GB1R and GB2R. Furthermore, the GB3R is explicitly written only in unusual cases (Section 7.2.3.2). See Sections 6.2.12 and 6.2.13 for more information.

Figure 4–31 shows the Bresenham 1 register (GB1R) format, and Table 4–30 describes its fields.

Figure 4–31 GB1R Format

31 ... 16	15 ... 0
Address Increment 1	Error Increment 1

Table 4–30 GB1R Field Description

Bits	Field	Type	Description
31:16	Address Increment 1	RW	The signed value added to the current address when the Bresenham error term is < 0 (a major axis step).
15:0	Error Increment 1	RW	The positive value added to the error term when the Bresenham error term is < 0 (a major axis step).

The GB1R specifies the address and error increments used by the internal Bresenham line-drawing engine when the cumulative error is negative.

When the cumulative error is negative:

- Error increment 1 (<15:0>) is added to the cumulative error.
- Address increment 1 (<31:16>) is added to the current internal address, to point to the next pixel address to be written along the line.

This is effectively one step along the major axis of the line.

The GB1R is cleared at reset.

4.4.11 Bresenham 2 Register

See the note at the beginning of the GB1R description, Section 4.4.10.

Figure 4–32 shows the GB2R format, and Table 4–31 describes its fields.

Figure 4–32 GB2R Format

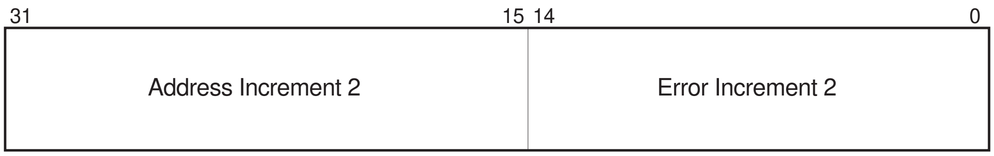


Table 4–31 GB2R Field Description

Bits	Field	Type	Description
31:16	Address Increment 2	RW	The signed value added to the current address when the Bresenham error term is ≥ 0 (a step along the major and minor axes).
15:0	Error Increment 2	RW	The positive value subtracted from the error term when the Bresenham error term is ≥ 0 (a step along the major and minor axes).

The GB2R specifies the address and error increments used by the internal Bresenham engine when the cumulative Bresenham error value is greater than or equal to zero:

- Error increment 2 (<15:0>) is subtracted from the error term.
- Address increment 2 (<31:16>) is added to the current address.

This is effectively one step along the major and minor axes of the line.

The GB2R is cleared at reset.

4.4.12 Bresenham 3 Register

Figure 4–33 shows the Bresenham 3 register (GB3R) format, and Table 4–32 describes its fields.

Figure 4–33 GB3R Format

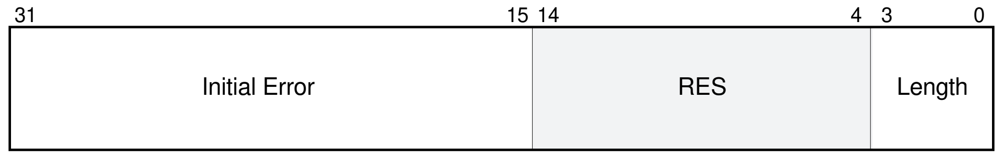


Table 4–32 GB3R Field Description

Bits	Field	Type	Description
31:15	Initial Error	RW	The signed initial value stored in the Bresenham error accumulator.
14:4	RES	RAZ/IGN	Reserved.
3:0	Length	RW	The length, in pixels, of the line segment to be drawn. A value of 0_{16} = 16 pixels.

The GB3R specifies:

- The initial error (<31:15>) used by the Bresenham error logic to determine how to step along the line segment
- The length (<3:0>), which specifies the number of pixels to draw in the line

Although software can write the GB3R directly to set either parameter, it is usually unnecessary for software to write the GB3R under any circumstances. A write to a slope or slope-no-go register sets the parameters to the appropriate value as a function of rise and run of the line. However, when drawing clipped lines and lines under Windows NT that specify subpixel endpoints, software might have to adjust the initial error term by writing the GB3R to draw the appropriate pixels (see Sections 7.2.3.1 and 7.2.3.2 for more information).

Section 6.2.12 describes how the 21030 hardware presets the initial error as a function of the slope, octant, and whether the line is being drawn in a Win32 or X graphics environment.

The GB3R is cleared at reset.

4.4.13 Bresenham Width Register

Figure 4–34 shows the Bresenham width register (GBWR) format, and Table 4–33 describes its fields.

Figure 4–34 GBWR Format

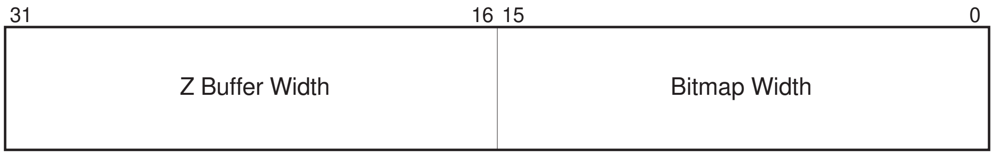


Table 4–33 GBWR Field Description

Bits	Field	Type	Description
31:16	Z-Buffer Width	RW	The width of the Z-buffer. Value is number of bytes for 8-bit packed pixels and number of Dwords for all other pixel types.
15:0	Bitmap Width	RW	The width, in bytes, of the destination bitmap.

The GBWR specifies the width of the destination bitmap and the Z-buffer used in the line drawing operation. Bitmap width (<15:0>) is required for all line drawing, but Z-buffer width (<31:16>) is required only for the Z-buffering line modes.

The 21030 Bresenham setup hardware uses both fields to calculate the increments to the pixel's destination bitmap address and the Z-address, on steps along the minor and major axes as the line is drawn.

Note

The GBWR must be written before writing a slope or slope-no-go register.

The GBWR is cleared at reset.

4.4.14 Stencil Mode Register

Figure 4–35 shows the GSMR format, and Table 4–34 describes its fields.

Figure 4–35 GSMR Format

31	30 28	27 25	24 22	21 19	18 16	15 8	7 0
Z U	Z Test	D Pass	D Fail	S Fail	S Test	S Read Mask	S Write Mask

Table 4–34 GSMR Field Description

Bits	Field	Type	Description
31	ZU	RW	Z-buffer update—When set, the Z-buffer is not updated after the Z-test; when clear, the Z-buffer is updated.
30:28	Z Test	RW	Z-buffer test—Specifies the type of reference and stored value comparison (Table 4–36).
27:25	D Pass	RW	Depth-test pass—Specifies the action to be taken when the comparisons specified in the S-test and Z-test fields both pass (Table 4–37). Table 4–35 defines the codes and actions.
24:22	D Fail	RW	Depth-test fail—Specifies the action to be taken when the comparison specified in the S-test field passes and the comparison specified in the Z-test field fails (Table 4–37). Table 4–35 defines the codes and actions.
21:19	S Fail	RW	Stencil test fail—Specifies the action to be taken if the comparison specified in the S-test field fails. Table 4–35 defines the codes and actions.
18:16	S Test	RW	Stencil test—Specifies the type of reference and stored value comparison (Table 4–36).
15:8	S Read Mask	RW	Stencil read mask—Specifies which of the bits in the 8-bit reference-stencil and stored-stencil values are masked before performing the S-test. Set bits in this field correspond to unmasked bits; clear bits correspond to masked bits.

(continued on next page)

Table 4–34 (Cont.) GSMR Field Description

Bits	Field	Type	Description
7:0	S Write Mask	RW	Stencil write mask—Determines which bits of the 8-bit stencil buffer can be modified on update. Set bits in this field enable update for corresponding bits in the stencil buffer; clear bits disable update.

Table 4–35 defines the GSMR pass and fail field codes and actions.

Table 4–35 GSMR Pass and Fail Fields Codes

Code*	Action	
000	KEEP	No update to stored stencil
001	ZERO	0s $\Rightarrow$ stored stencil
010	REPLACE	Reference stencil $\Rightarrow$ stored stencil
011	INCR	Stored stencil +1 $\Rightarrow$ stored stencil
100	DECR	Stored stencil -1 $\Rightarrow$ stored stencil
101	INVERT	Invert stored stencil $\Rightarrow$ stored stencil
Unused codes are reserved		

*Code in GSMR fields D pass, D fail, and stencil fail (<27:25,24:22,21:19>).

Table 4–36 defines the GSMR test field codes and comparisons.

Table 4–36 GSMR Test Fields Codes

Code*	Comparison	
000	GEQUAL	Reference value $\geq$ stored value
001	ALWAYS	Always pass
010	NEVER	Never pass
011	LESS	Reference value < stored value
100	EQUAL	Reference value = stored value
101	LEQUAL	Reference value $\leq$ stored value
110	GREATER	Reference value > stored value
111	NOTEQUAL	Reference value $\neq$ stored value

*Code in GSMR fields Z-test and stencil buffer test (<30:28,18:16>).

The stencil mode register (GSMR) controls the behavior of the stencil-buffer and Z-buffer logic in simple-Z mode (Section 6.2.2) and the 3D line-drawing modes (Section 6.2.14). The GSMR-defined stencil and Z-buffer operations are compatible with the OpenGL API.

The Z-buffer-test and stencil-test fields (<30:28,18:16>) specify a comparison; both tests compare a potential reference value with the stored value currently in the buffer. The Z-buffer update bit (<31>) determines whether the Z-buffer is updated after the Z-buffer compare.

The stencil read mask (<15:8>) specifies which bit-planes of the reference- and stored-stencil values are masked prior to the comparison.

The depth-test pass, depth-test fail, and stencil-test fields (<27:25,24:22,21:19>) define the how the stencil buffer is updated under the conditions listed in Table 4–37.

Table 4–37 Stencil Buffer Update Conditions

S-Test Result	Z-Test Result	Update According to Field:
Pass	Pass	D Pass (<27:25>)
Pass	Fail	D Fail (<24:22>)
Fail	Pass or Fail	S Fail (<21:19>)

See Sections 6.2.2 and 6.2.14 for a description of stencil- and Z-buffer operations in the simple-Z and 3D line-drawing modes. See Section 6.1.5 for a description of the frame buffer memory format for the buffers.

The GSMR is cleared at reset.

4.4.15 Z-Base-Address Register

Figure 4–36 shows the Z-base-address register (GZBR) format, and Table 4–38 describes its fields.

Figure 4–36 GZBR Format

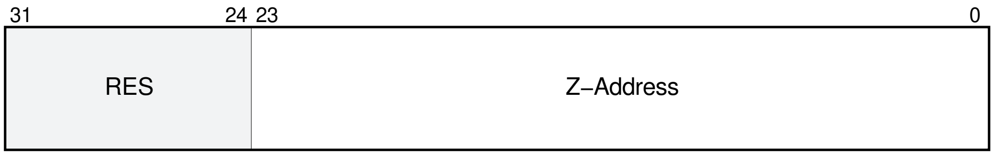


Table 4–38 GZBR Field Description

Bits	Field	Type	Description
31:24	RES	RAZ/IGN	Reserved.
23:0	Z-Address	RW	The starting Z-address for Z-buffered line drawing operations.

The GZBR specifies the starting Z-buffer address for a Z-buffered line or span drawing operation. The GZBR is used in all Z-buffered line-mode operations.

To initiate a Z-buffered line or span drawing operation, software writes a slope register or the GSWR (Section 4.3.2). This action usually, but not always (Section 6.2.14), causes the value in the Z-address field (<23:0>) to be loaded into the internal Z-interpolator hardware, to specify the first Z-address for the line or span drawing operation. The Z-interpolator calculates subsequent Z addresses by Bresenham-stepping through the Z-buffer. (See Section 6.2.14 for more information about using the GZBR to draw Z-buffered lines and spans.)

Note

Writes to the GZBR alter the contents of the DMA base address register (GDBR, Section 4.4.7), and writes to the GDBR alter the contents of the GZBR.

The GZBR is cleared at reset.

4.4.16 Z-Value High and Low Registers

Figure 4–37 shows the Z-value high register (GZVR-H) and Z-value low register (GZVR-L) formats, and Table 4–39 describes their fields.

Figure 4–37 GZVR-H and GZVR-L Formats

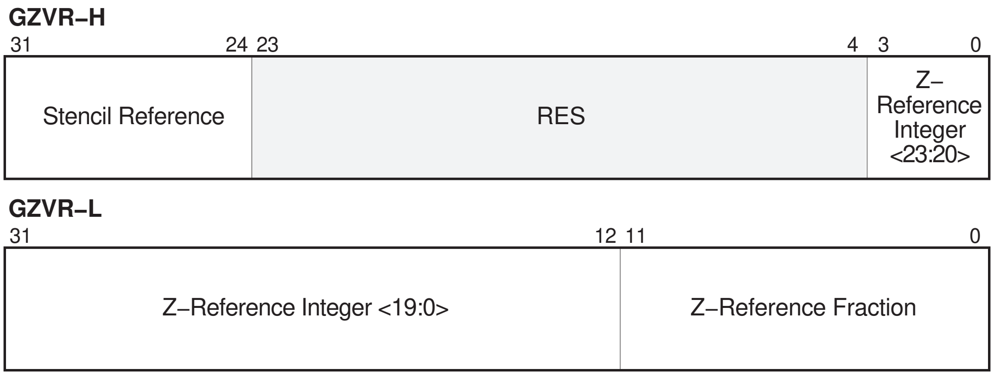


Table 4–39 GZVR-H and GZVR-L Field Description

Bits	Field	Type	Description
GZVR-H			
31:24	Stencil Reference	RW	The stencil value to be used in all stencil-buffer operations.
23:4	RES	RAZ/IGN	Reserved.
3:0	Z-Reference Integer <23:20>	RW	The integer MSBs of the starting Z-value for a Z-buffered line drawing operation.
GZVR-L			
31:12	Z-Reference Integer <19:0>	RW	The integer LSBs of the starting Z-value for a Z-buffered line drawing operation.
11:0	Z-Reference Fraction	RW	The fractional part of the starting Z-value for a Z-buffered line drawing operation.

The GZVR-H and GZVR-L specify the starting 36-bit Z-value for a Z-buffered line or span drawing operation. The GZVR-H and GZVR-L are used in all Z-buffered line-mode operations.

To initiate a Z-buffered line or span drawing operation, software writes a GSLR (Section 4.3.1) or the GSWR (Section 4.3.2). This action usually, but not always (Section 6.2.14), causes the 24-bit Z-reference integer value (GZVR-H <3:0> and GZVR-L <31:12>) and 12-bit Z-reference fraction value (GZVR-L <11:0>) to be loaded into the internal Z-interpolator hardware, to specify the first Z-address for the line or span drawing operation.

The Z-interpolator calculates subsequent Z-values (in full 24.12 precision) by adding the 36-bit Z-increment integer and fraction value from the Z-increment registers (GZIR-L and GZIR-H) to the accumulated Z-value at each step across the span or line.

The GZVR-H also specifies the stencil reference value (<31:24>) used in all stencil-buffer operations. The stencil buffer is packed with the Z-buffer in each frame buffer Dword, such that the Z value at that pixel location resides in the 3 low bytes and the stencil value resides in the high byte. Stencil operations are enabled and parameterized in conjunction with Z operations in the stencil mode register (GSMR, Section 4.4.14). When read, the stencil reference field returns arbitrary data.

(See Section 6.1.5 for more information about stencil and Z-buffer organization. See Section 6.2.14 for more information about using the GZVR-L and GZVR-H in Z-buffered and stencil operations.)

The GZVR-H and GZVR-L are cleared at reset.

4.4.17 Z-Increment High and Low Registers

Figure 4–38 shows the Z-increment high register (GZIR-H) and Z-increment low register (GZIR-L) formats, and Table 4–40 describes their fields.

Figure 4–38 GZIR-H AND GZIR-L Formats

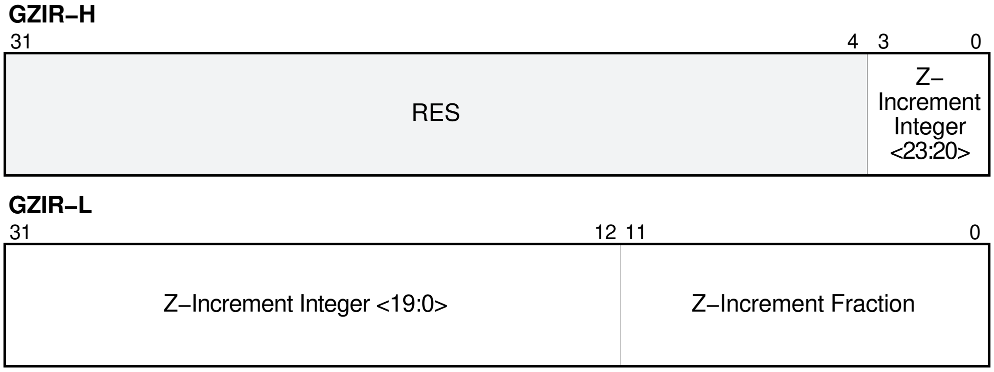


Table 4–40 GZIR-H and GZIR-L Field Description

Bits	Field	Type	Description
GZIR-H			
31:4	RES	RAZ/IGN	Reserved.
3:0	Z-Increment Integer <23:20>	RW	The integer MSBs of the Z-increment value for a Z-buffered line drawing operation.
GZIR-L			
31:12	Z-Increment Integer <19:0>	RW	The integer LSBs of the Z-increment value for a Z-buffered line drawing operation.
11:0	Z-Increment Fraction	RW	The fractional part of the Z-increment value for a Z-buffered line drawing operation.

The GZIR-H and GZIR-L specify the 36-bit Z-increment value in 24.12 precision for a Z-buffered line or span drawing operation. The GZIR-L and GZIR-H are used in all Z-buffered line-mode operations.

To initiate a Z-buffered line or span drawing operation, software writes a GSLR (Section 4.3.1) or the GSWR (Section 4.3.2). This action always causes the 24-bit Z-increment integer value (GZIR-H <3:0> and GZIR-L <19:0>) and 12-bit Z-increment fraction value (GZIR-L <11:0>) to be loaded into the internal Z-interpolator hardware, to specify the Z-increment for the line or span drawing operation.

During Z-buffered operations, the Z-interpolator calculates the next pixel's Z value by adding the 36-bit Z-increment integer and fraction value to the previous pixel's Z value at each step across the span or line.

(See Section 6.2.14 for more information about using the GZIR-L and GZIR-H in Z-buffered operations.)

The GZIR-H and GZIR-L are cleared at reset.

4.4.18 Foreground Register

Figure 4–39 shows the foreground register (GFGR) format, and Table 4–41 describes its field.

Figure 4–39 GFGR Format

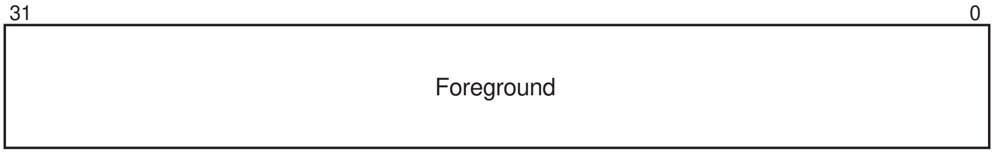


Table 4–41 GFGR Field Description

Bits	Field	Type	Description
31:0	Foreground	RW	Defines the foreground color (or set of colors) used in pixel substitution in any of the transparent or opaque stipple, line, or fill modes.

The GFGR defines foreground pixel colors. In any of the transparent or opaque stipple, line, or fill modes, foreground color is substituted for ones in the data. The data can be any of the following:

- Write data on a write to the frame buffer or GCTR
- Data in the GDAR on a write to a GSLR or the GSWR
- Data in the GDAR on a write to the frame buffer in a fill mode

The foreground field is a 32-bit quantity regardless of the depth of the bitmap type currently being drawn to. Consequently, software must compensate for the actual depth by replicating the color across the foreground field for bitmap depths less than 32-bpp (Figure 4–40). For example, to present the same color to each possible buffer in 8-bpp mode, the foreground color must be replicated four times across foreground field. Similarly, in 12-bpp mode, the foreground color must be replicated across both sets of RGB values. (The bitmap formats are described in Sections 6.1.5 through 6.1.5.3.)

When drawing to 12-bpp bitmaps in a 32-bpp frame buffer, the plane mask registers (GPMR, Section 4.4.20) can be used with the GFGR and background register (GBGR) to draw to only the target bitmap while masking off the other bitmap.

Note

A write to the red or blue increment register (GRIR, Section 4.4.23 or GBIR, Section 4.4.27) alters the contents of the GFGR.

The GFGR is initialized to 00000000_{16} at reset.

Figure 4–40 shows the GFGR and GBGR contents as a function of the bitmap depth in 8-bpp and 32-bpp frame buffers.

Figure 4–40 Foreground and Background as a Function of Bitmap Depth

8–bpp Frame Buffer

	31–24	23–16	15–8	7–0
8–bpp Bitmap	index	index	index	index

32–bpp Frame Buffer

	31–24	23–20	19–16	15–12	11–8	7–4	3–0
8–bpp Bitmap	index	index		index		index	
12–bpp Bitmap	tag	red	red	green	green	blue	blue
24–bpp Bitmap	tag	red		green		blue	

4.4.19 Background Register

Figure 4–41 shows the background register (GBGR) format, and Table 4–42 describes its field.

Figure 4–41 GBGR Format

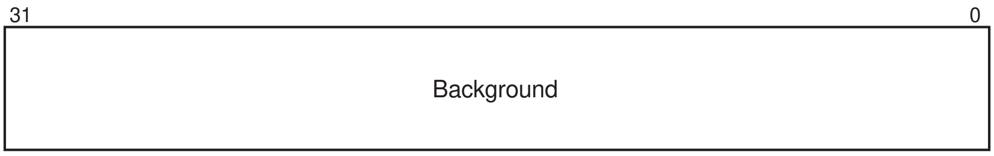


Table 4–42 GBGR Field Description

Bits	Field	Type	Description
31:0	Background	RW	Defines the background color (or set of colors) used in pixel substitution in any of the opaque stipple, line, or fill modes.

The GBGR defines background pixel colors. In any of the opaque-stipple, opaque-line, or opaque-fill modes, background color is substituted for zeros in the stipple mask or line mask. The stipple mask and line mask data can be any of the following:

- Write data on a write to the frame buffer or GCTR
- Data in the GDAR on a write to a GSLR or the GSWR
- Data in the GDAR on a write to the frame buffer in a fill mode

The background field is a 32-bit quantity regardless of the depth of the destination bitmap currently being drawn to. Consequently, software must arrange the colors or indices based on the actual depth, as it does for the foreground register (GFGR, Section 4.4.18). See Figure 4–40.

Note

A write to the green or blue increment register (GGIR, Section 4.4.25 or GBIR, Section 4.4.27) alters the contents of the GBGR.

The GBGR is initialized to 00000000_{16} at reset.

4.4.20 Plane Mask Registers

Note

> Several copies of the plane mask registers (GPMRs) are physically resident in the VRAMs rather than the 21030, and are undefined at reset. Because most standard DRAMs do not support persistent write-per-bit, GPMR writes should not be performed to back buffers populated by such DRAMs.

Figure 4–42 shows the GPMR format, and Table 4–43 describes the field.

Figure 4–42 GPMR Format

31 … 0
Plane Mask

Table 4–43 GPMR Field Description

Bits	Field	Type	Description
31:0	Plane Mask	RW	A depth-dependent mask. Writes are enabled for bits in the pixel value that correspond to set mask bits, and disabled for bits in the pixel value that correspond to clear mask bits.

The GPMRs specify the bits within each pixel value that are affected by a write to the frame buffer. Each GPMR bit determines whether the corresponding bit in a pixel value is updated on a write. A mask bit value of one enables the corresponding pixel value bit to be updated, and a mask bit value of zero disables updating.

Although drawing to an 8-bpp destination requires only an 8-bit plane mask, the 21030 requires that the 8-bit mask be replicated across the 32-bit register (Figure 4–43).

The GPMRs are used primarily to:

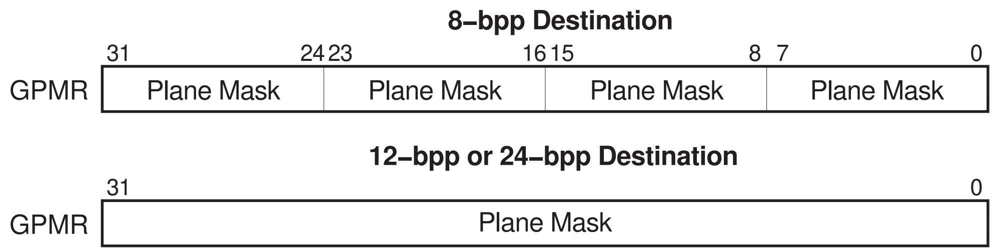
- Disable writes to one or more 8-bpp or 12-bpp color bitmaps in a 32-bpp frame buffer
- Isolate writes to overlay planes, window tags, and so on
- Isolate Z-buffer access to either the Z value or the stencil value

(See Chapters 6 and 7 for more information about using the plane mask registers.)

The GPMRs are undefined at reset.

Figure 4–43 shows the plane-mask format for 8-bpp, 12-bpp, and 24-bpp destinations.

Figure 4–43 Plane Mask Formats



4.4.21 Pixel Mask Register

The pixel mask register (GPXR) is used to mask pixels in opaque-stipple, simple, and simple-Z modes; its format is mode-dependent.

4.4.21.1 Opaque-Stipple Mode

Figure 4–44 shows the GPXR opaque-stipple mode format, and Table 4–44 describes its field.

Figure 4–44 GPXR Opaque-Stipple Mode Format

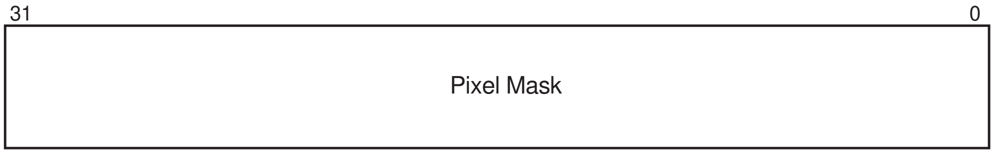


Table 4–44 GPXR Opaque-Stipple Mode Format Field Description

Bits	Field	Type	Description
31:0	Pixel Mask	RW	The mask data for each 32-pixel stippled span. Writes are enabled for pixels that correspond to set mask bits, and disabled for pixels that correspond to clear mask bits.

In opaque-stipple mode, the frame buffer write data determines whether each of the 32 pixels beginning at that address should be filled with foreground or background color, and the GPXR determines which pixels are written. Prior to the frame buffer write, the 32-bit mask is written to the GPXR to selectively write-enable each pixel on the subsequent opaque-stipple operation.

4.4.21.2 Simple and Simple-Z Modes

Figure 4–45 shows the GPXR simple and simple-Z modes format, and Table 4–45 describes its fields.

Figure 4–45 GPXR Simple and Simple-Z Modes Format

Table 4–45 GPXR Simple and Simple-Z Modes Field Description

Bits	Field	Type	Description
31:4	RES	RAZ/IGN	Reserved.
3:0	Mask GPXR	RW	Mask data for each 32-bit frame buffer write. Writes are enabled for pixels that correspond to set mask bits, and disabled for pixels that correspond to clear mask bits.

The mask GPXR field (<3:0>) determines which data bytes are to be written in the next frame buffer write. The field is logically ANDed with the incoming PCI byte mask, to create the byte mask that is ultimately used in simple and simple-Z modes. When writing to 12-bpp or 24-bpp destinations, <0> determines whether to write the pixel.

Pixel-mask data for simple and simple-Z modes is primarily useful in systems based on Alpha AXP microprocessors. Because the Alpha AXP instruction set does not support byte granularity, a true PCI byte mask might not be available.

4.4.21.3 Any Mode

The GPXR is mapped into the 21030 register space twice: as a persistent GPXR and as a one-shot GPXR (Table 2–2). When written as a one-shot GPXR, the value in the GPXR is used only for the next operation. After that operation is complete, the GPXR reinitializes to an inactive state of FFFFFFFF. When written as a persistent GPXR, the GPXR retains its value until next written at either address. The GPXR state bit in the mode register (GMOR <23>, Section 4.4.1) indicates the current state of GPXR.

The GPXR is initialized to FFFFFFFF at reset.

4.4.22 Red-Value Register

The function of the red-value register (GRVR) depends on whether the graphics mode is a color-interpolated or a sequential-interpolated line mode.

4.4.22.1 Color-Interpolated Line Mode

Figure 4–46 shows the GRVR color-interpolated line-mode format, and Table 4–46 describes its fields.

Figure 4–46 GRVR Color-Interpolated Line-Mode Format

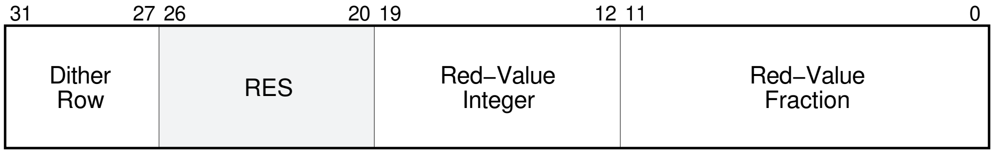


Table 4–46 GRVR Color-Interpolated Line-Mode Format Field Description

Bits	Field	Type	Description
31:27	Dither Row	RW	The row pointer into the 32 × 32 dither matrix.
26:20	RES	RAZ/IGN	Reserved.
19:12	Red-Value Integer	RW	The integer part of the starting red value for a color-interpolated line drawing operation.
11:0	Red-Value Fraction	RW	The fractional part of the starting red value for a color-interpolated line drawing operation.

The GRVR specifies the starting 20-bit red value for a color-interpolated line or span drawing operation. The GRVR is used in all color-interpolated line mode operations.

To initiate a color-interpolated line or span drawing operation, software writes a slope register (GSLR<7:0>) or the span width register (GSWR). This action usually loads the red-value integer and fraction (<19:12,11:0>) into the internal color-interpolator hardware as the starting red value for the line or span drawing operation. However, the red-value parameters are sampled only if the address register (GADR) was written since the last drawing operation (see Section 6.2.14.4 for more information).

The color interpolator calculates subsequent red values for each pixel in the span or line (in full 8.12 precision) by adding the 20-bit red-increment integer and fraction values from the red increment register (GRIR, Section 4.4.23) to the accumulated red value at each step across the span or line. (See Section 6.2.14 for more information about using the GRVR in color-interpolated line and span drawing operations.)

The dither-row field (<31:27>) specifies the row pointer into the internal 32 × 32 dither matrix, which is used in all dithered line-drawing modes. The pointer addresses the matrix row and, together with the column pointer from the green-value register (GGVR, Section 4.4.24), produces the dither offsets added to each color prior to decimation when drawing in a dithering mode.

4.4.22.2 Sequential-Interpolated Line Mode

Figure 4–47 shows the GRVR sequential-interpolated line-mode format, and Table 4–47 describes its fields.

Figure 4–47 GRVR Sequential-Interpolated Line-Mode Format

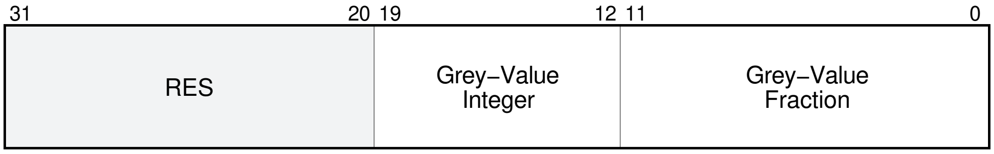


Table 4–47 GRVR Sequential-Interpolated Line-Mode Format Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Grey-Value Integer	RW	The integer part of the starting grey-scale value for a sequentially interpolated line drawing operation.
11:0	Grey-Value Fraction	RW	This is the fractional part of the starting grey-scale value for a sequentially interpolated line drawing operation.

The GRVR specifies the starting 20-bit grey-scale value for a sequentially interpolated line drawing operation to an 8-bpp destination bitmap. The GRVR is used in all sequential-interpolated line mode operations.

To initiate a sequentially interpolated line or span drawing operation, software writes a GSLR or the GSWR. This action usually loads grey-value integer and fraction (<19:12,11:0>) into the internal sequential-interpolator hardware as the starting grey-scale value for the line or span drawing operation. However, the grey-value parameters are sampled only if the address register GADR was written since the last drawing operation (see Section 6.2.14.4 for more information).

The sequential interpolator calculates subsequent grey-scale values for each pixel in the span or line (in full 8.12 precision) by adding the 20-bit grey-increment integer and fraction value from the GRIR to the accumulated grey-scale value at each step across the span or line. Sequentially interpolated lines can also be drawn without using the GSLRs.

(See Section 6.2.14 for more information about using the GRVR in sequentially interpolated line and span drawing operations.)

The GRVR is cleared at reset.

4.4.23 Red-Increment Register

The function of the red-increment register (GRIR) depends on whether the graphics mode is a color-interpolated or a sequential-interpolated line mode.

Note

Writes to the foreground register (GFGR, Section 4.4.18) alter the contents of the GRIR.

4.4.23.1 Color-Interpolated Line Mode

Figure 4–48 shows the GRIR color-interpolated line-mode format, and Table 4–48 describes its fields.

Figure 4–48 GRIR Color-Interpolated Line-Mode Format

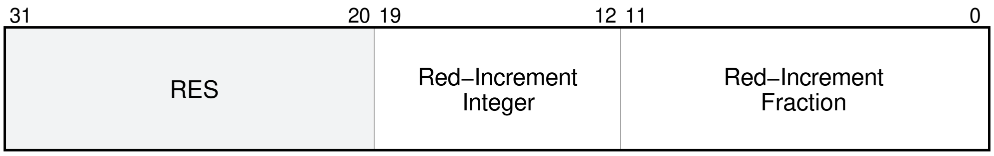


Table 4–48 GRIR Color-Interpolated Line-Mode Format Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Red-Increment Integer	RW	The integer part of the red-increment value for a color-interpolated line drawing operation.
11:0	Red-Increment Fraction	RW	The fractional part of the red-increment value for a color-interpolated line drawing operation.

The GRIR specifies the 20-bit red-increment value for a color-interpolated line or span drawing operation. The GRIR is used in all color-interpolated line mode operations.

To initiate a color-interpolated line or span drawing operation, software writes a slope register (GSLR<7:0>) or the span width register (GSWR). This action loads the red-increment integer and fraction values (<19:12,11:0>) into the internal color-interpolator hardware as the red-increment for the line or span drawing operation.

During color-interpolated operations, the color interpolator calculates the next pixel's red value by adding the 20-bit red-increment integer and fraction value to the previous pixel's red value, at each step across the span or line. (See Section 6.2.14 for more information about using the GRIR in color-interpolated line and span drawing operations.)

4.4.23.2 Sequential-Interpolated Line Mode

Figure 4–49 shows the GRIR sequential-interpolated line-mode format, and Table 4–49 describes its fields.

Figure 4–49 GRIR Sequential-Interpolated Line-Mode Format

31 … 20	19 … 12	11 … 0
RES	Grey–Increment Integer	Grey–Increment Fraction

Table 4–49 GRIR Sequential-Interpolated Line-Mode Format Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Grey-Increment Integer	RW	The integer part of the grey-scale increment value for a sequentially interpolated line drawing operation.
11:0	Grey-Increment Fraction	RW	The fractional part of the grey-scale increment value for a sequentially interpolated line drawing operation.

The GRIR specifies the 20-bit grey-scale increment value for a sequentially interpolated line drawing operation to an 8-bpp destination bitmap. The GRIR is used in all sequential-interpolated line mode operations.

To initiate a sequentially interpolated line or span drawing operation, software writes a GSLR or the GSWR. This action loads the grey-increment integer and fraction values (<19:12,11:0>) into the internal sequential-interpolator hardware as the grey-increment for the line or span drawing operation.

During sequentially interpolated operations, the sequential interpolator calculates the next pixel's grey-scale value by adding the 20-bit grey-increment integer and fraction value to the previous pixel's grey-scale value, at each step across the span or line.

(See Section 6.2.14 for more information about using the GRIR in sequentially interpolated line and span drawing operations.)

The GRIR is cleared at reset.

4.4.24 Green-Value Register

Figure 4–50 shows the green-value register (GGVR) format, and Table 4–50 describes its fields.

Figure 4–50 GGVR Format

31 27	26 20	19 12	11 0
Dither Column	RES	Green–Value Integer	Green–Value Fraction

Table 4–50 GGVR Field Description

Bits	Field	Type	Description
31:27	Dither Column	RW	The column pointer into the 32 × 32 dither matrix.
26:20	RES	RAZ/IGN	Reserved.
19:12	Green-Value Integer	RW	The integer part of the starting green value for a color-interpolated line drawing operation.
11:0	Green-Value Fraction	RW	The fractional part of the starting green value for a color-interpolated line drawing operation.

The GGVR specifies the starting 20-bit green value for a color-interpolated line or span drawing operation. The behavior and use of the GGVR are the same as the GRVR in color-interpolated line mode (Section 4.4.22), except that the GGVR specifies the following:

- Green value rather than red value
- Column pointer, rather than row pointer, into the dither matrix

The GGVR is cleared at reset.

4.4.25 Green-Increment Register

Figure 4–51 shows the green-increment register (GGIR) format, and Table 4–51 describes its fields.

Figure 4–51 GGIR Format

31 — 20	19 — 12	11 — 0
RES	Green–Increment Integer	Green–Increment Fraction

Table 4–51 GGIR Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Green-Increment Integer	RW	The integer part of the green-increment value for a color-interpolated line drawing operation.
11:0	Green-Increment Fraction	RW	The fractional part of the green-increment value for a color-interpolated line drawing operation.

The GGIR specifies the 20-bit green-increment value for a color-interpolated line or span drawing operation. The behavior and use of the GGIR are the same as the GRIR in color-interpolated line mode (Section 4.4.23), except that the GGIR specifies the green increment rather than the red increment.

Note

Writes to the background register (GBGR, Section 4.4.19) alter the contents of the GGIR.

The GGIR is cleared at reset.

4.4.26 Blue-Value Register

Figure 4–52 shows the blue-value register (GBVR) format, and Table 4–52 describes its fields.

Figure 4–52 GBVR Format

31 … 20	19 … 12	11 … 0
RES	Blue–Value Integer	Blue–Value Fraction

Table 4–52 GBVR Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Blue-Value Integer	RW	The integer part of the starting blue value for a color-interpolated line drawing operation.
11:0	Blue-Value Fraction	RW	The fractional part of the starting blue value for a color-interpolated line drawing operation.

The GBVR specifies the starting 20-bit blue value for a color-interpolated line or span drawing operation. The behavior and use of the GBVR are the same as the GRVR in color-interpolated line mode (Section 4.4.22), except that the GBVR specifies the blue value and does not specify an index into the dither matrix.

The GBVR is cleared at reset.

4.4.27 Blue-Increment Register

Figure 4–53 shows the blue-increment register (GBIR) format, and Table 4–53 describes its fields.

Figure 4–53 GBIR Format

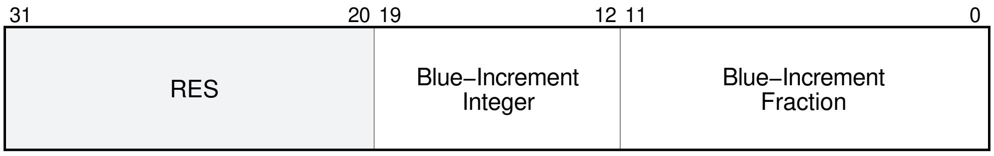


Table 4–53 GBIR Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:12	Blue-Increment Integer	RW	The integer part of the blue-increment value for a color-interpolated line drawing operation.
11:0	Blue-Increment Fraction	RW	The fractional part of the blue-increment value for a color-interpolated line drawing operation.

The GBIR specifies the 20-bit blue-increment value for a color-interpolated line or span drawing operation. The behavior and use of the GBIR are the same as the GRIR in color-interpolated line mode, except that the GBIR specifies the blue increment rather than the red increment.

Note

Writes to the foreground or background register (GFGR, Section 4.4.18 or GBGR, Section 4.4.19) alter the contents of the GBIR.

The GBIR is cleared at reset.

4.4.28 Deep Register

Figure 4–54 shows the deep register (GDER) format, and Table 4–54 describes its fields.

Figure 4–54 GDER Format

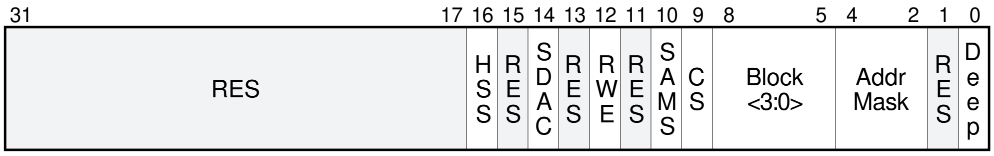


Table 4–54 GDER Field Description

Bits	Field	Type	Description
31:17	RES	RAZ/IGN	Reserved.
16	HSS	RW	Horizontal sync select—When set, **vsync_l** carries vertical sync pulses and **hsync_l** carries horizontal sync pulses; when clear, **vsync_l** carries a composite sync signal and **hsync_l** carries a stereo control signal.
15	RES	RAZ/IGN	Reserved.
14	SDAC	RW	Slow DAC—When set, the RAMDAC MPU port delay is enabled; when clear, the delay is disabled.
13	RES	RAZ/IGN	Reserved.
12	RWE	RW	ROM write enable—When set, the optional, external EEPROM can be written; otherwise, writes to the EEPROM are disabled.
11	RES	RAZ/IGN	Reserved.
10	SAMS	RW	Serial-access memory size—Set for SAMs with 256 entries; clear for SAMs with 512 entries.
9	CS	RW	Column size—Set for 256 columns (128K $\times$ *n* parts); clear for 512 columns (256K $\times$ *n* parts).
8:5	Block <3:0>	RW	For each block bit *n*: When set, eight columns are enabled for VRAMs in segment *n*; when clear, four columns are enabled for VRAMs in segment *n*.

(continued on next page)

Table 4–54 (Cont.) GDER Field Description

<table>
<tr><th>Bits</th><th>Field</th><th>Type</th><th>Description</th></tr>
<tr><td>4:2</td><td>Addr Mask</td><td>RW</td><td>Address mask—Determines which bits of the incoming PCI address are masked according to the size of the 21030 address space, as follows:
<table>
<tr><th><4:2></th><th>Mask</th><th>Core Map Size</th></tr>
<tr><td>000</td><td>Mask <24:22></td><td>4MB</td></tr>
<tr><td>001</td><td>Mask <24:23></td><td>8MB</td></tr>
<tr><td>011</td><td>Mask bit 24</td><td>16MB</td></tr>
<tr><td>111</td><td>No masking</td><td>32MB</td></tr>
</table>
Unused codes are reserved.</td></tr>
<tr><td>1</td><td>RES</td><td>RAZ/IGN</td><td>Reserved.</td></tr>
<tr><td>0</td><td>Deep</td><td>RW</td><td>Set for a 32-bpp frame buffer and clear for an 8-bpp frame buffer.</td></tr>
</table>

The GDER specifies the type and configuration of the frame buffer. The 21030 supports 8-bpp and 32-bpp frame buffers, comprising various types of VRAM parts. The GDER also determines whether video sync is for a standard or stereo display, enables RAMDAC access delay, and write-enables the external EEPROM. The GDER must be written before the first frame buffer access (it is typically written once, at initialization time).

The horizontal sync select bit (<16>) specifies the type of video sync pulses on both the **vsync_l** and **hsync_l** pins. When this bit is set, the 21030 generates separate horizontal and vertical sync pulses; when cleared, the 21030 generates a composite vertical sync signal and a stereo display output control signal.

The slow DAC bit (<14>) specifies whether the 21030 inserts a delay period after each RAMDAC MPU access cycle to comply with the minimum MPU cycle specifications of certain RAMDACs. The delay occurs while the **dacce<1:0>** signals are deasserted and the delay period is 4 frame buffer clocks.

The ROM write enable bit (<12>) is set to write-enable the external EEPROM. When <12> is clear, external EEPROM writes are disabled; however, the cycle is externally visible and can be used to implement a write-only parallel port (Section 8.3).

Note

The 21030 supports block write and persistent plane mask, but most DRAMs do not. If DRAMs that do not support these features are used, plane masking and block-mode operations cannot be used.

The SAM size bit, column size bit, and block field (<10,9,8:5>) identify the type of VRAM.

The SAM size and column size bits (<10,9>) are set according to the size (128K $\times$ *n* or 256K $\times$ *n*) of the VRAM parts that populate the frame buffer.

The block field (<8:5>) defines the format of block-write data expected by the VRAMs. Depending on the vendor and the memory organization, a VRAM can be enabled to write eight or four columns per block-write CAS cycle. However, the 21030 writes only four columns at a time, though the different types of VRAM require slightly different control. The 4-bit width of the block field allows different VRAM devices to occupy different segments in the same frame buffer bank. VRAMs map to the 21030 frame buffer segments and banks (Section 8.1).

The address mask (<4:2>) determines how incoming PCI address bits <24:22> are masked to index into the address space. (See Chapter 2 for more information.)

The deep bit (<0>) specifies the physical depth of the frame buffer. Note that this bit does not always indicate the depth of the displayed bitmap in a 32-bpp frame buffer, which supports various bitmap depths. The visual depth is specified separately in the GMOR (Section 4.4.1). (Sections 6.1.5 through 6.1.5.3 describe the bitmaps supported by the 21030.)

The GDER is cleared at reset.

4.5 Video Timing Registers

The video timing registers control the screen and VRAM shift-register timing. They specify the parameters to generate composite sync and blank and VRAM shift clocks, and determine whether stereo display is enabled.

4.5.1 Horizontal Control Register

Figure 4–55 shows the horizontal control register (VHCR) format, and Table 4–55 describes its fields.

Figure 4–55 VHCR Format

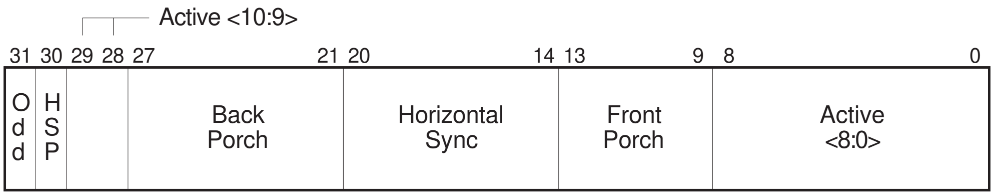


Table 4–55 VHCR Field Description

Bits	Field	Type	Description
31	Odd	RW	When clear, enables the last 4 pixels on a scan line to be displayed if the value of active (<29:28,8:0>) is an odd number. When set, the last 4 pixels are not displayed.
30	HSP	RW	Horizontal sync polarity. When set, horizontal sync is asserted high. When clear, horizontal sync is asserted low. (Describes 21030 step B functionality. In the 21030 step A, this bit is reserved and horizontal sync is low.)
29:28	Active <10:9>	RW	Active field MSBs. See bits <8:0>.
27:21	Back Porch	RW	The value of this field is ≥ 2. It specifies the number of pixels between the deassertion of horizontal sync and the deassertion of horizontal blank, as follows: *Number of pixels = back porch value* $\times$ *4*

(continued on next page)

Table 4–55 (Cont.) VHCR Field Description

Bits	Field	Type	Description
20:14	Horizontal Sync	RW	The value of this field is $\geq$2. It specifies the number of pixels between the assertion of horizontal sync and the deassertion of horizontal sync, as follows: *Number of pixels = horizontal sync value* $\times$ *4*
13:9	Front Porch	RW	The value of this field is $\geq$2. It specifies the number of pixels between the assertion of horizontal blank and the assertion of horizontal sync, as follows: *Number of pixels = front porch value* $\times$ *4*
8:0	Active <8:0>	RW	Active field LSBs. The value of <29:28,8:0> (Active <10:0>) is $\geq$2. It specifies the number of pixels between the deassertion of horizontal blank and the assertion of horizontal blank (that is, the number of active display pixels on the scan line), as follows: *Number of pixels = active value* $\div$ *4* Although 000_{16} and 001_{16} = <2, they are exceptions, as follows:

<8:0>	Pixels
002	8
003	12
⋮	⋮
7FF	8188
000	8192
001	8196

The VHCR contains the horizontal timing parameters for the monitor in use. The parameters specify various times between the assertion and deassertion of the horizontal sync and blank signals. The values are specified in multiples of 4 pixels.

The odd bit (<31>) determines whether the last 4 pixels of all scan lines are displayed when active (<29:28,8:0>) value MOD 2 = 1. For example, if the scan-line active width is 1028 pixels, the odd bit determines whether pixels 1024..1027 are displayed. This allows the specification of active field values that are 4 pixels longer than expected by the monitor. It also effectively skews the scan lines by 4 pixels when they are mapped into frame buffer memory. Because the 21030 memory controller comprises four independent memory controllers, skewing maps vertically contiguous screen pixels into memory in a way that allows such pixels to be drawn simultaneously. This greatly improves the drawing rates for tall, thin areas (that is, lines). (See Section 7.2.3.3 for information about programming this feature. See Section 8.1 for information about mapping the visible screen using the 21030 memory controller.)

Note

The VHCR must be initialized before enabling video in the video valid register (VVVR, Section 4.5.4).

The VHCR is cleared at reset.

4.5.2 Vertical Control Register

Figure 4–56 shows the vertical control register (VVCR) format, and Table 4–56 describes its fields.

Figure 4–56 VVCR Format

31	30	29 28	27 22	21 16	15 11	10 0
SE	VSP	RES	Back Porch	Vertical Sync	Front Porch	Active Lines

Table 4–56 VVCR Field Description

Bits	Field	Type	Description
31	SE	RW	Stereo enable—When set, the refresh pointer is reloaded with the video base address after every other frame; when clear, it is reloaded after every frame.
30	VSP	RW	Vertical sync polarity. When set, vertical sync is asserted high. When clear, vertical sync is asserted low. (Describes 21030 step B functionality. In the 21030 step A, this bit is reserved and vertical sync is low.)
29:28	RES	RAZ/IGN	Reserved.
27:22	Back Porch	RW	The value of this field is ≥ 1. It specifies the number of lines between the deassertion of vertical sync and the deassertion of vertical blank.
21:16	Vertical Sync	RW	The value of this field is ≥ 1. It specifies the number of lines between the assertion of vertical sync and the deassertion of vertical sync.
15:11	Front Porch	RW	The value of this field is ≥ 0. It specifies the number of lines between the assertion of vertical blank and the assertion of vertical sync.
10:0	Active Lines	RW	The value of this field is ≥ 1. It specifies the number of lines between the deassertion of vertical blank and the assertion of vertical blank; that is, the number of active scan lines in the frame.

The VVCR contains the vertical timing parameters for the monitor in use. The parameter values are specified in number of scan lines.

The *refresh pointer* is an address pointer in the 21030's timing logic. It contains the frame buffer location at which the screen is being refreshed. The stereo enable bit (<31>) determines when to reload the working-screen refresh pointer from the video base-address field in the video base-address register (VVBR <8:0>, Section 4.5.3). In nonstereo modes, the 21030 reloads the refresh pointer with the video base address at vertical sync time to reset the frame.

The 21030 implements stereo by dividing the visible screen into left-eye and right-eye portions that are contiguous in frame buffer memory. Each portion is then *stretched* to fill one full left or right screen in one-half of the normal frame time. The video pointer is reloaded with the video base-address value after every two frames, displaying left, then right; left, then right; and so on.

The VVCR is cleared at reset and must be initialized before enabling video in the video valid register (VVVR, Section 4.5.4).

4.5.3 Video Base-Address Register

Figure 4–57 shows the video base-address register (VVBR) format, and Table 4–57 describes its fields.

Figure 4–57 VVBR Format

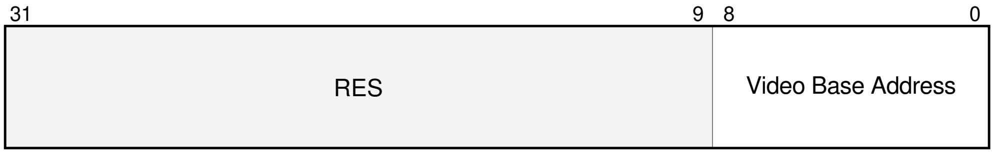


Table 4–57 VVBR Field Description

Bits	Field	Type	Description
31:9	RES	RAZ/IGN	Reserved.
8:0	Video Base Address	WO	The row address of the start of the visible frame in 21030 frame buffer space.

The VVBR specifies the frame buffer row-address that is the start of the visible portion of video memory. The video base-address field (<8:0>) contains a pixel address aligned to either 2K pixels or 4K pixels, depending on whether the frame buffer is populated with 128K $\times$ n or 256K $\times$ n VRAMs; that is, depending on the value of the column size bit in the deep register (GDER <9>, Section 4.4.28).

Table 4–58 shows the required video base-address alignment as a function of the VRAM size.

Table 4–58 Video Base-Address Alignment According to VRAM Size

VRAM Size	Frame Buffer Depth	Row Size Pixels	Row Size Bytes	Frame Buffer Address Bits
128KB	8-bpp	2K	2K	<19:11>
128KB	24-bpp	2K	8K	<21:13>
256KB	8-bpp	4K	4K	<20:12>
256KB	24-bpp	4K	16K	<22:14>

Note

Software must ensure that sufficient video memory exists beyond the video base address to display the entire screen.

The write-only VVBR is cleared at reset.

4.5.4 Video Valid Register

Figure 4–58 shows the video valid register (VVVR) format, and Table 4–59 describes its fields.

Figure 4–58 VVVR Format

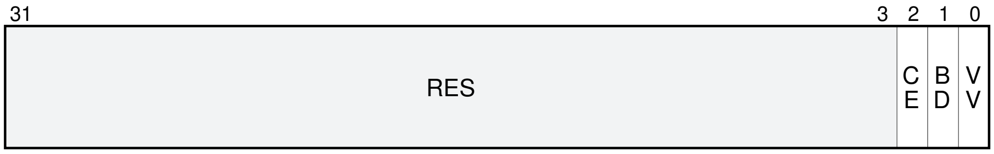


Table 4–59 VVVR Field Description

Bits	Field	Type	Description
31:3	RES	RAZ/IGN	Reserved.
2	CE	RW	Cursor enable—When set, the 21030's on-chip $64 \times 64 \times 2$ cursor is enabled; when clear, it is disabled.
1	BD	RW	Blank Disable
0	VV	RW	Video Valid These bits enable the display of the 21030 frame buffer, as follows: 0 0 Video disabled (but not blanked) 0 1 Active display 1 0 Screen blanked 1 1 Screen blanked (but active sync signals)

The VVVR enables display of the 21030 frame buffer and on-chip cursor.

The blank disable bit (<1>) determines whether **blank_l** (composite blank) is asserted. When blank disable is set, **blank_l** is asserted. When blank disable is clear, **blank_l** is asserted as a function of the video blank circuits. Its edges are controlled by the VHCR (Section 4.5.1) and VVCR (Section 4.5.2).

The video-valid bit (<0>) enables the 21030 video sync and VRAM clock control circuits and enables the assertion of the corresponding control pins: **hsync_l**, **vsync_l**, **hold_l**, and **toggle**. The video-valid bit indicates that the VHCR and VVCR parameters are valid; that is, the correct parameters for the screen mode and type of monitor.

Note

The VHCR (Section 4.5.1) and VVCR (Section 4.5.2) must be initialized before video is enabled in the VVVR.

The VVVR is cleared at reset.

4.5.5 Video Shift-Address Register

Figure 4–59 shows the video shift-address register (VSAR) format, and Table 4–60 describes its fields.

Figure 4–59 VSAR Format

31 : 20	19 : 10	9 : 0
RES	Last Shift Address	Interrupt Shift Address

Table 4–60 VSAR Field Description

Bits	Field	Type	Description
31:20	RES	RAZ/IGN	Reserved.
19:10	Last Shift Address	RO	Returns the shift address most recently used in a split read-transfer cycle.
9:0	Interrupt Shift Address	RW	The address of a VRAM split read-transfer cycle at which an interrupt should be issued.

The VSAR specifies the address of a VRAM split read-transfer cycle at which an interrupt should be issued. Every time the 21030 executes a split read-transfer cycle it compares the address issued with the interrupt shift-address (<9:0>). If a match is detected and shift-address interrupts are enabled in the interrupt status register (SISR <17>, Section 4.7.2), the 21030 posts an interrupt on the **inta_l** pin.

A split read-transfer cycle transfers one-half of the row specified by the shift-address VRAMs in display banks (for 8-bpp frame buffers) or display segments (for 24-bpp frame buffers). Each split read-transfer cycle loads 1024 or 2048 pixels, depending on the VRAM size, into the VRAM SAMs. Table 4–61 indicates the number of pixels that are loaded and how the interrupt shift-address is mapped to the frame buffer byte address.

Table 4–61 Interrupt Shift-Address to Frame Buffer Byte-Address Map

VRAM Depth	Frame Buffer Depth	Pixels Loaded per Split Read Transfer	Shift-Address Bits in Frame Buffer Byte-Address
128KB	8-bpp	1K pixels	<19:10>
256KB	8-bpp	2K pixels	<20:11>
128KB	32-bpp	1K pixels	<21:12>
256KB	32-bpp	2K pixels	<22:13>

Requesting an explicit interrupt at a specific shift address or polling for the last shift address can be useful in scheduling the off-screen buffer clear, copy, and draw operations that are typical in performing animations to a portion of the visible screen (a window).

(See Section 8.1 for more information about mapping display screens to the physical VRAM devices in 8-bpp and 32-bpp frame buffers.)

The VSAR is cleared at reset.

4.6 Cursor Registers

The cursor registers specify the screen location of the on-chip $64 \times 64 \times 2$ cursor and the frame buffer location of the cursor image. The cursor output pins (**cursor<7:0>**) are typically driven into the overlay ports of the RAMDAC. (See Section 7.2.8 for more information about programming the cursor.)

For system applications in which the RAMDAC contains a cursor generator, the 21030 cursor registers and display function can be disabled by clearing the cursor enable bit in the video valid register (VVVR <2>, Section 4.5.4).

4.6.1 Cursor XY Register

Figure 4–60 shows the cursor XY register (CXYR) format, and Table 4–62 describes its fields.

Figure 4–60 CXYR Format

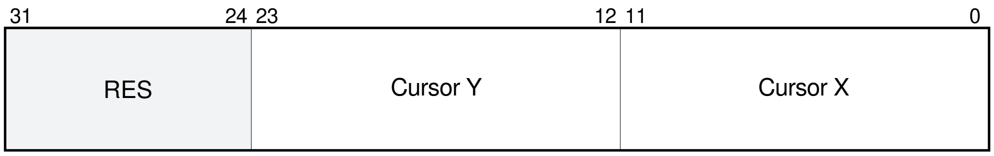


Table 4–62 CXYR Field Description

Bits	Field	Type	Description
31:24	RES	RAZ/IGN	Reserved.
23:12	Cursor Y	RW	The Y coordinate of the top-most pixels inside the cursor.
11:0	Cursor X	RW	The X coordinate of the left-most pixels inside the cursor.

The CXYR specifies the location of the displayed cursor.

Cursor X (<11:0>) and cursor Y (<23:12>) specify the top-left pixel inside the cursor. The cursor is displayed in the rectangular region from *cursor x, cursor y* through *cursor x + 63, cursor y + 63*.

The 21030 imposes limits on the range of programmable *x* and *y* coordinates for the cursor. Table 4–63 specifies the minimum and maximum values.

Table 4–63 Cursor Coordinate Limits

Limit	Value
Cursor X minimum =	(Number of pixels between the start of horizontal sync and the end of horizontal back porch) -63
Cursor X maximum =	(Number of pixels between the start of horizontal sync and the start of horizontal front porch) -1

(continued on next page)

Table 4–63 (Cont.) Cursor Coordinate Limits

Limit	Value
Cursor Y minimum =	(Number of lines between the start of vertical sync and the end of vertical back porch) -63
Cursor Y maximum =	(Number of lines between the start of vertical sync and the start of vertical front porch) -1

The CXYR is cleared at reset.

4.6.2 Cursor Base-Address Register

Figure 4–61 shows the cursor base-address register (CCBR) format, and Table 4–64 describes its fields.

Figure 4–61 CCBR Format

31 16	15 10	9 4	3 0
RES	Cursor Rows −1	Cursor Base Address	RES

Table 4–64 CCBR Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.
15:10	Cursor Rows −1	RW	Specifies the number of cursor rows to display −1.
9:4	Cursor Base Address	RW	Specifies the value of frame buffer byte-address bits <9:4> that points to the bottom of the cursor array in memory. Note that this corresponds to the top of the displayed cursor.
3:0	RES	RAZ/IGN	Reserved.

The CCBR specifies the frame buffer address of the first visible row of the cursor. The cursor data is arranged in a linear array in the first 1KB of VRAM. Cursor data is stored as consecutive bits, starting with the first 2 bits of the upper left pixel, continuing horizontally across the 64 pixels of the row, and then to the next row down.

Each row of the 64-pixel-wide cursor consumes 16 bytes of memory. The cursor base address (<9:4>) must be aligned to the start of a cursor row. Therefore, the cursor base address is specified with bits <9:4> of the full byte-address (bits <3:0> are all zero).

The cursor rows −1 field (<15:10>) defines the number of rows in the cursor display. It is usually set to 63.

Constraints imposed by certain monitor-timing specifications might make it necessary to specify a cursor base address that does not correspond to the first row of the 64-row cursor, and to specify a cursor rows -1 value other than 63. (See Section 7.2.8 for more information about the cursor display function.)

The CCBR is cleared at reset.

4.7 Status Registers

The status registers return information on the current status of chip processing and pending interrupts. They can be written to enable interrupts and provide a synchronization mechanism for scheduling commands.

4.7.1 Command Status Register

Figure 4–62 shows the command status register (SCSR) format, and Table 4–65 describes its fields.

Figure 4–62 SCSR Format

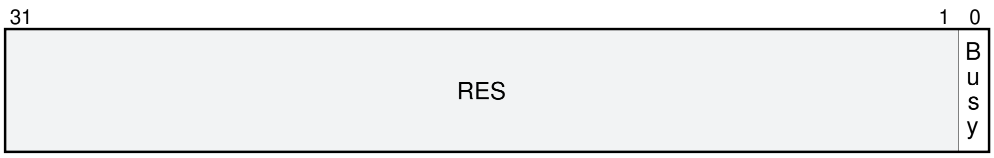


Table 4–65 SCSR Field Description

Bits	Field	Type	Description
31:1	RES	RAZ/IGN	Reserved.
0	Busy	RW	When set, the 21030 is processing commands from the command FIFO; when clear, the 21030 is idle.

When read, the SCSR returns the state of the busy bit (<0>). The busy bit indicates whether the chip is processing commands or has completed all command processing and the command FIFO is empty.

The SCSR and the PCI configuration registers are the only registers that are immediately accessible for read; that is, the command FIFO does not have to be flushed before completing a read of the SCSR.

The 21030 is optimized as a primarily write-only device and implements pipelined processing. In typical graphics operations, the driver can stream writes and commands to the chip without overflowing the command FIFO. The 21030's PCI retry mechanism combined with short command processing times prevents most writes from stalling. Hardware retries any writes that do stall and software polling is unnecessary.

However, in many cases software should poll the busy bit and wait for the 21030 to become idle before continuing. Although the 21030 provides hardware interlocks to ensure coherency for most operations (such as holding a frame buffer read until the write buffer is flushed), waiting for the 21030 to be idle is necessary for unsupported interlocks and to synchronize hardware and software processing. The following situations are examples of when it is practical or necessary to wait for the 21030 to be idle.

Software should wait for the 21030 to be idle:

- As an interlock for not updating the GDAR until a DMA operation is complete
- To avoid unnecessary retries on the PCI bus while long commands complete
- For algorithms that use DMA, to indicate that a DMA operation is complete and main memory can be unlocked

The SCSR acts as a write memory barrier. The 21030 inserts a write to the SCSR into the command FIFO as a flag to ensure that preceding commands and writes are completely processed before subsequent commands and writes are unloaded from the command FIFO. The command parser unloads commands and writes from the command FIFO, performs some initial processing, and then passes graphics processing requests to the pixel pipeline (Section 3.5). The command parser provides a hardware interlock mechanism to ensure that any writes it processes do not affect processing in-progress downstream in the pipeline. The SCSR interlock mechanism is an additional precaution, in case the hardware interlock fails or cannot handle a particular operation.

The SCSR is cleared at reset.

4.7.2 Interrupt Status Register

Figure 4–63 shows the interrupt status register (SISR) format, and Table 4–66 describes its fields.

Figure 4–63 SISR Format

31 21	20	19 18	17	16	15 5	4	3 2	1	0
RES	TIE	RES	SAIE	EOFIE	RES	TIP	RES	SAIP	EOFIP

Table 4–66 SISR Field Description

Bits	Field	Type	Description
31:21	RES	RAZ/IGN	Reserved.
20	TIE	RW	Timer interrupt enable—When set, enables timer interrupts; when clear, the interrupts are disabled.
19:18	RES	RAZ/IGN	Reserved.
17	SAIE	RW	Shift-address interrupt enable—When set, enables shift-address interrupts; when clear, the interrupts are disabled.
16	EOFIE	RW	End-of-frame interrupt enable—When set, enables end-of-frame interrupts; when clear, the interrupts are disabled.
15:5	RES	RAZ/IGN	Reserved.
4	TIP	R/W1C	Timer interrupt pending—When set, indicates that a timer interrupt is pending. Writing a one to this bit clears the interrupt.
3:2	RES	RAZ/IGN	Reserved.
1	SAIP	R/W1C	Shift-address interrupt pending—When set, indicates that a shift-address interrupt is pending. Writing a one to this bit clears the interrupt.
0	EOFIP	R/W1C	End-of-frame interrupt pending—When set, indicates that an end-of-frame interrupt is pending. Writing a one to this bit clears the interrupt.

The SISR enables all supported 21030 interrupts and returns status for all pending interrupts. For each interrupt, the register contains a read-write enable bit and a read-only pending status bit. Each pending status bit and its corresponding interrupt can be cleared by writing a one to the bit.

The 21030 supports three types of interrupts:

- End-of-frame—If enabled, this interrupt is posted at the start of vertical sync.
- Shift address—If enabled, this interrupt is posted immediately after a split read-transfer cycle in which the address matched the interrupt shift-address in the VSAR (Section 4.5.5).
- Timer—If enabled, this interrupt is posted 256 frame buffer clocks after the interrupt is cleared.

The SISR is cleared at reset.

4.8 External Device Registers

The 21030 has several interfaces to the external devices, including an EEPROM interface and glueless interfaces to several Brooktree RAMDACs and a clock generator. The external device registers provide an access window to these external devices. (See Chapter 8 for more information on the hardware interface to the external devices.)

4.8.1 EEPROM Write Register

Figure 4–64 shows the EEPROM write register (ERWR) format, and Table 4–67 describes its fields.

Figure 4–64 ERWR Format

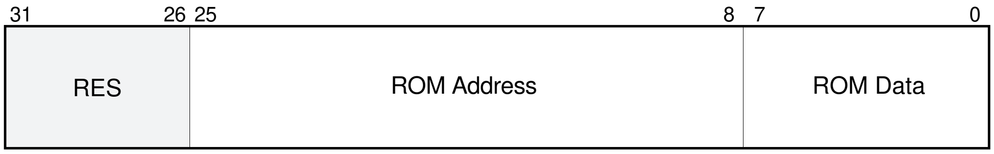


Table 4–67 ERWR Field Description

Bits	Field	Type	Description
31:26	RES	RAZ/IGN	Reserved.
25:8	ROM Address	RW	The EEPROM address to be written.
7:0	ROM Data	RW	The EEPROM write data.

The ERWR provides a mechanism to write an external EEPROM (flash ROM), up to 256K × 8 in size. On a write to the ERWR, the 21030 writes the EEPROM data byte (<7:0>) to the EEPROM address (<25:8>). (See Sections 2.2.3 and 4.2.6 for more information about accessing the EEPROM.)

The ERWR is cleared at reset.

4.8.2 Palette and DAC Setup Register

Figure 4–65 shows the palette and DAC setup register (EPSR) format, and Table 4–68 describes its fields.

Figure 4–65 EPSR Format

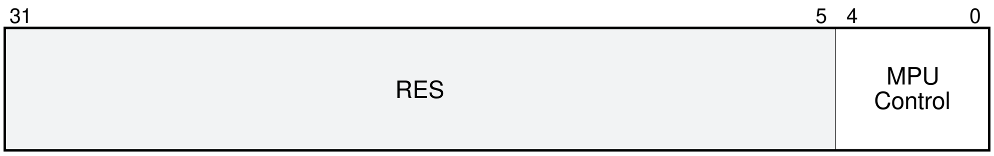


Table 4–68 EPSR Field Description

Bits	Field	Type	Description
31:16	RES	RAZ/IGN	Reserved.
4:0	MPU Control	RW	The control and strobe signals for accessing the RAMDAC. The bits in this field are mapped as shown in Table 4–69.

The EPSR specifies the control bits to be used when accessing the external RAMDAC. The palette and DAC data register (EPDR, Section 4.8.3) writes the data to and reads the data from the RAMDAC. Before accessing the EPDR, the EPSR MPU control field (<4:0>) must be written to select the appropriate data.

The function of the MPU control field depends on the specific RAMDAC being used. The 21030's pin interface is glueless for both PC-class and workstation-class Brooktree RAMDACs. The pin interface to the RAMDACs comprises the **dacce_l<1:0>**, **dacrw**, and **dacc<2:0>** pins. These pins are mapped to the EPSR read-control bits (<4:0>) as shown in Table 4–69.

The Brooktree RAMDACs in the PC-class include the Bt485 and Bt484; and in the workstation-class include the Bt458, Bt459, and Bt463. Two Bt458-class RAMDACs can be used to implement a glueless, dual-headed subsystem.

Table 4–69 shows how the EPSR MPU control field is mapped to the 21030 pins, and how the 21030 pins connect to the Bt485-class and Bt458-class RAMDAC pins.

Table 4–69 EPSR MPU Control Field Mapping

MPU Control		21030	RAMDAC Pin	
Bit	Function	Pin	Bt458	Bt485
4	Control	**dacc2**	C2†	RS3
3	Control	**dacc1**	C1	RS2
2	Control	**dacc0**	C0	RS1
1	Control	**dacrw**	RW	RS0
0 = 1	Strobe	**dacce_l1**	CE* (head 1)	WR*
0 = 0	Strobe	**dacce_l0**	CE* (head 0)	RD*

*Brooktree designation for active-low pins is an asterisk.

†Not required on all Bt458-class RAMDACs.

For example, to read the byte selected by the value RS3:RS0 = 1010 from a Bt458, software can write 00000014_{16} to the EPSR, then read the EPDR.

(See Section 8.3 for more information about the hardware interface.)

The EPSR is cleared at reset.

4.8.3 Palette and DAC Data Register

Figure 4–66 shows the palette and DAC data register (EPDR) read format, and Table 4–70 describes its fields.

Figure 4–66 EPDR Read Format

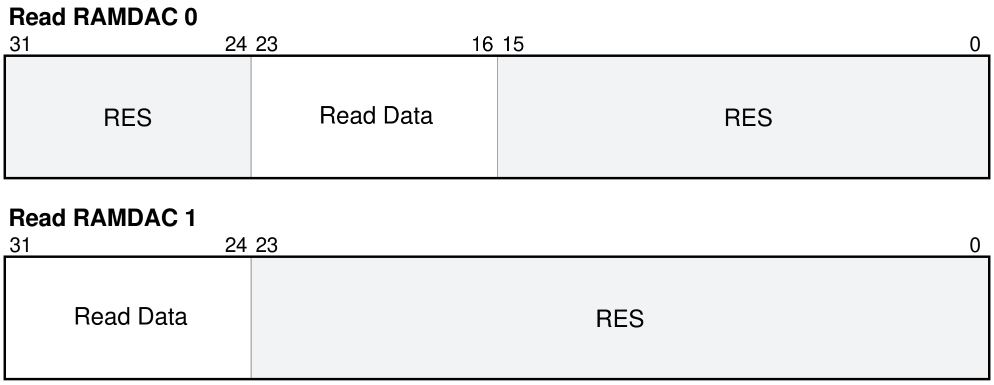


Table 4–70 EPDR Read-Format Field Description

Number of Bits	Field	Type	Description
24	RES	RAZ/IGN	Reserved. The exact position of the 24 reserved bits is application-specific.
8	Read Data	RW	The data byte read from the RAMDAC. The exact position of the read byte is application-specific.

Figure 4–67 shows the EPDR write format, and Table 4–71 describes its fields.

Figure 4–67 EPDR Write Format

Table 4–71 EPDR Write-Format Field Description

Bits	Field	Type	Description
31:8	RES	RAZ/IGN	Reserved.
7:0	Write Data	RW	The data byte to be written to the RAMDAC.

The EPDR provides the read and write window into the external RAMDAC. On a read, software writes the MPU control field (EPSR <4:0>) with the appropriate value for a RAMDAC read, and reads the requested data in the EPDR.

The location of the returned data byte in the EPDR depends on which RAMDAC was read and is application-specific. That is, the application determines which of the **data<31:0>** pins are connected to the RAMDAC's MPU data bus. The format shown in Figure 4–66 is a result of connecting the data bus from RAMDAC 1 to **data<31:24>** and RAMDAC 0 to **data<23:16>**.

On a write, software writes the EPSR MPU control field with the appropriate value for a RAMDAC write and writes the EPDR write data (<7:0>). The 21030 then writes the RAMDAC.

The EPDR is cleared at reset.

4.8.4 Clock Generator Register

Figure 4–68 shows the clock generator register (ECGR) format, and Table 4–72 describes its fields.

Figure 4–68 ECGR Format

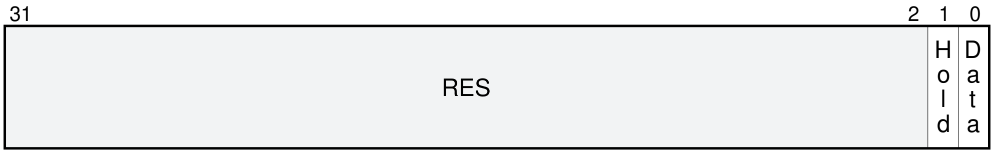


Table 4–72 ECGR Field Description

Bits	Field	Type	Description
31:2	RES	RAZ/IGN	Reserved.
1	Hold	RW	Set on the last of 56 writes to the ECGR; otherwise, clear.
0	Data	RW	The data bit to be written through the serial port to the clock generator.

The write-only ECGR provides a serial interface to an external ICS1562 clock generator chip. The interface consists of one data bit (<0>) and a strobe-control bit (<1>). To program the ICS1562 chip, 56 consecutive data bits must be written to ECGR <0>. The hold bit must be clear the first 55 writes. On the last write, the hold bit must be set, to strobe all of the bits from the clock generator chip's holding register.

Although this interface is designed for the ICS1562 chip, the register and hardware port can also be used to communicate with some other serial device. (Additional external logic might also be required.)

(See Section 8.3 for more information about the clock generator interface.)

The ECGR is cleared at reset.

5
PCI Operations

This chapter describes the PCI functions supported by the 21030. The PCI signals are described in Section 8.4. See the *PCI Local Bus Specification, Revision 2.0* for more information about the PCI bus transactions described in this chapter.

5.1 Configuration Operations

Prior to normal device operation, configuration firmware must write several configuration registers to define the following:

- Device address space
- Associated ROM address space
- Bus access privilege
- Bus ownership duration

To allow system configuration software to access the configuration registers, the 21030 supports the following PCI configuration transactions:

- Configuration write
- Configuration read

When the 21030 detects either transaction, it uses address bits **ad<7:3>** to index into the PCI configuration space header block (Section 4.2). Writes to reserved configuration addresses are ignored, and reads return zeros. The 21030 will not terminate a configuration cycle.

5.2 Target Operations

As a target, the 21030 responds to the following PCI memory transactions:

- Memory read
- Memory write

It responds to any memory read or memory write cycle in which the address falls within the address space defined by the PCI device base-address register (PDBR, Section 4.2.2). Additionally, the 21030 responds to any memory read cycle in which the address falls within the address space defined by the PCI expansion ROM base-address register (PRBR, Section 4.2.6). If the 21030 detects a write to a reserved location in the 21030 address space, it responds to and completes the bus cycle, but ignores the data. Similarly, the 21030 responds to and completes a read transaction of a reserved location, but returns zeros.

The 21030 also responds to the following types of memory transactions, treating them as one of the simpler supported types:

- Memory write and invalidate (operates as memory write)
- Memory read line (operates as memory read)
- Memory read multiple (operates as memory read)

5.2.1 Access Granularity

As a target, the 21030 supports arbitrary, subDword (less than 32-bits) read and write accesses. The 21030 handles all possible permutations of byte masks presented on the **cbe_l<3:0>** pins during both read and write accesses, with the following restrictions:

- Writes to 21030 registers are limited to Dword access. Byte masks are ignored.
- Expansion ROM reads, through the alternate ROM space (Section 2.2.3), return only Dword-aligned data.

(The byte mask restrictions are different in the 21030 step A. See Section A.4.1.)

5.2.2 Transaction Termination

As a target, the 21030 supports arbitrary burst-length, memory write cycles to the 21030 PCI memory space. If the internal command FIFO fills during a burst write, the 21030 disconnects to avoid losing write data.

The 21030 does not support burst memory-read cycles or any burst transactions to PCI configuration space. The 21030 disconnects such transactions after one successful transfer.

The PCI interface loads all writes into the internal 16-entry command FIFO. If the 21030 detects a memory write cycle to its address space and no command FIFO entries are available, it stalls for up to 8 PCI clocks, waiting for an entry to become available. If an entry is still not available, the 21030 issues a target-disconnect termination. The 21030 does not initiate a target-abort termination.

5.3 Master Operation

The 21030 masters the PCI to move image data between display memory and system memory. To support this function, the PCI interface initiates memory read and memory write PCI transactions.

In response to a host read or write request, the 21030 attempts to read or write in bursts of arbitrary length, according to the command it received. The 21030 responds in DMA-read copy or DMA-write copy mode. The specified length can be between 1 and 2048 transfers (that is, burst read between 4 bytes and 8KB and burst write between 8 bytes and 16KB). The 21030 attempts to string together the largest burst possible, but allows the PCI target to regulate the access through its target-disconnect mechanism.

The 21030 monitors the number of transfers remaining to complete the DMA request, making the number of separate burst-transfers transparent to the driver. If the initial attempt to transfer the entire burst length is disconnected, the 21030 attempts to remaster the bus as many times as necessary to complete the request without driver assistance. For example, if a DMA read requests 100 bytes, the 21030 attempts one burst-read of 100 bytes. However, depending on the speed of the target (for example, a bridge to system memory), the transfer could comprise 10 bursts averaging 10 bytes each, or 20 bursts averaging 5 bytes each, and so on.

5.3.1 Transaction Termination

The 21030 supports the PCI-master latency timer mechanism that limits a master's tenure in the presence of other bus requests. The 21030 limits its bus ownership to the number of PCI clocks programmed in the PCI latency timer register (PLTR, Section 4.2.5). The timer is cleared and disabled when the 21030 is not asserting **frame_l**. While **frame_l** is asserted, the timer counts. If the count equals the value in the PLTR and **gnt_l** is deasserted (that is, another agent needs the bus), the 21030 terminates the transaction as soon as the target asserts **trdy_l** for the current data transfer.

When initiating a memory transaction, the 21030 issues a master abort if it does not detect the assertion of **devsel_l** within 6 PCI clocks after it asserts **frame_l**. In such cases, the 21030 terminates the transaction, relinquishes PCI bus ownership, and sets the master-abort bit in the PCI command and status register (PCSR <29>, Section 4.2.1).

Cycles terminated by a target abort are handled similarly. If a target signals target abort, the 21030 immediately terminates the cycle, relinquishes bus ownership and sets the target-abort bit (PCSR <28>).

5.3.2 Aborted DMA Transaction Termination

The 21030 treats an aborted DMA-read copy transaction as a successfully completed transaction, but it sets the appropriate abort-bit status in the PCSR. The 21030 immediately completes all subsequent DMA transfers internally (no PCI activity) until the abort bit is cleared.

An aborted DMA-write copy operation is handled differently. If the target aborts a DMA-write copy operation, the 21030 relinquishes the bus, sets the target-abort bit (PCSR <28>), and begins to flush its internal DMA write FIFO. It does not begin another DMA-write copy operation until the target-abort bit is cleared.

As a master, the 21030 supports all types of target-initiated terminations defined by the *PCI Local Bus Specification, Revision 2.0*.

5.4 Parity

The 21030 generates and drives parity on the **par** pin. However, the 21030 does not support parity-error checking and notification, because it is not required to check for parity errors.

As a master, the 21030:

- Generates parity across 36 bits (**ad<31:0>** and **cbe_l<3:0>**) for all address and write-data cycles
- Ignores parity received on **par** during read-data cycles

As a target, the 21030:

- Generates parity for all read-data cycles
- Ignores parity received on **par** during address and write-data cycles

5.5 Address and Data Stepping

The 21030 does not drive the **ad<31:0>** bus in 1 PCI clock tick. The 21030 requires 2 PCI clock cycles to assert signals across this bus during the data phase of target reads, the address phase of master reads and writes, and the data phase of master writes. The 21030 asserts **frame_l**, **irdy_l**, and **trdy_l** to validate the assertions. The bus-stepping bit in the PCI command and status register (PCSR <7>, Section 4.2.1) is set to indicate this behavior.

5.6 Bus Parking

The 21030 supports PCI bus parking. The central PCI arbitration resource can select the 21030 to actively drive much of the PCI bus to a known state while the bus is idle, to prevent the bus from floating. When the arbiter asserts the **gnt_l** input, the 21030 drives pins **ad<31:0>**, **cbe_l<3:0>**, and, at least 1 clock later, **par**, to an arbitrary state. The 21030 can enable these drivers over several PCI clocks. When **gnt_l** is deasserted, the 21030 tristates **ad<31:0>** and **cbe_l<3:0>** on the next clock, and tristates **par** 1 clock later.

5.7 Functions Not Supported

The 21030 does not support and ignores special cycle and interrupt acknowledge PCI transactions. As a target, the 21030 does not support exclusive accesses (LOCK cycles) for any of its registers or for display memory. The 21030 does not request exclusive access as a master.

6 Graphics Operations

This chapter describes the 21030 general graphics functions and specific graphics modes.

6.1 Overview

The accelerated graphics operations are specified by mode and initiated by a write to either of the following:

- The frame buffer address space (standard)
- Any graphics command register (alternative)

6.1.1 Frame Buffer Writes

Writing to the frame buffer address space is the standard way to invoke a graphics function. In general, the 21030 responds to and interprets write data according to the mode specified in the mode register (GMOR, Section 4.4.1). When the 21030 detects a write to its frame buffer address space, it starts the mode-specified graphics operation at the specified address using control parameters passed in the write data, and possibly, one or more graphics control registers.

Table 6–1 describes the graphics functions that can be invoked in each mode on a write to the frame buffer. (Table 4–17 lists all the modes.)

Table 6–1 Mode-Dependent Frame Buffer Write Operations

Mode	Action Initiated on Frame Buffer Write
Simple	Write pixels.
Simple-Z	Conditionally write pixels based on Z-buffer comparison.
Transparent stipple	Draw masked, monotone 32-pixel spans.

(continued on next page)

Table 6–1 (Cont.) Mode-Dependent Frame Buffer Write Operations

Mode	Action Initiated on Frame Buffer Write
Opaque stipple	Draw masked, bitonal 32-pixel spans.
Block stipple	Draw masked, patterned * 32-pixel spans.
Block fill	Draw unmasked, solid or patterned* spans up to 2K pixels.
Opaque fill	Fill bitonal span up to 2K pixels.
Transparent fill	Fill solid span up to 2K pixels.
Transparent line	Draw masked (styled), monotone 16-pixel lines.
Opaque line	Draw masked (styled), bitonal 16-pixel lines.
Copy	Fill the copy buffer with a 32-pixel, masked, 8-bpp span or a 16-pixel, masked, 32-bpp span; or, empty the copy buffer to a 32-pixel, masked, 8-bpp span or a 16-pixel, masked, 32-bpp span.†
DMA-write copy	Transfer an unaligned, edge-masked span up to 16KB from display memory to PCI addressable memory.
DMA-read copy	Transfer an unaligned, edge-masked span up to 8KB from PCI addressable memory to display memory.

*No raster operation. Some additional restrictions apply. See the specific mode description in Sections 6.2.1 through 6.2.14.

†Whether the copy buffer is filled or emptied depends on the state of the copy hardware.

In several graphics modes, writing to the frame buffer to initiate an operation does not take full advantage of the 21030's speed or range. For example, a write to the frame buffer in copy mode uses only half of the 64-byte on-chip copy buffer for an 8-bpp span. For another example, the 21030 memory interface supports very fast line-drawing rates, but writing to the frame buffer in line mode burdens the CPU with processing Bresenham-style setup code.

6.1.2 Graphics Command Register Writes

For better performance, the graphics command registers can be used to initiate graphics operations. They give software a faster and simpler way to invoke graphics operations.

Similar to writing directly to the frame buffer, writing to a graphics command register invokes a mode-dependent graphics operation, but the frame-buffer address is provided in a register rather than on the write. Writes to graphics command registers cannot invoke all mode operations, but can and should be used to generate the graphics functions listed in Table 6–2.

Table 6–2 describes the graphics operations that can be initiated by writing to a graphics command register.

Table 6–2 Graphics Command Register Write Operations

Register	Mode	Action Initiated on Register Write
Slope<7:0> (GSLR<7:0>) Span width (GSWR)	Line *	Initializes the Bresenham engine and then draws a mode-dependent 16-pixel 2D line (Table 6–1) or a 3D span or line (Table 6–3).
Slope-no-go<7:0> (GSNR<7:0>)	Line*	Initializes the Bresenham engine.†
Continue (GCTR)	Line*	Continues the current line another 16 pixels.
	Other	In any mode other than a line mode, initiates an operation based on the specified mode (Table 6–1), conditionally using the address from the GADR.
Copy 64 source (GCSR)	Copy	Fills up to 64 bytes of the copy buffer from the specified frame buffer address.
Copy 64 destination (GCDR)	Copy	Empties up to 64 bytes from the copy buffer to the specified frame buffer address.

*Any line mode.

†The GSNRs are included because they initialize the Bresenham engine, but they do *not* initiate line drawing and are not graphics command registers.

Additionally, several graphics command registers provide exclusive access to the 3D line and span drawing functions. These functions cannot be invoked by direct frame-buffer writes. The GSLRs, GSWR, and GCTR are used to invoke 3D line drawing operations, which can be programmed to be Z-buffered, color-interpolated, masked, and dithered. The GSWR is an alias for GSLR0 with the slope set to zero to invoke a span rather than an arbitrary line.

Table 6–3 lists the functions generated on a write to these registers in the supported 3D line and span modes.

Table 6–3 Graphics Command Register Write Operations in 3D Line Modes

Line Mode	Action Initiated on Register Write*
Color-interpolated, nondithered	Draw masked, depth-cued (shaded), 16-pixel line (span) without dithering.

*To GSLRs, GSWR, and potentially, GCTR

(continued on next page)

Table 6–3 (Cont.) Graphics Command Register Write Operations in 3D Line Modes

Line Mode	Action Initiated on Register Write*
Color-interpolated, dithered	Draw masked, depth-cued (shaded), 16-pixel line (span) with dithering.
Sequential-interpolated	Draw masked, grey-scale-shaded, 16-pixel line (span).†
Z-buffered	Draw masked, monotone, 16-pixel, Z-buffered line or span.
Z-buffered, color-interpolated, nondithered	Draw masked, Z-buffered, depth-cued (shaded), 16-pixel line (span) without dithering.
Z-buffered, color-interpolated, dithered	Draw masked, Z-buffered, depth-cued (shaded), 16-pixel line (span) with dithering.
Z-buffered, sequential-interpolated	Draw masked, Z-buffered, grey-scale-shaded, 16-pixel line (span).†

*To GSLRs, GSWR, and potentially, GCTR

†8-bpp only

6.1.3 Invoking Graphics Operations

To invoke a graphics function in any supported mode, the basic sequence is as follows:

1. Set the mode for the desired operation.
2. Write the required mode-specific parameters to the appropriate graphics control registers.
3. Initiate the operation with a write to the frame buffer or to a graphics command register.

This sequence of writes is grouped as one command packet. Each packet typically contains none to several control parameters, followed by the operation that initiates the write. Software streams command packets to the 21030 where they are stored in the 16-entry command FIFO. The 21030 unloads the packets from the FIFO one at a time, and executes them as specified by the mode.

The order of the control parameters is usually not important, but they all must be written before the final write that initiates the operation. All 21030 drivers must maintain this level of ordering. In particular, Alpha AXP drivers present special problems because the CPU write buffer does not enforce write ordering. (See Section 7.3.1 for more information about 21030 support for the write buffer in Alpha AXP CPUs.)

The 21030 uses a different set of control parameters for each operating mode. The parameters are provided by the graphics control registers and also by the data that the operation-initiating write passes to the frame buffer or to the graphics command registers.

In each mode, the 21030 can operate on a variety of on-screen and off-screen visual bitmaps; and in the 3D line modes, on off-screen Z-buffers and stencil buffers. Supported bitmaps include an 8-bpp format in an 8-bpp frame buffer, and packed 8-bpp, unpacked 8-bpp, 12-bpp, and 24-bpp formats in a 32-bpp frame buffer. (See Section 6.1.5 for more information about the supported bitmap and buffer formats, and the mechanisms for handling them.)

6.1.4 Register Load Synchronization

In general, software can write the frame buffer, any graphics control register, or any graphics command register, without regard to the internal state of the chip. The order of the writes within each command packet is important to the extent that all control registers must be set before the frame buffer or graphics command registers are written to initiate the graphics operation. However, in all but a few cases, software need not send register data or command packets in synchronism with the previous operation's completion.

The 21030 does not schedule a command packet for execution until the previous command has finished executing. Most of the graphics registers are double-buffered, such that, while one set is being loaded from a command packet, the other set can be used for graphics processing without interference. Therefore, software can usually issue register and packet writes indefinitely, without polling the state of the 21030 graphics processing hardware or registers. The following exceptions require the chip to be idle (that is, processing complete with the command buffer empty) before a write can occur:

- Write to the deep register (GDER) in any mode
- Write to the data register (GDAR) in DMA-write copy mode

For the few cases where it is required, register-load synchronization can be done in either of the following ways:

- Software can poll the busy bit in the command status register (SCSR <0>, Section 4.7.1) and write the register only when the value of busy is zero.

- Software can insert a synchronization barrier into the command stream. A write to the SCSR effectively causes the 21030 to wait for the busy bit to go to logical zero. A write to the SCSR goes into the command buffer along with all other writes. But when the SCSR write is removed from the command buffer for processing, the operation stalls until all previous graphics processing is completed. For example, before writing the GDAR in DMA-write copy mode, software can first write the SCSR and then write the GDAR, rather than polling the busy bit and waiting for the chip to become idle.

6.1.5 Visual Bitmap and Buffer Formats

The graphics functions support three types of frame buffer objects:

- Visual bitmaps
- Z-buffers
- Stencil buffers

The formats of all supported frame-buffer objects are described in the context of the frame buffer in which they can exist. The supported visual bitmaps can have various depths and organizations depending on whether they exist in an 8-bpp or a 32-bpp frame buffer. The supported Z-buffer and stencil-buffer objects exist only within the domain of 32-bpp frame buffers, and have two specific formats (Section 6.1.5.3).

6.1.5.1 8-bpp Frame Buffer

For an 8-bpp frame buffer, the value of the deep bit in the deep register (GDER <0>, Section 4.4.28) must be zero. A packed 8-bpp bitmap (PB8) is the only supported type of frame-buffer object in an 8-bpp frame buffer; stencil and Z-buffers are not supported.

With an 8-bpp frame buffer, the entire screen can be viewed as one large 8-bpp bitmap. The pixel values (typically color map indices) are byte-packed such that each Dword in the frame buffer contains four adjacent pixel values.

Figure 6–1 shows the format of a packed 8-bpp bitmap.

Figure 6–1 Packed 8-bpp Bitmap

PB8

31 24	23 16	15 8	7 0
Pixel 3	Pixel 2	Pixel 1	Pixel 0

6.1.5.2 32-bpp Frame Buffer

For a 32-bpp frame buffer, the value of the deep bit (GDER <0>, Section 4.4.28) must be one. A 32-bpp frame buffer supports the full range of visual bitmaps (Figures 6–1 through 6–4) as well as Z-buffers and stencil buffers.

Table 6–4 lists the bitmap formats supported in 32-bpp frame buffers.

Table 6–4 32-bpp Frame Buffer Supported Bitmaps

Mnemonic	Description	Figure
PB8	Packed 8-bpp bitmap	6–1
$UB8_0$ $UB8_1$ $UB8_2$	Unpacked 8-bpp bitmaps	6–2
$DC12_0$ $DC12_1$	12-bpp direct-color bitmaps	6–3
TC24	24-bpp true-color bitmap	6–4
Z24	24-bpp Z-buffer/8-bpp stencil	6–5
Z16	16-bpp Z-buffer/up to 8-bpp stencil	6–6

The 32-bpp frame buffer supports only 8-, 12-, and 24-bit bitmap depths. In a 32-bpp frame buffer, 1 Dword in the frame buffer corresponds to 1 pixel location, except in the PB8 bitmap. As a result, the 21030 can pack three 8-bit pixels or two 12-bit pixels into a Dword allocated for a pixel (1 byte is reserved for the tag field).

In addition to the packed 8-bpp and true-color 24-bpp bitmaps, three unpacked 8-bpp bitmap formats and two 12-bpp bitmap formats are supported in 32-bpp frame buffers.

Figure 6–2 8-bpp Unpacked Bitmap Formats

$UB8_0$

31–24	23–8	7–0
Tag	RES	Pixel

$UB8_1$

31–24	23–16	15–8	7–0
Tag	RES	Pixel	RES

$UB8_2$

31–24	23–16	15–0
Tag	Pixel	RES

Figures 6–1 through 6–4 show the content of a pixel Dword for all the possible bitmap formats. The tag field is common to all and is application-dependent.

Figure 6–3 12-bpp Bitmap Formats

$DC12_0$

31–24	23–20	19–16	15–12	11–8	7–4	3–0
Tag	RES	Red	RES	Green	RES	Blue

$DC12_1$

31–24	23–20	19–16	15–12	11–8	7–4	3–0
Tag	Red	RES	Green	RES	Blue	RES

Figure 6–4 24-bpp True-Color Bitmap Format

TC24

31–24	23–16	15–8	7–0
Tag	Red	Green	Blue

The application-specific tag field can also be used to store such things as overlay or window-ID information, on a per-pixel basis. It can facilitate in-place, double-buffered or triple-buffered, 8- and 12-bpp bitmaps in a 32-bpp frame buffer.

The color content in each 8-bpp and 12-bpp format aligns to mutually exclusive fields within the Dword. While stored in the frame buffer, the multiple bitmaps are merged, such that each Dword can contain 1 byte from each of three 8-bpp bitmaps, or three nibbles from each of two 12-bpp bitmaps. In effect, the reserved fields of one bitmap are aliases for the valid color fields in the other bitmaps.

Because multiple pixels can be packed into 1 Dword, operating on 8-bpp or 12-bpp bitmaps in a 32-bpp frame buffer requires that the desired source be specified and the desired destination be isolated, without corrupting another bitmap within the Dword. For example, to copy from bitmap $UB8_0$ to bitmap $UB8_2$ (Figure 6–2), the low byte must be specified as the source and written to the upper byte, without writing either bitmap $UB8_0$ or bitmap $UB8_1$. The 21030 can be programmed to isolate the appropriate source and destination bitmaps for each graphics operation (Section 6.1.6).

The supported bitmap formats depends on the RAMDAC, and all bitmap formats cannot be used in all applications. For example, a frame buffer that uses a Bt489 can display packed 8-bpp and 24-bpp formats, but cannot handle unpacked formats. Conversely, a Bt463 can handle unpacked formats, but cannot display the packed 8-bpp format.

The unpacked bitmap formats supported by the 21030 are compatible with the Bt463 RAMDAC, which supports 8-bit pseudo-color, 12-bit direct-color, and 24-bit true-color visual types. The Bt463 allows the color fields to be located at different bit positions within each pixel's Dword as a function of the window ID.

In a 21030 subsystem, bitmaps can be specified on a per-window basis by using the tag field to specify different window IDs. This allows in-place double- or triple-buffering without copying the displayed bitmaps, by specifying a different 8-bpp or 12-bpp bitmap in alternating frames. (See Chapter 7 for more information about using the 21030 and Bt463 to display the different bitmap formats and do in-place double-buffering.)

6.1.5.3 Z-Buffer and Stencil-Buffer Formats

Hardware support for Z and stencil buffers is limited to 32-bpp frame buffers. The 21030 supports two formats for the Z and stencil buffers: Z24 and Z16. The Z16 bit in the mode register (GMOR <14>, Section 4.4.1) specifies the format.

The Z24 format provides full, 24-bit Z resolution and is desirable for 3D image quality. Full-screen, Z-buffering with the Z24 format requires an additional 4MB or 8MB of physical memory, depending on the screen resolution. In the Z24 format, the 24-bit Z-buffer data and 8-bit stencil-buffer data are stored packed in one Dword (Figure 6–5). In Z-buffering modes, the 21030 expects to find stencil and Z information in this format and updates it accordingly.

Figure 6–5 shows the Z24 format for Z and stencil buffers.

Figure 6–5 Z24 Z and Stencil Buffer Format

31 24	23 0
Stencil Stored	Z Stored

Alternatively, the Z16 format limits the resolution of the Z-values to 16 bits; but unlike the Z24 format, it allows full-screen, Z-buffering of an 8-bpp image in a 32-bpp frame buffer with only 4MB (1024 $\times$ 768) or 8MB (1280 $\times$ 1024 or 1600 $\times$ 1280) of physical memory. The Z16 format is limited to use in 8-bpp bitmaps in a 32-bpp frame buffer.

Figure 6–6 shows the Z16 format for Z and stencil buffers.

Figure 6–6 Z16 Z and Stencil Buffer Format

31 24	23 8	7 0
Stencil Stored	Z Stored	Pixel Value

A separate destination bitmap is used when drawing with a Z24-format buffer, but not with a Z16-format buffer. The Z16 format is special in that both the Z-value and pixel value are packed in the Z-buffer Dword. In effect, the Z16 format is a variant of the $UB8_0$ format. When drawing with the Z16 format, software must specify the destination-bitmap field value as 01_2 and the destination-byte field value as 00_2.

6.1.6 Source and Destination Operands

The 21030 references a source and a destination operand for every graphics operation.

Table 6–5 shows the specific source and destination operands according to mode.

Table 6–5 Source and Destination Operands According to Mode

Mode	Source	Source Byte	Destination	Destination Byte
Simple Simple-Z	PCI write data	No	Frame buffer bitmap	Yes
Opaque stipple Opaque line Transparent stipple Transparent line	GFGR or GBGR	No	Frame buffer bitmap	Yes
Block stipple Block fill	GBCR<7:0>	No	Frame buffer bitmap	Yes
All interpolated line modes	Interpolation engines	No	Frame buffer bitmap	Yes
Copy	Frame buffer bitmap	Yes	Frame buffer bitmap	Yes
DMA-read copy	PCI memory bitmap	No	Frame buffer bitmap	Yes
DMA-write copy	Frame buffer bitmap	Yes	PCI memory bitmap	No

In most cases, the source and destination operands are simply pixel values that are read from or written to a bitmap. For example, a copy mode operation reads a pixel value from a source bitmap and writes it to a destination bitmap. In general, the graphics operations support source and destination bitmaps in any format described in Section 6.1.5, with the following restrictions:

- The source and destination bitmaps must have an equal number of bits-per-pixel.
- Every mode does not support every format.

For example, an operation that references a format $UB8_2$ source bitmap can reference a format PB8 destination bitmap, but not a format TC24 bitmap.

Table 6–6 lists the bitmap formats not supported by a particular mode.

Table 6–6 Unsupported Bitmap Formats According to Mode

Mode	Unsupported Source Format	Unsupported Destination Format
DMA-write copy	None	All unpacked 8-bpp bitmaps
DMA-read copy	All unpacked 8-bpp bitmaps	None

(continued on next page)

Table 6–6 (Cont.) Unsupported Bitmap Formats According to Mode

Mode	Unsupported Source Format	Unsupported Destination Format
Block stipple Block fill	None	PB8 (only in 32-bpp frame buffers) *

*The block modes can be used to fill packed 8-bpp bitmaps in a 32-bpp frame buffer, but not by setting the destination to PB8. Software must simulate PB8 support by setting the destination bitmap to TC24 and setting the block color registers accordingly.

The programmable graphics control register fields listed in Table 6–7 allow software to independently specify the source and destination bitmaps, within the previously stated limitations.

Table 6–7 Source, Destination, and Plane Mask Fields

Field	Register	Mnemonic	Bits	Description
Source Bitmap	Mode	GMOR	10:8	Section 4.4.1
Source Byte	Mode	GMOR	12:11	Section 4.4.1
Destination Bitmap	Raster operation	GOPR	9:8	Section 4.4.3
Destination Byte	Raster operation	GOPR	11:10	Section 4.4.3
Plane Mask	Plane mask	GPMR	31:0	Section 4.4.20

The source and destination bitmap fields determine whether the respective bitmaps are 8-bpp packed, 8-bpp unpacked, 12-bpp, or 32-bpp. For an 8-bpp source or destination in a 32-bpp frame buffer, the source bitmap and source byte fields specify the specific format (PB8, $UB8_0$, $UB8_1$, and so on). For 12-bpp destination bitmaps, the plane mask must be used on writes to isolate the desired bitmap ($DC12_0$ or $DC12_1$) from the unwanted bitmap.

6.1.6.1 Address Alignment Requirements

Each graphics mode requires the address of a source or destination bitmap operand to be aligned to some number of bytes or pixels. The number varies as a function of the mode and bitmap type. (Sections 6.2.1 through 6.2.14 describe the alignment required for all supported bitmap formats in each mode.)

6.1.6.2 24-bpp Bitmap Operands

The TC24 format is the *only* 21030-supported 24-bpp bitmap format. To specify TC24 source and destination bitmaps, the source and destination bitmap fields (Table 6–7) should be set to 011_2 and 11_2, respectively. The source and destination byte fields are ignored when manipulating 24-bpp bitmaps.

6.1.6.3 12-bpp Bitmap Operands

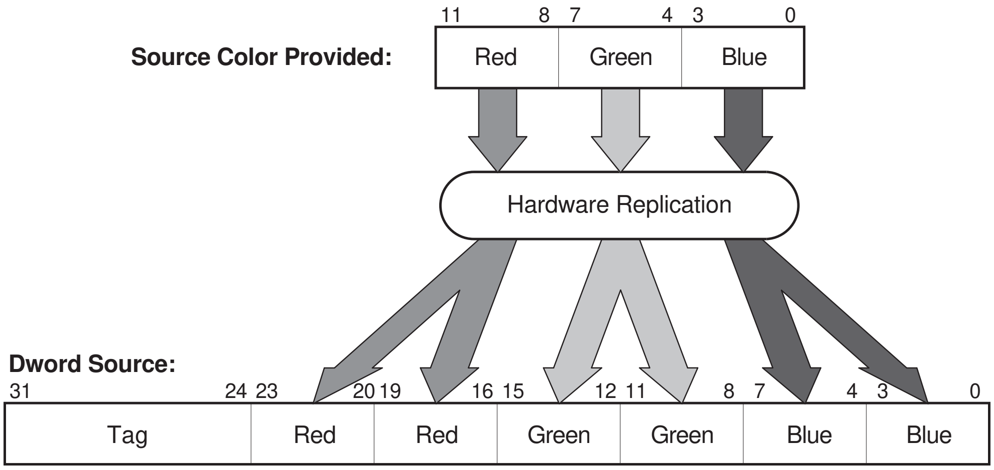
When manipulating 12-bpp source and destination bitmaps, software must specify the appropriate codes for the source and destination bitmap fields (Table 6–7), and ensure the following:

- The source data is aligned to match the format of the destination bitmap.
- The plane mask (GPMR) is set to update only the desired bitmap.

For the modes in which the PCI write data or a register (for example, foreground, background, or block color register) provides the source data, software must explicitly align the source data to the destination bitmap format (Figure 6–3). In the simplest case, to align to format $DC12_0$ or $DC12_1$, data can be replicated across the Dword.

In color-interpolated line, copy, DMA-read copy, and DMA-write copy modes, the 21030 provides the source data. To allow the source data to be mapped to one of the 12-bpp format bitmaps (Figure 6–3), the 21030 duplicates the source bitmap data (either $DC12_0$ or $DC12_1$, as specified by the source bitmap field) into both the $DC12_0$ and the $DC12_1$ format, as shown in Figure 6–7.

Figure 6–7 Hardware Replication of 12-bpp Source Bitmap to Destination Dword



In either case, when writing to a 12-bpp bitmap destination, software must ensure that the plane mask (Table 6–7) is set to update only the desired bitmaps. For example, to update only the $DC12_0$ bitmap color data, software must set the plane mask to 000F0F0F; and to update the $DC12_1$ bitmap color data, the plane mask must be 00F0F0F0. The source and destination byte fields are ignored for 12-bpp operations.

6.1.6.4 8-bpp Bitmap Operands

The 21030's flexible byte-access into 32-bpp frame buffers allows packed and unpacked 8-bpp bitmap formats to be manipulated almost transparently to the software. Software can set independent values in the source bitmap and byte fields and the destination bitmap and byte fields (Table 6–7), to specify any combination of packed and unpacked 8-bpp bitmaps as the source and destination of a graphics operation. Independent-byte access into each bank of memory and the unique layout of the bitmaps in the frame buffer provides this capability.

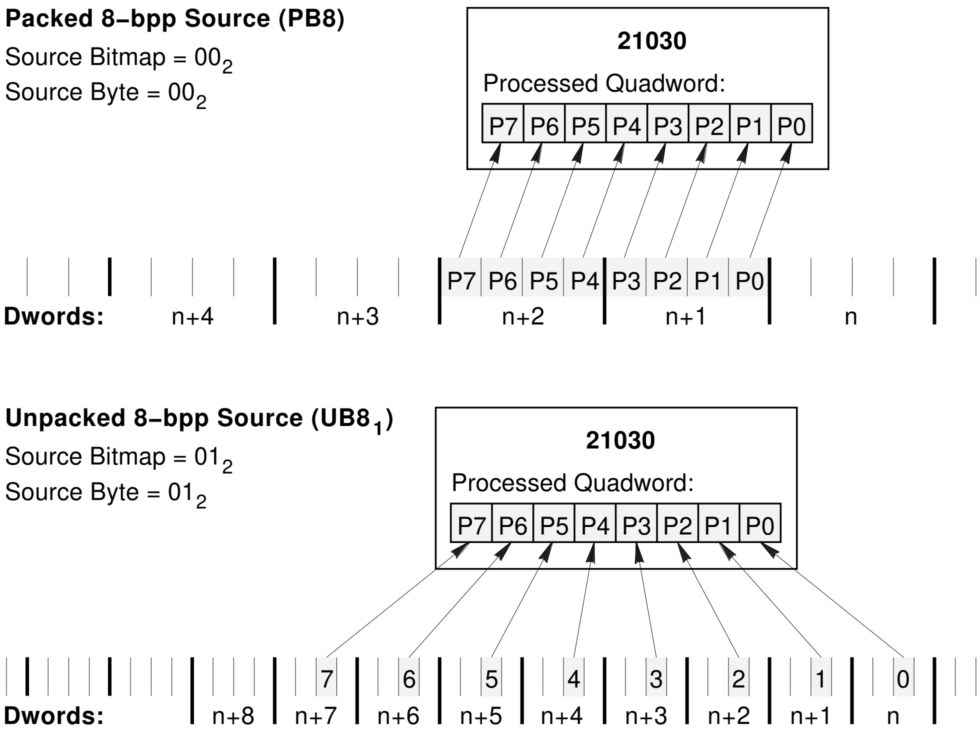
Each time the 21030 accesses the frame buffer through its 64-bit memory port, it can retrieve eight packed, or eight unpacked, 8-bit pixels, according to the source bitmap value specified in the GMOR. In the simple case of a packed source, the 21030 reads 1 contiguous quadword (64 bits) from the frame buffer, returning 8 packed pixels to the 21030 for processing (Figure 6–8). On the other hand, each Dword corresponds to 1 pixel in an unpacked source, and each byte within the Dword corresponds to a different 8-bpp bitmap (Figure 6–2).

When reading from an unpacked 8-bpp source, the 21030 effectively reads 1 byte from each of 8 pixel Dwords (Figure 6–8). The 8 bytes are packed as they are read into the chip, such that the source data inside the chip appears as if it was read from a packed 8-pixel quadword. Internal to the chip, all 8-bpp processing is handled in a packed format, regardless of the type of source and destination bitmaps stored in the frame buffer.

Similarly, the 21030 can write 8 pixels at a time to a packed or unpacked destination, as specified by the destination bitmap value in the GOPR. On a write to an unpacked destination, the 8 internally packed pixels are effectively unpacked, 1 pixel in each of 8 consecutive Dwords.

Figure 6–8 shows how the 21030 accesses 8-bpp packed (PB8) and unpacked ($UB8_1$) bitmaps as a function of the source bitmap and byte fields. If the direction of the arrows was reversed and destination was substituted for source, the figure would show how the 21030 accesses the same 8-bpp bitmaps on a write to destination.

Figure 6–8 8-bpp Bitmap Access in 32-bpp Frame Buffers



When the source and destination field values specify an unpacked 8-bpp bitmap, the source byte field specifies the byte read from each Dword and the destination byte field value specifies the byte written to each Dword. In other words, when the source or destination bitmap field specifies an unpacked 8-bpp source or destination, the source byte field specifies the bitmap format as either $UB8_0$, $UB8_1$, or $UB8_2$. For example, if all the following are true:

- The source bitmap field specifies an 8-bpp unpacked bitmap.
- The source byte field value is 10_2.
- The 21030 reads the source as part of a copy operation.

then the 21030 reads the third bytes from 8 consecutive Dwords and packs them into 1 quadword for on-chip processing; in other words, the 21030 reads 8 pixels from bitmap $UB8_2$.

The inverse is true for writing to an unpacked 8-bpp destination. When the destination bitmap field specifies an unpacked 8-bpp destination, the destination byte field effectively selects the byte from the packed on-chip quadword that is written to each of 8 consecutive Dwords. For example, if the destination byte value is 01_2, the 21030 writes one byte to the second byte in each of 8 consecutive Dwords; in other words, it writes 8 pixels to bitmap $UB8_1$.

When the source and destination bitmap fields specify other than an unpacked 8-bpp bitmap, the source and destination byte fields are ignored.

Table 6–8 summarizes the source and destination bitmap and byte field values for specifying 8-bpp source and destination bitmaps in a 32-bpp frame buffer (Table 6–7 points to the field descriptions).

Table 6–8 8-bpp Source and Destination Bitmap and Byte Field Values

Bitmap Format	Source Bitmap GMOR<10:8>	Source Byte GMOR<12:11>	Destination Bitmap GOPR<9:8>	Destination Byte GOPR<11:10>
PB8	000	00	00	00
$UB8_0$	001	00	01	00
$UB8_1$	001	01	01	01
$UB8_2$	001	10	01	10

Additionally, if the respective source or destination byte value is 11_2, all 8 consecutive tag fields from an unpacked bitmap can either be read as a source or written as a destination.

The hardware source-byte function affects only operations for which the source bitmap is resident in the frame buffer (as in copy and DMA-write copy modes). Otherwise, depending on the mode, the source is specified by the 21030, specified explicitly by software, or fetched over the PCI bus.

Similarly, the hardware destination-byte function affects operations for which the destination bitmap is resident in the frame buffer. This includes operations in all modes except the DMA-write copy mode (which does not support drawing to unpacked 8-bpp destination bitmaps, and therefore, does not require a destination byte function).

Table 6–5 shows the modes that support the source and destination byte functions. The table assumes that the corresponding source and destination bitmaps are unpacked 8-bpp bitmaps.

In the dithered, interpolated-line, and sequential-interpolated modes, the on-chip interpolation engines generate the source pixel data. The 21030 directs the 8-bit result to the correct byte according to the destination alignment specified in the destination-byte field.

In general, reading an unpacked 8-bpp source or writing an unpacked 8-bpp destination requires a more rigid address alignment than packed 8-bpp formats, depending on the mode. (Sections 6.2.1 through 6.2.14 describe each mode and the specific addressing requirements for applicable source and destination bitmaps.)

See Section 8.1 for more information about the frame buffer organization.

6.2 Graphics Modes

Sections 6.2.1 through 6.2.14 describe the graphics modes. Each section describes the mode's invocation, required parameter sets, and functional operation. The descriptions include the standard invocation mechanism (directly writing the frame buffer), and for applicable modes, the alternate graphics command register mechanism.

Note

The functional-algorithm pseudo-code examples in the following sections are for descriptive purposes and do not describe the exact logic implementation.

6.2.1 Simple Mode

In the simple mode, a PCI write to the frame-buffer address space writes 4 independently masked bytes of data to the frame buffer at the Dword-aligned write address. The 21030 performs the write as a function of the parameters listed in Table 6–9.

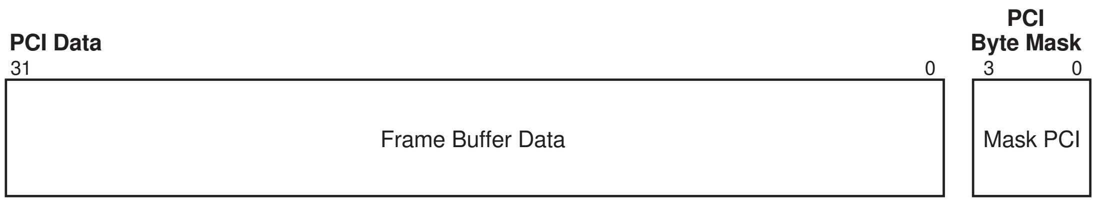
```
Simple Write (Frame Buffer Address, Frame Buffer Data, Mask PCI, Plane Mask,
              Mask GPXR, Raster Op, Destination Bitmap, Destination Byte);
```

Table 6–9 Simple Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Frame Buffer Data`	PCI write data	<31:0>	—
`Mask PCI`	PCI data byte mask	<3:0>	—
`Mask GPXR`	Pixel mask register	GPXR <31:0>	4.4.21
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

Figure 6–9 shows the PCI write data format for the `Frame Buffer Data` and `Mask PCI`.

Figure 6–9 Simple Mode PCI Write-Data Format



In the simple mode, the 21030 acts as a generic 32-bit memory controller with the following exceptions:

- The GPXR specifies a byte mask (`Mask GPXR`) to be used in addition to the mask passed over PCI (`Mask PCI`).
- The `Raster Op` programmed in the GOPR is applied.

- The Plane Mask specified in the GPMR is applied.

For every write in the simple mode, the 21030 ANDs Mask PCI and Mask GPXR to generate the final byte mask that determines whether to write each byte. As specified by the Raster Op, the write conditionally combines the Frame Buffer Data and the data stored at the Frame Buffer Address. Only the bit-planes that are enabled by the Plane Mask are written.

The Destination Bitmap and Destination Byte parameters allow access to all types of destination bitmaps (Section 6.1.5). The Frame Buffer Address must be aligned to 4 bytes (Dword-aligned) for all destinations except unpacked 8-bpp bitmaps, which must be aligned to 16 bytes. (The Source Bitmap and Source Byte parameters are not used in the simple mode.)

The following pseudo-code represents the basic algorithm for the simple mode:

```
Write Mask = Mask PCI & Mask GPXR;
Write Pixel (Frame Buffer Address, Frame Buffer Data, Raster Op, Plane Mask,
             Write Mask, Destination Bitmap, Destination Byte);
```

The 21030 always uses the GPXR to specify which Dword bytes are to be written, but software does not always write the GPXR. Because hardware resets the GPXR to FFFFFFF (all bytes unmasked) after every operation, software must write the GPXR only if a different value is required. Additionally, the 21030 always performs the Raster Op specified in the GOPR when writing the frame buffer data (the Raster Op retains its value from operation to operation).

The simple mode can be used to write arbitrary data in the frame buffer. When writing visual, Z, or stencil data to the frame buffer, the Frame Buffer Data must adhere to the 21030-supported formats described in Section 6.1.5.

6.2.2 Simple-Z Mode

In the simple-Z mode, a PCI write to the frame buffer address space writes a Z-buffered set of 4 independently masked bytes of data to the frame buffer at the Dword-aligned address. The 21030 performs the Z-buffered write as a function of the parameters listed in Table 6–10.

```
Simple Z Write (Frame Buffer Address, Frame Buffer Data, Z Address,
               Z Reference, Stencil Reference, Mask PCI, Mask GPXR,
               Plane Mask, Raster Op, Z Test, Z Update, Stencil Read Mask,
               Stencil Write Mask, D Pass, D Fail, S Fail, Z16,
               Destination Bitmap, Destination Byte);
```

Table 6–10 Simple-Z Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Frame Buffer Data`	PCI write data	<31:0>	—
`Mask PCI`	PCI write data byte mask	<3:0>	—
`Z Address`	Z-base-address register	GZBR <23:0>	4.4.15
`Z Reference`	Z-value register	GZVR-H <31:0> GZVR-L <3:0>	4.4.16 4.4.16
`Stencil Reference`	Z-value register	GZVR-L <31:24>	4.4.16
`Mask GPXR`	Pixel mask register	GPXR <31:0>	4.4.21
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3
`Z16`	Mode register	GMOR <14>	4.4.1
`Stencil Write Mask`	Stencil mode register	GSMR <7:0>	4.4.14
`Stencil Read Mask`	Stencil mode register	GSMR <15:8>	4.4.14
`S Test`	Stencil mode register	GSMR <18:16>	4.4.14
`S Fail`	Stencil mode register	GSMR <21:19>	4.4.14
`D Fail`	Stencil mode register	GSMR <24:22>	4.4.14
`D Pass`	Stencil mode register	GSMR <27:25>	4.4.14
`Z Test`	Stencil mode register	GSMR <30:28>	4.4.14

(continued on next page)

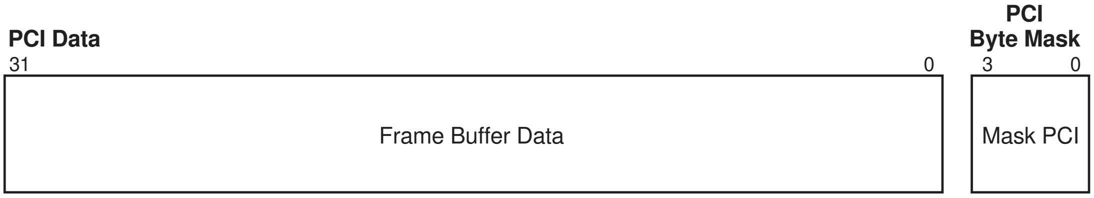
Table 6–10 (Cont.) Simple-Z Mode Parameters

Parameter	Source		Section
`Z Update`	Stencil mode register	GSMR <31>	4.4.14

The simple-Z mode works in almost the same way as the simple mode, except that the write to the `Frame Buffer Data` is conditional, based on the result of fetching and comparing the `Z Buffer` and `Stencil Buffer` values.

As in the simple mode, the PCI write addresses the color value in the bitmap at the `Frame Buffer Address` and passes the `Frame Buffer Data` and `Mask PCI` in the PCI write data. Figure 6–10 shows the format of the PCI write data.

Figure 6–10 Simple-Z Mode PCI Write-Data Format



To perform a Z-buffered frame buffer write operation, the 21030 does the following:

1. Fetches the `Z Stored` and `Stencil Stored` values and compares both to the `Stencil Reference` and `Z Reference` values from the 21030 registers.
2. Conditionally updates the stencil buffer and the Z-buffer, based on the results of the comparisons and the state of several control fields in the GSMR.
3. Conditionally writes the `Frame Buffer Data` to the `Frame Buffer Address`.

For the stencil and Z logic operations, the 21030 does the following:

- Reads the `Z Stored` and `Stencil Stored` values from the frame buffer at the `Z Address`. The 21030 extracts a 16-bit or 24-bit Z-value from each Dword, depending on the Z-buffer format specified by `Z16`. (See Section 6.1.5 for more information about the Z16 and Z24 formats.)
- For the stencil operation, logically ANDs the `Stencil Read Mask` with the `Stencil Stored` and `Stencil Reference` values.

- Compares the masked versions of the `Stencil Stored` and `Stencil Reference` values according to `S Test`.

  The 21030 also compares the `Z Stored` value with the `Z Reference` value as specified by `Z Test`.

  – If `S test` fails, the `Stencil Stored` value is updated as specified by `S Fail`.

  – If `S Test` passes and `Z Test` fails, the `Stencil Stored` value is updated as specified by `D Fail`.

  – If both `Z Test` and `S Test` pass, the `Stencil Stored` value is updated as specified by `D Pass`.

  Standard plane masking is disabled on Z and stencil accesses. However, the `Stencil Write Mask` parameter can be used as a plane mask for the stencil. On any write back to `Stencil Stored` location, the only bit positions modified are those specified by `Stencil Write Mask`. Per-bit enables are not available for writes to `Z Stored`.

  – If `Z Test` passes and `Z Update` is enabled, the `Z Reference` value is written back to the stored Z location.

  – If both `Z Test` and `S Test` pass, the `Frame Buffer Data` is written to the `Frame Buffer Address` as a function of the `Raster Op`, `Plane Mask`, `Mask GPXR` and `Mask PCI` parameters as in the simple mode.

  On any write back to `Stencil Stored` location, only the bit positions specified by the `Stencil Write Mask` are modified.

The `Destination Bitmap` and `Destination Byte` parameters allow access to all types of destination bitmaps (Section 6.1.5). The `Frame Buffer Address` must be aligned to 4 bytes for all destinations except unpacked 8-bpp bitmaps, which must be aligned to 16 bytes. (The `Source Bitmap` and `Source Byte` parameters are not used in the simple-Z mode.)

The simple-Z mode also supports a 16-bit or 24-bit Z-buffer format (Z16 or Z24) as specified by the Z16 bit (GMOR <14>, Section 4.4.1). In the Z16 format, the `Pixel Value` and `Z Stored` value can be embedded in the same Dword (Figure 6–5). If they are, software must explicitly set the `Frame Buffer Address` and the `Z Address` to the same location.

The following pseudo-code represents the basic algorithm for the simple-Z mode:

```
Read Z (Z Address, Stencil Stored, Z Stored);
Stencil Reference = Stencil Reference & SRdMask;
Stencil Stored = Stencil Stored & SRdMask;
if  ! S Test (Stencil Stored, Stencil Reference)
{
Conditional Write Stencil (Stencil Stored, S Fail, Stencil Write Mask);
}
else if Z Test (Z Stored, Z Reference)
{
Conditional Write Stencil (Stencil Stored, D Fail, Stencil Write Mask);
}
else
{
Conditional Write Stencil (Stencil Stored, D Pass, Stencil Write Mask);
}
if Z Update && Z Test (Z Stored, Z Reference)
{
Write Z(Stencil Stored);
}
/* conditional write pixel */
Write Mask = Mask PCI & Mask GPXR;
if  Z Test (Z Stored, Z Reference) && S Test (Stencil Stored, Stencil Reference)
{
Write Pixel (Frame Buffer Address, Frame Buffer Data, Raster Op,
             Plane Mask,Write Mask,Destination Bitmap,Destination Byte);
}
```

The simple-Z mode primarily allows the host to draw to Z-buffered bitmaps without the overhead involved in Z read, Z compare, conditional Z write, and conditional color write, which is usually required in Z-buffered drawing. Simple-Z mode operations are allowed only in the context of 32-bpp frame buffers.

6.2.3 Opaque-Stipple Mode

In the opaque-stipple mode, a PCI write to the frame buffer address space draws a bitonal, masked span of 32 contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–11.

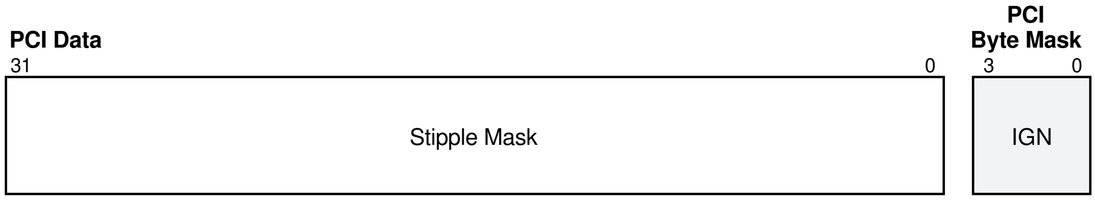
```
Opaque Stipple Span (Frame Buffer Address, Stipple Mask, Plane Mask,
                     Pixel Mask, Raster Op, Foreground, Background,
                     Destination Bitmap, Destination Byte);
```

Table 6–11 Opaque-Stipple Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Stipple Mask`	PCI write data	<31:0>	—
`Pixel Mask`	Pixel mask register	GPXR <31:0>	4.4.21
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Background`	Background register	GBGR <31:0>	4.4.19
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle to the `Frame Buffer Address` initiates the drawing operation and specifies the `Stipple Mask` in the format shown in Figure 6–11.

Figure 6–11 Opaque-Stipple Mode PCI Write-Data Format



The 21030 expands the 32-bit `Stipple Mask` to 32 pixels, masking each pixel according to the `Pixel Mask` (Figure 6–12), as follows:

- Write is enabled for each pixel that corresponds to a `Pixel Mask` bit = 1.
- Pixels that correspond to a `Pixel Mask` bit = 0 are unmodified.
- The 21030 writes the foreground color to each pixel that corresponds to a `Stipple Mask` bit = 1.
- The 21030 writes the background color to each pixel that corresponds to a `Stipple Mask` bit = 0.

The 21030 applies the specified `Raster Op` and `Plane Mask` on the write to the frame buffer.

The `Destination Bitmap` and `Destination Byte` parameters allow access to all types of destination bitmaps (Section 6.1.5). (The `Source Bitmap` and `Source Byte` parameters are not used in opaque-stipple mode.)

The destination address for 8-bpp destinations must be aligned to 4 pixels. Consequently, the `Frame Buffer Address` must be aligned to 4 bytes for drawing to 8-bpp packed bitmaps, and to 16 bytes for unpacked bitmaps; 12-bpp and 24-bpp destinations must be aligned to 8 bytes (1 quadword).

The following pseudo-code represents the basic algorithm for the opaque-stipple mode:

```
for (n = 0; n <= 31; n++)
{
if (Pixel Mask<n> = 1)
  {
  Pixel = (Stipple Mask<n> ? Foreground : Background;
  Write Frame Buffer (Frame Buffer Address, Pixel, Raster Op, Plane Mask,
                      Destination Bitmap, Destination Byte);
  }
Increment Pixel Address (Frame Buffer Address);
}
```

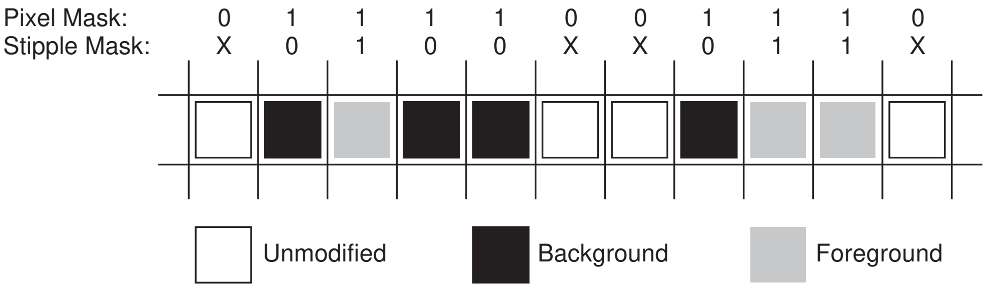
The 21030 optimizes the algorithm by writing 64 bits at a time; that is, up to 8 pixels to an 8-bpp packed bitmap, or 2 pixels to 8-bpp unpacked, 12-bpp, or 24-bpp bitmaps. The 21030 also increases performance by skipping over leading and trailing strings of zeros in the `Pixel Mask`. A result of this optimization is that the completion of opaque-stipple (and transparent-stipple) mode operations can leave the internal pixel-processing address incorrectly set for the next span (unlike line mode, Section 6.2.12).

The 21030 requires 4-byte or 8-byte address alignment and does not implicitly mask span edges; software must align addresses and mask left and right span-edges. Therefore, before delivering the `Stipple Mask` and `Pixel Mask` parameters to the 21030, software must:

- Align the `Stipple Mask`
- Align and logically combine the intended `Pixel Mask` with the desired left and right edge masks

The opaque-stipple operation writes a bitonal pattern to a bitmap. Figure 6–12 is an example of drawing in opaque-stipple mode.

Figure 6–12 Opaque-Stipple Mode Operation



The 21030 does the following operations in the opaque-stipple mode:

- Under X, does opaque stippling and tiling operations
- Under Windows, paints a region with an arbitrary bitonal brush
- In certain cases, draws text
- Quickly fills a solid region (however, in most cases, the block-stipple and block-fill modes are quicker)

(See Section 7.2.1 for more examples of opaque and transparent stipple mode applications.)

6.2.4 Transparent-Stipple Mode

In the transparent-stipple mode, a PCI write to the frame buffer address space draws a solid, masked span of 32 contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–12.

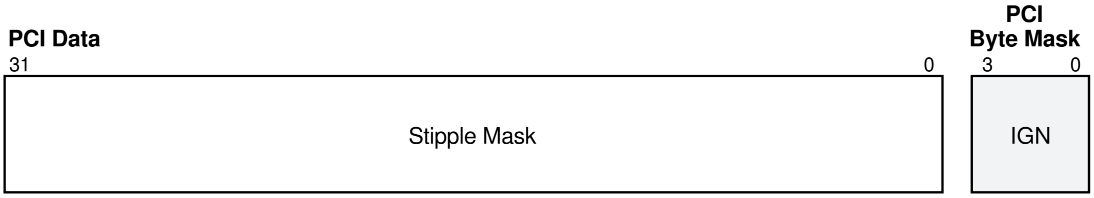
```
Transparent Stipple Span (Frame Buffer Address, Stipple Mask, Plane Mask,
                          Raster Op, Foreground, Destination Bitmap,
                          Destination Byte);
```

Table 6–12 Transparent-Stipple Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Stipple Mask`	PCI write data	—	—
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle to the `Frame Buffer Address` initiates the drawing operation and specifies the `Stipple Mask` in the format shown in Figure 6–13.

Figure 6–13 Transparent-Stipple Mode PCI Write-Data Format



Transparent-stipple operations are, in effect, a simpler version of opaque-stipple operations, and operate in the same way with the following exceptions:

- The `Pixel Mask` is not specified.

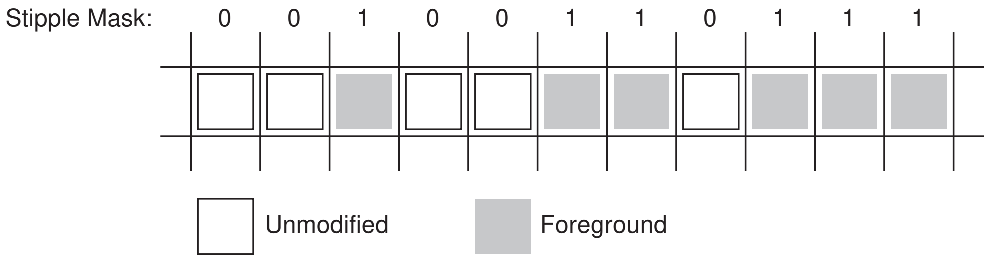
- The `Stipple Mask` bits determine whether foreground color is written or write is disabled for the corresponding pixels, rather than determining whether foreground or background color is written.

The transparent-stipple mode basic algorithm differs slightly from the opaque-stipple mode basic algorithm, and is represented by the following pseudo-code:

```
for (n = 0; n <= 31; n++)
{
if (Stipple Mask<n> = 1)
    Write Frame Buffer(Frame Buffer Address, Foreground, Raster Op, Plane Mask,
                       Destination Bitmap, Destination Byte);
Increment Pixel Address(Frame Buffer Address);
}
```

Figure 6–14 is an example of drawing in the transparent-stipple mode.

Figure 6–14 Transparent-Stipple Mode Operation



The transparent-stipple mode does the following operations:

- Fills regions in X transparent-stipple mode
- Under Windows, paints a region with a monochrome brush
- In many cases, draws text
- In some cases, fills solid regions

(See Section 7.2.1 for more examples of opaque-stipple and transparent-stipple mode applications.)

6.2.5 Block-Stipple Mode

In the block-stipple mode, a PCI write to the frame buffer address space draws a masked span of 32 contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–13.

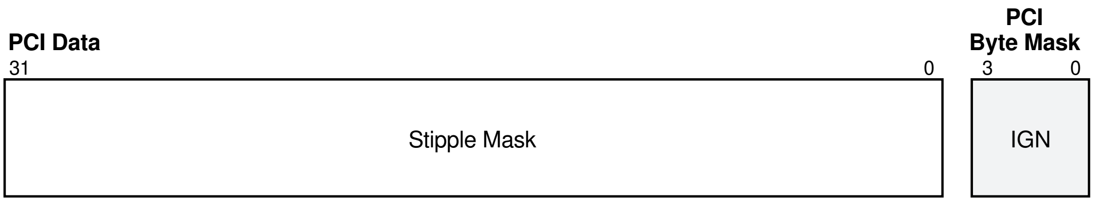
```
Block Stipple Span (Frame Buffer Address, Stipple Mask, Block Color Pattern <7:0>,
                    Plane Mask, Destination Bitmap, Destination Byte);
```

Table 6–13 Block-Stipple Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Stipple Mask`	PCI write data	<31:0>	—
`Block Color Pattern <7:0>`	Block color registers	GBCR<7:0> <31:0>	4.4.4
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle to the `Frame Buffer Address` initiates the drawing operation and specifies the `Stipple Mask` in the format shown in Figure 6–15. The `Frame Buffer Address` must be aligned to 1 pixel.

Figure 6–15 Block-Stipple Mode PCI Write-Data Format



The block-stipple mode is functionally similar to the transparent-stipple mode, with the following exceptions:

- The `Raster Op` is not specified.
- The GBCRs specify an 8-pixel color pattern that is repeated (tiled) four times across the 32-pixel span, rather than substituting foreground color on a per-pixel basis (Figure 6–16).

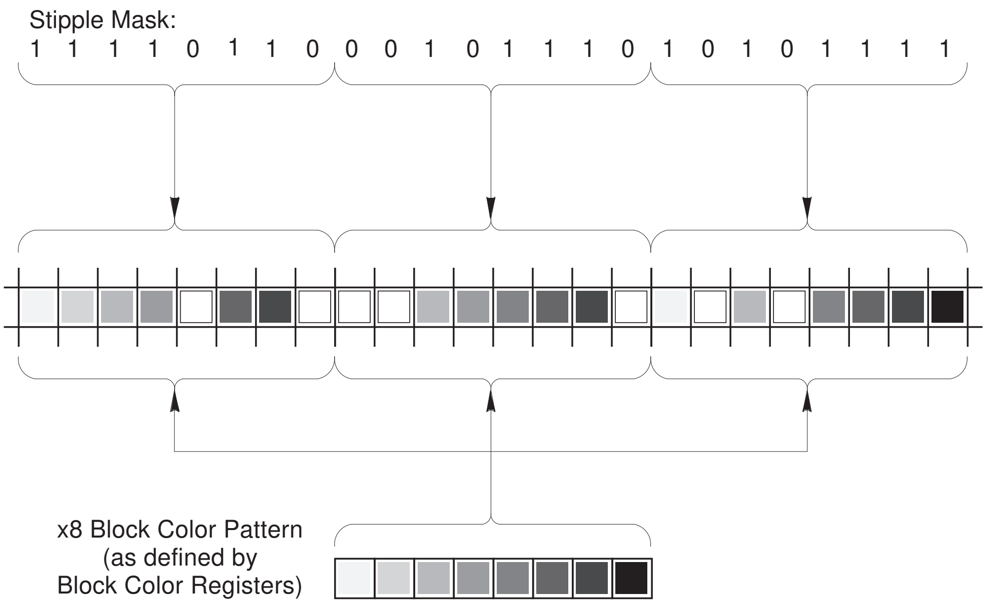
- The block-stipple mode draws four times faster than the transparent-stipple mode.

The 21030 generates a 32-pixel pattern by replicating the specified 8-pixel `Block Color Pattern` four times. The 21030 then writes the 32-pixel pattern, masked by `Stipple Mask`, to the frame buffer beginning at the `Frame Buffer Address`.

The 21030 writes the `Block Color Pattern` color to each pixel that corresponds to a `Stipple Mask` bit = 1. Pixels that correspond to a `Stipple Mask` bit = 0 are unmodified (Figure 6–16).

Graphically, the block-stipple operation writes a 32-pixel masked span, composed of any arbitrary, 8-pixel, repeating color pattern, to a bitmap. Figure 6–16 is an example of this operation.

Figure 6–16 Block-Stipple Mode Operation



The `Plane Mask` is enabled during block-stipple operations, but the `Raster Op` is not enabled. In effect, the raster operation function is hardwired in the block-stipple mode. This is because the GBCRs are physically resident in the VRAMs and the `Block Color Pattern` (the source operand) is always written directly to the memory (the destination operand). In effect, the block-stipple mode raster-op function is hardwired to *dest* ⇐ *src*.

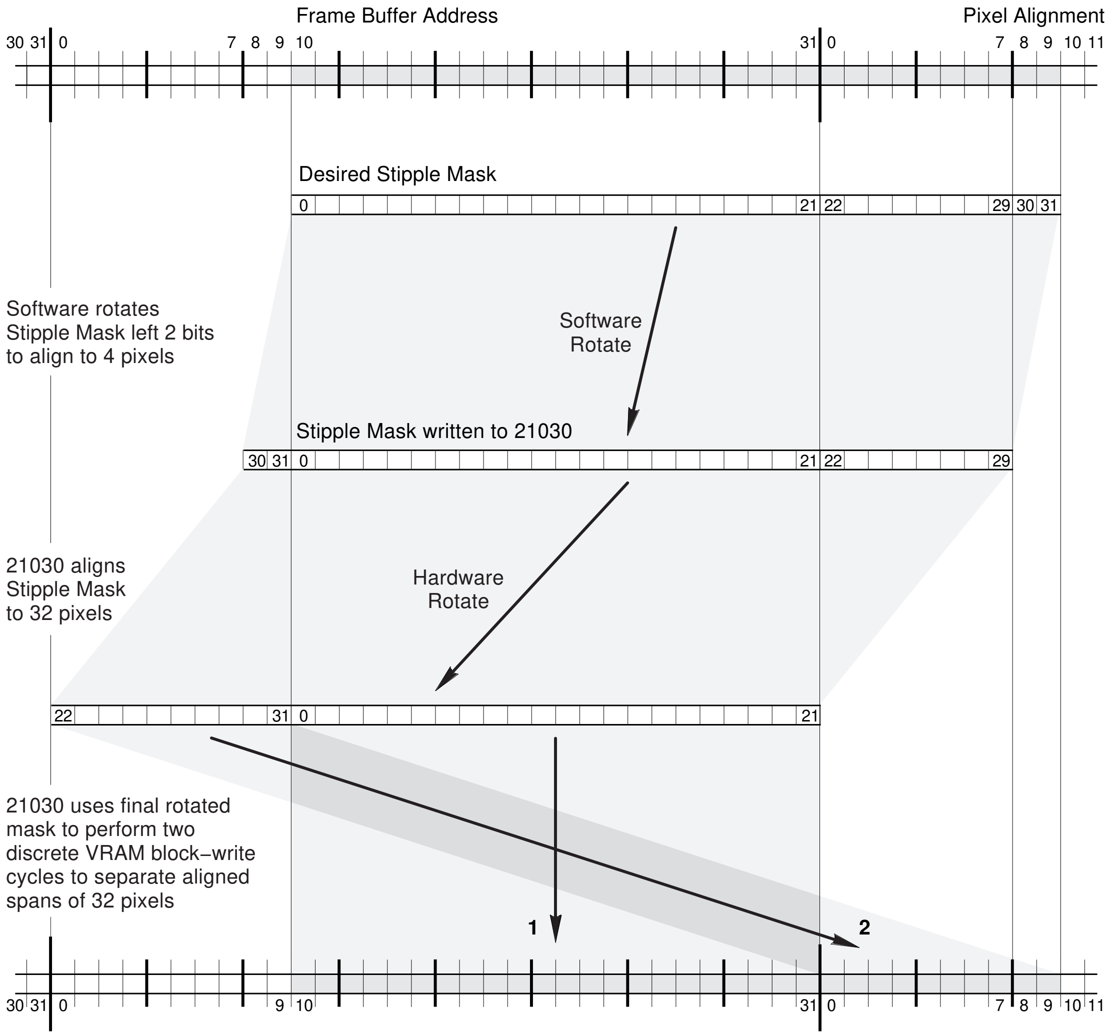
The following pseudo-code represents the basic algorithm for the block-stipple mode:

```
for (i = 0; i <= 3; i++)
{
  for (j = 0; j <= 7; j++)
  {
    if (Stipple Mask<i*8+j> = 1)
    Write Frame Buffer(Frame Buffer Address, Block Color Pattern j, Raster Op,
                       Plane Mask, Destination Bitmap, Destination Byte);
  }
  Increment Pixel Address(Frame Buffer Address);
}
```

The 21030 optimizes the algorithm by using the block-write feature of the VRAMs to write all 32 pixels in one or two write cycles. The number of write cycles required depends on the span alignment, the destination bitmap depth, and whether a non-trivial mask (that is, not all ones) is used. If the `Frame Buffer Address` is aligned to 32 pixels in an 8-bpp frame buffer, or aligned to 32 pixels and unmasked in a 32-bpp frame buffer, the 21030 writes the entire span in one VRAM write cycle; otherwise, the 21030 breaks the span into two VRAM write cycles. If arbitrary masking is used, block writes to 32-bpp bitmaps can take as many as eight cycles per 32 pixels. However, drawing to 8-bpp bitmaps in either 8-bpp or 32-bpp frame buffers takes a maximum of two write cycles per 32-pixels.

Figure 6–17 shows how the hardware breaks a span that is not aligned to 32-pixels.

Figure 6–17 Block-Stipple Mode Address and Mask Alignment



The VRAM block-write feature allows the block-stipple mode to draw up to four times faster than the transparent-stipple or opaque-stipple modes. Additionally, the 21030 increases performance by skipping over leading and trailing strings of zeros in the `Stipple Mask`.

The block-stipple mode supports drawing to all types of destination bitmaps, except packed 8-bpp bitmaps in a 32-bpp frame buffer. All other destination bitmaps can be specified through the `Destination Bitmap` and `Destination Byte` parameters (Section 6.1.5). Software must initialize the `Block Color Pattern` in the GBCRs according to the specific destination bitmap. (See Section 4.4.4 for a description of the GBCR formats as a function of destination bitmap format.)

The block-stipple mode requires the following parameter alignments:

`Frame Buffer Address`	Aligned to 1 pixel
`Stipple Mask`	Aligned to 4 pixels
`Block Color Pattern`	Aligned to 8 pixels

6.2.5.1 Frame Buffer Address Alignment

The `Frame Buffer Address` must be aligned to 1 pixel. In terms of bytes, the `Frame Buffer Address` must be aligned to 1 byte for drawing to 8-bpp packed bitmaps (on 8-bpp frame buffers only), and to 4 bytes for all other destination bitmaps.

When drawing to 24-bpp, 12-bpp, and unpacked 8-bpp bitmaps, this restriction is insignificant, because the pixel address naturally aligns to 4 bytes and can be specified in a normal PCI write to the frame buffer address space. Drawing operations to 4-pixel-aligned packed 8-bpp destinations can be also initiated with a direct write to the frame buffer.

However, the `Frame Buffer Address` alignment requirement becomes significant when drawing to 1-pixel-aligned packed 8-bpp bitmaps. Such drawing implies an address alignment of 1 byte, while the address of a PCI write is Dword-aligned (aligned to 4 bytes). Consequently, software cannot use block-stipple mode to draw to packed 8-bpp bitmaps and initiate the drawing operation by writing to the frame buffer. The drawing operation can be initiated only by writing to the continue register (GCTR, Section 4.3.3), with the byte address explicitly specified in the address register (GADR, Section 4.4.2).

Ultimately, the memory controller executes one or two VRAM block-write cycles to complete one block-stipple mode operation. When the `Frame Buffer Address` is aligned to 32 pixels, only one block-write cycle is required; otherwise, the 21030 breaks the operation into two-consecutive block-write cycles. In either case, `Stipple Mask` must be aligned to 32-pixels in order to apply the mask on the block-write cycle.

6.2.5.2 Stipple Mask Alignment

Software and hardware share responsibility for aligning the `Stipple Mask` to 32-pixels. Software must first align the `Stipple Mask` to 4 pixels. To do this, software must rotate the mask such that mask bit 0 aligns with the first pixel in a naturally aligned 4-pixel span. The chip completes the alignment to 32 pixels through a hardware rotate operation. Figure 6–17 shows the two stages of `Stipple Mask` rotation for a particular value of `Frame Buffer Address` and a given pixel alignment. Note that the hardware initiates two VRAM block-write cycles for unaligned values of the `Frame Buffer Address`.

This `Stipple Mask` alignment restriction significantly reduces chip complexity, while requiring a minimum of additional CPU computation. For performance-critical operations that use the block-stipple mode (for example, masked, large, area fills), the CPU should not be the bottleneck, and the extra computation should have no effect on throughput.

6.2.5.3 Block Color Pattern Alignment

Because the `Block Color Pattern` is fixed in the VRAM-resident GBCRs, software must prealign the `Block Color Pattern` to 8-pixel boundaries.

6.2.5.4 Using Block Stipple Mode

The block-stipple mode can be used in place of the transparent-stipple and opaque-stipple modes in the following situations:

- The fill span is of significant size.

  An appreciable amount of overhead is involved when the CPU generates and aligns the `Block Color Pattern`, and writes the pattern to the GBCRs. Consequently, large, solid or tiled, fill regions can benefit greatly from block-stipple operations (any benefit is unlikely for narrow spans).

- The desired raster-operation function does not include destination.

  In the block-stipple mode, raster-operation functions are not used. When generating the `Block Color Pattern`, software can perform some Boolean operation on the source operand but not on the destination operand.

- The desired color pattern repeats on intervals of 2^j pixels, where $j \leq 3$.

  Because the `Block Color Pattern` width is fixed at 8 pixels, repeating color patterns that are 1, 2, 4, or 8 pixels wide are the only patterns that produce the correct block stipple when mapped to the `Block Color Pattern`. More specifically, the pattern width must equal 2^j, where $j \leq 3$. For example, the opaque-stipple or transparent-stipple mode must be used in order for software to fill a span with a pattern that is 5 pixels wide. Fortunately, the most common X and Windows pattern-fill operations can take advantage of the block-stipple mode. For example, under Windows, the most common brush patterns are 8 pixels wide.

6.2.6 Block-Fill Mode

In the block-fill mode, a PCI write to the frame buffer address space writes a span of up to 2K contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–14.

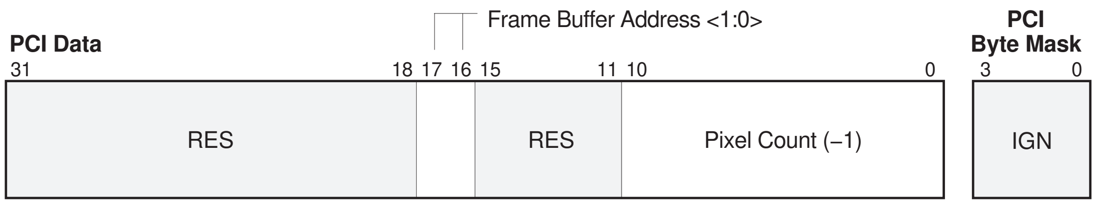
```
Block Fill Span (Frame Buffer Address, Pixel Count(-1), Frame Buffer Address <1:0>,
                Block Color Pattern<7:0>, Plane Mask, Fill Mask,
                Destination Bitmap, Destination Byte);
```

Table 6–14 Block-Fill Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Pixel Count (-1)`	PCI write data	<10:0>	—
`Frame Buffer Address <1:0>`	PCI write data	<17:16>	—
`Block Color Pattern <7:0>`	Block color registers	GBCR<7:0> <31:0>	4.4.4
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Fill Mask`	Data register	GDAR <31:0>	4.4.8
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle to the `Frame Buffer Address` initiates the drawing operation and specifies the `Pixel Count` and `Frame Buffer Address <1:0>` in the format shown in Figure 6–18. As in the block-stipple mode, the `Frame Buffer Address` must be aligned to 1 pixel.

Figure 6–18 Block-Fill Mode PCI Write-Data Format



The block-fill mode is functionally similar to the block-stipple mode. The primary differences in the block-fill mode are as follows:

- The span is limited to 2K pixels (rather than 32 pixels) and the `Pixel Count` is encoded in the PCI write data.
- The `Fill Mask` is specified once (passed in GDAR) and repeated at 32-pixel intervals across the span.
- `Frame Buffer Address <1:0>` (the starting address LSBs) is specified to simplify drawing to packed 8-bpp bitmaps.

Drawing a 32 $\times$ *n*-pixel span in the block-fill mode is effectively the same as drawing *n* 32-pixel spans in the block-stipple mode.

The 21030 replicates the 8-pixel wide `Block Color Pattern` four times to generate a 32-pixel color pattern. Starting at the `Frame Buffer Address`, the 21030 writes the 32-pixel pattern through the `Fill Mask` to the frame buffer as the first of *n* segments needed to fill an entire span (up to 2K pixels). For each segment, the 21030 writes the `Block Color Pattern` color to each pixel that corresponds to a `Fill Mask` bit = 1. Pixels that correspond to a `Fill Mask` bit = 0 are unmodified (Figure 6–19).

After writing each masked 32-pixel span segment, the 21030 decrements the `Pixel Count` by 32. If the full span operation is not complete (that is, the `Pixel Count` is positive), the 21030 updates the `Frame Buffer Address` to point to the next contiguous span, and again draws the span segment. The 21030 repeats the process until a full span of `Pixel Count` pixels is drawn.

All of the span segments (with the possible exception of the last segment) are identical because the same `Block Color Pattern` is written through the same `Fill Mask` for each segment drawn. If the `Pixel Count` is not an integer multiple of 32, the last segment is `Pixel Count` MOD 32 and is masked by the corresponding `Fill Mask` LSBs.

The block-fill mode alignment requirements for the `Frame Buffer Address`, `Fill Mask`, and `Block Color Pattern` are the same as the block-stipple mode alignment requirements (replacing `Stipple Mask` with `Fill Mask`). However, unlike the block-stipple mode, the block-fill mode allows software to provide `Frame Buffer Address <1:0>` (the two LSBs) as part of the PCI write data. This allows drawing to packed 8-bpp bitmaps that are not aligned to four pixels without using the GADR and the GCTR.

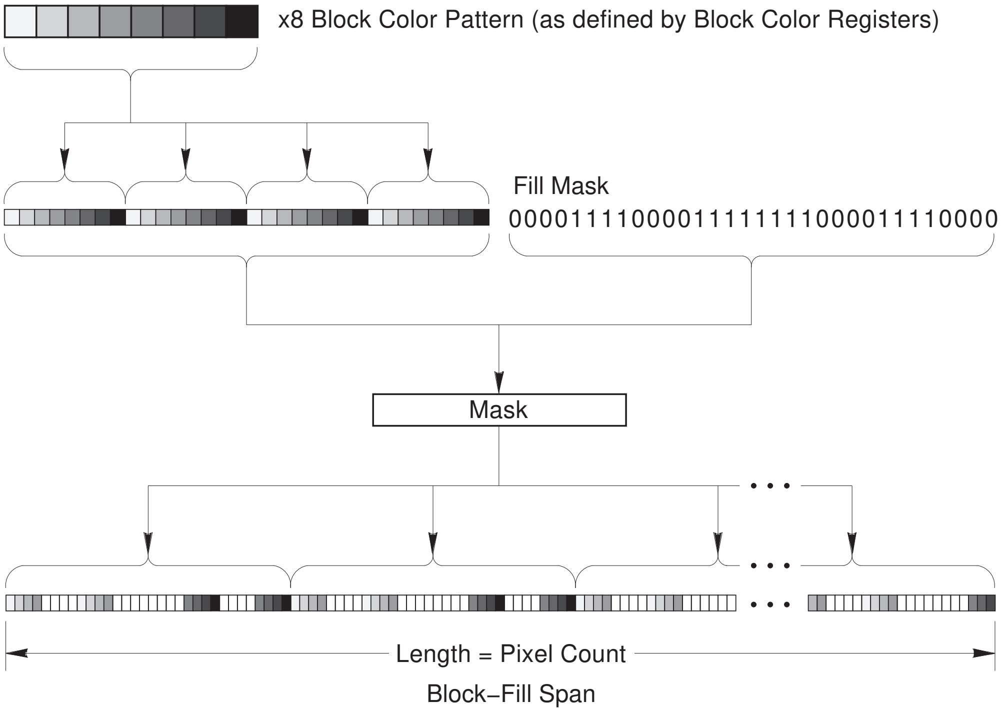
The block-fill mode supports the same destination bitmaps as the block-stipple mode. All bitmap formats are supported except packed 8-bpp (PB8) in a 32-bpp frame buffer. Software must initialize the `Block Color Pattern` in the GBCRs according to the specific destination bitmap. (See Section 4.4.4 for a description of the GBCR formats as a function of destination bitmap format.)

The following pseudo-code represents the basic algorithm for the block-fill mode:

```
while (Pixel Count > 32)
{
  Block Stipple Span(Frame Buffer Address, Fill Mask, Block Color);
  Get Next Contiguous Span Address(Frame Buffer Address);
  pixel count -= 32;
}
Block Stipple Span(Frame Buffer Address, Fill Mask < pixel count:0>, Block Color);
```

Graphically, the block-fill operation writes a span to a bitmap. The span can be up to 2K pixels, composed of any arbitrary, 8-pixel, repeating color pattern, and masked by a repeating, 32-bit, fill mask. Figure 6–19 is an example of this operation.

Figure 6–19 Block-Fill Mode Operation



Typically, the block-fill mode, rather than the block-stipple mode, is used when the `Stipple Mask` repeats at an interval of 2^i, where $i \leq 5$. That is, the block-fill mode might work as well as or better than the block-stipple mode if the single `Fill Mask` can be passed as the `Stipple Mask` for multiple span-segments.

The most common block-fill mode application is filling a large screen area with a solid color, or a tile or brush with a width = 2^j, where $j \leq 3$. Many X tile and Windows brush sizes meet this requirement. Using the block-fill mode, software can fill an entire window or the screen with one write per scan line to the 21030.

The block-fill mode can also be used with appropriate `Plane Mask` values to clear and select between in-place visual buffers in a 32-bpp frame buffer, accelerating graphic animation rates. (See Section 7.2 for more information about block-fill mode applications.)

6.2.7 Opaque-Fill Mode

In the opaque-fill mode, a PCI write to the frame buffer address space writes a bitonal, unmasked span of up to 2K contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–15.

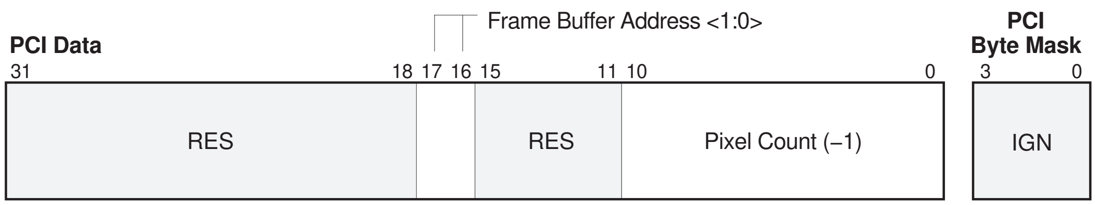
```
Opaque Fill Span (Frame Buffer Address, Pixel Count, Frame Buffer Address <1:0>,
                  Foreground, Background, Raster Op, Plane Mask, Pixel Mask,
                  Fill Mask, Destination Bitmap, Destination Byte);
```

Table 6–15 Opaque-Fill Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Pixel Count (-1)`	PCI write data	<10:0>	—
`Frame Buffer Address <1:0>`	PCI write data	<17:16>	—
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Background`	Background register	GBGR <31:0>	4.4.19
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Pixel Mask`	Pixel mask register	GPXR <31:0>	4.4.21
`Fill Mask`	Data register	GDAR <31:0>	4.4.8
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle to the `Frame Buffer Address` initiates the drawing operation and specifies the `Pixel Count` and `Frame Buffer Address <1:0>` in the format shown in Figure 6–20.

Figure 6–20 Opaque-Fill Mode PCI Write-Data Format



Functionally, the opaque-fill mode is a slower, but more flexible variation of the block-fill mode. The primary differences in the opaque-fill mode are as follows:

- The 21030 uses standard page-mode write cycles, rather than fast ($\times$ 4) block-write cycles.
- The applied colors are generated from the `Foreground` and `Background` parameters (as in the opaque-stipple mode) and the `Block Color` parameter is not used.
- The Boolean operation defined by `Raster Op` is applied.

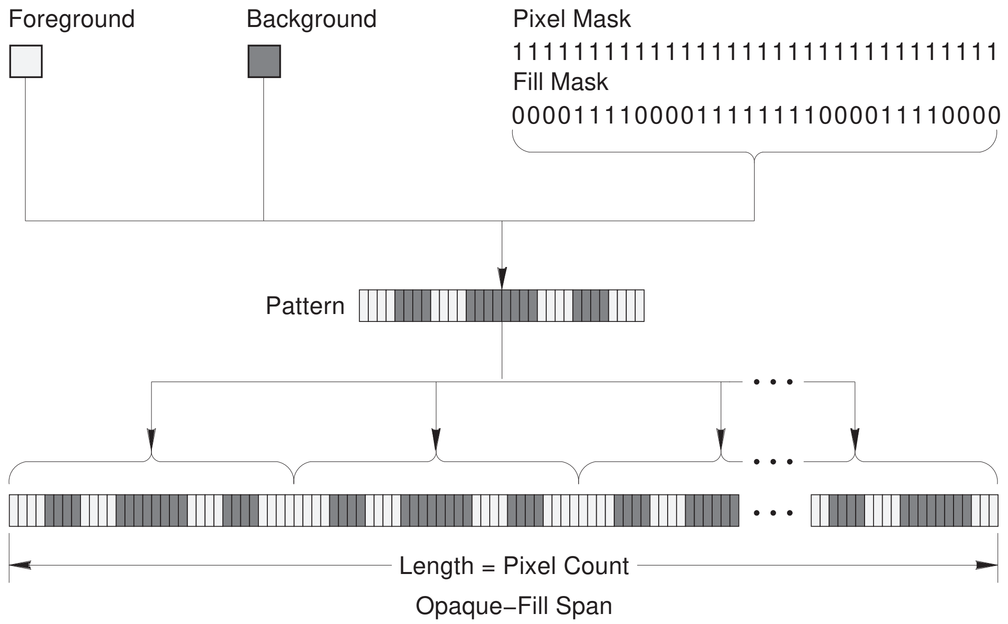
Also (unlike an opaque-stipple operation), the `Pixel Mask` parameter is not specified; therefore, individual pixels cannot be masked in the opaque-fill mode.

The opaque-fill mode fills a span of `Pixel Count` (up to 2K) pixels with a repeating, bitonal, 32-pixel pattern, as follows:

1. The pattern is defined by the `Foreground`, `Background`, `Pixel Mask`, and `Fill Mask` parameters.
2. `Fill Mask` enables the color for each pixel that corresponds to a `Pixel Mask` bit = 1. Pixels that corresponds to a `Pixel Mask` bit = 0 are not modified.
3. The 21030 writes the `Foreground` color to each pixel that corresponds to a `Fill Mask` bit = 1, and writes the `Background` color to each pixel that corresponds to a `Fill Mask` bit = 0.
4. The 32-pixel pattern is repeated as many times as necessary to fill up to `Pixel Count` pixels (Figure 6–21).

As in the block-fill mode, the `Frame Buffer Address` must be aligned to 1 pixel (1 byte in 8-bpp frame buffers and 4 bytes on 32-bpp frame buffers) and both `Pixel Mask` and `Fill Mask` must be aligned to 4 pixels. The `Frame Buffer Address` <`1:0`> parameter (two LSBs) provides byte-granularity on 8-bpp frame buffers.

Figure 6–21 Opaque-Fill Mode Operation



The opaque-fill mode can be used in place of the opaque-stipple or block-fill mode for larger, bitonal fill operations that do not require per-pixel masking, but do require a Boolean logic operation (the block-fill mode supports only $dest \Leftarrow src$ Boolean operations).

6.2.8 Transparent-Fill Mode

In the transparent-fill mode, a PCI write to the frame buffer address space writes a solid, masked span of up to 2K contiguous pixels starting at that address. The 21030 draws the span as a function of the parameters listed in Table 6–16.

```
Transparent Fill Span (Frame Buffer Address, Pixel Count, Frame Buffer Address <1:0>,
                       Foreground, Raster Op, Plane Mask, Fill Mask,
                       Destination Bitmap, Destination Byte);
```

Table 6–16 Transparent-Fill Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Pixel Count (-1)`	PCI write data	<10:0>	—
`Frame Buffer Address <1:0>`	PCI write data	<17:16>	—
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Fill Mask`	Data register	GDAR <31:0>	4.4.8
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The transparent-fill mode fills a span of `Pixel Count` (up to 2K) pixels with a repeating, bitonal, 32-pixel pattern defined by the `Foreground`, `Background`, and `Fill Mask` parameters.

The transparent-fill mode is almost identical to the opaque-fill mode. The `Foreground` color is written to each pixel that corresponds to a `Fill Mask` bit = 1; but in the transparent-fill mode, pixels that correspond to a `Fill Mask` bit = 0 are unmodified (rather than being written with the `Background` color).

As in the opaque-fill mode, the 32-pixel pattern is repeated as many times as necessary to fill up to `Pixel Count` pixels.

The transparent-fill mode can be used in place of the transparent-stipple or the block-fill mode for larger, solid-fill operations that require a Boolean logic operation (the block-fill mode supports only $dest \Leftarrow src$ Boolean operations).

6.2.9 Copy Mode

In the copy mode, a set of two consecutive PCI writes to the frame buffer address space copies a contiguous span of up to 64 bytes from the first address to the second address. The span can be copied in either direction: left-to-right (increasing addresses) or right-to-left (decreasing addresses). The 21030 performs the copy as a function of the parameters listed in Table 6–17.

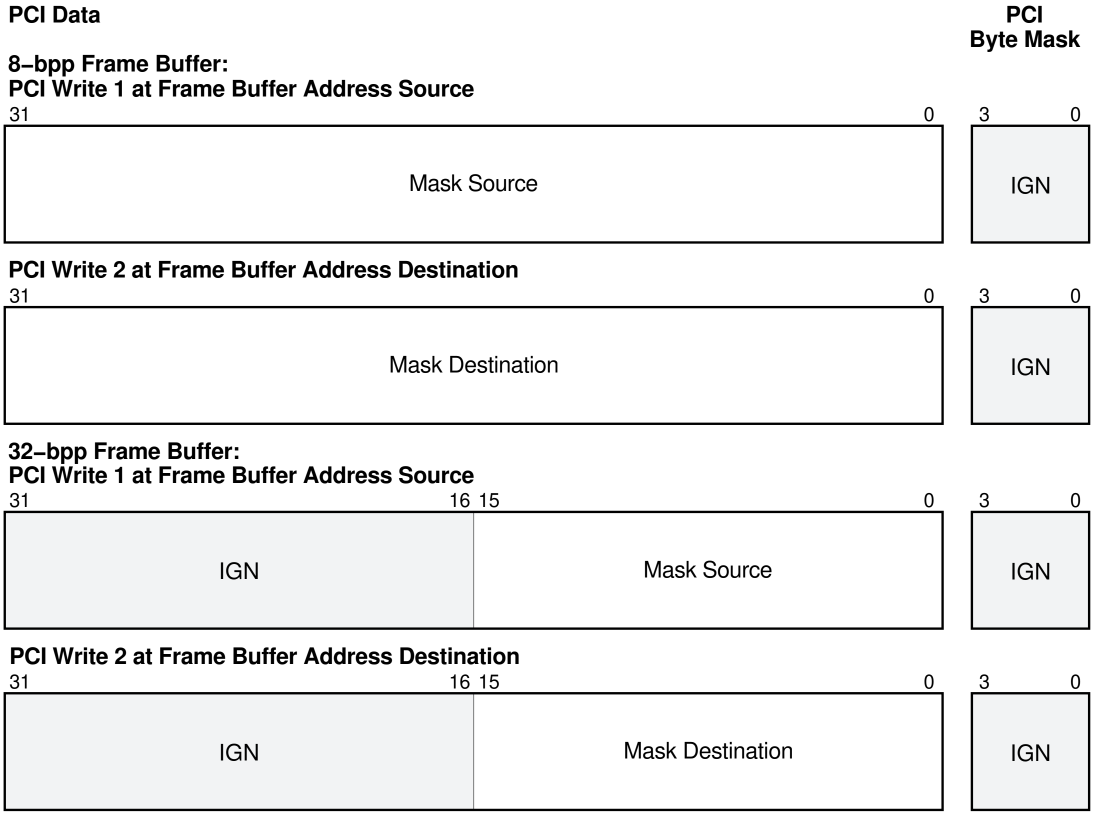
```
Copy Span (Frame Buffer Address Source, Frame Buffer Address Destination,
          Mask Source, Mask Destination, Plane Mask, Raster Op, Source Bitmap,
          Source Byte, Destination Bitmap, Destination Byte);
```

Table 6–17 Copy Mode Parameters

Parameter	Source		Section
`Frame Buffer Address Source`	PCI write address 1	—	—
`Mask Source`	PCI write data 1	—	—
`Frame Buffer Address Destination`	PCI write address 2	—	—
`Mask Destination`	PCI write data 2	—	—
`Pixel Shift`	Pixel shift register	GPSR <3:0>	4.4.5
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Source Bitmap`	Mode register	GMOR <10:8>	4.4.1
`Source Byte`	Mode register	GMOR <12:11>	4.4.1
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

Two PCI write cycles are necessary to copy one span locally in the frame buffer. The first PCI write addresses the location of the source span (`Frame Buffer Address Source`) and passes a read mask for the source (`Mask Source`) as data. The second PCI write addresses the location of the destination span (`Frame Buffer Address Destination`) and passes a write mask for the destination (`Mask Destination`) as data. Figure 6–22 shows the format of the PCI write operations.

Figure 6–22 Copy Mode PCI Write Data Formats



The maximum span limit of 64 bytes is set by the depth of the internal copy buffer and is independent of the frame buffer size (Table 6–18). Consequently, the maximum span size is 16 pixels when using copy mode in a 32-bpp frame buffer, regardless of whether the bitmap involved is 24-bpp, 12-bpp, or unpacked 8-bpp.

In an 8-bpp frame buffer, the size of the copy buffer allows 64-pixel spans, but each PCI write can supply only 32 bits to specify the mask. Consequently, the maximum span is 32-pixels when copying a masked span. (A different 21030 mechanism allows 64-pixel unmasked spans to be copied in an 8-bpp frame buffer, Section 6.2.9.5.)

Table 6–18 Copy Mode Span Limits

Frame Buffer Depth	Span Limit (Masked)	Span Limit (Unmasked)
8-bpp	32 pixels	64 pixels
32-bpp	16 pixels	16 pixels

In an 8-bpp frame buffer, the depth and byte parameter values must = 0 for all copies. In a 32-bpp frame buffer, the depth and byte parameters are used when copying between different format source and destinations; for example, copying from an unpacked, 8-bpp bitmap to an off-screen, packed, 8-bpp bitmap. For copies between a 24-bpp source and destination, the depth parameter values must = 7, and the byte parameter values must = 0. (See Section 6.2.9.8 for more information about depth and byte values, and copying between bitmaps of different depths and formats in a 32-bpp frame buffer.)

Basically, the 21030 performs a masked, span copy operation in two stages, as follows:

1. On the first PCI write, the 21030 reads up to 64 bytes from the `Frame Buffer Address Source`, selectively aligning and depositing those bytes into the copy buffer, starting at the bottom and filling upward. Only pixels that correspond to a `Mask Source` bit = 1 are read.
2. On the second PCI write, the 21030 unloads up to 64 bytes from the copy buffer, starting at the bottom and draining upward. Each pixel is conditionally stored as a function of the `Mask Destination`, starting at the `Frame Buffer Address Destination`. Only pixels that correspond to a `Mask Source` bit = 1 are written.

   On the final write to the destination, the `Plane Mask` and the Boolean operation specified by the `Raster Op` parameter are applied.

Copy mode can handle any span including the following:

- Copies with aligned or unaligned source and destination
- Copies that require backward (right-to-left) processing in addition to forward (left-to-right) processing

Arbitrarily aligned sources and destinations require the 21030 to shift source data as it is processed. Backward processing is necessary in certain alignments of overlapping copies, and requires the 21030 to increment and decrement its addresses as it steps through the span.

6.2.9.1 Source and Destination Alignment

For most source and destination bitmaps, the Frame Buffer Address Destination and Frame Buffer Address Source must be aligned to 8 bytes rather than arbitrary pixel boundaries. However, unpacked 8-bpp source and destination bitmaps in a 32-bpp frame buffer must be aligned to 32 bytes (see Section 6.2.9.8 for more information on copy mode support for various bitmap formats).

To copy a 32-pixel span, the 21030 reads up to 4 successive quadwords from the Frame Buffer Address Source masked by Mask Source; and writes up to 4 successive quadwords to the Frame Buffer Address Destination masked by Mask Destination. Consequently, copies are simple when both the Frame Buffer Address Source and the Frame Buffer Address Destination lie on natural quadword boundaries. However, graphics software (graphics applications, window managers, and so on) is not limited to specifying only quadword-aligned source and destination addresses. Therefore, the 21030 display driver must handle arbitrarily aligned source and destination addresses. The 21030 driver and hardware share the responsibility for ensuring that all possible combinations of desired source and destination are correctly handled.

Software must first adjust (that is, decrement) the desired source-pixel and destination addresses to quadword boundaries, such that the adjusted addresses can be passed as the Frame Buffer Address Source and Frame Buffer Address Destination. In addition, Mask Source and Mask Destination must be bit-shifted by the number of bytes that the addresses were decremented. That number is defined as Source Align and Destination Align for the source and destination. They are calculated as follows:

```
Source Align      = (desired source address) & Align Mask;
Destination Align = (desired destination address) & Align Mask;
```

In the previous equations,

Align Mask = 00000007_{16} for 8-bpp frame buffers
00000004_{16} for 32-bpp frame buffers

In general, the specified source and the destination alignment is random, with Source Align and Destination Align taking values from 0 to 7 in 8-bpp frame buffers, and 0 or 4 in 32-bpp frame buffers. In an aligned copy, Source Align and Destination Align take the same value; however, the 21030 display driver must usually process an unaligned copy in which Source Align and Destination Align take different values.

To process unaligned spans, the 21030 includes a hardware byte-shifter that aligns quadword source read data to the destination prior to filling the copy buffer. The `Pixel Shift` parameter is a signed 4-bit value ($-8 \leq$ `Pixel Shift` ≤ 7) that specifies the number of bytes to shift. Embedded in the byte-shifter is a 64-bit residue register that stores the previous quadword read from the copy source. (The residue register cannot be directly read or written.) The byte-shift function in conjunction with the residue register allows 21030 to process all possible combinations of the `Source Align` and `Destination Align` values.

In an unaligned copy, at least 1 pixel from each of 2 successive quadwords read from the source must be merged into 1 quadword in the destination. Consequently, in each quadword the 21030 writes to the destination, it must extract some subset of pixels from source quadword *n* and merge them with a complementary subset of pixels from source quadword *n–1*. This amounts to a 1-stage source-read pipeline, in which the residue register always stores the last quadword read.

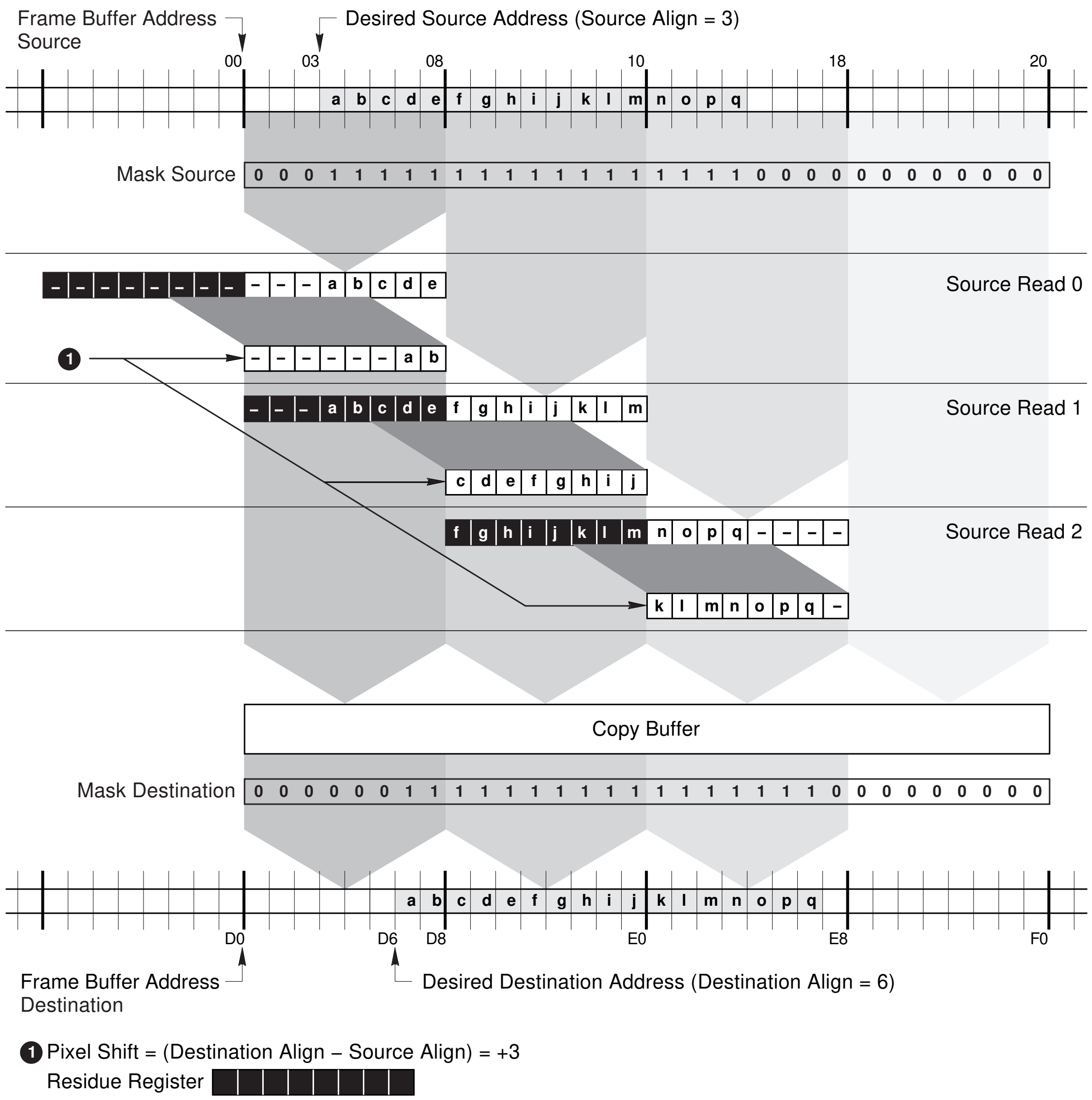
Starting at the `Frame Buffer Address Source`, the 21030 does the following:

1. Reads 1 to 4 quadwords, depending on the value in `Mask Source`.
2. Concatenates the 64 bits read from each quadword with the residue register.
3. Rotates the resulting 128-bit quantity through the byte-shifter by the amount specified by the `Pixel Shift` value (a negative value rotates left, and a positive value rotates right).
4. Extracts a quadword (now properly aligned with the destination) from the bit positions corresponding to the position of the read data before rotation.
5. Loads the extracted quadword into the copy buffer in the next available quadword entry.
6. Stores the last quadword read into the residue register.
7. Moves on to the next source quadword and repeats the process until the span is complete.

Figure 6–23 is an example of an unaligned forward (left-to-right) copy with an 8-bpp packed source and destination. The individual pixels are labeled **a** through **q**. Three quadwords are read through `Mask Source` and three are written through `Mask Destination`. For each read, the figure shows a ''snapshot'' of the contents of the residue register and the resultant byte-shifter output quadword. An example of a copy with a 24-bpp source and destination would be almost the same, with the following exceptions:

- Letters **a** through **q** would correspond to bytes within a pixel.
- The `Source Align` and `Destination Align` values would be 0 or 4.

Figure 6–23 Forward Span Copy



6.2.9.2 Backward Copies

In addition to arbitrary alignments, the 21030 must process forward (left-to-right) and backward (right-to-left) copies. Spans that overlap require the graphics server to pick a direction, to avoid corrupting a portion of the source before it is read. Consequently, the 21030 selectively increments (forward) or decrements (backward) source and destination quadword addresses in order to step through the span. The sign of the `Pixel Shift` value determines the direction of the span copy, as follows:

–8 ≤ `Pixel Shift` ≤ –1 for backward copies
0 ≤ `Pixel Shift` ≤ 7 for forward copies

For a negative `Pixel Shift` value, the 21030 does the following:

- Begins reading at the `Frame Buffer Address Source` and writing at the `Frame Buffer Address Destination`.
- Decrements the `Frame Buffer Address Source` after each quadword is read.
- Decrements the `Frame Buffer Address Destination` after each quadword is written.

For a positive `Pixel Shift` value, the 21030 also begins at `Frame Buffer Address Source` and `Frame Buffer Address Destination`, but it increments the respective addresses as it steps through the span.

The sign of the `Pixel Shift` value also determines the direction that the byte-shifter rotates incoming source data (with residue): negative for rotate left and positive for rotate right. Therefore, the assignment of the `Pixel Shift` value must take into account that all incoming source data is rotated to the right in a forward copy and to the left in a backward copy.

Table 6–19 shows how the `Pixel Shift` value is calculated as a function of alignment and copy direction.

Table 6–19 Assigning the Pixel Shift Value

Direction	Destination Align ≥ Source Align
Forward	Destination Align – Source Align
Backward	(Destination Align – Source Align) – 8
Direction	**Source Align > Destination Align**
Forward	8 – (Source Align – Destination Align)
Backward	Destination Align – Source Align

6.2.9.3 Priming and Flushing the Residue Register

Certain combinations of alignment and copy direction require one additional adjustment to be made prior to starting the copy mode operation. Two types of copies fall into this category:

- Forward copies when `Source Align` > `Destination Align`
- Backward copies when `Destination Align` > `Source Align`

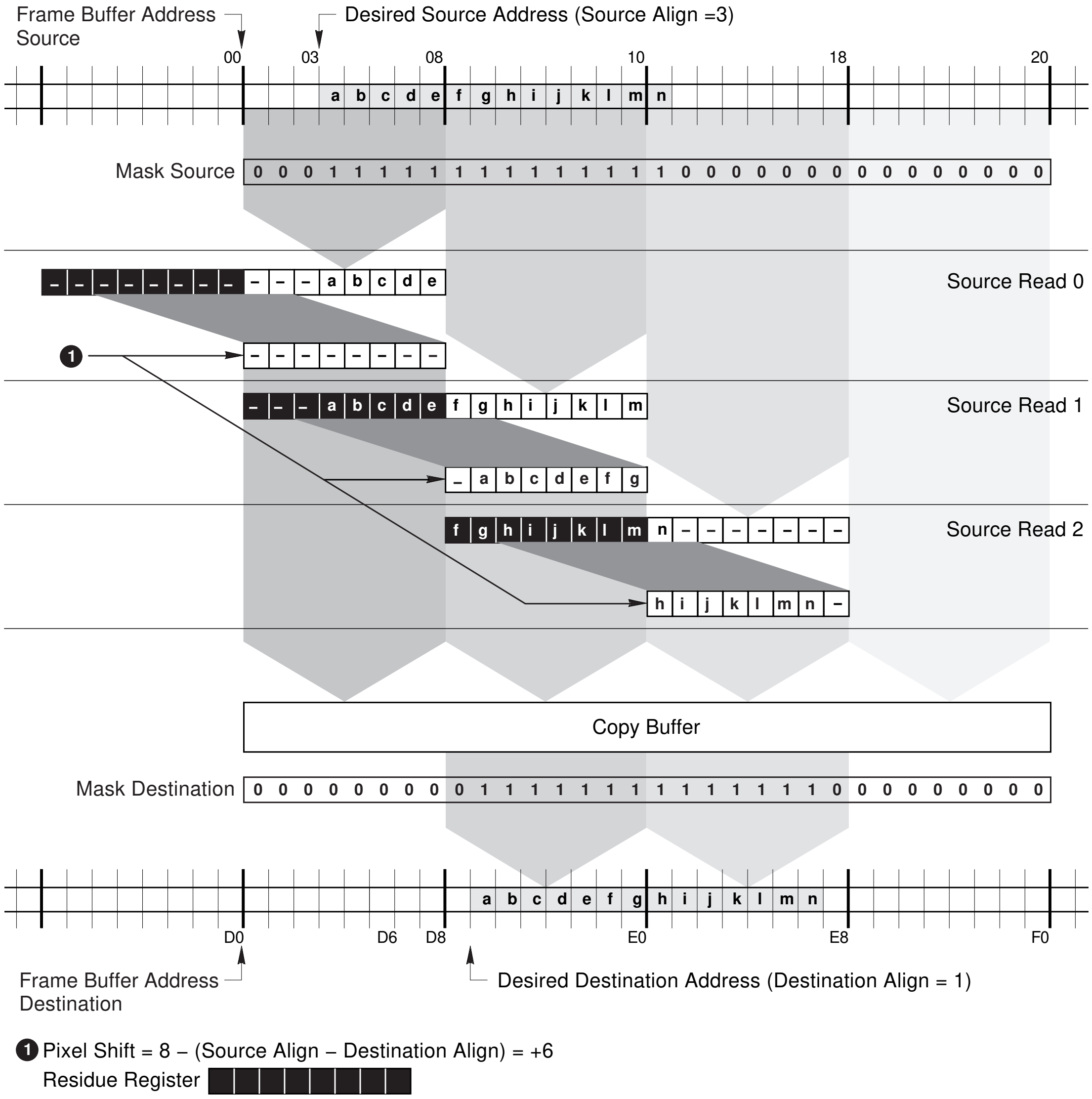
In either case, the first quadword written to the destination takes some bytes from both the first and second quadwords read from the source. The `Pixel Shift` value is set such that none of the valid pixels from the first read are rotated into the first quadword generated by the byte-shifter. The byte-shifter does not generate the proper quadword for the first destination quadword until the second source quadword is read. In effect, the first read primes the residue register, and every subsequent source read generates a valid destination quadword to store in the copy buffer. This amounts to a 1-stage read-data path pipeline.

To compensate for priming the residue register, software must adjust the `Frame Buffer Address Destination` and `Mask Destination` by an additional quadword. The `Frame Buffer Address Destination` must be decremented (forward copies) or incremented (backward copies) by 8 and the `Mask Destination` must be bit-shifted 8 bits to the right (forward) or left (backward) (Figure 6–24).

In some cases, including those in which priming is not required, the pipeline delay introduced by the residue register has a side effect. In such cases, the residue register must be flushed after the last unmasked quadword has been read, because it may contain leftover valid destination-pixels. Consider a span copy similar to that shown in Figure 6–23, but with the the source span extended 2 pixels to the right. In that case, the 21030 hardware flushes the residue register when necessary, to generate the last destination-quadword written into the copy buffer; software need not do anything special. However, residue-register priming and flushing must also be considered in the DMA-read and DMA-write copy modes. In those modes, flushing requires software intervention (see Sections 6.2.10 and 6.2.11 for more information).

Figure 6–24 is an example of a forward copy in which priming is necessary.

Figure 6–24 Primed Forward Span Copy



The following pseudo-code represents the basic algorithm for copying a span in copy mode, including source and destination alignment:

On PCI write 1:

```
for (i = 0; i <= 3; i++)
{
  Read Quadword (Frame Buffer Address Source, Mask Source <i*8+7:i*8>, Data Source,
                Source Bitmap, Source Byte);
  Byte Rotate (Data Source, Residue, Pixel Shift, Data Shift);
  Load Copy Buffer (Data Shift);
  Residue = Data Source;
  Frame Buffer Address Source += 8*Sign(Pixel Shift);
}
```

On PCI write 2:

```
for (i = 0; i <= 3; i++)
{
  Unload Copy Buffer (Data Out);
  Store Quadword (Frame Buffer Address Destination, Mask Destination<i*8+7:i*8), Data Out,
                 Destination Bitmap, Destination Byte);
  Frame Buffer Address Destination += 8*Sign(Pixel Shift);
}
```

The algorithm is for descriptive purposes and does not address all details of the copy-mode operation. For example, the 21030 does not necessarily read or write all four quadwords. The 21030 monitors leading and trailing zeros in `Mask Source` and `Mask Destination`, to save time when copying. The 21030 jumps to the first unmasked pixel to start reading, and terminates the read and copy-buffer fill after the last unmasked pixel.

6.2.9.4 Copy Direction Flag

In the copy-mode process, software does not pass an explicit parameter to indicate whether the address and mask parameters passed on a PCI write to the frame buffer correspond to the source or the destination. In the copy mode, the 21030 requires a strict ordering of alternating source-reads and destination-writes to the frame buffer, and uses the copy direction flag to indicate the next operation. The copy direction flag is a 2-state, internal, hardware pointer (GMOR <20>). The flag state, `Source Next` or `Destination Next`, determines whether the next incoming PCI write to the `Frame Buffer Address` space should trigger a read to, or a write from, the copy buffer.

Software neither reads the copy direction flag nor writes it directly. The flag is initialized to `Source Next` on a write to the GPSR or copy buffer. Therefore, when software sets the `Pixel Shift` parameter before starting the copy, the hardware is ready to read the first span. Each time software writes the frame buffer in the copy mode, the copy direction flag changes state. As long as software properly initializes the GPSR and alternates source-reads and destination-writes, the hardware always does the appropriate operation without explicit software control. If necessary, software can rewrite the GPSR to reset the copy direction flag.

6.2.9.5 64-Byte Unmasked Span Copies

The 32-bit masks passed in the copy mode limit the span to 32 bytes in the packed 8-bpp format. This uses only half of the 64-byte copy buffer. To overcome this limitation, the 21030 has a separate mechanism for copying spans of 64, unmasked, contiguous bytes (masked copies are not supported). In other words, this mechanism cannot be used to copy any span segment in which either the source or destination includes an edge that is not naturally aligned to an 8-byte boundary, because any such span must be masked.

Copying 64-byte spans involves a 2-stage operation: one PCI write to load the copy buffer starting at a specified `Frame Buffer Address Source` aligned to 8 bytes; and a second PCI write to unload the copy buffer starting at the `Frame Buffer Address Destination`. However in this case, rather than writing to the frame buffer, software writes the copy-64 source register (GCSR) to load the copy buffer, using the `Frame Buffer Address Source` as data. Similarly, to write the copy buffer contents to the frame buffer, software writes the copy-64 destination register (GCDR), using the `Frame Buffer Address Destination` as data.

Loading and unloading the copy buffer in this way always moves 64 bytes. Although the GCSR and GCDR are not normally used for span segments containing an edge, they can be used to fill interior span segments in a large copy operation, where the edge segments are copied using alternating writes to the frame buffer source and destination addresses.

6.2.9.6 Copy Buffer Operation

The 21030 copy buffer contains 8 quadword entries (Figure 6–25). The 21030 loads and unloads the copy buffer in copy-mode operations as follows:

- A write to the frame buffer in the copy mode with the copy direction flag pointing to `Source Next`

  For this operation, the 21030 loads the copy buffer with up to 8 quadwords from a 32-pixel span, starting at copy buffer entry0 and filling contiguously up to entry7. For 8-bpp frame buffers, a 32-pixel span consists of 32 bytes and only entries 0 through 3 are filled. The `Mask Source` parameter specifies which pixels in the span are enabled to be loaded into the copy buffer, but does not affect how each pixel is mapped to a copy-buffer entry. In effect, each pixel in the quadword-aligned source span is mapped to a specific byte (8-bpp) or Dword (32-bpp) of a specific entry. Zeros in the `Mask`

`Source` parameter affect only the leading and trailing ends of the span; the 21030 saves time by not reading pixels that will be masked.

For example, on a write to an 8-bpp frame buffer in the copy mode with a `Mask Source` value of FFFFFFFF, the 21030 loads all bytes in all copy-buffer entries, with the first pixel of the span loaded in the least significant byte of entry0 and the last pixel in the most significant byte of entry7. On the other hand, on a write with a `Mask Source` value of 00FFFF00, only entries 1 and 2 are filled.

- A write to the frame buffer in copy mode with the copy direction flag pointing to `Destination Next`

  For this operation, the 21030 unloads up to 8 quadwords from the copy buffer to a 32-pixel span, starting with entry0 and draining contiguously up to entry7. For 8-bpp frame buffers, a 32-pixel span consists of 32 bytes and only entries 0 through 3 are drained. On the write, `Mask Destination` bits enable each byte in an 8-bpp frame buffer and each Dword in a 32-bpp frame buffer. On the drain, the 21030 uses the `Mask Destination` to optimize frame buffer accesses, skipping leading and trailing zeros. The pixel masking does not affect how each copy buffer entry is mapped to the quadword; for example, entry6 is always mapped to the starting quadword-address + 7.

- A write to the copy-64 source register (GCSR)

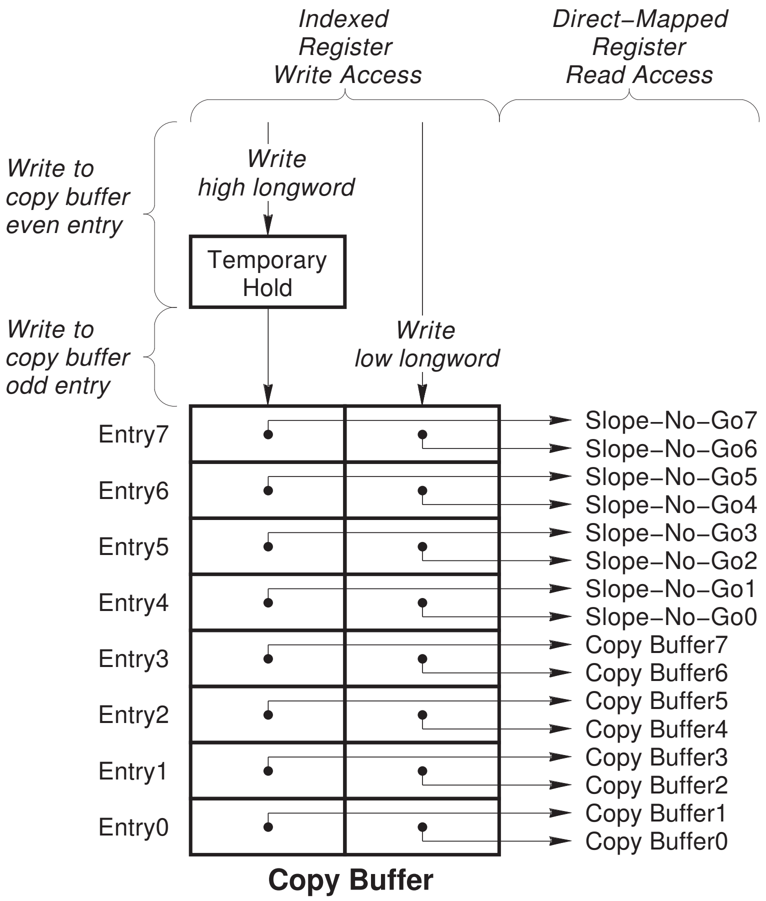
  For this operation, the 21030 fills the copy buffer with exactly 8 quadwords from a quadword-aligned address, starting with entry0 and filling contiguously up to entry7. Masking to read fewer than 8 quadwords is not done.

- A write to the copy-64 destination register (GCDR)

  For this operation, the 21030 drains exactly 8 quadwords from the copy buffer to a quadword-aligned address, starting with entry0 and draining contiguously up to entry7. Masking to write fewer than 8 quadwords is not done.

Figure 6–25 shows how the copy buffer registers (GCBR<7:0>) and slope-no-go registers (GSNR<7:0>) are mapped to the copy buffer entries.

Figure 6–25 Copy Buffer Layout



Programmed I/O Copy Buffer Operation

The copy buffer is also available for programmed I/O read and write operations. The host can sequentially fill or random-access-read all entries of the copy buffer through the GCBRs and GSNRs.

- Programmed I/O write to the copy buffer

  The host can write to the copy buffer only by filling sequentially, starting with entry0. A write to any even-numbered GCBR specifies, but does not load, the low Dword of the next empty copy-buffer entry. A subsequent write to any odd-numbered GCBR loads that Dword into the high Dword of the next entry and loads the previously specified Dword into the low Dword of the same entry. The effect of writes to a full copy is undefined.

- Programmed I/O read of the copy buffer

  Unlike programmed I/O writes, programmed I/O reads directly and randomly access individual halves of each quadword entry (Figure 6–25). The eight GSNRs are direct-mapped to copy buffer entries 7 through 4 and the eight GCBRs are direct-mapped to copy buffer entries 3 through 0. The GSNRs are mapped to the copy buffer only on reads.

6.2.9.7 Fast Frame Buffer Access Using the Copy Buffer Registers

The best way to copy back-and-forth between host and screen is to use the DMA-read and DMA-write copy modes. While DMA should provide the best performance for large operations, it incurs an appreciable amount of overhead that can make it inefficient for small regions. On the other hand, simple mode, particularly when reading, is too slow for extended, dumb frame buffer access or image transfer between the host and screen. However, the copy mode, in conjunction with direct software access to the copy buffer, allows localized regions in the frame buffer to be quickly read and written.

Standard screen-to-screen copies involve groups of two alternating PCI writes, either to the source and destination frame buffer addresses (the standard copy mechanism) or to the GCSR and GCDR. However, each write can individually load or unload the copy buffer from or to the frame buffer. Additionally, software can directly read and write the copy buffer (Section 6.2.9.6). For example, by interleaving writes to the GCSR and GCDR with programmed I/O reads and writes, software can rapidly transfer image data between host memory and the frame buffer.

To transfer a bitmap from host memory to the frame buffer, software can do the following:

1. Write the GCBRs.
2. Write the resulting contents of the copy buffer to the frame buffer with either a write to the destination address with the copy direction flag set to write `Destination Next` or a write to the GCDR.

Similarly, to transfer a bitmap from the frame buffer to host memory, software can load the copy buffer with either a write to the source address with the copy direction flag set to read `Source Next` or a write to the GCSR.

In either transfer, using the copy direction flag can be awkward, because the copy direction flag can be manipulated only through writes to the frame buffer and the GPSR.

6.2.9.8 Copy Mode Source and Destination Bitmaps

In the preceding sections, the description of the copy mode is limited to moving pixel data between bitmaps that have the same format. The examples in Figures 6–23 and 6–24 show copying data between packed 8-bpp bitmaps (PB8 format); copying between 24-bpp bitmaps is similar. However, the copy mode supports all types of source and destination bitmaps (Section 6.1.5). The only restriction is that the depth of the source and destination bitmaps must be the same. That is, spans copied from a 24-bpp bitmap source can only be written to a 24-bpp destination, a 12-bpp source can be specified only with a 12-bpp destination, and an 8-bpp source can be specified only with an 8-bpp destination.

In the copy mode, the source and destination bitmaps can be specified independently by setting some subset of the following parameters:

- `Source Bitmap` and `Destination Bitmap`
- `Source Byte` and `Destination Byte` (8-bpp formats only, otherwise ignored)
- `Plane Mask` (12-bpp formats only)

For each supported permutation, an associated `Source Bitmap` and `Destination Bitmap` encoding is assigned (Section 4.4.1). Software must specify the appropriate values for both bitmaps prior to initiating the copy. After the values have been specified, the `Source Byte` and the `Destination Byte` or `Plane Mask` values can be specified, based on the respective `Source Bitmap` and `Destination Bitmap` specified encodings, and the specific source and destination formats. (See Section 6.1.5 for more information about using the source and destination bitmap and byte fields.)

- 24-bpp bitmaps

  To copy from one 24-bpp bitmap to another, the `Source Bitmap` and `Destination Bitmap` values must be set to 011_2 and 11_2 respectively.

- 12-bpp bitmaps

  When copying 12-bpp spans between format $DC12_0$ and $DC12_1$ bitmaps, software must set the `Source Bitmap` value to specify the specific bitmap and the `Destination Bitmap` value to 10_2 (to specify a 12-bpp destination).

  However, the `Destination Bitmap` value does not explicitly specify one of the two possible formats. Instead, the 21030 extracts the 4:4:4 (R:G:B) pixel data from the specific source bitmap and replicates it into both of the possible destination bitmap formats (Section 6.1.6). Consequently, software needs to set only the `Plane Mask` to write-enable only the desired bitmap.

Table 6–20 defines the `Source Bitmap`, `Destination Bitmap`, and typical `Plane Mask` settings for each permutation of 12-bpp copies.

Table 6–20 Format Parameters for 12-bpp Bitmaps

Source Format	Destination Format	Source Bitmap Value*	Destination Bitmap Value*	Typical Plane Mask Value
$DC12_0$	$DC12_0$	010	10	000F0F0F
$DC12_0$	$DC12_1$	010	10	00F0F0F0
$DC12_1$	$DC12_0$	110	10	000F0F0F
$DC12_1$	$DC12_1$	110	10	00F0F0F0

*Binary

The `Source Byte` and `Destination Byte` parameters do not apply and are ignored when using 12-bpp bitmaps.

- 8-bpp bitmaps

  In the copy mode, the appropriate values of the `Source Bitmap`, `Source Byte`, `Destination Bitmap`, and `Destination Byte` parameters support any combination of 8-bpp source and destination.

  Table 6–21 shows all possible permutations of 8-bpp source and destination bitmaps within the context of a 32-bpp frame buffer. The 21030 writes only to the destination bitmap as specified by the `Destination Byte` parameter; therefore, the `Plane Mask` parameter should not be used to specify the destination. The `Plane Mask` parameter can be used to mask individual pixel bits for unpacked 8-bpp bitmaps, in the same way that it is used for packed bitmaps.

  Unpacked 8-bpp bitmaps referenced as a source or destination require extra attention in terms of `Frame Buffer Address Source` and `Frame Buffer Address Destination` alignment. Unlike all other bitmap formats (which require address alignment to 8 bytes), 8-bpp unpacked bitmaps require alignment to 32 bytes.

Table 6–21 Format Parameters for 8-bpp Bitmaps in a 32-bpp Frame Buffer

Source Format	Destination Format	Source Bitmap Value*	Source Byte Value*	Destination Bitmap Value*	Destination Byte Value*
$UB8_0$	$UB8_0$	001	00	01	00
$UB8_0$	$UB8_1$	001	00	01	01
$UB8_0$	$UB8_2$	001	00	01	10
$UB8_1$	$UB8_0$	001	01	01	00
$UB8_1$	$UB8_1$	001	01	01	01
$UB8_1$	$UB8_2$	001	01	01	10
$UB8_2$	$UB8_0$	001	10	01	00
$UB8_2$	$UB8_1$	001	10	01	01
$UB8_2$	$UB8_2$	001	10	01	10
$UB8_0$	PB8	001	00	00	00
$UB8_1$	PB8	001	01	00	00
$UB8_2$	PB8	001	10	00	00
PB8	$UB8_0$	000	00	01	00
PB8	$UB8_1$	000	00	01	01
PB8	$UB8_2$	000	00	01	10
PB8	PB8	000	00	00	00

*Binary

6.2.10 DMA-Read Copy Mode

In the DMA-read copy mode, a PCI write to the frame buffer address space copies a contiguous span of up to 2K Dwords (8KB) from external PCI memory to the frame buffer. The 21030 copies the span as a function of the parameters listed in Table 6–22.

```
DMA Read Copy Span (DMA Address, Frame Buffer Address Destination, Read Count (-1),
                    Mask Left <1:0>, Mask Right <1:0>, Plane Mask,
                    Pixel Shift, Raster Op, Destination Bitmap, Destination Byte)
```

Table 6–22 DMA-Read Copy-Mode Parameters

Parameter	Source		Section
`Frame Buffer Address Destination`	PCI write address	—	—
`Read Count (-1)`	PCI write data	<31:16>	—
`Mask Left 0`	PCI write data	<11:8>	—
`Mask Left 1`	PCI write data	<15:12>	—
`Mask Right 0`	PCI write data	<3:0>	—
`Mask Right 1`	PCI write data	<7:4>	—
`DMA Address`	DMA base address register	GDBR <31:0>	4.4.7
`Pixel Shift`	Pixel shift register	GPSR <3:0>	4.4.5
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Source Bitmap`	Mode register	GMOR <10:8>	4.4.1
`Source Byte`	Mode register	GMOR <12:11>	4.4.1
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle initiates a DMA-read copy of one span from PCI external memory into the frame buffer. The PCI write addresses the location of the destination span (`Frame Buffer Address Destination`). The PCI write data consists of a Dword `Read Count` and four read masks for the destination. Figure 6–26 shows the format of the PCI write data.

Figure 6–26 DMA-Read Copy-Mode PCI Write-Data Format

PCI Data

31 27	26 16	15 12	11 8	7 4	3 0
IGN	Read Count (–1)	Mask Right 1	Mask Right 0	Mask Left 1	Mask Left 0

PCI Byte Mask

3 0
IGN

On the PCI write, the 21030 requests and then masters the PCI bus. It then reads `Read Count` Dwords from PCI external memory starting at the `DMA Address` and writes the Dwords to the frame buffer, starting at the `Frame Buffer Address`. On each successful transfer, the 21030 reads and writes one full Dword as follows:

- First Dword
  1. Reads the Dword from PCI external memory.
  2. Writes the Dword to the frame buffer at the `Frame Buffer Address Destination`. On the write, `Mask Left 0` masks the individual bytes of the first Dword.
  3. Decrements the `Read Count`.
- Second Dword
  4. Reads the Dword from PCI external memory.
  5. Writes the Dword to the frame buffer at the next frame buffer address. On the write, `Mask Left 1` masks the individual bytes of the second Dword.
  6. Updates the frame buffer address.
  7. Decrements the `Read Count`.
- Third Dword through next-to-last Dword
  8. Reads the Dword from PCI external memory.
  9. Writes the Dword to the frame buffer at the next frame buffer address. No bytes are masked.
  10. Updates the frame buffer address.
  11. Decrements the `Read Count`.

- Last Dword
  12. Reads the Dword from PCI external memory.
  13. Writes the Dword to the frame buffer at the next frame buffer address. On the write, `Mask Right 0` masks the individual bytes of the last Dword.
  14. Updates the frame buffer address.
  15. Decrements the `Read Count`.
- After the last Dword is read and written, the 21030 writes the contents of the residue register to the frame buffer at the next frame buffer address, masked by `Mask Right 1`.
- For each write to the frame buffer destination, the 21030 also executes the specified `Raster Op` and filters data through the `Plane Mask`.

The DMA-read copy mode is functionally similar to the copy mode. However, the DMA-read copy mode differs in the following ways:

- Addresses are aligned to Dword (4 bytes) rather than quadword (8 bytes).
- The copy source is located in PCI external memory.
- External memory does not support all bitmap formats.
- The span can contain as many as 2K Dwords.
- The PCI write data passes two sets of mask data to mask the span's left and right edges (per-pixel masking is not allowed).
- The source can be optionally dithered.

The DMA-read copy operation can be considered to be a copy-mode operation in which the source is accessed across the PCI bus and the granularity of the operation is 32 bits rather than 64 bits. Both the `Frame Buffer Address Destination` and `DMA Address` must be aligned to 4 bytes. The process of reading the source, rotating using the residue register, and writing the destination occurs in groups of 4 bytes rather than 8 bytes.

Because the `Frame Buffer Address Destination` and the `DMA Address` must be aligned to 4 bytes, software must adjust the desired source and destination addresses and masks for unaligned copies to the next whole Dword. However, all bitmaps, except packed 8-bpp bitmaps, are naturally aligned to 4 bytes.

Each Dword read from PCI external memory is concatenated with a 32-bit version of the residue register, and rotated by `Pixel Shift` bytes to produce the destination Dword written to the frame buffer. In the DMA-read copy mode, the `Pixel Shift` is calculated as in the copy mode (Table 6–19). However, unlike the copy mode, the `Pixel Shift` value range is 0 to +3, because backward copies are unnecessary and the granularity is 4 bytes rather than 8 bytes.

In the DMA-read and DMA-write copy modes, the copy buffer is not used, and the destination Dword is written directly to the frame buffer, using the specified `Raster Op` and `Plane Mask`. Residue-register priming and flushing is similar to the copy mode (Section 6.2.9).

6.2.10.1 Priming and Flushing the Residue Register

The two left-edge masks compensate for residue-register priming. For the copy alignments that require residue register priming, the following occur:

- The `Frame Buffer Address Destination` is decremented one Dword (that is, the destination span's left edge is extended 4 bytes).
- `Mask Left 0` masks out the additional Dword.
- `Mask Left 1` contains the desired edge mask.

For alignments that do not require residue-register priming, `Mask Left 0` usually contains the desired edge mask and `Mask Left 1` is set to 1111_2.

The two right-edge masks compensate for copy alignments that require-residue register flushing. As in the copy mode, the pipelined nature of the source-read data path causes valid source data to remain in the residue register under certain conditions. Depending on the alignment and location of the span's right edge, this also applies to the DMA-read copy mode. But unlike the copy mode, the DMA-read copy mode requires explicit software attention to flush the residue register. Specifically, explicit residue-register flushing is required for alignments in which `Source Aligned` and `Destination Aligned` are the desired address alignments of the end of the span and one of the following is true:

`Source Align > Destination Align` and `Source Aligned < Destination Aligned`

or

`Source Align < Destination Align` and `Source Aligned > Destination Aligned`

To flush the residue register in such cases, `Mask Right 1` contains the desired edge mask and `Mask Right 0` is set to 1111_2.

Table 6–23 shows how the four edge-mask parameters are set according to the requirement to prime and flush the residue register.

Table 6–23 Edge Mask Settings in DMA-Read Copy Mode

Residue Register	Mask Left 0	Mask Left 1	Mask Right 0	Mask Right 1
Prime	0000_2	Left-edge mask	—	—
No prime	Left-edge mask	1111_2	—	—
Flush	—	—	1111_2	Right-edge mask
No flush	—	—	Right-edge mask	0000_2

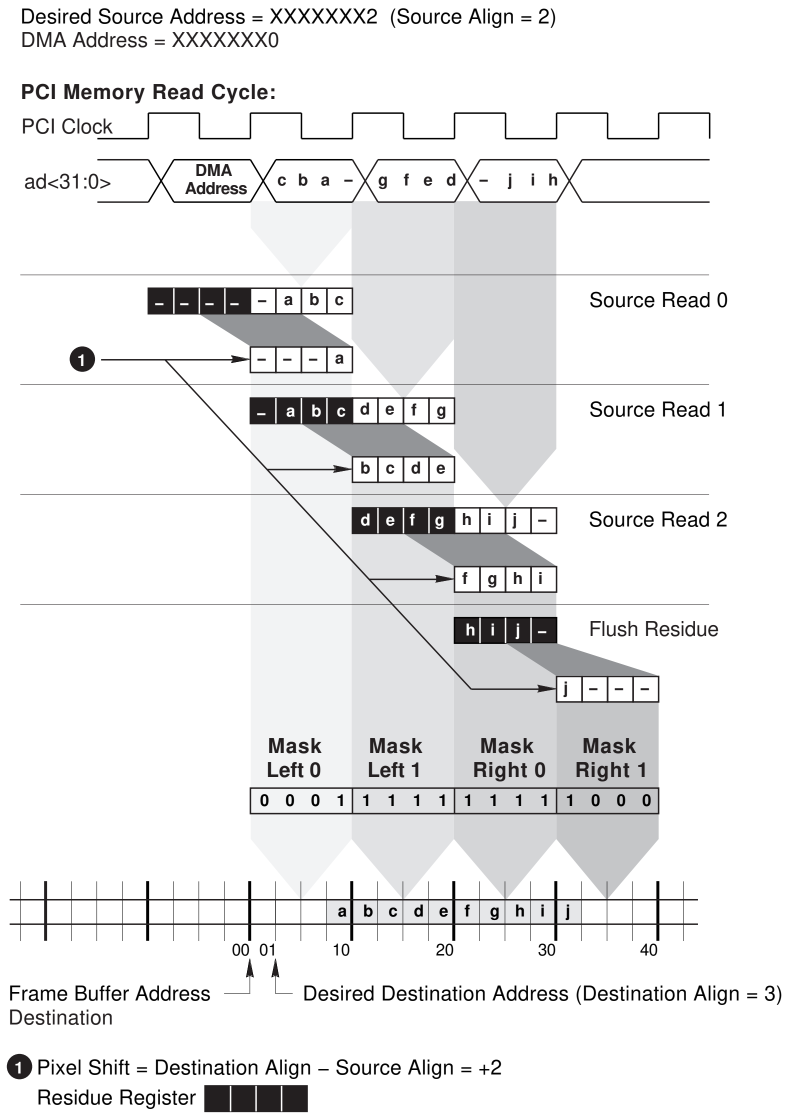
For short spans, in which fewer than 3 Dwords are read across the PCI, all edge masks are not used. (However, the DMA copy modes are seldom used to copy such small spans and the limitation can usually be ignored.) Table 6–24 lists the masks used for such spans.

Table 6–24 Edge Mask for Short Spans in DMA-Read Copy Mode

Read Count	Mask Left 0	Mask Left 1	Mask Right 0	Mask Right 1
≥ 3	Yes	Yes	Yes	Yes
2	Yes	No	Yes	Yes
1	No	No	Yes	Yes

Figure 6–27 is an example of a packed, 8-bpp, short span copied to the frame buffer over the PCI bus in DMA-read copy mode. The alignment requires an extra frame buffer write to flush the residue register. For descriptive purposes, the PCI cycle shown assumes a fast target response with no read latency.

Figure 6–27 DMA-Read Copy



6.2.10.2 Bitmap Formats Supported in DMA-Read Copy Mode

Unlike the copy mode, the DMA-read copy mode does not support all 21030 bitmap formats. Specifically, the formats listed below are not supported as an external memory source bitmap. This is because a bitmap in PCI memory cannot be accessed in the same way as a frame buffer bitmap. `Source Bitmap` and `Source Byte` can be set as described in Section 6.2.9.8, with the following source formats not allowed:

- $UB8_0$
- $UB8_1$
- $UB8_2$
- $DC12_0$

Note that in each of the disallowed formats, each pixel occupies one Dword; therefore, the unsupported formats could be read from a 24-bpp source. `Source Bitmap` and `Source Byte` are effectively ignored in the DMA-read copy mode.

In the DMA-read copy mode, all 21030 bitmap formats can be specified as frame buffer destination bitmaps with `Destination Bitmap` and `Destination Byte`. When accessing unpacked 8-bpp destination bitmaps $UB8_0$, $UB8_1$ or $UB8_2$, the `Frame Buffer Address Destination` must be aligned to 16 bytes rather than 4 bytes (Dword) as required by all other formats.

Although the DMA-read copy mode does not support the $DC12_0$ 12-bpp format in external memory, it does support the $DC12_0$ and $DC12_1$ formats in the frame buffer. If the `Destination Bitmap` specifies a 12-bpp format, the 21030 replicates the high nibble of each byte to the low nibble; that is, it replicates $DC12_0$ to $DC12_1$. Therefore, software must set the `Plane Mask` to write only the specified format.

6.2.10.3 Dithering in DMA-Read Copy Mode

The mode field (GMOR <6:0>) enables DMA-read copy mode in two ways:

- The source is dithered to the destination.
- Dithering is disabled.

When dithering is enabled, the 21030 expects the source to be a TC24 format bitmap (Figure 6–4). The 21030 dithers from the 24-bpp source to the destination format specified by the `Destination Bitmap` parameter. To allow the 21030 dithering logic to address the dither matrix, software must initialize the dither row and dither column values in the red- and green-value registers (GRVR <31:27>, Section 4.4.22 and GGVR <31:27>, Section 4.4.24).

6.2.11 DMA-Write Copy Mode

In the DMA-write copy mode, a PCI write to the frame buffer address space copies a contiguous span of up to 2K quadwords (16KB) from the frame buffer to the external PCI memory. The 21030 copies the span as a function of the parameters listed in Table 6–25.

```
DMA Write Copy Span (DMA Address, Frame Buffer Address Source, Read Count (-1),
                     Mask Left, Mask Right, AND Mask, Source Bitmap, Source Byte,
                     Destination Bitmap, Destination Byte);
```

Table 6–25 DMA-Write Copy-Mode Parameters

Parameter	Source		Section
`Frame Buffer Address Source`	PCI write address	—	—
`Read Count (-1)`	PCI write data	<31:16>	—
`Mask Left`	PCI write data	<15:8>	—
`Mask Right`	PCI write data	<7:0>	—
`DMA Address`	DMA base address register	GDBR <31:0>	4.4.7
`Pixel Shift`	Pixel shift register	GPSR <3:0>	4.4.5
`AND Mask`	Data register	GDAR <31:0>	4.4.8
`Source Bitmap`	Mode register	GMOR <10:8>	4.4.1
`Source Byte`	Mode register	GMOR <12:11>	4.4.1
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3

The PCI write cycle initiates a DMA-write copy of one span from the 21030 frame buffer to PCI external memory. The PCI write addresses the location of the source span (`Frame Buffer Address Source`). The PCI write data consists of a read count and two read masks (`Read Count`, `Mask Left`, and `Mask Right`) for the destination. Figure 6–28 shows the format of the PCI write data.

Figure 6–28 DMA-Write Copy-Mode PCI Write-Data Format

PCI Data

31–27	26–16	15–8	7–0
IGN	Read Count (–1)	Mask Left	Mask Right

PCI Byte Mask

3–0
IGN

On the PCI write, the 21030 reads `Read Count` quadwords from the frame buffer beginning at the `Frame Buffer Address Source`. Each quadword read is selectively shifted, then written one Dword at a time over the PCI bus, beginning at the `DMA Address`. `Mask Left` is the left-edge byte mask and `Mask Right` is the right-edge mask data. As in the DMA-read copy mode, only forward copies are necessary — backward copies are not supported.

In general, the DMA-write copy mode behaves as the inverse of the DMA-read copy mode, with the following major differences:

- The source bitmap address (`Frame Buffer Address Source`) must be quadword-aligned, except for unpacked 8-bpp source bitmaps, which must be aligned to 32 bytes (Section 6.2.10.2).
- The `Frame Buffer Address Source` must be adjusted to always prime the residue register.
- If residue-register flushing is required, the `Read Count` must be adjusted to read an additional quadword.

The 21030 reads the first quadword from the `Frame Buffer Address Source`, concatenates it with the residue register, and rotates the result by `Pixel Shift` bytes (as in the copy mode). It then decrements `Read Count`. The first quadword generated by the first read-concatenate-rotate operation is discarded because the 21030 always assumes that the first quadword primes the residue register, whether or not the particular alignment requires it. Consequently, software must conditionally decrement the `Frame Buffer Address Source` by 8 for alignments that do not require residue-register priming.

The 21030 repeats the process for the second quadword and again decrements `Read Count`, but this time it writes the quadword generated by the rotate operation through `Mask Left` onto the PCI bus. Prior to writing, the 21030 masters the PCI bus, drives the `DMA Address`, and sets up a memory-write cycle to burst Dwords until the last quadword is read from the frame buffer. (See Section 5.3 for more information about the 21030's behavior as a PCI master.)

The 21030 maps 1 bit of Mask Left to each byte of the first quadword written. Because the PCI is 1 Dword (4 bytes) wide, the 21030 processes one-half of the quadword at a time, as follows:

- If the lower nibble (<3:0>) of Mask Left is nonzero, the 21030 ignores the low Dword and moves on to the second Dword.
- If the lower nibble of Mask Left is zero, the 21030 transfers the low Dword over the PCI bus, specifying Mask Left <3:0> as the data byte enable.
- The 21030 then transfers the high Dword, specifying Mask Left <7:4> as the data byte enable.

For each Dword successfully transferred, the 21030 decrements its internal copy of the current PCI address. The 21030 monitors the PCI address because the burst can be terminated on any cycle. If the burst is terminated, the 21030 must reacquire the bus and issue the address of the next untransferred Dword, to resume the burst transfer.

After the second source quadword has been processed, and one or both Dwords written, the 21030 repeatedly reads quadwords until the Read Count = 1. For each quadword read, the 21030 performs the rotate operation with the residue register, decrements the Read Count, and writes both Dwords of the destination quadword over the PCI, twice incrementing its copy of the current PCI address. (Note that when writing to a destination across the PCI, the Raster Op and Plane Mask parameters are not used.)

Finally, the 21030 reads the last source-quadword from the frame buffer and performs the concatenate and rotate operation. It then writes the destination quadword through Mask Right onto the PCI bus. Again, each bit of Mask Right corresponds to 1 byte of the quadword. The 21030 writes 1 Dword at a time, starting with the lower Dword; however, if the upper nibble of Mask Right is zero, the 21030 does not write the last Dword. The 21030 then terminates the PCI memory write cycle, completing the DMA-write copy operation.

The following pseudo-code represents the basic algorithm for the DMA-write copy operation:

```
/* process first quadword and ignore (priming) */
Read Quadword (Frame Buffer Address Source, Quadin, Source Bitmap, Source Byte);
Byte Shifter (Quadin, Residue, Pixel Shift, Quadout);
Read Count-- ;
Frame Buffer Address Source += 8;
/* process second quadword and write through left edge mask*/
Read Quadword (Frame Buffer Address Source, Quadin, Source Bitmap, Source Byte);
Byte Shifter (Quadin, Residue, Pixel Shift, Quadout);
Frame Buffer Address Source += 8;
if (Mask Left<3:0> != 0)
{
    Write PCI (Quadout<31:0>, Mask Left<3:0>);
    Read Count-- ;
}
Write PCI (Quadout<63:32>, Mask Left<7:4>);
/* process middle of span without masks*/
while (Read Count > 1)
{
     Read Quadword (Frame Buffer Address Source, Quadin, Source Bitmap, Source Byte);
     Byte Shifter (Quadin, Residue, Pixel Shift, Quadout);
     Write PCI (Quadout<31:0>, Mask Right<3:0>);
     Write PCI (Quadout<63:32>, Mask Right<7:4>);
     Frame Buffer Address Source += 8;
     Read Count-- ;
}
/* process last quadword and write through right edge mask*/
Read Quadword (Frame Buffer Address Source, Quadin, Source Bitmap, Source Byte);
Byte Shifter (Quadin, Residue, Pixel Shift, Quadout);
Write PCI (Quadout<31:0>, Mask Right<3:0>);
if (Mask Right<7:4> != 0)
{
    Write PCI (Quadout<63:32>, Mask Right<7:4>);
    Read Count-- ;
}
```

The preceding code does not necessarily indicate the exact behavior of the 21030 hardware. For example, when copying short spans, in which fewer than 3 Dwords are read across the PCI, all edge masks are not used. Table 6–26 lists the masks used for such spans. (However, the DMA-read and DMA-write copy modes are seldom used to copy such small spans and the limitation can usually be ignored.)

Table 6–26 Edge Mask for Short Spans in DMA-Write Copy Mode

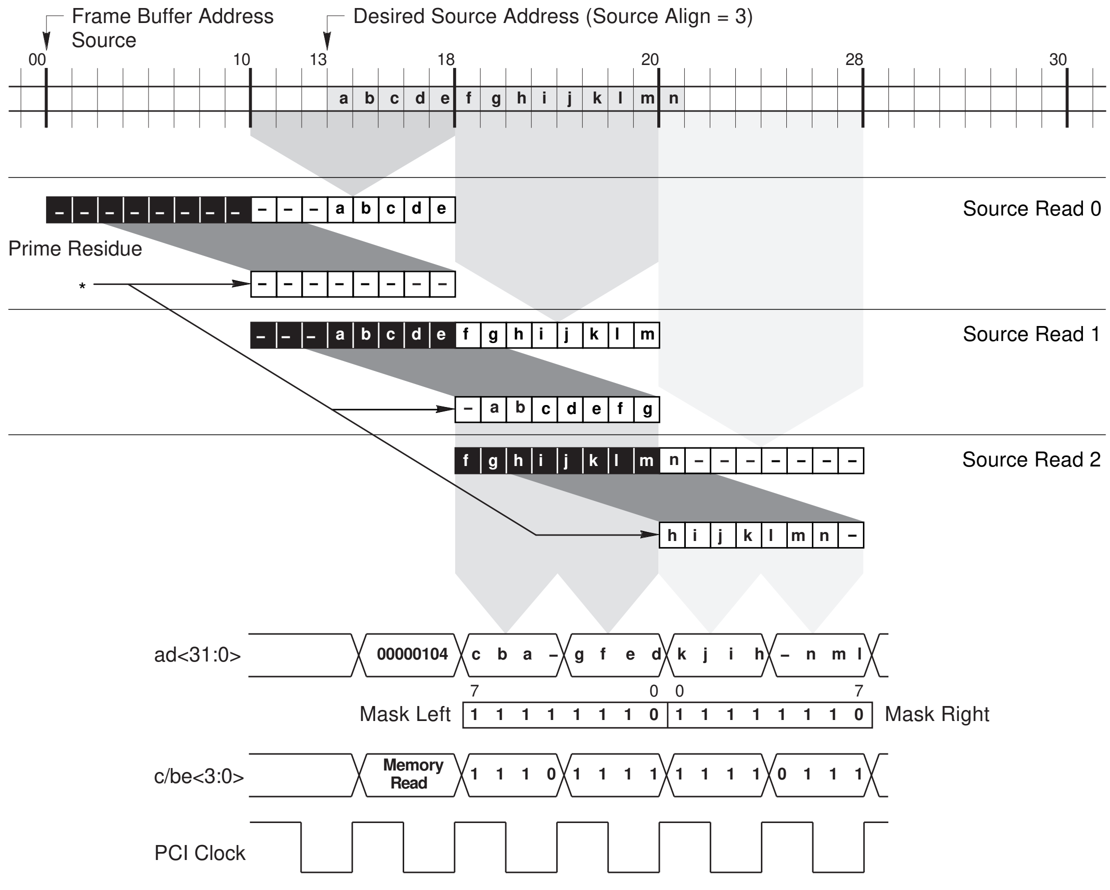
Read Count	Mask Left	Mask Right
≥ 3	Yes	Yes
2	Yes	No
1	No	No

6.2.11.1 Priming and Flushing the Residue Register

Because the hardware does nothing special to flush the residue register, alignments that need residue register flushing require software to increment the `Read Count` by 1 to force an extra read to flush the residue register.

Figure 6–29 is an example of a DMA-write copy mode operation. In the example, the alignments require residue-register priming; therefore, the `Frame Buffer Address Source` requires no backward adjustment. Because residue-register flushing is not required, the `Read Count` is not incremented.

Figure 6–29 DMA-Write Copy



Desired Destination Address = 00000105 (Destination Align = 1)
DMA Address = 00000104 (Dword Aligned)

* Pixel Shift = 8 – (Source Align – Destination Align) = +6
Residue Register

6.2.11.2 Bitmap Formats Supported in DMA-Write Copy Mode

Similar to the DMA-read copy mode, the DMA-write copy mode does not support all 21030 bitmap formats. Specifically, it does not support any of the formats listed below as an external-memory destination bitmap. `Destination Bitmap` and `Destination Byte` can be set as described in Section 6.2.9.8, with the following destinations not allowed:

- $UB8_0$
- $UB8_1$
- $UB8_2$
- $DC12_0$

In the DMA-write copy mode, all 21030 bitmap formats can be specified as frame buffer source bitmaps with `Source Bitmap` and `Source Byte`. When accessing unpacked 8-bpp source bitmaps $UB8_0$, $UB8_1$ or $UB8_2$, the `Frame Buffer Address Source` must be aligned to 32 bytes rather than 8 bytes as required by all other bitmap formats.

When copying to destinations across the PCI bus, normal plane masking is not available. In its place, the 21030 provides the `AND Mask` from the GDAR. The `AND Mask` can be used to clear bits in the destination-Dword as it is written. For example, on a write to a format $DC12_1$ destination bitmap, software can specify a mask value of 00F0F0F0 to zero the inactive fields.

The GDAR can be modified in the DMA-write copy mode only when the chip is idle. Before writing the GDAR, software should wait for busy to be deasserted in the command and status register (SCSR <0>, Section 4.7.1).

6.2.12 Opaque-Line Mode

Note

The first part of this description is included for continuity. It describes opaque-line mode operations initiated by the standard frame buffer write mechanism. However, the same functionality is more efficiently implemented with the alternate slope register write mechanism described in Section 6.2.12.1.

In the opaque-line mode, a PCI write to the frame buffer address space draws a masked, 16-pixel, bitonal line segment starting at the specified address. For this description, a *line segment* is defined as a string of 16 contiguous pixels drawn along an arbitrary slope; a *line* is made up of multiple segments and its length is arbitrary.

The 21030 draws the line segment as a function of the parameters listed in Table 6–27.

```
Opaque Line (Frame Buffer Address, Frame Buffer Address <1:0>, Line Mask, Raster Op,
       Plane Mask, Foreground, Background, Address Increment 1,
       Address Increment 2, Error Increment 1, Error Increment 2, Initial Error,
       Length, Destination Bitmap, Destination Byte. Cap Ends);
```

Table 6–27 Opaque-Line Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Frame Buffer Address <1:0>`	PCI write data	<17:16>	—
`Line Mask`	PCI write data	<15:0>	—
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Background`	Background register	GBGR <31:0>	4.4.19
`Address Increment 1`	Bresenham 1 register	GB1R <31:16>	4.4.10
`Error Increment 1`	Bresenham 1 register	GB1R <15:0>	4.4.10
`Address Increment 2`	Bresenham 2 register	GB2R <31:16>	4.4.11
`Error Increment 2`	Bresenham 2 register	GB2R <15:0>	4.4.11

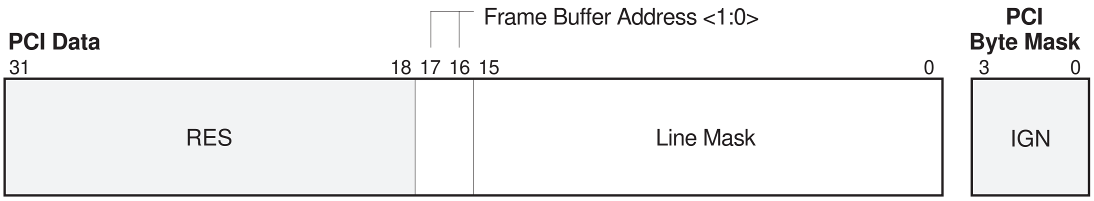
(continued on next page)

Table 6–27 (Cont.) Opaque-Line Mode Parameters

Parameter	Source		Section
`Initial Error`	Bresenham 3 register	GB3R <31:16>	4.4.12
`Length`	Bresenham 3 register	GB3R <3:0>	4.4.12
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3
`Cap Ends`	Mode register	GMOR <15>	4.4.1

The PCI write cycle initiates the line segment drawing operation to the 21030 frame buffer. The PCI write addresses the start of the segment (`Frame Buffer Address`) and passes as data the two address LSBs (`Frame Buffer Address <1:0>`) and a 16-bit `Line Mask` to pattern the line. Figure 6–30 shows the format of the PCI write data.

Figure 6–30 Opaque-Line Mode PCI Write-Data Format



The `Frame Buffer Address` must be aligned to 1 pixel. For drawing to packed 8-bpp bitmaps, the two LSBs of the frame buffer address (`Frame Buffer Address <1:0>`) are part of the PCI write data. For drawing to any other bitmap, the address is Dword-aligned (by default) and `Frame Buffer Address <1:0>` is ignored.

Before writing to the frame buffer in a line mode, software must ensure that the `Address Increment`, `Error Increment`, `Length`, and `Initial Error` values stored in the Bresenham registers are appropriate to the slope, octant, and length of the line segment. Software can write these parameters directly or initialize them indirectly by writing the GSLRs or GSNRs (Section 6.2.12.1).

Before starting to draw the line segment, the 21030 uses the `Frame Buffer Address` (concatenated with `Frame Buffer Address <1:0>`, if necessary) to initialize the address stored in its Bresenham engine.

The 21030 draws a line segment as follows:

1. To draw the first pixel, the 21030 checks the bits from the `Line Mask` as follows:
   - If the first `Line Mask` bit = 1, `Foreground` color is written.
   - If the first `Line Mask` bit = 0, `Background` color is written.
2. On any write in opaque-line mode, the 21030 does the specified `Raster Op` and uses `Plane Mask` to mask the writes to individual pixel bits.
3. The Bresenham engine then takes one step along the line, as follows:
   - If the current error term is <0, the engine adds `Address Increment 1` to the current address and adds `Error Increment 1` to the current error term to take one step along the major axis of the line segment.
   - If the current error term is ≥0, the engine adds `Address Increment 2` to the current address and subtracts `Error Increment 2` from the current error term to take one step along the major and minor axes of the line segment.
4. The 21030 then decrements `Length` and repeats the process for each pixel along the line, until the segment `Length` = 0. Once initialized by `Frame Buffer Address <1:0>`, the 21030 internally monitors which Dword-byte is to be written to a packed 8-bpp bitmap as it steps through the line.

The following pseudo-code represents the basic algorithm for opaque-line mode:

```
while (Length > 0)
{
    Pixel = (Extract Bit(Line Mask,Length)) ? Foreground : Background;
    Write Frame Buffer(Frame Buffer Address, Pixel, Raster Op, Plane Mask,
                       Destination Bitmap, Destination Byte);
    /* Bresenham step along line */
    if (Error < 0)
    {
        Frame Buffer Address += Address Increment 1;
        Error += Error Increment 1;
    }
    else
    {
        Frame Buffer Address += Address Increment 2;
        Error -= Error Increment 2;
    }
    Length --;
}
```

6.2.12.1 Drawing Lines with the Slope Registers

Drawing lines as described in the preceding sections results in a bottleneck for the following reasons:

- Overall line throughput in the CPU or I/O is slow, due to the software overhead incurred in setting up and writing all of the Bresenham address and error terms for each line.
- The 21030's high-performance, 64-bit memory bus can draw at a rate faster than an Alpha AXP CPU can supply commands and data.

To avoid bottlenecks, the GSLRs are a faster and simpler mechanism for drawing lines with less computation and fewer writes.

Table 6–28 is the modified list of parameters used in drawing lines with the GSLRs.

Table 6–28 Opaque-Line Mode Parameters Using Slope Registers

Parameter	Source		Section
`Absolute Dy`	Slope register	GSLR*n* <31:16>	4.3.1
`Absolute Dx`	Slope register	GSLR*n* <15:0>	4.3.1
`Frame Buffer Address`	Address register	GADR <31:0>	4.4.2
`Line Mask`	Data register	GDAR <15:0>	4.4.8
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Background`	Background register	GBGR <31:0>	4.4.19
`Bitmap Width`	Bresenham width register	GBWR <15:0>	4.4.13
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3
`Cap Ends`	Mode register	GMOR <15>	4.4.1
`Deep`	Deep register	GDER <0>	4.4.28

Drawing lines with the GSLRs is similar to the standard line drawing mechanism, with the following exceptions:

- A write to a GSLR, rather than to the frame buffer, initiates the drawing operation.
- The address and line-mask data are specified in registers.

- Software must initialize the Bresenham width register (GBWR, Section 4.4.13) instead of the GB1R, GB2R and GB3R registers.

Each GSLR corresponds to one drawing octant and contains two 16-bit fields, one for the absolute value of the slope rise (`Absolute Dy`) and the other for the absolute value of the slope run (`Absolute Dx`).

On a write to a GSLR, the 21030 calculates the Bresenham terms and then starts the standard Bresenham line drawing algorithm. Because the PCI write that initiates the drawing operation addresses a GSLR and passes slope information as data, the `Frame Buffer Address` and `Line Mask` parameters are specified in the GADR and GDAR, rather than in the PCI write data.

Given the slope and octant information, the 21030 does all of the Bresenham setup. It calculates all of the Bresenham error and address terms and stores them in the appropriate Bresenham register fields. The 21030 implements a slightly different setup algorithm depending on whether the line must comply with Win32 or be compatible with existing Digital conventions for lines drawn under X.

The following pseudo-code represents the basic hardware setup algorithm:

```
Pixel Bytes = (deep ? 4 : 1);
dxGEdy = (Absolute Dx >= Absolute Dy);
dxGE0 = (Absolute Dx > 0);
dyGE0 = (Absolute Dy > 0);
dmajor = (dxGEdy ? Absolute Dx : Absolute Dy)
dminor = (dxGEdy ? Absolute Dy : Absolute Dx)
majorGE0 = (dxGEdy ? dxGE0 : dyGE0);
minorGE0 = (dxGEdy ? dyGE0 : dxGE0);
amajor = (dxGEdy ? PixelBytes : BitmapWidth);
aminor = (dxGEdy ? BitmapWidth : PixelBytes);
if Graphics Environment
     {
     errinc = (dxGEdy ? dyGE0 : !dxGE0);
     }
else
     {
    errinc = majorGE0;
     }
/* Initial Bresenham terms */
Length = dmajor + Cap Ends mod16;
Error Increment 1 = dminor;
Error Increment 2 = dmajor + ~dminor + 1;
Initial Error = ((dminor<<1) + ~dmajor + errinc) >>1;
Address Increment 1 = (majorGE0 ? amajor : ~amajor + 1);
Address Increment 2 = (minorGE0 ? aminor : ~aminor + 1) + Address Increment 1;
```

`Cap Ends` and `Deep` are additional parameters specified in GMOR and GDER, respectively. Results are undefined if `Absolute Dx` and `Absolute Dy` are both set to zero.

Note that `Length` is set to the major axis length MOD 16. Therefore, the 21030 draws up to, but not necessarily exactly, 16 pixels when a GSLR is written. For example, if the major axis length is 19, writing a GSLR causes a 3-pixel line to be drawn (assuming `Cap Ends` is set).

Because Win32 has strict requirements on which pixels must be illuminated for a particular line, while X does not, the `Initial Error` term is calculated differently for Win32 display drivers than for Digital X servers. Win32 lines must comply with Microsoft's grid intersect quantization (GIQ) specification:

> ''That is, the geometric line from the starting point to the ending point is imagined as drawn on a grid with p(ix)els at the grid intersections. Whenever the geometric line crosses the grid, the nearest p(ix)el is illuminated. In the case where two p(ix)els are equidistant, the upper or left p(ix)el is illuminated, unless the slope of the line is exactly one, in which case the upper or right p(ix)el is illuminated.''

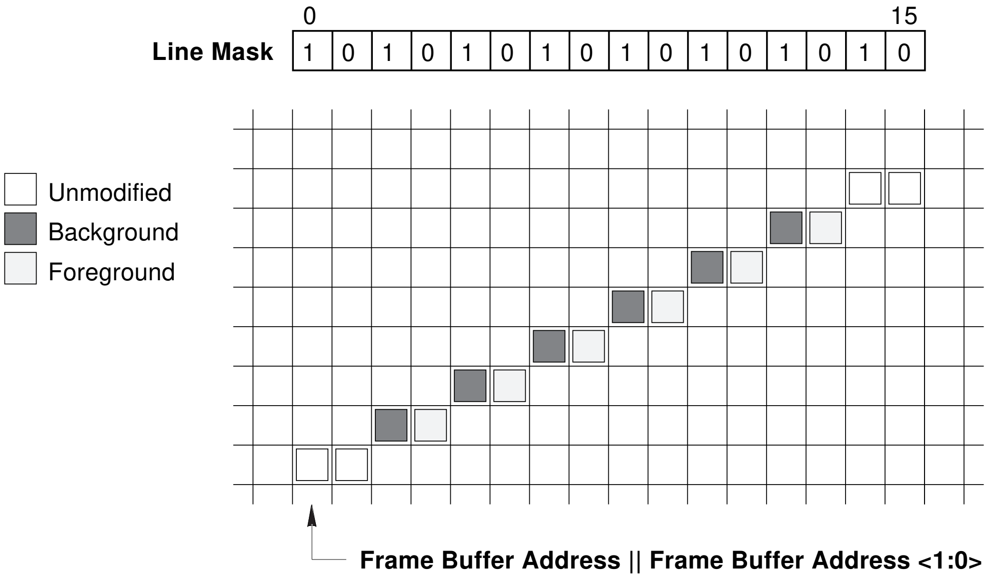
While Win32 lines are generally X-compliant, they do not comply with Digital's traditional way of drawing lines under X. Traditionally, Digital's X servers draw X-compliant lines that are not always Win32-compliant; specifically, the upper or left pixel is not always illuminated in accordance with the GIQ specification. Consequently, the 21030 line setup compensates for the difference by setting `Initial Error` as a function of the graphics environment. If there is no need to adhere to traditional practice, the 21030 draws X-compliant lines, including when the graphics environment is Win32.

In any line mode, drawing lines by writing to the GSLRs is almost always faster than drawing lines by writing to the frame buffer. However, additional restrictions imposed when drawing Win32-compliant lines prevent using the GSLRs. Therefore, some lines can be drawn only by directly writing to the frame buffer. These restrictions affect the 21030 display driver rather than the hardware (Section 7.2.3).

The slope-no-go registers (GSNR<7:0>) mimic the behavior of the GSLRs, but they do not initiate drawing. That is, on a write to a GSNR, the 21030 processes the slope data, generates the Bresenham terms, and loads them into the Bresenham registers, but the line is not drawn. The GSNRs are useful for drawing clipped lines, in which some portion of the line is not drawn.

Figure 6–31 is an example opaque-line mode operation.

Figure 6–31 Opaque Line Drawing



6.2.12.2 Destination Bitmap Support in Opaque-Line Mode

Opaque-line mode drawing supports the destination bitmap formats described in Section 6.1.5. The `Destination Bitmap` and `Destination Byte` parameters must be set for the desired destination bitmap. The `Source Bitmap` and `Source Byte` parameters are ignored, but software must align the `Foreground` and `Background` to the desired destination. (See Section 6.1.6 for more information.)

6.2.12.3 Extending and Linking 2D Lines

The 21030 processes up to 16-pixels per line-segment drawing operation, but graphics applications do not limit line drawing requests to lines that are 16 or fewer pixels. Additionally, applications can request a string of lines, with each subsequent line starting at the end of the preceding line. The line drawing hardware supports two ways of linking 16-pixel line segments:

- A previously drawn line can be extended up to 16-pixels along the same slope.
- A new line drawing can start at the end of a previously drawn line.

For example, to draw a 50-pixel line, the 21030 software must link four segments along the same line. The 21030 allows multiple segments of the same line and multiple lines to be drawn without software reassigning the address and other parameters for each segment. The Bresenham engine has several features to facilitate such operations.

The Bresenham engine contains a working set of the parameters `Initial Error`, `Length`, and `Frame Buffer Address`. When a line-segment drawing operation is initiated, the Bresenham engine conditionally loads the primary parameter values into its working set. (The drawing operation can be initiated by a write to a GSLR or the GCTR.) During the line-stepping process, the Bresenham engine operates only on the working set.

On completion of the segment drawing operation, the Bresenham engine leaves its working set in a state suitable for linking to the next segment or line. Specifically, the Bresenham engine's working set of parameters is managed as follows:

- If a GSLR was written since the last line segment was drawn, the Bresenham engine updates its working copies of `Length` and `Initial Error` from the register before drawing a line segment. Otherwise, the line segment is drawn without updating the working set parameters.
- If a new address was specified in the GADR since the last line segment was drawn, the Bresenham Engine updates its working copy of the `Frame Buffer Address` before drawing a line segment. Otherwise, the line segment is drawn without updating the working copy of the `Frame Buffer Address`.
- On completion of a line-segment drawing operation, the Bresenham engine does the following:
  – Leaves its working copy of the `Frame Buffer Address` at the address of the next pixel along the line.
  – Resets the value of its working copy of `Length` to 16.
  – Sets its copy of `Initial Error` to the error term for the next pixel along the line.

In other words, the Bresenham engine uses new values of `Initial Error` and `Length` only if A GSLR was reloaded after the last line segment was drawn; otherwise, the engine does not sample either parameter, but uses the current working set values.

Similarly but independently, the Bresenham engine uses a new address only if a new address was specified by a write to the GADR after the last line segment was drawn.

Extending a Single Line

By taking advantage of the Bresenham engine behavior, software can extend the current opaque line up to 16 pixels by writing the `Line Mask` for the next segment to the GCTR. Because software does not write a GSLR, the Bresenham engine's working parameters correspond to the next pixel in the line, with `Length` reset to 16. Given the new line mask, the 21030 extends the line 16 pixels.

In summary, the fastest way for software to extend the current line by one more 16-pixel segment is to write the following:

1. Any relevant registers, except the GADR and GSLR
2. The segment's `Line Mask` to the GCTR

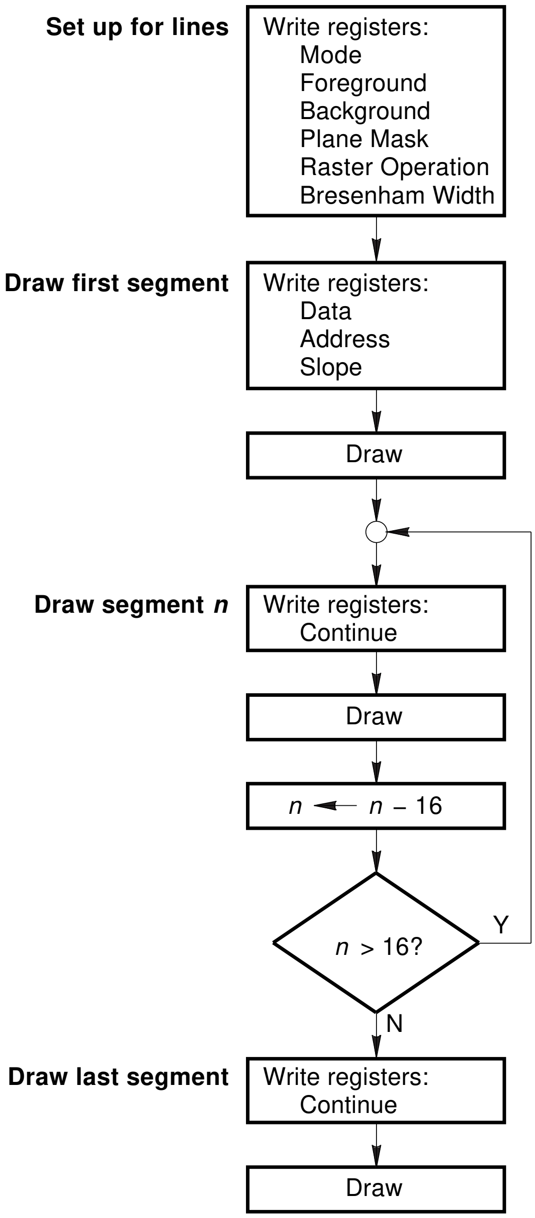
This process can be repeated as many times as necessary to draw lines of arbitrary length. Usually, the first segment is drawn by writing to a GSLR, and all subsequent segments along the same line are drawn by writing to the GCTR.

Figure 6–32 shows a typical sequence for drawing a line of length *n* by drawing the first segment and then drawing as many extending segments as necessary.

Note

Other than `Length`, `Initial Error`, and `Frame Buffer Address`, all relevant opaque-line mode parameters (such as `Foreground` and `Background`) are sampled every time a line segment drawing operation is initiated.

Figure 6–32 Opaque-Line Drawing Sequence



Linking Multiple Lines

The Bresenham engine behavior also allows software to link multiple lines. Each line can have different color, mask, or slope attributes, but the lines must be drawn end-to-end (a polyline). In this case, software writes a GSLR, rather than the GCTR, and does not write the GADR. This effectively reinitializes all of the engine's slope parameters, including `Initial Error`, but does not change the working copy of the `Fame Buffer Address`.

In summary, to write the first segment of a new line where the previous line terminated, software writes the following:

1. Any relevant registers except the GADR
2. A GSLR

Specifying Cap Ends

Whether extending or linking lines, software must specify the appropriate value for `Cap Ends` (GMOR <15>). When the value of `Cap Ends` = 0, the last pixel in the line is not drawn; otherwise, the last pixel is drawn. Therefore, to extend or link line segments as described above, software must set `Cap Ends` = 0, so that the last pixel in the previous line segment is not drawn. If the value of `Cap Ends` = 1, the last pixel in the previous line segment and the first pixel in the next line segment will be drawn at the same place, possibly with undesired results.

6.2.13 Transparent-Line Mode

In the transparent-line mode, a PCI write to the frame buffer address space draws a masked, 16-pixel solid-line segment starting at the specified address. The 21030 draws the line segment as a function of the parameters listed in Table 6–29.

```
Transparent Line (Frame Buffer Address, Frame Buffer Address <1:0>, Line Mask,
                  Raster Op, Plane Mask, Foreground, Address Increment 1,
                  Address Increment 2, Error Increment 1, Error Increment 2,
                  Initial Error, Length, Destination Bitmap, Destination Byte,
                  Cap Ends);
```

Table 6–29 Transparent-Line Mode Parameters

Parameter	Source		Section
`Frame Buffer Address`	PCI write address	—	—
`Frame Buffer Address <1:0>`	PCI write data	<17:16>	—
`Line Mask`	PCI write data	<15:0>	—
`Raster Op`	Raster operation register	GOPR <3:0>	4.4.3
`Plane Mask`	Plane mask register	GPMR <31:0>	4.4.20
`Foreground`	Foreground register	GFGR <31:0>	4.4.18
`Address Increment 1`	Bresenham 1 register	GB1R <31:16>	4.4.10
`Error Increment 1`	Bresenham 1 register	GB1R <15:0>	4.4.10
`Address Increment 2`	Bresenham 2 register	GB2R <31:16>	4.4.11
`Error Increment 2`	Bresenham 2 register	GB2R <15:0>	4.4.11
`Initial Error`	Bresenham 3 register	GB3R <31:16>	4.4.12
`Length`	Bresenham 3 register	GB3R <3:0>	4.4.12
`Destination Bitmap`	Raster operation register	GOPR <9:8>	4.4.3
`Destination Byte`	Raster operation register	GOPR <11:10>	4.4.3
`Cap Ends`	Mode register	GMOR <15>	4.4.1

The transparent-line mode works in the same way as the opaque-line mode, and is similarly more efficient when operations are initiated by writing a slope register rather than the frame buffer. Transparent-line mode differs in that `Line Mask` determines whether the `Foreground` color is written (`Line Mask` bit = 1) or write is disabled (`Line Mask` bit = 0), rather than determining whether foreground or background color is written.

6.2.14 3D Line and Span Modes

In any 3D line mode, a PCI write to a slope register (GSLR<7:0>) or the span width register (GSWR), draws a masked, 16-pixel 3D line segment starting at the specified address.

For this description, a *3D line segment* is defined as a string of 16 contiguous pixels drawn along a line of arbitrary slope and length. Additionally, line segments drawn in any of the 3D line modes are described in the context of a *generic 3D line segment*, and that segment can be Z-buffered, color-interpolated or stippled, and dithered, depending on the mode specified. Span segments are included in the context, and a *3D span segment* is defined as a 3D line segment in which the slope is zero.

The mode field (GMOR <6:0>) specifies the following 3D line modes:

- Z-buffered opaque line
- Z-buffered transparent line
- Z-buffered, opaque, color-interpolated, dithered line
- Z-buffered, opaque, color-interpolated, nondithered line
- Z-buffered, opaque, sequential-interpolated line
- Z-buffered, transparent, color-interpolated, dithered line
- Z-buffered, transparent, color-interpolated, nondithered line
- Z-buffered, transparent, sequential-interpolated line
- Color-interpolated, dithered line
- Color-interpolated, nondithered line
- Sequential-interpolated line

Unlike other modes, 3D line mode drawing operations *cannot* be initiated by a write to the frame buffer. Software must initiate 3D line mode drawing operations by writing to a GSLR, GSNR, or the GSWR, to initialize the hardware for color- or sequential-interpolation and Z-buffering.

The GCTR can be used to extend and link 3D lines in a way similar to 2D lines (Section 6.2.14.4).

The 21030 draws a 3D segment as a function of some or all of the parameters listed in Table 6–30; *all* of the 3D line modes do not require *all* of the parameters.

```
3D Line (Frame Buffer Address, Frame Buffer Address <1:0>, Line Mask, Raster Op,
         Plane Mask, Z Address, Z Reference Integer, Z Reference Fraction,
         Z Increment Integer, Z Increment Fraction, Red Value, Red Increment,
         Green Value, Green Increment, Blue Value, Blue Increment, Z Buffer Width,
         Bitmap Width, Destination Bitmap, Destination Byte, Z16, Cap Ends,
         Stencil Write Mask, Stencil Read Mask, S Test, S Fail, D Fail, D Pass,
         Z Test, Z Update);
```

Table 6–30 3D Line Mode Parameters

Parameter	Source Register		Section	Note
`Absolute Dy`	Slope	GSLR*n* <31:16>	4.3.1	❶
`Absolute Dx`	Slope	GSLR*n* <15:0>	4.3.1	❶
`Frame Buffer Address`	Address	GADR <31:0>	4.4.2	❶
`Line Mask`	Data	GDAR <15:0>	4.4.8	❶
`Raster Op`	Raster operation	GOPR <3:0>	4.4.3	❶
`Plane Mask`	Plane mask	GPMR <31:0>	4.4.20	❶
`Pixel Mask`	Pixel mask	GPXR <31:0>	4.4.21	❷
`Foreground`	Foreground	GFGR <31:0>	4.4.18	❸
`Background`	Background	GBGR <31:0>	4.4.19	❷
`Z Address`	Z-base-address	GZBR <23:0>	4.4.15	❹
`Z Reference Integer`	Z-value high Z-value low	GZVR-H <3:0> GZVR-L<31:12>	4.4.16 4.4.16	❹
`Z Reference Fraction`	Z-value low	GZVR-L <11:0>	4.4.16	❹
`Z Increment Integer`	Z-increment high Z-increment low	GZIR-H <3:0> GZIR-L <31:12>	4.4.17 4.4.17	❹
`Z Increment Fraction`	Z-increment low	GZIR-L <11:0>	4.4.17	❹
`Red Value`	Red value	GRVR <19:0>	4.4.22	❺
`Red Increment`	Red increment	GRIR <19:0>	4.4.23	❺
`Green Value`	Green value	GGVR <19:0>	4.4.24	❺
`Green Increment`	Green increment	GGIR <19:0>	4.4.25	❺
`Blue Value`	Blue value	GBVR <19:0>	4.4.26	❺
`Blue Increment`	Blue increment	GBIR <19:0>	4.4.27	❺
`Dither Column`	Green value	GGVR <31:27>	4.4.24	❻
`Dither Row`	Red value	GRVR <31:27>	4.4.22	❻

(continued on next page)

Table 6–30 (Cont.) 3D Line Mode Parameters

Parameter	Source Register		Section	Note
`Bitmap Width`	Bresenham width	GBWR <15:0>	4.4.13	❶
`Z Buffer Width`	Bresenham width	GBWR <31:16>	4.4.13	❹
`Z16`	Mode	GMOR <14>	4.4.1	❹
`Destination Bitmap`	Raster operation	GOPR <9:8>	4.4.3	❶
`Destination Byte`	Raster operation	GOPR <11:10>	4.4.3	❶
`Cap Ends`	Mode	GMOR <15>	4.4.1	❶
`Stencil Reference`	Z-value high	GZVR-H <31:24>	4.4.16	❹
`Stencil Write Mask`	Stencil mode	GSMR <7:0>	4.4.14	❹
`Stencil Read Mask`	Stencil mode	GSMR <15:8>	4.4.14	❹
`S Test`	Stencil mode	GSMR <18:16>	4.4.14	❹
`S Fail`	Stencil mode	GSMR <21:19>	4.4.14	❹
`D Fail`	Stencil mode	GSMR <24:22>	4.4.14	❹
`D Pass`	Stencil mode	GSMR <27:25>	4.4.14	❹
`Z Test`	Stencil mode	GSMR <30:28>	4.4.14	❹
`Z Update`	Stencil mode	GSMR <31>	4.4.14	❹

The notes for Table 6–30 list the 3D line modes in which the parameters are used, as follows:

❶ All

❷ Z-buffered opaque

❸ Opaque line and transparent line

❹ All Z-buffered

❺ All interpolating

❻ All dithering

In general, the 21030 generates 3D segments in the same way as 2D segments, with the following exceptions:

- Write can be disabled to each pixel based on the result of comparing Z or stencil values.
- Color can be supplied in one of the following ways:
  - Color-interpolated and selectively dithered by the color interpolators

- Sequential-interpolated by the sequential interpolator
- As solid or patterned shades from the `Foreground` and `Background` parameters

In the color-interpolated modes, the on-chip color interpolators use independent red, green, and blue starting values and their respective incremental values to generate an arbitrarily smooth-shaded, 24-bit, true-color value for each pixel in the segment. The 24-bit value can be selectively dithered to 12-bpp or 8-bpp.

In the sequential-interpolated modes, the sequential interpolator uses one 8-bpp grey-scale value and 8-bpp grey-scale increment per line, to generate one unique grey-scaled value per 8-bit pixel. Because the pixel resolution is limited to 8 bits, sequential-interpolated modes support drawing to only 8-bpp bitmaps. Dithering is neither useful nor available in these modes.

In the noninterpolated opaque or transparent modes, colors are supplied as a function of the `Line Mask`, `Pixel Mask`, `Foreground`, and `Background` parameters in the same way as in the 2D line modes. Noninterpolated opaque or transparent 3D lines cannot be dithered.

6.2.14.1 Color Interpolation

The 21030 color interpolators contain a hidden working-set of parameters that shadow the primary parameters `Red Value`, `Green Value`, and `Blue Value` (see Table 6–30). When any 3D segment drawing operation is initiated, the color interpolators conditionally load the primary parameter values into the working set from their respective registers (GRVR, GGVR, and GBVR). In a color-interpolated line mode, a write to a GSLR or the GCTR initiates a segment drawing operation. During the drawing process, the color interpolators operate only on the working set. On completion of the segment drawing operation, the color interpolators leave the working set in a state suitable for linking to the next segment or line (similar to the way that drawn 2D line segments are ready for extension or linking). (See Section 6.2.14.4 for the conditions for loading the working set.)

For this description, the color interpolators' working-set of color values are initialized to the primary color values stored in the GRVR, GGVR, and GBVR. The color interpolators run in lock-step with the Bresenham engine to generate either a 24-bit color-interpolated value or an 8-bit sequential-interpolated value for each pixel in the segment. Each time the Bresenham engine steps to the next pixel in a color-interpolated mode, the color interpolators add the color increments `Red Increment`, `Green Increment`, and `Blue Increment` to the respective working color values, generating 8-bit red, green, and blue values for the pixel.

Note

The 21030 does not clamp interpolated values to avoid overflow and underflow; software must do the clamping.

Because the color interpolators process 8 bits of each color for each pixel, writing interpolator output to a 24-bpp bitmap is simple (Figure 6–33). Each 8-bit channel is mapped to the corresponding field specified by bitmap format TC24 (Figure 6–4).

Figure 6–33 shows the color-interpolator output for a 24-bpp destination.

Figure 6–33 Color Interpolator Output for a 24-bpp Destination

31 24	23 16	15 8	0
Undefined	Red	Green	Blue

Because the `Destination Bitmap` can specify 8-bpp or 12-bpp bitmap destinations as well as 24-bpp, the 21030 reduces 24-bpp color to 8-bpp or 12-bpp in either of the following ways:

- Dithering through the on-chip dither logic
- Truncating each channel to the required depth

The specified mode determines whether to enable dithering. If a dithering mode is specified, software must also specify the starting `Dither Row` and `Dither Column` parameters for the segment, to specify the initial row and column indices into the internal dither matrix. These parameters are usually the LSBs of the starting *y* (row) and *x* (column) coordinates. After the initial `Dither Row` and `Dither Column` values are sampled, the Bresenham engine updates them modulo 32 as it walks the slope of the line. The addressed matrix data is then be added to each channel, and the resultant value truncated. (See Section 6.2.14.4 for information about the dithering algorithm.)

If a dithering mode is not specified, the 21030 simply truncates each channel to the appropriate number of bits, leaving only the MSBs. Whether or not dithering is enabled, the 21030 reduces the color interpolator output to the number of shades shown in Table 6–31.

Table 6–31 Reduced Color Interpolator Output for 8-bpp and 12-bpp Destinations

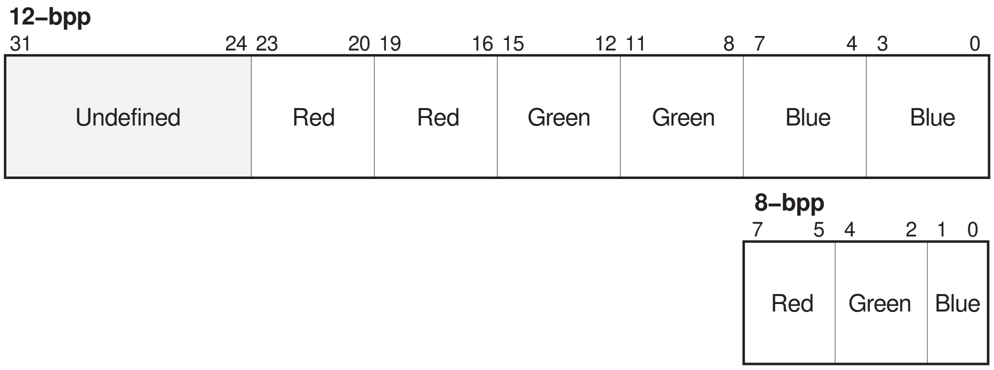
	Number of Shades (Bits)		
Destination	**Red**	**Green**	**Blue**
8-bpp	8 (3)	8 (3)	4 (2)
12-bpp	16 (4)	16 (4)	16 (4)

When drawing to a 12-bpp destination, the 21030 disperses and replicates the reduced colors across the output Dword. This effectively allows software to specify the bitmap format $DC12_0$ or $DC12_1$ by simply applying the appropriate value `Plane Mask` (000F0F0F or 00F0F0F0, respectively).

When drawing to an 8-bpp destination, the 21030 simply packs the reduced fields into one byte. The 21030 routes the resultant byte to the proper byte location according to the current address LSBs and the `Destination Bitmap` and `Destination Byte` parameters; special plane masking is unnecessary.

Figure 6–34 shows the destination-specific color output for 8-bpp and 12-bpp bitmaps.

Figure 6–34 Color Interpolators Output for 12-bpp and 8-bpp Destinations



6.2.14.2 Sequential Interpolation

The sequential interpolator operates in the same way as the color interpolators, except that it maintains a working copy of only one `Grey Value` per pixel. When a sequential-interpolated line drawing operation is initiated, the sequential interpolator conditionally loads the value of the primary `Grey Value` from the GRVR into the working copy. During the drawing process, the sequential interpolator operates only on the working copy. On completion of the segment drawing operation, the interpolator leaves its working copy in a state suitable for linking to the next segment or line (similar to the way that drawn 2D line segments are ready for extension or linking). The conditions for loading the `Grey Value` working copy are identical to the conditions for the independent red, green, and blue values (Section 6.2.14.4).

In the same way as the color interpolators, the sequential interpolator runs in lock-step with the Bresenham engine to generate an 8-bit grey-scale value for each pixel in the segment. In a sequential-interpolated mode, each time the Bresenham engine steps to the next pixel, the sequential interpolator adds the `Grey Increment` value to the working `Grey Value`, generating a new 8-bit grey-scale value for the pixel.

Note

The 21030 does not clamp interpolated values to avoid overflow and underflow; software must do the clamping.

As in drawing color-interpolated values dithered to 8 bits, the 21030 aligns the resultant Grey Value byte to the proper byte location according to the current address LSBs and the Destination Bitmap and Destination Byte parameters.

6.2.14.3 Z-Buffer and Stencil-Buffer Operation

If a Z-buffering mode is specified, the 21030 performs both a Z-buffer and a stencil-buffer reference test for each pixel along the segment. The results of the tests determine whether and how the stored-Z and stored-stencil values are updated, as well as whether to write the corresponding pixel value. The stored values are defined as those resident in frame buffer memory before the operation. A Stencil Reference value (used for the entire segment) and a Z Reference value are taken from the Z-value registers (GZVR-H and GZVR-H) and the output of the Z-interpolator hardware. Z-buffering modes are allowed only in 32-bpp frame buffers. (See Section 6.1.5 for more information about the stored and reference stencil and Z formats.)

The Z-interpolator and its associated stencil and Z logic work in a way similar to the color interpolators. The Z-interpolator runs in lock-step with the Bresenham engine and operates with a working copy of the Z Reference parameter. It can sample the primary Z Reference parameter stored in the Z-value registers at the start of a 3D-segment drawing operation. The Bresenham engine also contains a working copy of the Z Address and conditionally samples the primary parameter stored in the Z-base-address register (GZBR). In certain cases, the working copies of the Z Reference parameter and Z Address are not updated at the time of drawing (Section 6.2.14.4).

For this description, the working copies of the Z Reference parameter and Z Address are initialized to their primary values. On a write to a GSLR, the Bresenham engine executes its setup sequence (Section 6.2.12.1). In the same way that it generates the pixel Address Increment 1 and Address Increment 2 as part of its setup procedure, in a Z-buffering mode the Bresenham engine generates the Z Address Increment 1 and Z Address Increment 2, as a function of the Z Buffer Width.

For each pixel step through the segment, the Bresenham engine updates the Z Address at the same time it updates the Frame Buffer Address. Similarly, on each pixel step, the Z-interpolator adds the Z Increment Integer and Z Increment Fraction values to its working copy of the Z Reference value. Therefore, new values of Z Reference and Z Address are available for each pixel as the Bresenham engine steps through the segment.

The `Z Address Increment 1` and `Z Address Increment 2` parameters can be supplied only by the Bresenham engine as a result of executing its setup — software cannot explicitly specify these parameters; however, both parameters can be read through the GCTR.

At each pixel, the stencil and Z logic reads the stored stencil and Z values from the frame buffer at the `Z Address.` A 24-bit or 16-bit Z value is read, depending on the value the `Z16` parameter. (See Section 6.1.5 for more information about the two supported Z-buffer formats.)

For the stencil operation, the 21030 logically ANDs both the `Stencil Stored` and `Stencil Reference` values with the `Stencil Read Mask`. It then compares the masked versions of `Stencil Stored` and `Stencil Reference` values as specified by the `S Test`.

- If the `S Test` fails, the `Stencil Stored` value is updated as specified by the `S Fail` parameter.
- If the `S test` passes and the `Z Test` fails, the `Stencil Stored` value is updated as specified by the `D Fail` parameter.
- If both the `Z Test` and `S Test` pass, the `Stencil Stored` value is updated as specified by the `D Pass` parameter.

  On any write back to the `Stencil Stored` location, only the bit positions specified by the `Stencil Write Mask` are modified.

For the Z operation, the 21030 compares the `Z Stored` and `Z Reference` values as specified by the `Z Test`.

- If the `Z Test` passes and `Z Update` is enabled, the `Z Reference` value is written back to the stored Z location.
- If the `Z Test` passes, the `S Test` passes, and the appropriate bit in the `Line Mask` is set, the color from the color interpolator (interpolated modes) or the `Background` and `Foreground` parameters (noninterpolated opaque or transparent modes) is written to the `Frame Buffer Address`.

The standard plane mask is disabled for writes to the `Stencil Stored` and `Z Stored` locations.

The foregoing description of Z-buffering assumes a transparent mode of operation; that is, the Z and stencil tests determine whether the calculated pixel value is written. In an opaque mode of operation, the Z and stencil values are tested and updated as specified by the GSMR, but the pixel value is always written, regardless of the Z and stencil test results.

(See the GSMR description in Section 4.4.14 for more information about specifying the D, S, and Z test, pass, and fail parameters.)

The following pseudo-code represents the basic algorithm for a Z-buffered, color-interpolated, dithered line. The algorithm for other 3D line modes is similar, with Z-buffering, color-interpolation, or dithering individually optional according to the specified mode.

```
/* generate all of the initial Bresenham error, error increment, and address
   increment terms for color and Z */
Bresenham Setup (Absolute Dx, Absolute Dy, Bitmap Width, Z Buffer Width);
while (Length > 0)
{
     /* generate next 24-bit color */
     Red Value += Red Increment;
     Green Value += Green Increment;
     Blue Value += Blue Increment;
     Dither (Dither Row, Dither Column, Red Value, Green Value, Blue Value,
             Depth, Dither Out);
     /* stencil and Z ops */
     Z Reference += Z Increment;
     ReadZ (Z Address, Stencil Stored, Z Stored);
     Stencil Reference = Stencil Reference & Stencil Read Mask;
     Stencil Stored = Stencil Stored & Stencil Read Mask;
     if  ! S Test (Stencil Stored, Stencil Reference)
     {
          Cond Write Stencil (Stencil Stored, S Fail, Stencil Write Mask);
     }
     else if Z Test (Z Stored, Z Reference)
     {
         Cond Write Stencil (Stencil Stored, D Fail, Stencil Write Mask);
     }
     else
     {
         Cond Write Stencil (Stencil Stored, D Pass, Stencil Write Mask);
     }
     if Z Update && Z Test (Z Stored, Z Reference)
     {
          Write Z (Stencil Stored);
     }
     /* conditional write pixel */
     if ((Line Mask (Length) && Transparent) &&
               ((Z Test (Z Stored, Z Reference) &&
                 S Test (Stencil Stored, Stencil Reference))
               || Opaque))
     {
          Write Pixel (Frame Buffer Address, Pixel, Raster Op, Plane Mask,
                       Destination Bitmap, Destination Byte);
     }
     /* Bresenham step along line */
     if (Error < 0)
     {
          Frame Buffer Address += Address Increment 1;
          Z Address += Z Address Increment 1;
```

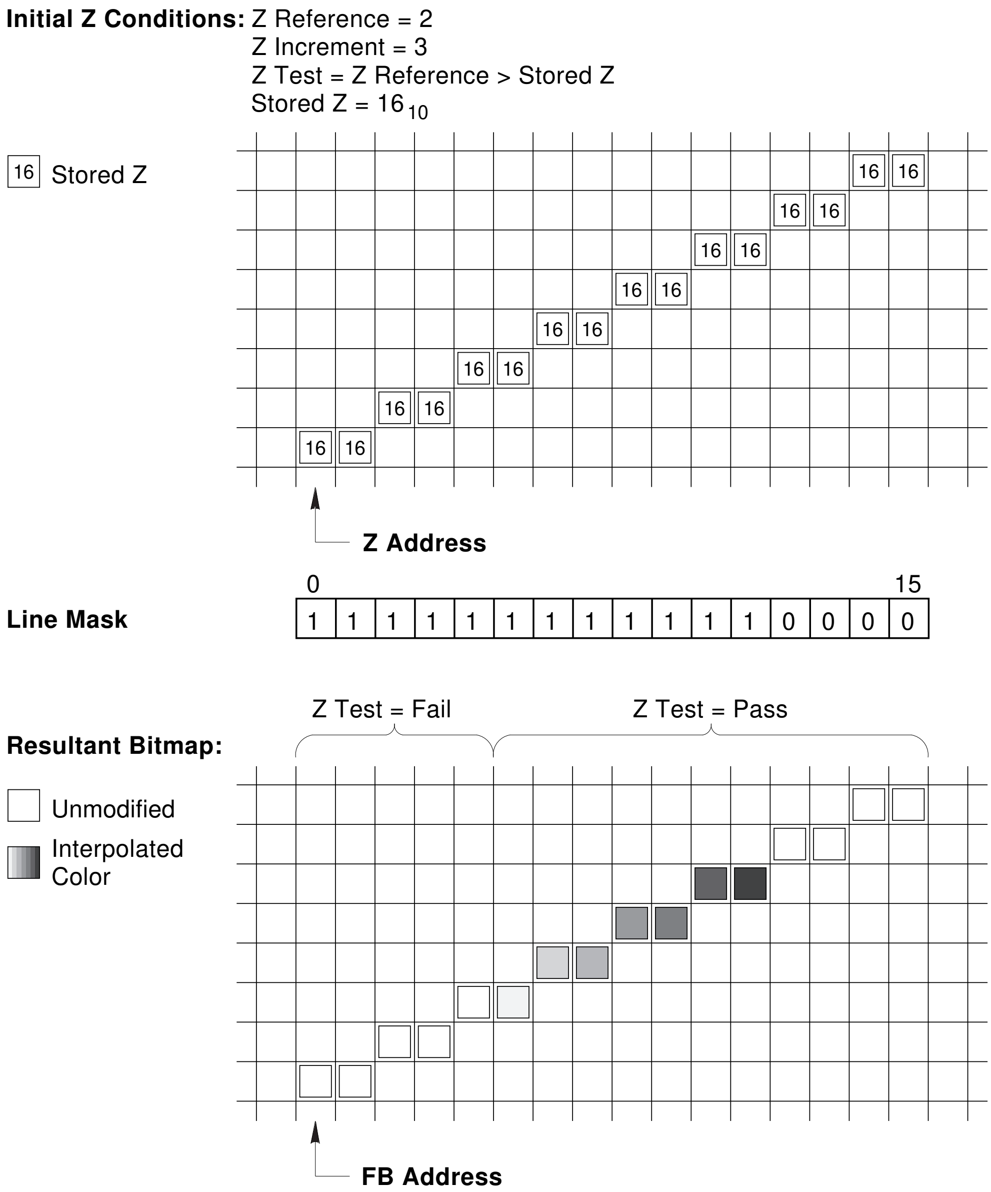
```
          Error += Error Increment 1;
          Adjust Dither Major (Dither Row, Dither Column);
     }
     else
     {
          Frame Buffer Address += Address Increment 2;
          Z Address += Z Address Increment 2;
          Error -= Error Increment 2;
          Adjust Dither Minor (Dither Row, Dither Column);
     }
     Length --;
}
```

The pseudo-code does not reflect the exact logic implementation, but provides a general description of the function. The 21030 actually optimizes the process as follows:

- It minimizes the number of RAS cycles necessary to draw the segment.
- It tries to group the maximum number of reads or writes to the same row before moving on to the next row.
- Because the frame buffer port is 64 bits wide, the 21030's write buffer tries to merge, without collapsing, Dwords and bytes into quadwords, to minimize the number of CAS cycles.

Figure 6–35 is an example of a Z-buffered, color-interpolated line-segment drawing operation. The figure shows the initial state of the Z buffer in off-screen frame buffer memory and the result of the drawing operation.

Figure 6–35 Z-Buffered, Color-Interpolated Line Segment



6.2.14.4 Extending and Linking 3D Lines

3D lines, as well as 2D opaque and transparent lines, can be extended and linked. To efficiently extend the current 3D line, or start a new line where the previous line ended, software can use 3D line mode features and also take advantage of the Bresenham engine features available in the 2D modes. The same mechanism works for spans. (See Section 6.2.12.3 for more information about the features common to the 2D line modes.)

The following hardware features simplify 3D line-segment linking:

- Before drawing a line segment, the working copies of the following parameters are updated with their primary values only if a new `Frame Buffer Address` was written to the GADR after the last segment drawing operation.
  - `Red Value`, `Green Value`, and `Blue Value` (color interpolators)
  - `Grey Value` (sequential interpolator)
  - `Z Reference` (Z-interpolator)
  - `Z Address` (Bresenham engine)
- On completion of a segment drawing operation, the working copies for the above parameters are defined for the next pixel along the line.

To extend the line another 16 pixels, software can simply write the `Line Mask` for the next segment to the GCTR.

To start a new line with different color and Z-slope values where the previous line ended, software does the following:

- Rewrites the `Red Increment`, `Green Increment`, `Blue Increment`, and `Z Increment` or `Grey Increment` values as required.
- Writes a GSLR, but does not write the GADR.

7

Programming Guide

This chapter contains programming information about the 21030 configuration firmware, graphics drivers and servers, video support functions, and functions to support Alpha AXP systems.

7.1 PCI Configuration Firmware

The 21030 hardware implements the full set of required PCI configuration registers, and is fully configurable by generic PCI-compliant system firmware. The 21030 is not limited to motherboard applications, but behaves as a generic plug-and-play PCI option for all PCI-compliant systems independent of operating system. (Section A.4.2.1 addresses systems that require dedicated support for the 21030 in the base system firmware.)

7.1.1 Device Address Mapping

Configuration firmware can map the 21030 device and enable response to that mapping by manipulating fields in the PCI device base address register (PDBR, Section 4.2.2) and PCI command and status register (PCSR, Section 4.2.1). Table 7–1 describes the fields to be manipulated.

Table 7–1 21030 Base Address and Memory Space Enable Fields

Field	Register	Bits	Field Description
Device base address	PDBR	<31:4>	The PCI memory address defined as the base address of the 21030 address space.
Memory space enable	PCSR	<1>	When set, enables the 21030 to respond to memory space accesses.

The PDBR and PCSR are written in the following sequence:

1. Configuration firmware probes the PDBR to determine where the 21030 is to be mapped and the amount of space to allocate to it; that is, firmware

writes all ones to the PDBR and then reads back the value. The 21030 returns zeros in <26:4> to indicate the following 21030 requirements:

- It requires at least 128MB of total address space (PDBR <26:4> = 000000_{16}).
- It must be mapped to PCI memory space (PDBR bit 0 = 0).

2. Firmware allocates 32MB of naturally aligned PCI memory space and writes the base address to the PDBR.
3. Firmware sets memory space enable (PCSR <1>) to enable device response. (Usually, memory space enable should not be set until the PDBR has been properly initialized as described in steps 1 and 2.)

(The PDBR and PCSR write sequence differs for the 21030 step A. See Section A.4.2.)

After the PDBR is written and memory space enable is set in the PCSR, the 21030 can respond as a normal PCI target (Section 5.2).

7.1.2 Bus Mastering

The 21030 supports DMA operations to rapidly transfer image data from PCI-accessible memory to display memory. To invoke 21030 DMA operations, the 21030 must be able to master the PCI bus. Configuration firmware must write fields in the PCSR and PCI latency timer register (PLTR, Section 4.2.5), to enable the 21030 to be a PCI bus master. Table 7–2 describes the fields to be written.

Table 7–2 PCI Latency Timer and Master Enable Fields

Field	Register	Bits	Field Description
Latency timer	PLTR	<15:8>	21030 bus ownership is limited to the number of PCI clocks specified in this field.
Master enable	PCSR	<2>	When set, enables the 21030 to become bus master. It must be set to enable DMA operations, but should not be set until the PLTR is initialized.

DMA operations usually involve a long (hundreds of bytes) burst transfer. Therefore, a high latency-timer value helps improve performance. However, the benefit of a high latency-timer value depends on the PCI bridging structure, and is limited, for example, by PCI bridges that terminate transfers on cache-line boundaries.

7.1.3 Interrupt Routing

Configuration firmware is also responsible for mapping system interrupt lines to PCI devices that require interrupt services (as does the 21030). After the interrupt lines are mapped, configuration firmware must write the routing information to the interrupt line field in the PCI line interrupt register (PLIR <7:0>, Section 4.2.7). During subsequent normal graphics operation, display drivers or the operating system can determine interrupt vectors and priorities either by reading the PLIR or through the GET_DEVICE_INTERRUPT BIOS routine.

7.1.4 VGA Pass-Through

The 21030 supports VGA in a pass-through mode. In this mode, the VGA video stream is brought into the 21030 graphics subsystem through a VESA standard-VGA feature connector and multiplexed into the main video stream prior to color lookup and digital-to-analog conversion. (See Chapter 6 for more information about VGA support.)

Many systems that use the 21030 also use VGA for the boot display. Boot code requests display before any 21030-specific code can execute. Consequently, the 21030 is initialized into a VGA pass-through state without explicit firmware initialization.

All VGA pass-through control is embedded in the PCI VGA redirect register (PVRR, Section 4.2.8). The PVRR fields enable VGA color-register snooping in I/O space, and define how the snooped VGA I/O addresses are translated into MPU-port RAMDAC accesses (Table 7–3).

Table 7–3 VGA Redirect Register Fields

Field	Bits	Description
VGA mask	3:0	The redirected address for the VGA pixel-mask register (3C6). Initialized to 2_{16} at reset.
VGA address	7:4	The redirected address for the VGA pixel address register (3C8). Initialized to 0_{16} at reset.
VGA data	27:24	The redirected address for the VGA pixel data register (3C9). Initialized to 1_{16} at reset.
VGA enable	31	When set, VGA color-register snooping is enabled; otherwise snooping is disabled. Set at reset.

Each of the three snooped I/O cycles has a corresponding MPU-write cycle in which the address of the write cycle is extracted from one of the three PVRR redirected-address fields.

The PVRR fields are initialized as follows:

- VGA enable is set.
- VGA address, VGA data, and VGA mask pointers are made compatible with the Bt485 RAMDAC.

Therefore, if the RAMDAC used is compatible with the Bt485, the 21030 is initialized in pass-through mode without firmware intervention. (See Sections 4.2.8, 4.8.2, and 6.1.5.2 for more information about RAMDAC compatibility.)

When a 21030 subsystem is implemented on the system motherboard, the base firmware can modify the PVRR (or other registers) before display to support RAMDACs that are not compatible with the Bt485 configuration.

Systems that do not require VGA for the boot display can disable pass-through by clearing the PVRR VGA enable bit (this is unnecessary if the system I/O space is PCI-compatible).

7.1.5 Expansion ROM

The 21030 supports an external EEPROM that conforms to PCI expansion ROM specifications. See the *PCI Local Bus Specification, Revision 2.0* for more information.

7.2 Graphics Drivers and Servers

Sections 7.2.1 through 7.2.7.2 describe the implementation of standard graphics server and driver functions using the 21030. The descriptions suggest the 21030 modes and functions that are most appropriate or specifically intended to implement typical API functions. The descriptions assume familiarity with the function and operation of each 21030 drawing mode, including parameter lists, operation invocation, and expected results (Chapter 6).

7.2.1 Bit-Block Transfers

Bit-block transfers (BitBlts) can be implemented as screen-to-screen copies and host-to-screen copies.

7.2.1.1 Screen-to-Screen Copy

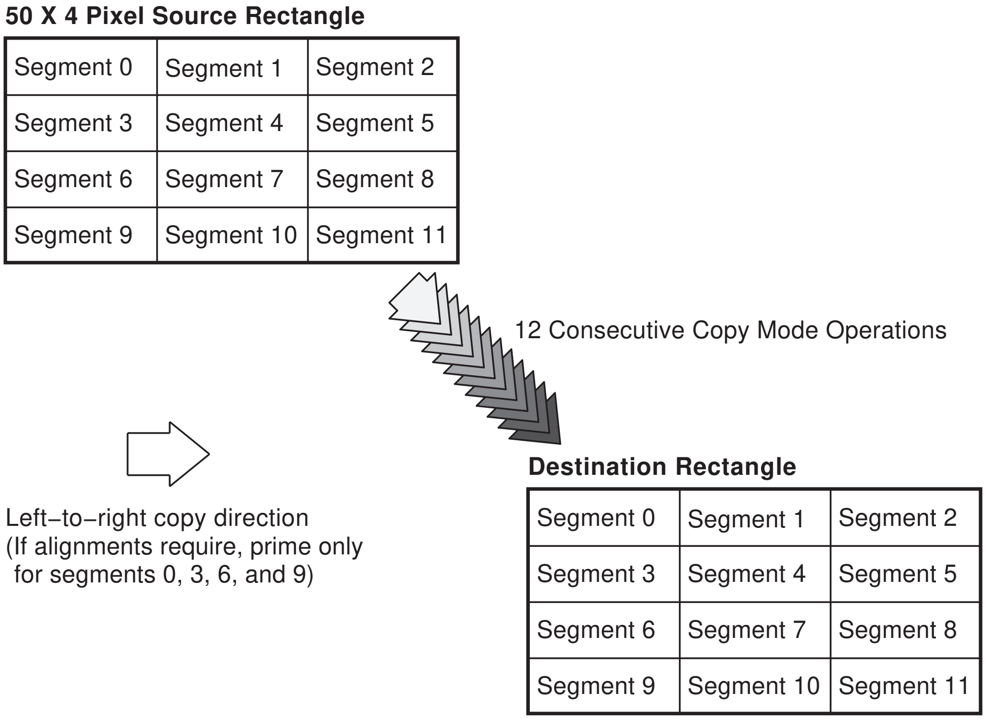
For high performance, screen-to-screen copy is the most important function to accelerate. The 20130 copy mode, 64-bit memory port, and 64-byte copy buffer all contribute to the high speed screen-to-screen copies.

Typically, driver-level calls move a rectangular source region to a destination region, and, possibly, apply a Boolean raster operation to the source and destination. Because the copy mode (Section 6.2.9) supports only span copies, software must break the rectangle into as many individual spans as necessary, with the width of each span equal to the width of the rectangle. Furthermore, it must break each arbitrary-width span into as many individual segments as necessary, with the length of each segment equal to 16, 32 or 64 pixels, depending on the frame buffer, bitmap, and masking used.

For overlapping source and destination spans, software must choose the proper copy direction (right-to-left or left-to-right), so that the source is not corrupted before it is read. The copy mode supports both directions, and maintains the internal state of the residue register for unaligned copies. Therefore, software must prime the residue register for only the first span segment (if necessary), rather than for each segment copy. Priming for subsequent segments occurs on the last read of the previous segment.

Figure 7–1 shows how an arbitrary rectangle can be broken into segments and where priming and flushing occur, if necessary.

Figure 7–1 BitBlt Using Copy Mode Example



For the most efficient copying of 8-bpp spans, the copy-64 source and destination registers (GCSR and GCDR, Section 4.3.4) use the entire copy buffer. Although the GCSR and GCDR can copy only aligned, unmasked span segments, they can be used to copy the interior of a large unaligned copy, where the left and right edges are copied by direct writes to the frame buffer.

The 21030 provides 16 raster-operation encodings to support the full set of 2-operand Boolean operations specified by X and OpenGL, but not the full set of Win32 graphics operations. (Win32 supports 256 ternary operations, two of which can be specified in a particular operation by a mask operand.) However, the 21030 raster operation encodings do include the most commonly used Win32 Boolean operations (such as srccopy and patcopy). (See the raster operation register description, Section 4.4.3.) Therefore, under Win32, if the Boolean operation passed in the DrvBitBlt call is not supported by the 21030, it can be broken into supported operations (if possible and desirable), or handled by the graphics device interface (GDI). Note that handling unsupported raster operations is not specific to BitBlts — raster operations are called into every Win32 device-driver interface (DDI) graphics call.

7.2.1.2 Host-to-Screen Copy

An image or bitmap can be copied between host memory and the 21030 frame buffer with X PutImage or GetImage calls or a Win32 DrvCopyBits call. Ideally, the DMA-read copy or DMA-write copy mode can be used, depending on direction. If DMA-read or DMA-write copy mode cannot be used, the image or bitmap can be burst-written directly into the 21030 frame buffer space using simple mode and standard programmed I/O. A final (and likely slower than simple mode) option is to write the copy buffer in standard programmed I/O, then unload the copy buffer with a write to the GCDR.

To use the DMA-read and DMA-write copy modes, the source rectangle must be broken into spans (as in the case of local frame buffer copies). However, because span lengths in the DMA modes are larger (>2K pixels) than any supported screen width, spans need not be broken down to less than 64K pixels. The residue register is primed and flushed as necessary, with appropriate address and pixel-count values. (See Sections 6.2.10 and 6.2.11 for more information about the DMA-read and DMA-write copy modes.)

7.2.2 Fills

Sections 7.2.2.1 through 7.2.2.3 describe filling, stippling, and tiling functions.

7.2.2.1 Solid

A region can be solid-filled with X FillSpan or PolyFillRect calls, or under Win32 with a DrvPaint call and a solid brush. The best way to do a solid fill is to use the block-fill, transparent-fill, or opaque-fill mode. The specific mode used depends on the following conditions:

- Fill region size
- Required raster operation
- Destination bitmap

The block-fill mode fills up to four times faster than the transparent- or opaque-fill modes and is preferable for larger fills. The block-fill mode can be used if the following conditions apply:

- The destination bitmap is not a packed 8-bpp bitmap in a 32-bpp frame buffer.
- Only a trivial raster operation, such as $dest \Leftarrow src$, is required.

Because the block- and transparent-fill modes only fill spans, software must do the following:

1. Break the fill region into spans no longer than 2K pixels.
2. In the block-fill mode, replicate the solid color across the block-color registers (GBCR<7:0>, Section 4.4.4).

   In the transparent-fill mode, replicate the solid color as necessary across the foreground register (GFGR, Section 4.4.18).
3. Write the frame buffer as many times as necessary to fill the spans.

(The fill modes are described in Sections 6.2.6 through 6.2.8.)

7.2.2.2 Stippling or Filling with a Monochrome Brush

When stippling or filling with a monochrome brush, a 1-bpp bitmap is expanded into a foreground (and optionally, background) color to tile a solid or bitonal pattern across a region. The opaque-stipple, opaque-fill, block-stipple, or block-fill mode can be used, depending on the following conditions:

- Size of the fill region
- Number of pixels at which the pattern repeats
- Raster operation
- Destination bitmap
- Masking

The block-fill mode runs up to four times faster than the transparent- or opaque-fill mode, and is preferable for larger regions. The block-fill mode can be used if the following conditions apply:

- The pattern repeats at intervals of 2^i and $i \leq 3$.
- Only a trivial raster operation, such as $dest \Leftarrow src$, is required.
- The destination is not a packed 8-bpp bitmap in a 32-bpp frame buffer.
- The mask, if used, repeats at intervals of 2^j and $j \leq 5$.

Filling a region with a 4×4 or 8×8 monochrome brush is a common Windows operation. The block-fill or block-stipple modes work in many typical cases and fill extremely fast. The block-fill mode is appropriate for larger regions because of the initial overhead required to set up the block-color pattern. If arbitrary per-pixel masking is required, the block-stipple mode, rather than the block-fill mode, can be used with a separate stipple-mask passed for every 32-pixel span.

When either the block-fill or block-stipple mode is used, software must do the following:

1. Expand the pattern into a full-depth 8-pixel block-color pattern.
2. Rotate the pattern to an 8-pixel alignment and write it to the GBCRs.
3. Break the region into spans of up to 2K pixels (block fill) or 32 pixels (block stipple) in length.
4. For every span, write the frame buffer once in the appropriate mode.

If the block-fill or block-stipple mode cannot be used because a nontrivial raster operation is required, the opaque- or transparent-fill mode can be used. In such cases, the pattern must repeat at intervals of 2^i and $i \leq 5$, and the foreground and background color, rather than the block-color pattern, must be specified.

If none of the fill modes can be used or the region is very small, the opaque-stipple (or transparent-stipple) mode can be used, in conjunction with the foreground and background registers.

7.2.2.3 Tiling or Filling with a Non-Monochrome Brush

When tiling or filling with a non-monochrome brush, a tile or brush pattern that is the same depth as the destination bitmap is repeated across the fill region. The width and number of colors in the pattern can be arbitrary. The block-stipple and block-fill modes are preferable, because of their fast fill rates, but cannot be used in all cases. The simple, copy, or DMA-read copy mode can be used to accelerate operations in which block mode operations are not possible.

The block-fill mode is preferable for large fill operations because of the initial overhead required to set up the block-color pattern. The block-fill mode can be used if the following conditions apply:

- The width of the brush pattern (or tile) is 2^j and $j \leq 3$.
- Only a trivial raster operation, such as $dest \Leftarrow src$, is required.
- The destination is not a packed 8-bpp bitmap in a 32-bpp frame buffer.

Whether to use the block-fill or block-stipple mode depends on the type of masking required for each pixel. The block-fill mode can be used if masking is not used or the mask repeats at intervals of 32 pixels; otherwise, the block-stipple mode can be used. In either case, software must do the following:

1. Expand and align the pattern to 8 pixels (if not previously so aligned).
2. Write the resulting 8-pixel block-color pattern to the GBCRs.

3. Break the region into spans of up to 2K pixels (block fill) or 32 pixels (block stipple) in length.
4. For every span, write the frame buffer once in the appropriate mode.

If either block mode is inappropriate, the copy mode or DMA-read copy mode can be used to recopy the same pattern from off-screen memory or main memory, respectively, to the destination as many times as necessary.

In the DMA-read copy mode, the 21030 can read more than 100 MB per second from the PCI bus. It can write the frame buffer at approximately the same rate, depending on the length of the copy. Therefore, the DMA-read copy mode can theoretically tile (brush) at approximately 100 MB per second; however, the actual rate in a specific system implementation varies as a function of the PCI bus performance (that is, latency, burst lengths, use, and so on). By comparison, the standard copy mode fill rate is more than 50 MB per second.

The simple mode can also be used, and might be the best choice for small regions.

7.2.3 2D Lines

The X PolyLine or PolySegment calls or the Win32 DrvStrokePath call can request 2D lines. The 21030 can draw lines in either of the following ways:

- Standard mechanism — Software initializes the Bresenham terms and then writes the frame buffer to initiate the drawing operation.
- Alternate mechanism — Software writes a slope register (GSLR<7:0>, Section 4.3.1) and the 21030 automatically generates the Bresenham terms and initiates the drawing operation.

Drawing with the GSLRs is preferred because it is significantly faster than drawing lines using the standard mechanism. In either case, the continue register (GCTR) can be used to extend lines to an arbitrary length.

Typically, all X lines can be drawn using the GSLRs. Conversely, the GSLRs cannot always be used to draw Win32 lines.

7.2.3.1 Line Drawing Under X

The following sequences list the steps for drawing various types of 2D lines under X.

- **Solid or Bitonal Lines**
  1. Set the mode to opaque- or transparent-line mode, as desired.
  2. Set the foreground and background colors in the foreground register (GFGR) and background register (GBGR, Section 4.4.19).

3. Write the starting address to the address register (GADR, Section 4.4.2).
4. Initialize the data register (GDAR, Section 4.4.8) to XXXXFFFF to draw all pixels (X = unused).
5. Write the appropriate GSLR.
6. Use the GCTR to extend the line to the desired length.

- **Patterned or Styled Lines**

  Do the solid or bitonal lines sequence described above, but write the desired pattern, rather than XXXXFFFF, to the GDAR.

- **Connected Lines**

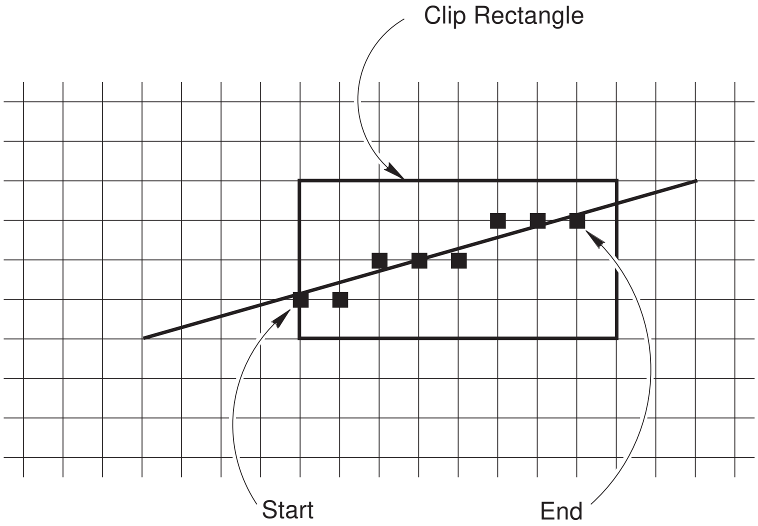
  Do the solid or bitonal lines sequence described above, but do not write to the GADR. The 21030 will draw the new line starting 1 pixel beyond the end of the previous line.

- **Clipped Lines**

  Figure 7–2 shows a clipped line drawn through a clipping rectangle.

  1. Write slope-no-go register 7 (GSNR7, Section 4.4.9) to draw in octant 7.
  2. Write the starting pixel address to the GADR.
  3. Write the initial error and line length to the Bresenham 3 register (GB3R).
  4. Write the GCTR to draw the line (and repeat for rectangle wider than 16 pixels).
  5. Repeat steps 2 through 4 for each rectangle in the clip list.

Figure 7–2 Drawing Clipped Lines



7.2.3.2 Line Drawing Under Win32

The coordinate constructs supported by Win32 do not allow the GSLRs to be used to draw all lines that the GDI might request from the display driver. To improve the appearance of rendered lines, Win32 supports subpixel coordinates. Each coordinate is in the 28.4 format (28 integer bits and 4 fraction bits). The coordinate system can be visualized as a grid in which an endpoint can reside at any grid intersection, but pixels reside at every sixteenth pixel in both X and Y. (For more information, see the Win32 device-driver kit documentation.) The following is a more detailed description of this ''problem'' and its ''solution.''

Problem

On a write to a GSLR, the hardware sets up the line; that is, it translates the absolute *dx* and absolute *dy* information into the Bresenham address and error increment values and initial error term. However, absolute *dx* and absolute *dy* are 16-bit quantities, assumed to be 16 bits of integer and 0 bits of fraction. This presents two problems:

- The setup hardware cannot properly correct a subpixel endpoint to the pixel centers.

- Lines passed by the GDI can be too long for the Bresenham engine to render without the risk of introducing error in the digital data-analysis (DDA, or pixel-stepping) calculation.

The first problem is complex. Determining the starting pixel can be critical, and depends on the location of the endpoint in the subpixel grid. For some endpoints, the first pixel drawn is the pixel closest to the intersection of the geometric line and the first major-axis grid, rather than the pixel nearest to the specified endpoint. That is, the first pixel drawn can be a function of the starting subpixel endpoint and the slope.

The 21030 setup hardware cannot handle such constructs. It cannot correctly choose the address of the proper endpoint and cannot calculate the proper Bresenham initial error term. The initial error generated would be relative to the first subpixel rather than the first real pixel, which can be up to 16 subpixels away.

Secondly, if the 21030 could properly correct the starting address error term, the 21030 DDA, with 16 bits of resolution, cannot draw lines greater than 64K pixels in major-axis length with guaranteed accuracy. Consequently, the GSLRs cannot be used to draw any line with endpoints having nonzero fractional components or any line longer than 64K pixels.

Solution

To support all possible lines requested by the Win32 GDI, the 21030 Win32 display driver must use a combination of GSLR accesses, direct manipulation of the Bresenham registers, and writes to the frame buffer. The following is a suggested strategy for dealing with an arbitrary GDI line drawing request:

1. If the line is less than 64K pixels in the major-axis, go to step 2. Otherwise, do either of the following:
   - Default to the GDI.
   - Break long lines into smaller segments and go to step 2.
2. Screen for endpoints with nonzero fractional components.

   For integer endpoints, draw by writing the GSLR, passing a 16.0 format value for absolute *dx* and absolute *dy*.

   For noninteger endpoints, do the following (and refer to the Win32 device-driver kit documentation for more detail):

   a. Determine the starting pixel and calculate the address.

b. Write a GSNR to calculate the address and error increment terms (that is, the parameters in the GB1R and GB2R) passing 12.4 format values for absolute *dx* and absolute *dy*. The setup process will calculate these terms correctly, regardless of the number of fractional bits.

c. Adjust the initial error term relative to the starting pixel. This can be done by performing the DDA at subpixel increments until the first major-axis grid is reached (which might be necessary in any case) and scale the error term. Write the error term to the GB3R initial error field.

d. Write the address of the starting pixel to the GADR and draw the first 16-pixel segment with a write to the GCTR. Repeat this step for lines longer than 16 pixels.

In effect, this operation appears to be a clipped line with the edge of the clipping rectangle set at the first integer major-axis grid crossed by the geometric line.

7.2.3.3 21030 Turbo Lines

The unique 21030 RapiDraw feature greatly improves the drawing performance for tall, thin objects, such as lines.

The 21030's frame buffer interface implements four 16-bit memory controllers (Section 3.9). Each controller can independently address each 16-bit channel of the 64-bit frame buffer. When standard on-screen pixel widths of 1024 or 1280 are linearly mapped into the 21030's frame buffer, vertically contiguous pixels map into the same channel. Such mapping does not take advantage of the independent memory controllers' ability to draw consecutive pixels in parallel. However, specifying a visible width 4 pixels longer (for example, 1028 rather than 1024), effectively skews the scan lines by 4 pixels when mapped into the frame buffer memory. Skewing maps vertically contiguous screen pixels into different channels, allowing such pixels to be simultaneously drawn. The vertical line drawing speed is doubled, and line drawing performance increases approximately 50% compared to non-skewed mapping.

To take advantage of this *turbo* line support, software must do the following:

- When calculating screen addresses from coordinates, use a screen line width that is 4 pixels greater than the standard screen width.
- Specify a value in the horizontal-control register active field (VHCR <29:28,8:0>, Section 4.5.1) that is 4 pixels greater than the monitor requires.
- Set the odd bit (VHCR <31>) to inhibit the display of the 4 extra pixels during screen refresh.

For example, to set the parameters for a 1280 $\times$ 1024 monitor, the active value should be 1284 and the odd bit should be set.

7.2.4 Text

The 21030 stipple modes can process a request for any of the X text or glyph calls or the Win32 DrvTextOut call. The opaque-, transparent-, and, in some cases, block-stipple mode can be used, depending on the following:

- The destination bitmap
- Whether a nontrivial raster operation is required
- Whether the text foreground is filled with a solid, monochrome, or arbitrary patterned brush or tile

The block-stipple mode draws up to four times faster than the other stipple modes, and can be used if any of the following brushes is used:

- A solid brush is used, and it:
  - Requires only a trivial raster operation, such as $dest \Leftarrow src$
  - Does not draw to a packed 8-bpp bitmap in a 32-bpp frame buffer
- A monochrome or arbitrarily patterned brush is used, and the pattern repeats at intervals of 2^j and $j \leq 3$.

The sequence for drawing in block-stipple mode is as follows:

1. Expand and align the solid or properly repeating patterned brush to 8 pixels.
2. Write the result to the block-color pattern in the GBCRs.
3. Write the glyph, span-by-span, with writes to the frame buffer in block-stipple mode. Each write passes the glyph pattern for the span and writes up to 32 pixels. For example, rendering an 8 $\times$ 16 ($x \times y$) glyph requires 16 stipple-mode span-drawing operations.

If conditions prohibit block stipple-mode, opaque- or transparent-stipple mode can be used. Transparent-stipple mode is used for a solid brush, with the glyph mask specified as the stipple mask. Opaque-stipple mode is used for a monochrome brush, with the glyph mask specified as the pixel mask. In either case, if a mix raster operation is specified for the foreground under Win32, each raster operation requires two passes using transparent-stipple mode. For an arbitrarily patterned brush (that is, other than simple monochrome) that does not repeat at appropriate intervals, simple mode can be used to write the glyph foreground through the glyph mask.

All stipple modes allow up to 32 pixels to be drawn per operation. Therefore, it is advantageous to try to group spans from multiple glyphs that are contiguous in display memory. For example, rather than draw four 8×16 glyphs one at a time, draw all four in parallel, one span at a time — one write can draw one span from each glyph at the same time.

7.2.5 3D Lines

The 3D line modes (Section 6.2.14) generate any of the following standard 3D primitives:

- Gouraud-shaded spans, optionally Z-buffered or dithered to 8-bpp or 12-bpp by hardware
- Depth-cued lines, optionally Z-buffered or dithered to 8-bpp or 12-bpp by hardware
- 8-bpp smooth-shaded grey-scale lines, optionally Z-buffered by hardware

(Table 6–3 lists the modes to use for each primitive.)

The following drawing sequences are for several typical 3D spans and lines.

- **Flat-Shaded Line or Span**

  A flat-shaded line or span is visually the same as a 2D line and is programmed as such.

- **Flat-Shaded, Z-Buffered Line or Span**

  1. Write the following registers:
     a. Foreground register (GFGR) if changing color
     b. Z-increment registers (GZIR-H and GZIR-L)
     c. Z-value registers (GZVR-H and GZVR-L) if a noncontiguous segment
     d. Z-base-address register (GZBR) if a noncontiguous segment
     e. Address register (GADR) with the frame buffer address if a noncontiguous segment
     f. A slope register (GSLR) with absolute *dx* and absolute *dy* to begin drawing up to 16 pixels
  2. For longer than 16-pixels, repeatedly write the continue register (GCTR) to extend the line as necessary.

- **Depth-Cued Line**
  1. Write the following registers:
     a. Red-increment register (GRIR)
     b. Green-increment register (GGIR) if not grey-scale
     c. Blue-increment register (GBIR) if not grey-scale
     d. Address register (GADR) with the frame buffer address if a noncontiguous segment
     e. Red-value register (GRVR) if a noncontiguous segment
     f. Green-value register (GGVR) if not grey-scale and a noncontiguous segment
     g. Blue-value register (GBVR) if not grey-scale and a noncontiguous segment
     h. A slope register (GSLR) with absolute *dx* and absolute *dy* to begin drawing up to 16 pixels
  2. For longer than 16-pixels, repeatedly write the continue register (GCTR) to extend the line as necessary.
- **Depth-Cued Z-Buffered Lines**
  1. Write the following registers:
     a. Red-increment register (GRIR)
     b. Green-increment register (GGIR) if not grey-scale
     c. Blue-increment register (GBIR) if not grey-scale
     d. Z-increment registers (GZIR-H and GZIR-L)
     e. Red-value register (GRVR) if a noncontiguous segment
     f. Green-value register (GGVR) if not grey-scale and a noncontiguous segment
     g. Blue-value register (GBVR) if not grey-scale and a noncontiguous segment
     h. Z-value registers (GZVR-H and GZVR-L) if a noncontiguous segment
     i. Z-base-address register (GZBR) if a noncontiguous segment
     j. Address register (GADR) if a noncontiguous segment

k. Span width register (GSWR) to begin drawing up to 16 pixels

2. For longer than 16-pixels, repeatedly write the continue register to extend the span as necessary.

- **Gouraud-Shaded Z-Buffered Spans**

1. Write the following registers for each polygon:

   a. Red-increment register (GRIR)

   b. Green-increment register (GGIR) if not grey-scale

   c. Blue-increment register (GBIR) if not grey-scale

   d. Z-increment registers (GZIR-H and GZIR-L)

2. Write the following registers for each span:

   a. Red-value register (GRVR)

   b. Green-value register (GGVR) if not grey-scale

   c. Blue-value register (GBVR) if not grey-scale

   d. Z-base-address register (GZBR)

   e. Z-value registers (GZVR-H and GZVR-L)

   f. Address register (GADR)

   g. Span width register (GSWR) to begin drawing up to 16 pixels

3. For longer than 16-pixels, repeatedly write the GCTR to extend the span as necessary.

The colors in any of the preceding lines and spans can be optionally dithered and drawn to an 8-bpp or 12-bpp bitmap in a 32-bpp frame buffer.

The hardware Z-buffering is also advantageous for drawing that requires more advanced shading techniques, such as Phong shading or texture mapping. Rather than using one of the 3D line modes, the CPU can calculate the colors and Z-reference for each pixel, and write the pixels, one at a time, using simple-Z mode. However, depending on the primitive and application, this may be slower than doing all of the rendering and Z-buffering in software.

7.2.5.1 Software Z-Buffering

Although the 21030's performance is optimized to do Z-buffering in hardware, the line and span hardware also works well with Z-buffering performed in software. When this is done, a pixel mask is created to determine which pixels are written as a function of Z. The created Z-mask and the standard pixel mask are logically ANDed and then passed as a unified line mask either to the GDAR or on a PCI write in line mode.

7.2.6 3D Polygons

Software must break polygons into spans. It must walk all of the polygon edges and create a sequence of Gouraud-shaded (and optionally, Z-buffered) spans that can be programmed as described in Section 7.2.5. Note that the color and Z span-increment parameters need not be reloaded for each span.

7.2.7 Animations

The 21030 can use double-buffering to accelerate animation sequences. It supports two types of double-buffering: in-place double-buffering in 32-bpp frame buffer options and standard double-buffering. Software can draw to one buffer while displaying from the other.

In both types of double-buffering, software must be synchronized with the screen display to avoid tearing. Tearing occurs at the start of vertical retrace if the entire screen is double-buffered, or in the middle of the display if a window is being animated. Waiting until the display is complete eliminates tearing. Typically, the buffer swap begins as soon as the scan completes drawing to the region to be animated (for example, a window or the entire screen).

The 21030 provides two types of interrupts to synchronize software with screen refresh: end-of-frame interrupts and shift-address interrupts. Both types of interrupt are programmed in the interrupt status register (SISR, Section 4.7.2). The end-of-frame interrupt is used to wake up software at the start of vertical retrace, and the shift-address interrupt is used to wake up software at a programmable location in mid-screen.

7.2.7.1 Offscreen-Copy Double-Buffering

In a standard double-buffering scheme, the double-buffer is physically located apart (that is, off-screen) from the display buffer. After the current buffer has been displayed, the double-buffer is physically block-transferred to the screen. The following is a typical sequence for doing animation in an 8-bpp frame buffer.

On an end-of-frame or shift-address interrupt, do the following:

1. Use the copy mode to block-transfer the off-screen buffer to the on-screen buffer.
2. Clear the off-screen buffer. Use the block-fill mode to clear the back-buffer as quickly as possible.
3. Clear the Z-buffer (if doing Z-buffered drawing). Use the block-fill mode to clear the Z-buffer as quickly as possible.
4. Draw the (optionally Z-buffered) image to the off-screen buffer. Use the 21030's 3D line modes to draw 3D lines or polygons.
5. Wait for the next interrupt.

7.2.7.2 In-Place Double-Buffering

In an in-place double-buffering scheme, each 32-bpp pixel can have three 8-bpp pixels or two 12-bpp pixels, and each can be allocated to a separate unpacked buffer (or bitmap). Each pixel starts at a different offset within the 32 bits. Destination bitmap and destination byte can be specified as required to draw to each distinct bitmap.

When refreshing the screen, a Bt463 RAMDAC can select a different offset on a per-region (window) basis. For example, on one frame, software can specify an offset in the Bt463 to refresh from bitmap format $UB8_0$ while drawing to $UB8_1$. On the following frame, the offset can be switched to refresh from bitmap format $UB8_1$ while moving on to draw to $UB8_0$. All the buffer switches are accomplished by specifying a different offset; a BitBlt is not required (that is, the buffers remain in place).

In the 21030, the upper byte of a 32-bit pixel is a tag that can be used for anything the application desires. When doing in-place double-buffering, software should use at least some of the tag bits to identify the region to which the pixel belongs. When the tag is used in this way, the in-place buffers can be switched by simply writing a new tag (or tag subset) and instructing the Bt463 to refresh from the range of the 32-it pixel that corresponds to that tag.

In-place double-buffering is the optimal method of double-buffering in a 32-bpp frame buffer, provided that the RAMDAC can support a selectable range from the incoming 32-bit pixel. Because of the 21030's superior dithering, 8-bpp image quality is excellent, and 12-bpp image quality is indistinguishable from 24-bpp image quality. Consequently, animating with less than full-color resolution provides an increase in animation rates with fewer memory parts.

The following is a typical in-place double-buffering sequence:

On an end-of-frame or shift-address interrupt, do the following:

1. Switch the buffers and clear the back-buffer (that is, clear the buffer that was just displayed).

   Use the block-fill mode to perform both functions simultaneously. Specify a 32-bpp destination and set the GBCRs to the background color. Use the block-fill mode with a plane mask to isolate the tag and desired bitmap field.

2. Clear the Z-buffer (if doing Z-buffered drawing). Use the block-fill mode to clear the Z-buffer as quickly as possible.

3. Draw the (optionally Z-buffered) image to the back-buffer. Use the 21030's 3D line modes to draw 3D lines or polygons.

4. Wait for next interrupt.

7.2.8 Cursor Display

Moving the cursor off the top of the screen is often a problem because most monitors specify fewer than 64 lines between the assertion of vertical sync and the end of the vertical back porch (that is, the top of the displayable screen). Consequently, the entire cursor height cannot ''fit'' off the top of the screen. To compensate for this situation, the driver must specify the cursor *y* (CXYR <23:12>, Section 4.6.1) as cursor *y* minimum (Table 4–63) and then specify a cursor base address (CCBR <9:4>, Section 4.6.2) that offsets into the cursor array, such that the top of the cursor is ''cut off.'' The offset the driver adds to the cursor base address depends on the number of scans cut off the top of the cursor.

7.3 Programming for Alpha AXP CPUs

Sections 7.3.1 and Section 7.3.2\VALUE) describe special programming considerations when using the 21030 with Alpha AXP microprocessors.

7.3.1 Programmed I/O Through the CPU Write Buffer

The DECchip 21064, 21066, and 21068 Alpha AXP microprocessors contain an internal 4-entry write buffer. To optimize the use of system bus bandwidth, the write buffer attempts to collapse and merge quadwords (64 bits) and Dwords before they are written externally. This mechanism has an unwanted side-effect on write-ordering. Specifically, an ordered-packet of Dwords written by a simple string of STL instructions (as in writing a command packet to the 21030) is not necessarily written on the PCI bus in the same order or with all the Dwords intact.

To counter this unwanted side-effect, a 21030 driver running on an Alpha AXP microprocessor must:

- Avoid collapsing two separate writes to the same address
- Enforce write-ordering to order-critical 21030 registers

To enforce write-ordering, the Alpha AXP instruction set includes the memory barrier (MB) instruction that allows software to flush the write buffer between stores. However, the MB instruction significantly degrades performance when it is used as frequently as is necessary with an order-dependent, bandwidth-consuming, programmed I/O device such as the 21030. Therefore, to selectively enforce ordering and eliminate collapsed writes, the 21030 software can:

- Access multiple aliased regions in the 21030 address space
- Carefully order accesses within aligned hexawords (eight Dwords) as appropriate

The 21030 memory space provides multiple aliases to access the 21030 registers as well as the frame buffer. In most cases, the multiple address-space aliases can be used to work around the CPU write buffer's lack of ordering, without using MB instructions. (Memory space can be extended for 21030 options on Alpha AXP systems to create the appropriate number of core space aliases. See Sections A.1 and A.1.1.)

For example, rather than writing to the same register twice and issuing an explicit MB instruction, software can write to two aliases of the same register. The different addresses will reside in different write-buffer entries, such that the writes will not merge and will maintain ordering.

Ordering within each CPU write-buffer entry must also be carefully monitored. Each hexaword (eight Dwords) write-buffer entry empties from least significant to most significant Dword (or so it appears on the PCI bus). Therefore, stores to the same hexaword are in low-to-high order regardless of when they were written.

However, strict ordering is not necessary for all writes to the 21030. A typical graphics drawing command packet (Section 6.1.2) written to the 21030 consists of several order-independent register writes, followed by an ordered write to another register or the frame buffer. The first several writes can be arbitrarily reordered among themselves, but they all must appear after the previous command packet and before the last write of the current packet.

The 21030 register-space core map is organized by hexaword to map cleanly to the CPU’s write buffer. Within a typical command packet, order-independent register writes are mapped in the same hexaword, and the order-dependent register or frame buffer write is mapped either in the most-significant Dword location of the same hexaword or in another hexaword. If software needs to address another hexaword entry for the order-dependent write, it should choose a different alias for every fourth consecutive access. The order-dependent write then always appears after the order-independent writes.

7.3.2 Address and Continue Register Access

The alternate ROM space aliases of the address and continue registers (GADR and GCTR) is another mechanism for using the unenforcing write buffer in Alpha AXP processors. The GADR maps to all the even offsets in the first 512KB of alternate ROM space, and the GCTR maps to all the odd offsets (Section 2.2.3.2).

Any graphics operation invoked by a write to the frame buffer can also be invoked by a write to the GADR followed by a write to the GCTR. This allows the 210030 to be programmed by a continuous stream of alternating writes to the GADR and GCTR. By taking advantage of the odd and even aliases in alternate ROM space, software can effectively pack GADR-GCTR writes in the CPU write buffers. This also minimizes the translation-lookaside buffer (TLB) overhead in the CPU, because all the writes are local.

8 Hardware Interface

This chapter describes the frame buffer organization, external device interfaces, and external signals on the 21030 pins.

8.1 Frame Buffer Organization

Sections 8.1.1 through 8.1.3 describe the 8- and 32-plane frame buffer options and supported memory devices.

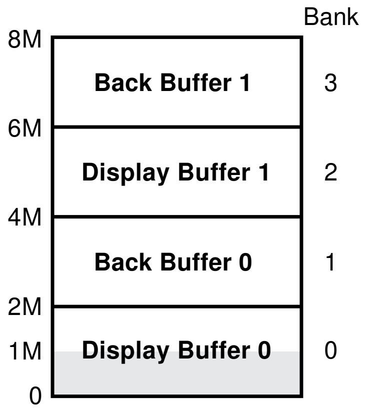
8.1.1 8-Plane Frame Buffer

The 8-plane frame buffer consists of one segment of memory which includes up to four contiguous banks. Figures 8–1 through 8–4 show typical 8-bpp frame buffer configurations.

Figure 8–1 Frame Buffer Option T8-01

Frame Buffer Depth: 8–bpp
Banks Populated: 1
Total Physical Size: 1MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	256	No	NA

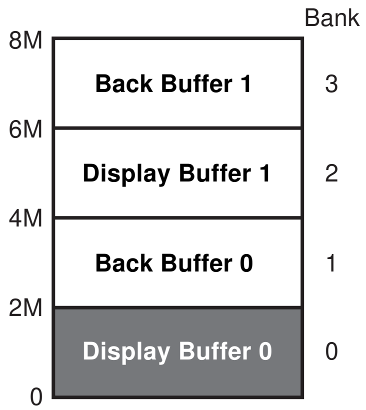


Unpopulated | 128K X *N* VRAM | 256K X *N* VRAM | 256K X *N* DRAM

Figure 8–2 Frame Buffer Option T8-02

Frame Buffer Depth: 8–bpp
Banks Populated: 1
Total Physical Size: 2MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	256	Yes	NA
1280 X 1024	256	No	
1600 X 1200	256	No	

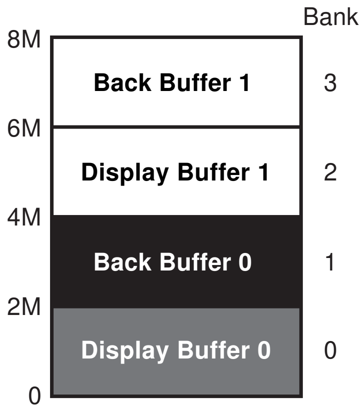


Unpopulated | 128K X *N* VRAM | 256K X *N* VRAM | 256K X *N* DRAM

Figure 8–3 Frame Buffer Option T8-22

Frame Buffer Depth: 8–bpp
Banks Populated: 2
Total Physical Size: 4MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	256	Yes	NA
1280 X 1024	256	Yes	
1600 X 1200	256	Yes	

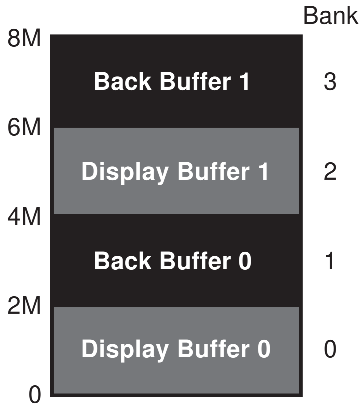


Unpopulated   128K X *N* VRAM   256K X *N* VRAM   256K X *N* DRAM

Figure 8–4 Frame Buffer Option T8-44

Frame Buffer Depth: 8–bpp
Banks Populated: 4
Total Physical Size: 8MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	256	Yes	NA
1280 X 1024	256	Yes	
1600 X 1200	256	Yes	



Unpopulated   128K X *N* VRAM   256K X *N* VRAM   256K X *N* DRAM

8.1.2 32-Plane Frame Buffer

The 32-plane frame buffer consists of two memory segments, each of which contains up to four contiguous 2MB banks. A 32-plane frame buffer can support a total of 16MB of physical memory.

In an 8-bpp frame buffer, four **cas_l<3:0>** pins select one of 16 channel-banks (4 banks $\times$ 4 channels). This is possible because **rasen_l<3:0>** enables RAS in only one bank per segment in an 8-bpp frame buffer. The bank on the selected channel that responds to the CAS cycle is selected by **rasen_l<3:0>**. In a 32-bpp frame buffer, RAS is enabled for an entire segment at a time. Because each segment requires 16 independent CAS controls (one control per channel-bank), the 21030 provides the **casen<1:0>** pins to help select the bank of VRAM that should respond to the **cas_l<3:0>** pin for that channel.

8.1.2.1 Horizontal Access Mode

When drawing to a 32-bpp frame buffer, the 21030 typically accesses two consecutive aligned Dwords (that is, two consecutive pixels). This translates to a 64-bit wide, horizontal access across one of the four banks. If 32-bpp frame buffer accesses were limited to this mode, **casen<1:0>** would be all that is required to indicate which of the four banks are enabled. However, the 32-bpp frame buffer options also support broadcast and diagonal pixel-access modes.

8.1.2.2 Broadcast Access Mode

The broadcast frame-buffer access-mode supports simultaneous access to all banks in the segment when all of the following are also supported:

- CAS-before-RAS dynamic memory refresh cycles
- Split and standard read-transfer cycles to support screen refresh
- Writes to the VRAM plane mask registers

8.1.2.3 Diagonal Access Mode

The diagonal frame-buffer access-mode supports access to unpacked 8-bpp bitmaps. When drawing to unpacked 8-bpp bitmaps in a 32-bpp frame buffer, the 21030 effectively reads 1 byte from each of 8 contiguous aligned Dwords.

In horizontal mode, each 8-byte read or write accesses 2 consecutive aligned Dwords. In contrast, each 8-byte read or write in diagonal mode accesses 1 byte from each of 8 consecutive Dwords. The 21030 uses diagonal mode when accessing unpacked 8-bpp bitmaps and horizontal mode when accessing all other types of bitmaps. The **casmode<1:0>** signals encode whether the access mode is diagonal, horizontal, or broadcast.

In a 32-bpp frame buffer, an external device (for example, a PLD) is required to generate the 16 independent channel-bank CAS controls from the **cas_l<3:0>**, **casen<1:0>**, and **casmode(1:0)** signals. In summary, **cas_l<3:0>** enables each channel for independent access and **casen<1:0>** and **casmode<1:0>** select the bank to respond when the **cas_l<3:0>** signal for that channel is active.

Figures 8–5 through 8–7 show typical 32-bpp frame buffer configurations.

Figure 8–5 Frame Buffer Option T32-04

Frame Buffer Depth: 32–bpp
Banks Populated: 2
Total Physical Size: 4MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	16M	8/12–bpp	No

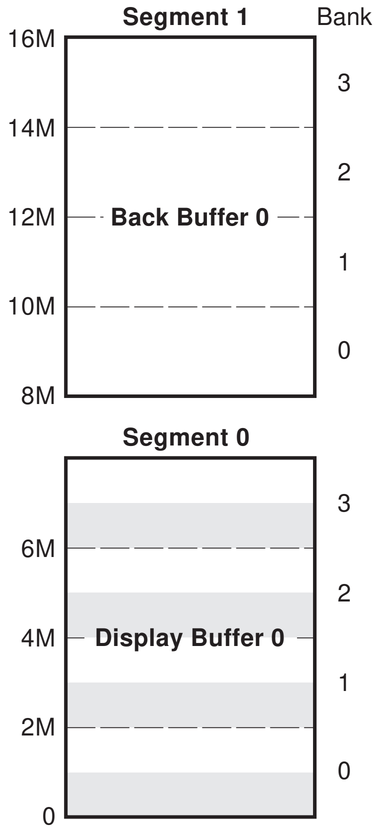
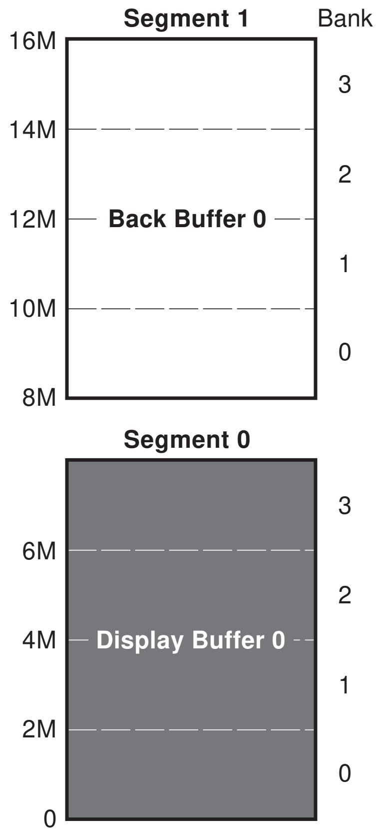


Figure 8–6 Frame Buffer Option T32-08

Frame Buffer Depth: 32–bpp
Banks Populated: 4
Total Physical Size: 8MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	16M	32–bpp	No
1280 X 1024	16M	8/12–bpp	No
1600 X 1200	16M	8/12–bpp	No



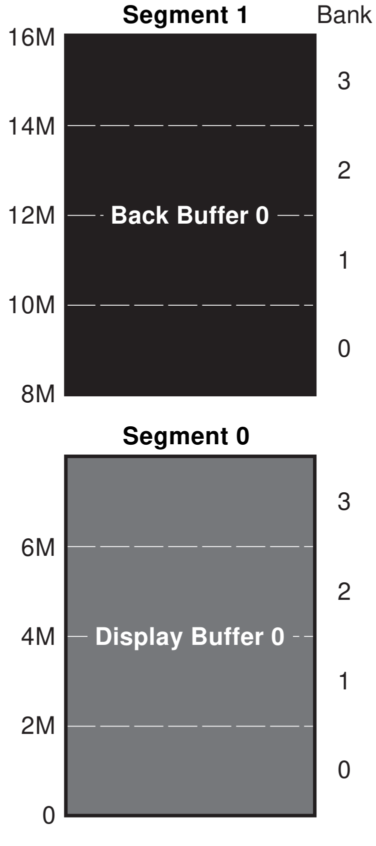
(continued on next page)

Figure 8–7 (Cont.) Frame Buffer Option T32-88

Figure 8–7 Frame Buffer Option T32-88

Frame Buffer Depth: 32–bpp
Banks Populated: 8
Total Physical Size: 16MB

Resolution	Colors	Double Buffer	Z Buffer
1024 X 768	16M	32–bpp	Yes
1280 X 1024	16M	8/12–bpp	Yes
1600 X 1200	16M	8/12–bpp	Yes



8.1.3 Supported Memory Devices

Display-buffer memory must be populated with VRAMs. Back-buffer memory can be populated with DRAMs but special care must be taken. The 21030 supports block-mode operations and the plane-mask function to accelerate and simplify drawing. Both features rely on memory device support for block-write cycles and persistent plane-mask, which are currently supported in only a few types of DRAM such as the 256K $\times$ 16 graphics DRAM (GRAM). To ensure the same performance with the same software whether drawing to back-buffer or display-buffer memory, back buffers must be populated with VRAMs or GRAMs. Conversely, if back buffers are populated with standard DRAMs, software must not specify block-mode operations or use a plane mask when drawing to the back buffers.

8.2 System Configurations with VGA

Systems using the 21030 can implement VGA pass-through support in several different ways, depending performance, cost, and option integration requirements. Additional information can be found in *VGA Pass-Through Support for DECchip 21030 Boards: An Application Note* which describes several possible configurations, including the following:

- The 21030 and ISA/EISA VGA on the motherboard
- The 21030 and PCI VGA on the motherboard
- The 21030 on a PCI option board and VGA on the motherboard
- The 21030 on a PCI option board and VGA in an ISA/EISA option slot
- The 21030 and VGA on a PCI option board

8.3 External Device Interfaces

Additional information about the RAMDAC, EEPROM, and clock generator interfaces can be found in *Interfacing to External Devices with DECchip 21030 Boards: An Application Note*. The application note also describes the implementation of the following alternate uses for these interfaces.

- Using the EEPROM interface as a serial communications port
- Using the EEPROM interface as an 8-bit parallel write-only port
- Using the EEPROM interface as an 8-bit parallel read-only port
- Using the RAMDAC interface as a serial communications port
- Using the Clock generator interface as an 8-bit parallel write-only port

8.4 Signal Descriptions

Sections 8.4.1 through 8.4.4 describe the frame buffer interface, PCI interface, video interface, and test signals.

8.4.1 Frame Buffer Interface Signals

Sections 8.4.1.1 through 8.4.1.12 describe the frame buffer interface signals in alphabetical order.

8.4.1.1 addr<17:0>

The **addr<17:0>**, **addren_l<1:0>**, **dsf<1:0>**, **oe_l<1:0>**, and **we_l<7:0>** signals are inputs to external logic that generates the VRAM address and control (DSF, DTOE, and WE) signals.

The **addr<17:0>** signals also directly address the optional, external EEPROM.

The **addr<17:0>** signals are output-only signals. They are driven and undefined at reset.

8.4.1.2 addren_l<1:0>

The **addren_l<1:0>**, **addr<17:0>**, **dsf<1:0>**, **oe_l<1:0>**, and **we_l<7:0>** signals are inputs to external logic that generates the VRAM address and control (DSF, DTOE, and WE) signals. The **addren_l<1:0>** signals enable the external logic.

The **addren_l<1:0>** signals are output-only signals. They are driven low at reset.

8.4.1.3 cas_l<3:0>, casen<1:0>, casmode<1:0>

The **cas_l<3:0>**, **casen<1:0>**, and **casmode<1:0>** signals are inputs to external logic that generates the VRAM CAS signals. (See Section 8.1.2 for more information.)

The **cas_l<3:0>**, **casen<1:0>**, and **casmode<1:0>** signals are output-only signals. They are driven high at reset.

8.4.1.4 dacc<2:0>, dacce_l<1:0>, dacrw

The **dacc<2:0>**, **dacce_l<1:0>**, and **dacrw** signals form the MPU interface to the various external RAMDACs supported by the 21030.

The exact function of these register-programmable signals depends on the type of external RAMDAC. Table 8–1 shows some examples of how these signals are connected.

The **dacc<1:0>** signals are also the data and hold inputs to the serial port of an optional, external, clock generator chip.

The **dacc<2:0>** and **dacrw** signals are output-only signals. They are driven and undefined at reset. The **dacce_l<1:0>** signals are output-only signals, and are driven high at reset.

Table 8–1 shows how the 21030 **dacc<2:0>**, **dacce_l<1:0>**, and **dacrw** signals are connected to the MPU interface pins of several typical RAMDACs.

Table 8–1 RAMDAC MPU Interface Connection

RAMDAC Type	dacc2	dacc1	dacc0	dacrw	dacce_l1	dacce_l0
Bt463, Bt459, Bt458	—	C1	C0	RW	Head 1	Head 0
Bt484, Bt485	RS3	RS2	RS1	RS0	RD	WR

(See Sections 4.8.2 and 4.8.3 for more information.)

8.4.1.5 data<63:0>

The **data<63:0>** signals carry data between the 21030, VRAM, RAMDAC, and optional EEPROM.

The **data<63:0>** signals are bidirectional. They are driven and undefined at reset.

8.4.1.6 dsf<1:0>

The **dsf<1:0>**, **addr<17:0>**, **addren_l<1:0>**, **oe_l<1:0>**, and **we_l<7:0>** signals are inputs to external logic that generates the VRAM address and control (DSF, DTOE, and WE) signals.

The **dsf<1:0>** signals are output-only signals. They are driven low at reset.

8.4.1.7 fbclk

The **fbclk** signal is used by the frame buffer, RAMDAC, optional EEPROM, and optional clock generator interface logic.

The **fbclk** signal is an input-only signal.

8.4.1.8 icsce_l

The **icsce_l** signal is the chip enable or data strobe for an optional, external, clock generator chip. If the clock generator is not programmable, then the function of this pin is application-specific. (See Section 8.3 for more information.)

The **icsce_l** signal is an output-only signal, and it is driven high at reset.

8.4.1.9 oe_l<1:0>

The **oe_l<1:0>**, **addr<17:0>**, **addren_l<1:0>**, **dsf<1:0>**, and **we_l<7:0>** signals are inputs to external logic that generates the VRAM address and control (DSF, DTOE, and WE) signals.

The **oe_l** signal is an output-only signal, and it is driven high at reset.

8.4.1.10 ras_l<3:0>, rasen_l<3:0>

The **ras_l<3:0>** and the **rasen_l<3:0>** signals are the data input and enable signals to external logic that generates the VRAM RAS signals. (See Section 8.1.2 for more information.)

The **ras_l<3:0>** are output-only signals, and are driven high at reset. The **rasen_l<3:0>** are also output-only signals, but are driven low at reset.

8.4.1.11 romce_l, romoe_l, romwe_l

romce_l is the chip enable signal, **romoe_l** is the output enable signal, and **romwe_l** s the write enable signal for the optional, external EEPROM.

The **romce_l**, **romoe_l**, and **romwe_l** signals are output-only signals, and are driven high at reset.

8.4.1.12 we_l<7:0>

The **we_l<7:0>**, **addr<17:0>**, **addren_l<1:0>**, **dsf<1:0>**, and **oe_l<1:0>** signals are inputs to external logic that generates the VRAM address and control (DSF, DTOE, and WE) signals.

The **we_l** signal is an output-only signal, and is driven high at reset.

8.4.2 PCI Signals

Sections 8.4.2.1 through 8.4.2.14 describe the PCI interface signals in alphabetical order.

8.4.2.1 ad<31:0>

PCI address and data are multiplexed on the **ad<31:0>** pins. During the first clock of a PCI transaction, the byte address is driven on the **ad<31:0>** pins. During subsequent clock cycles, data is driven on the **ad<31:0>** pins.

The **ad<31:0>** signals are bidirectional signals and are tristated at reset.

8.4.2.2 cbe_l<3:0>

PCI bus command codes and byte enables are multiplexed on the **cbe_l<3:0>** pins. During the address cycle of a PCI transaction, the PCI bus command code is driven on the **cbe_l<3:0>** pins. During data cycles, inverted byte enables are driven on the **cbe_l<3:0>** pins.

The **cbe_l<3:0>** signals are bidirectional signals and are tristated at reset.

8.4.2.3 devsel_l

The target of a PCI transaction asserts the **devsel_l** signal when it detects an address matching its programmed address space.

The **devsel_l** signal is a bidirectional signal and is tristated at reset.

8.4.2.4 frame_l

The **frame_l** signal is asserted at the beginning of a PCI transaction. It is also used to control the number of data transfers during the transaction (burst length). The **frame_l** signal is deasserted during the final data phase of a transaction.

The **frame_l** signal is a bidirectional signal and is tristated at reset.

8.4.2.5 gnt_l

The **gnt_l** signal is asserted by external arbitration logic when the 21030 is granted ownership of the PCI bus.

The **gnt_l** signal is an input-only signal.

8.4.2.6 idsel

The **idsel** signal is asserted when the 21030 has been selected for a configuration transaction.

The **idsel** signal is an input-only signal.

8.4.2.7 inta_l

The 21030 asserts the **inta_l** signal to request interrupt service.

The **inta_l** signal is an output-only signal and is tristated at reset.

8.4.2.8 irdy_l

The initiator of a PCI transaction asserts the **irdy_l** signal to indicate its ability to complete the current data phase of a PCI transaction. During a read cycle, the initiator asserts the **irdy_l** signal to indicate that it is ready to accept read data. During a write cycle, the initiator asserts the **irdy_l** signal to indicate that it is driving valid write data on the **ad<31:0>** pins. The current data phase is completed when both the **trdy_l** and **irdy_l** signals are sampled asserted.

The **irdy_l** signal is a bidirectional signal and is tristated at reset.

8.4.2.9 par

The **par** signal is the even parity signal for the **ad<31:0>** and **cbe_l<3:0>** signals.

The **par** signal is a bidirectional signal and is tristated at reset.

8.4.2.10 pciclk

The **pciclk** signal provides timing for all transactions on the PCI bus. All of the PCI signals, except the **rst_l** signal, are synchronous with the **pciclk** signal. Inputs are sampled on the rising edge of the **pciclk** signal. Outputs change state as a result of the rising edge of the **pciclk** signal.

The **pciclk** signal is an input-only signal.

8.4.2.11 req_l

The **req_l** signal is asserted when the 21030 needs to initiate a PCI transfer. External arbitration logic is required.

The **req_l** signal is an output-only signal and is tristated at reset.

8.4.2.12 rst_l

The **rst_l** signal is the PCI system reset signal.

The **rst_l** signal is an output-only signal and is asserted at reset.

8.4.2.13 stop_l

The target of a PCI transaction drives the **stop_l** signal to request that the initiator stop the current transaction.

The **stop_l** signal is a bidirectional signal and is tristated at reset.

8.4.2.14 trdy_l

The target of a PCI transaction asserts the **trdy_l** signal to indicate its ability to complete the current data phase of a PCI transaction. During a read cycle, the device asserts the **trdy_l** signal to indicate that valid data is being driven onto the **ad<31:0>** pins. During a write cycle, the device asserts the **trdy_l** signal to indicate that it is ready to accept write data. The current data phase is completed when both the **trdy_l** and **irdy_l** signals are sampled asserted.

The **trdy_l** signal is a bidirectional signal and is tristated at reset.

8.4.3 Video Interface Signals

Sections 8.4.3.1 through 8.4.3.7 describe the functions of the video interface signals in alphabetical order.

8.4.3.1 blank_l

The **blank_l** signal is the video blanking signal.

The **blank_l** signal is an output-only signal. It is driven low at reset.

8.4.3.2 cursor<7:0>

The **cursor<7:0>** signals provide cursor information to the overlay port of a RAMDAC.

The **cursor<7:0>** signals are output-only signals. They are driven and undefined at reset.

8.4.3.3 hsync_l

The **hsync_l** signal can be programmed to be either a video horizontal synchronization signal or a stereo goggle control signal. The 21030 is initialized during reset to generate the stereo goggle control signal.

The **hsync_l** signal is an output-only signal, and is driven high at reset.

8.4.3.4 hold_l

The **hold_l** and **toggle** signals are used with external logic to generate the VRAM serial-shift clock.

The **hold_l** signal is an output-only signal, and is driven low at reset.

8.4.3.5 toggle

The **toggle** and **hold_l** signals are used with external logic to generate the VRAM serial-shift clock. The **toggle** pin is also used for test information output when the 21030 is in test mode.

(See the *DECchip 21030 PCI Graphics Accelerator Data Sheet* for more information about the 21030 test mode.)

The **toggle** signal is an output-only signal, and is driven low at reset.

8.4.3.6 vidclk

The **vidclk** signal is the clock for the video logic.

The **vidclk** signal is an input-only signal.

8.4.3.7 vsync_l

The **vsync_l** signal can be programmed to be either a vertical synchronization signal or a composite synchronization signal. The 21030 is initialized during reset to generate the composite synchronization signal.

The **vsync_l** signal is an output-only signal, and is driven high at reset.

8.4.4 Test Signals

Sections 8.4.4.1 and 8.4.4.2 describe the test signals.

(See the *DECchip 21030 PCI Graphics Accelerator Data Sheet* for more information about the 21030 test mode.)

8.4.4.1 testin_l

When asserted, the **testin_l** signal places the 21030 in test mode. When in test mode, the integrity of the 21030 input receivers can be verified. The test result data is output on the **toggle** pin.

The **testin_l** signal is an input-only signal.

8.4.4.2 toggle

The **toggle** signal is normally used with the **hold_l** signals and external logic to generate the VRAM serial-shift clock. The **toggle** pin is also used for test information output when the 21030 is in test mode.

The **toggle** signal is an output-only signal, and is driven low at reset.

A

DECchip 21030 step A Differences

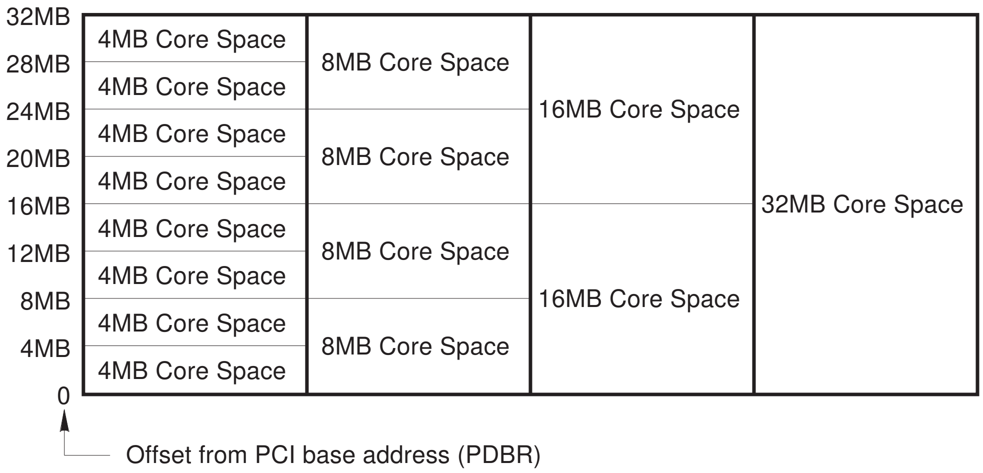
This appendix describes the 21030 step A functionality that differs from the 21030 step B functionality described in the main parts of this manual.

A.1 Memory Space

The size of the memory space is typically 32MB. It is mapped into the PCI memory space at the address specified in the PCI device base address register (PDBR). In certain cases, 64MB or 128MB can be allocated to the 21030 (Section A.1.1).

The 32MB memory space consists of one to eight copies of core space. Figure A–1 shows memory space mapping as a function of core space size.

Figure A–1 Memory Space Organization



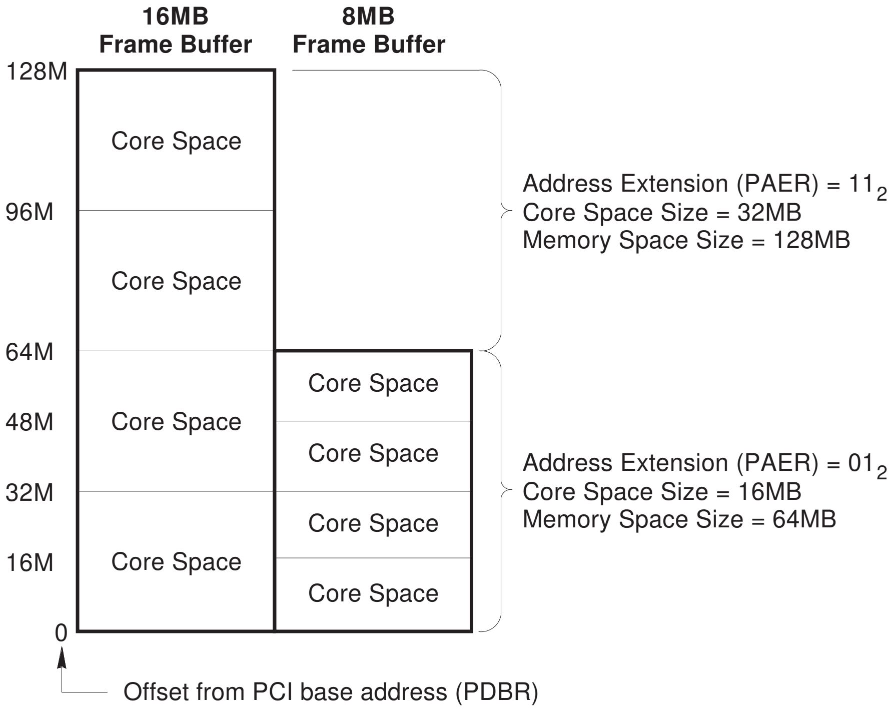
(See Section 2.1 for the equivalent 21030 step B functionality.)

A.1.1 Extending 21030 step A Memory Space

In platforms based on DECchip 21064, 21066, or 21068 Alpha AXP processors, address-space aliasing provides an effective way to enforce write ordering in the presence of the CPU's unenforcing write buffer. With one of these CPUs as the host, effective aliasing requires at least four 21030 address space aliases; but, the 32MB memory space is mapped no more than twice in the 8MB and 16MB 32-bpp frame buffer configurations (Figure 2–3). To support these 32-bpp configurations with four core space maps, the 21030 memory space can be expanded from 32MB to 64MB or 128MB. The address extension field in the PCI address extension register (PAER, Section A.2.2) can be written at configuration time to cause the 21030 to respond to 64MB or 128MB of PCI memory space, rather than 32MB. This effectively doubles or quadruples the number of core space aliases for the 8MB and 16MB 32-bpp configurations.

Figure A–2 shows the extended core space map for 32-bpp frame buffers.

Figure A–2 Extended 21030 step A Core Space Map for 32-bpp Frame Buffers



Normally, platforms not based on Alpha AXP processors do not need address-space aliasing, and the 8MB and 16MB 32-bpp configurations do not need address-space extension.

A.2 PCI Registers

The 21030 step A PDBR (Section A.2.1) is different than the 21030 step B PDBR and the PAER (Section A.2.2) is not used in the 21030 step B.

A.2.1 PCI Device Base Address Register

Figure A–3 shows the PCI device base address register (PDBR) format, and Table A–1 describes its fields.

Figure A–3 PDBR Format

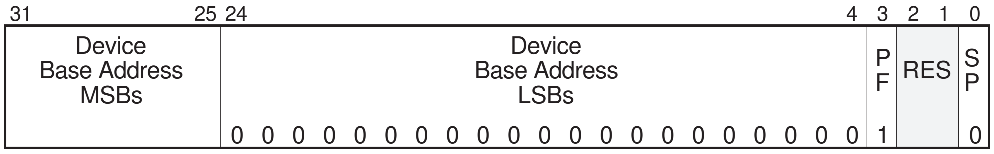


Table A–1 PDBR Field Description

Bits	Field	Type	Description
31:25	Device Base Address MSBs	RW	The most significant bits of the 21030 address-space base address.
24:4	Device Base Address LSBs	RO	The value of this field is 000000_{16}. It indicates that the base address must be aligned to 32MB or greater.
3	PF	RO	Prefetchable—Indicates that prefetched reads and merged writes to the 21030 address space are allowed. The value of this bit is 1.
2:1	RES	RAZ/IGN	Reserved.
0	SP	RO	Space—Specifies that the 21030 address space must be mapped into PCI memory space. The value of this bit is 0.

The device address space is mapped to the location specified in the PDBR.

The value of the space bit (<0>) is zero, indicating that the 21030 can be mapped only into memory space. The value of the 21 least significant address bits (<24:4>) is zero. This value indicates to configuration firmware that the 21030 and its associated memory requires 32MB of address space. Therefore, configuration firmware can map the 21030 address space into any naturally aligned, contiguous 32MB (or larger) region.

Configuration firmware for systems based on Alpha AXP microprocessors might also have to initialize the device-dependent PCI address extension register (PAER, Section A.2.2). (See Section A.1.1 for more information about expanding the 21030 memory space.)

The prefetchable bit (<3>) indicates that there are no side effects on reads to the 21030 address space. The 21030 returns all bytes on reads regardless of the byte enables, and host bridges can merge writes into this region without causing errors.

The value of the PDBR is 00000008_{16} at reset.

(See Section 4.2.2 for the equivalent 21030 step B functionality.)

A.2.2 PCI Address Extension Register

Figure A–4 shows the PCI address extension register (PAER) format, and Table A–2 describes its fields.

Figure A–4 PAER Format

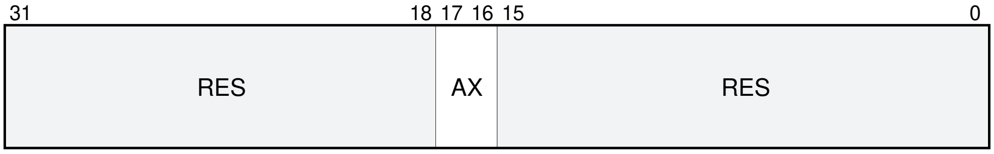


Table A–2 PAER Field Description

<table>
<tr><th>Bits</th><th>Field</th><th>Type</th><th>Description</th></tr>
<tr><td>31:18</td><td>RES</td><td>RAZ/IGN</td><td>Reserved.</td></tr>
<tr><td>17:16</td><td>AX</td><td>RW</td><td>Address extension—The code in this field determines the amount of additional PCI address space (above 32MB) that is allocated to the 21030, as follows:
<table>
<tr><th><17:16></th><th>Add</th><th>Total Address Space</th></tr>
<tr><td>0 0</td><td>None</td><td>32MB</td></tr>
<tr><td>0 1</td><td>32MB</td><td>64MB</td></tr>
<tr><td>1 0</td><td>Reserved</td><td>—</td></tr>
<tr><td>1 1</td><td>96MB</td><td>128MB</td></tr>
</table>
</td></tr>
<tr><td>15:0</td><td>RES</td><td>RAZ/IGN</td><td>Reserved.</td></tr>
</table>

In systems not based on Alpha AXP microprocessors, configuration firmware typically probes the PDBR (Section 4.2.2) to determine that the 21030 requires 32MB of memory and allocates that much memory to the device. Such systems should not need to access the PAER.

In systems based on Alpha AXP microprocessors, the display driver might need to access more than 32MB of memory. In such systems, the PAER allows configuration firmware to specify a 21030 address space allocation that is larger than that specified in the PDBR. To enable the allocation of more address space, system configuration firmware must explicitly write the PAER. (Note that the value of PDBR <0> is zero, specifying that the additional address space is mapped to memory space.) The PAER should be manipulated only by the PCI device mapping code. (For more information about modifying the PAER, see Section A.1.1.)

The PAER is cleared at reset.

A.3 Video Timing Registers

The 21030 step B allows horizontal and vertical sync to be asserted high or low according to the value of the horizontal and vertical sync polarity bits (VHCR <30>, Section 4.5.1 and VVCR <30>, Section 4.5.2). In the 21030 step A, the sync polarity bits are not active and both sync signals are asserted low.

A.4 PCI Operations

Sections A.4.1 through A.4.2.1 describe 21030 step A PCI support that differs from 21030 step B support.

A.4.1 Access Granularity

As a target, the 21030 supports arbitrary, subDword (less than 32-bits) read and write accesses. The 21030 handles all possible permutations of byte masks presented on the **cbe_l<3:0>** pins during both read and write accesses, with the following restrictions:

- Writes to 21030 registers are limited to Dword access. Byte masks are ignored.
- Expansion ROM reads, through the 256KB space defined by the PRBR, are limited to byte access.
- Expansion ROM reads, through the alternate ROM space (Section 2.2.3), returning only Dword-aligned data.

(See Section 5.2.1 for the equivalent 21030 step B functionality.)

A.4.2 Device Address Mapping

Configuration firmware can map the 21030 device and enable response to that mapping by manipulating fields in the PCI device base address register (PDBR) and PCI command and status register (PCSR). Table A–3 describes the fields to be manipulated.

Table A–3 21030 Base Address and Memory Space Enable Fields

Field	Register	Bits	Field Description
Device base address	PDBR	<31:4>	The PCI memory address defined as the base address of the 21030 address space.
Memory space enable	PCSR	<1>	When set, enables the 21030 to respond to memory space accesses.

The PDBR and PCSR are written in the following sequence:

1. Configuration firmware probes the PDBR to determine where the 21030 is to be mapped and the amount of space to allocate to it; that is, firmware writes all ones to the PDBR and then reads back the value. The 21030 returns zeros in <24:4> to indicate the following 21030 requirements:
   - It requires at least 32MB of total address space (PDBR <24:4> = 000000_{16}).
   - It must be mapped to PCI memory space (PDBR bit 0 = 0).
2. Firmware allocates 32MB of naturally aligned PCI memory space and writes the base address to the PDBR.
3. Firmware sets memory space enable (PCSR <1>) to enable device response. (Usually, memory space enable should not be set until the PDBR has been properly initialized as described in steps 1 and 2.)

After the PDBR is written and memory space enable is set in the PCSR, the 21030 can respond as a normal PCI target (Section 5.2).

(See Section 7.1.1 for the equivalent 21030 step B functionality.)

A.4.2.1 Address Mapping in Alpha AXP Systems

In systems based on Alpha AXP microprocessors, the 21030 can require more than 32MB of PCI memory space, depending on the amount of physical display-memory in the graphics subsystem. To minimize the use of memory barrier (MB) instructions in such systems, device driver-level software needs multiple apertures into the frame buffer. To support this driver requirement, the 21030 supports multiple aliases of its core space (that is, frame buffer and register space) in PCI memory space. (See Chapter 2 for more information about the memory space.)

Table A–4 shows the amount of PCI memory space required according to the total amount of physical display-memory included in the graphics subsystem.

Table A–4 PCI Address Space Requirements in Alpha AXP Systems

Physical Display Memory	PCI Memory Space Required
1MB	32MB
2MB	32MB
4MB	32MB
8MB	64MB
16MB	128MB

(Systems based on microprocessors other than an Alpha AXP microprocessor do not need to allocate more than 32MB of memory space to the 21030.)

To extend the 21030 memory space allocation beyond 32MB, configuration firmware must explicitly write the PCI address extension register (PAER). The initialization sequence to allocate 64MB or 128MB of naturally aligned, physical, PCI memory differs slightly from the sequence to allocate 32MB of memory space, as follows:

1. The base address is written to the PDBR.
2. The value of the PAER address extension field is set to 01_2 (64MB) or 11_2 (128MB).
3. Memory space enable (PCSR <1>) is set to enable device response.

Depending on how the PCI firmware standards evolve, the additional memory can be allocated either by transparently executing code from the 21030 expansion ROM prior to device address mapping or through dedicated support in the base system firmware.

B

Pin Summary

Tables B–1 through B–4 summarize the 21030 signal pins.

Table B–1 lists the frame buffer interface pins alphabetically within functional groups.

Table B–1 Frame Buffer Interface Pin Summary

Signal	Qty	Type	Function	Value at Reset
data<63:0>	64	I/O	Data bus	Driven, undefined
addr<17:0>	18	O	VRAM address	Driven, undefined
addren_l<1:0>	2	O	VRAM address enable	Driven, low
cas_l<3:0>	4	O	VRAM CAS	Driven, high
casmode<1:0>	2	O	VRAM CAS	Driven, low
casen<1:0>	2	O	VRAM CAS logic enable	Driven, high
dsf<1:0>	2	O	VRAM special function	Driven, low
oe_l<1:0>	2	O	VRAM output enable	Driven, high
ras_l<3:0>	4	O	VRAM RAS	Driven, high
rasen_l<3:0>	4	O	VRAM RAS logic enable	Driven, low
we_l<7:0>	8	O	VRAM write enable	Driven, high
dacc<2:0>	3	O	RAMDAC clock	Driven, undefined
dacce_l<1:0>	2	O	RAMDAC chip enable	Driven, high
dacrw	1	O	RAMDAC read/write	Driven, undefined
romce_l	1	O	EEPROM chip enable	Driven, high
romoe_l	1	O	EEPROM output enable	Driven, high
romwe_l	1	O	EEPROM write enable	Driven, high

(continued on next page)

Table B–1 (Cont.) Frame Buffer Interface Pin Summary

Signal	Qty	Type	Function	Value at Reset
fbclk	1	I	Clock for frame buffer, RAMDAC, optional ROM, and optional clock generator interface logic	Not applicable
icsce_l	1	O	Clock generator chip enable or data strobe	Driven, high

Table B–2 is an alphabetical list of the PCI interface pins.

Table B–2 PCI Interface Pin Summary

Signal	Qty	Type	Function	Value at Reset
ad<31:0>	32	I/O	PCI multiplexed address and data bus	Tristate
cbe_l<3:0>	4	I/O	PCI multiplexed cycle command and byte enables	Tristate
devsel_l	1	I/O	PCI device select	Tristate
frame_l	1	I/O	PCI cycle frame	Tristate
gnt_l	1	I	PCI bus grant	Not applicable
idsel	1	I	PCI initialization device select	Not applicable
inta_l	1	O	PCI interrupt request	Tristate
irdy_l	1	I/O	PCI initiator ready	Tristate
par	1	I/O	PCI even parity bit	Tristate
pciclk	1	I	PCI system clock	Not applicable
req_l	1	O	PCI bus request	Tristate
rst_l	1	I	PCI system reset	Asserted
stop_l	1	I/O	PCI target stop	Tristate
trdy_l	1	I/O	PCI target ready	Tristate

Table B–3 is an alphabetical list of the video interface pins.

Table B–3 Video Interface Pin Summary

Signal	Qty	Type	Function	Value at Reset
blank_l	1	O	Video blank signal	Driven, low
cursor<7:0>	8	O	Cursor information to RAMDAC	Driven, undefined
hold_l	1	O	VRAM serial-shift clock generation	Driven, low
hsync_l	1	O	Horizontal synchronization or stereo goggle control	Driven, high
toggle	1	O	VRAM serial-shift clock generation or test output	Driven, low
vidclk	1	I	Video logic clock	Not applicable
vsync_l	1	O	Vertical synchronization signal or composite synchronization	Driven, high

Table B–4 lists the test pins.

Table B–4 Test Pin Summary

Signal	Qty	Type	Function	Value at Reset
testin_l	1	I	21030 test mode select	Not applicable
toggle	1	O	VRAM serial-shift clock generation or test output	Driven, low

C

Register Summary

Tables C–1 through C–7 are a summary of the 21030 registers.

Table C–1 lists the PCI configuration registers in descending address order.

Table C–1 PCI Configuration Registers

Register	Mnemonic	Byte* Address Range	Value at Reset (Hexadecimal)
Reserved	—	FF..48	—
PCI address extension register†	PAER	47..44	Cleared
PCI VGA redirect register	PVRR	43..40	81000002
PCI interrupt line register	PLIR	3F..3C	Cleared
Reserved	—	3B..34	—
PCI expansion ROM base address register	PRBR	33..30	Cleared
Reserved	—	2F..14	—
PCI device base address register	PDBR	13..10	Cleared
PCI latency timer register	PLTR	0F..0C	Cleared
PCI class and revision register	PCRR	0B..08	03800001† 03800002‡
PCI command and status register	PCSR	07..04	028000A0
PCI identification register	PIDR	03..00	10110004

*In PCI configuration space

†Active only in the 21030 step A (Appendix A)

‡21030 step B

Table C–2 lists the graphics command registers in alphabetical order. For a list of the registers according to offset, see Table 2–2.

Table C–2 Graphics Command Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Continue register	GCTR	RW	04C	Cleared
Copy 64 destination register	GCDR	WO	164	Cleared
Copy 64 destination register	GCDR	WO	16C†	Cleared
Copy 64 destination register	GCDR	WO	174†	Cleared
Copy 64 destination register	GCDR	WO	17C†	Cleared
Copy 64 source register	GCSR	WO	160	Cleared
Copy 64 source register	GCSR	WO	168†	Cleared
Copy 64 source register	GCSR	WO	170†	Cleared
Copy 64 source register	GCSR	WO	178†	Cleared
Slope register 0	GSLR0	WO	120	Cleared
Slope register 1	GSLR1	WO	124	Cleared
Slope register 2	GSLR2	WO	128	Cleared
Slope register 3	GSLR3	WO	12C	Cleared
Slope register 4	GSLR4	WO	130	Cleared
Slope register 5	GSLR5	WO	134	Cleared
Slope register 6	GSLR6	WO	138	Cleared
Slope register 7	GSLR7	WO	13C	Cleared
Span width register	GSWR	WO	0BC	Cleared

*In core register space 3FF..000

†Register alias.

Table C–3 lists the graphics control registers in alphabetical order. For a list of the registers according to offset, see Table 2–2.

Table C–3 Graphics Control Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Address register	GADR	RW	03C	Cleared
Address register	GADR	WO	0AC†	Cleared
Background register	GBGR	RW	024	Cleared
Block color register 0	GBCR0	WO	140	Undefined
Block color register 1	GBCR1	WO	144	Undefined
Block color register 2	GBCR2	WO	148	Undefined
Block color register 3	GBCR3	WO	14C	Undefined
Block color register 4	GBCR4	WO	150	Undefined
Block color register 5	GBCR5	WO	154	Undefined
Block color register 6	GBCR6	WO	158	Undefined
Block color register 7	GBCR7	WO	15C	Undefined
Blue increment register	GBIR	RO	08C	Cleared
Blue value register	GBVR	RW	0B8	Cleared
Bresenham 1 register	GB1R	RW	040	Cleared
Bresenham 2 register	GB2R	RW	044	Cleared
Bresenham 3 register	GB3R	RW	048	Cleared
Bresenham width register	GBWR	RW	09C	Cleared
Copy buffer register 0	GCBR0	RW	000	Cleared
Copy buffer register 1	GCBR1	RW	004	Cleared
Copy buffer register 2	GCBR2	RW	008	Cleared
Copy buffer register 3	GCBR3	RW	00C	Cleared
Copy buffer register 4	GCBR4	RW	010	Cleared
Copy buffer register 5	GCBR5	RW	014	Cleared
Copy buffer register 6	GCBR6	RW	018	Cleared
Copy buffer register 7	GCBR7	RW	01C	Cleared
Data register	GDAR	RW	080	FFFFFFFF
Deep register	GDER	RW	050	Cleared
DMA base address register	GDBR	RW	098	Cleared

*In core register space 3FF..000

†Register alias.

(continued on next page)

Table C–3 (Cont.) Graphics Control Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Foreground register	GFGR	RW	020	Cleared
Green increment register	GGIR	RO	088	Cleared
Green value register	GGVR	RW	0B4	Cleared
Mode register	GMOR	RW	030	Cleared
Pixel mask (one shot) register	GPXR	RW	02C	Cleared
Pixel mask (persistent) register	GPXR	WO	05C	FFFFFFFF
Pixel shift register	GPSR	RW	038	Cleared
Plane mask register	GPMR	WO	028	Undefined
Raster operation register	GOPR	RW	034	00000003_{16}
Red increment register	GRIR	RO	084	Cleared
Red value register	GRVR	RW	0B0	Cleared
Slope-no-go register 0	GSNR0	WO	100	Cleared
Slope-no-go register 1	GSNR1	WO	104	Cleared
Slope-no-go register 2	GSNR2	WO	108	Cleared
Slope-no-go register 3	GSNR3	WO	10C	Cleared
Slope-no-go register 4	GSNR4	WO	110	Cleared
Slope-no-go register 5	GSNR5	WO	114	Cleared
Slope-no-go register 6	GSNR6	WO	118	Cleared
Slope-no-go register 7	GSNR7	WO	11C	Cleared
Stencil mode register	GSMR	RW	058	Cleared
Z base address register	GZBR	RW	0A8	Cleared
Z increment high register	GZIR-H	RW	094	Cleared
Z increment low register	GZIR-L	RW	090	Cleared
Z value high register	GZVR-H	RW	0A4	Cleared
Z value low register	GZVR-L	RW	0A0	Cleared

*In core register space 3FF..000

Table C–4 lists the video timing registers.

Table C–4 Video Timing Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Horizontal control register	VHCR	RW	064	Cleared. The VHCR must be initialized before video is enabled in the VVVR.
Vertical control register	VVCR	RW	068	Cleared. The VVCR must be initialized before video is enabled in the VVVR.
Video base address register	VVBR	RW	06C	Cleared
Video shift address register	VSAR	RW	078	Cleared
Video valid register	VVVR	RW	070	Cleared

*In core register space 3FF..000

Table C–5 lists the cursor control registers.

Table C–5 Cursor Control Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Cursor base address register	CCBR	RW	060	Cleared
Cursor XY register	CXYR	RW	074	Cleared

*In core register space 3FF..000

Table C–6 lists the status registers.

Table C–6 Status Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Command status register	SCSR	RW	1F8	Cleared
Interrupt status register	SISR	RW	07C	Cleared
Reserved	—	—	1FC	—

*In core register space 3FF..000

Table C–7 lists the external device registers.

Table C–7 External Device Register Summary

Register	Mnemonic	Access	Offset*	Reset State
Clock register	ECGR	WO	1E8	Cleared
EEPROM write register	ERWR	WO	1E0	Cleared
Palette and DAC interface register	EPDR	RW	1F0	Cleared
Palette and DAC setup register	EPSR	WO	0C0	Cleared
Reserved	—	—	1E4	—
Reserved	—	—	1EC	—
Reserved	—	—	1F4	—

*In core register space 3FF..000

D

Technical Support, Ordering, and Associated Literature

This appendix describes how to obtain DECchip information and technical support, as well as how to order DECchip products and associated literature.

Calling the DECchip Information Line for Information and Technical Support

Call the DECchip Information Line for information and technical support:

United States and Canada	**1–800–332–2717 (1–800–DEC–2717)**
TTY (United States only)	**1–800–332–2515 (1–800–DEC–2515)**
Outside North America	**+1–508–568–6868**

Ordering DECchip Products

To order the DECchip 21030 PCI Graphics Accelerator and DECchip 21030 8-bpp and 24-bpp Evaluation Boards, contact your local Digital sales office. When working with your sales representative, you may be able to take advantage of discounts and volume pricing.

You can order the following DECchip products from Digital:

Product	Order Number
DECchip 21030 PCI Graphics Accelerator	21030–AA
DECchip 21030 8-bpp Evaluation Board	PBXGA–AX
DECchip 21030 24-bpp Evaluation Board	PBXGA–BX

Ordering Associated DECchip Literature

The following table lists some of the DECchip literature that is available. For a complete list, and for information about ordering, contact the DECchip Information Line.

Title	Order Number
Alpha Architecture Reference Manual *	EY–L520E–DP–YCH
DECchip 21030 8-bpp Evaluation Board User's Guide	EC–N0681–72
DECchip 21030 24-bpp Evaluation Board User's Guide	EC–N2730–72
DECchip 21030 PCI Graphics Accelerator Hardware Reference Manual	EC–N0683–72
DECchip 21030 PCI Graphics Accelerator Product Brief	EC–N2714–72
Direct Memory Access for DECchip 21030 Boards: An Application Note	EC–N3126–72
Interfacing to External Devices with DECchip 21030 Boards: An Application Note	EC–N2746–72
VGA Pass-Through Support for DECchip 21030 Boards: An Application Note	EC–N0686–72

*To order and purchase the *Alpha Architecture Reference Manual*, call **1–800–DIGITAL** from the U.S. or Canada, or contact your local Digital office, or technical or reference bookstore where Digital Press books are distributed by Prentice Hall.

Associated Third-Party Literature

You can order the following third-party literature directly from the vendor.

Title	Vendor
PCI Local Bus Specification, Revision 2.0	PCI Special Interest Group M/S HF3–15A 5200 N.E. Elam Young Pkwy Hillsboro, Oregon 97124–6497 1–503–696–2000

Glossary

The 21030 accelerates graphics in both Win32 and X graphics environments. This manual uses terminology common to both environments, as well as general graphics terminology. This glossary defines the terms as they apply in this manual.

bitmap

An X term defining a pixmap of depth one. Analogous to a Windows 1-bpp bitmap.

A Windows bitmap can be 1-bpp, 4-bpp, 8-bpp, and so on. A 1-bpp bitmap is also called a monochrome bitmap. This also defines the term as it is used in this manual, unless otherwise noted.

bpp

Bits-per-pixel. Synonymous with number of planes or depth in X.

brush

A Windows term defining a bitmap that is used to fill the interior of a region. A monochrome brush is analogous to an X stipple, and an *n*-bpp brush (where $n > 1$) is analogous to an X *tile*.

depth

See *bpp*.

display driver

A Windows term analogous to an X *server*.

drawable

An X term usually referring to *pixmaps*.

GDI

Graphics device interface.

GUI

Graphics user interface.

pixmap

An X term usually referring to a two-dimensional array of pixels, where each pixel can have a depth of one to many bits. Also see *bitmap*.

PLD

Programmable logic device.

server

An X term for the instrument that, among other things, processes X graphical requests. Analogous to the Windows *display driver*.

stipple

An X *bitmap* used to *tile* a region with a solid (transparent) or bitonal (opaque) pattern. Analogous to a Windows monochrome *brush*.

tile

This X term describes the action of replicating a *pixmap* across a region. Analogous to using a Windows *brush* to fill a region.

Index

B

C

D

E

F

L

M

N

O

P

Q

R

S

T

U