



Radeon, Radeon VE, Mobility Radeon, Radeon 7500, and Mobility Radeon 7500 Software Development Guide

Technical Reference Manual

P/N: SDK-215R6-00-01

© 2001 ATI Technologies Inc.

CONFIDENTIAL MATERIAL

All information contained in this manual is confidential material of ATI Technologies Inc. Unauthorized use or disclosure of the information contained herein is prohibited.

You may be held responsible for any loss or damage suffered by ATI for your unauthorized disclosure hereof, in whole or in part. Please exercise the following precautions:

- Store all hard copies in a secure place when not in use.
- Save all electronic copies on password protected systems.
- Do not reproduce or distribute any portions of this manual in paper or electronic form (except as permitted by ATI).
- Do not post this manual on any LAN or WAN (except as permitted by ATI).

Your protection of the information contained herein may be subject to periodic audit by ATI. This manual is subject to possible recall by ATI.

The information contained in this manual has been carefully checked and is believed to be entirely reliable. No responsibility is assumed for inaccuracies. ATI reserves the right to make changes at any time to improve design and supply the best product possible.

ATI, Rage 128, Radeon, Mobility, and RAGE THEATER are trademarks and/or registered trademarks of ATI Technologies Inc. All other trademarks and product names are properties of their respective owners.

Record of Revisions

Release	Date
0.1	Sept 2001

Related Manuals

- Radeon Register Reference Manual
- Radeon Chip Specifications

Table of Contents

Chapter 1: Overview

1.1 Scope	1-1
1.2 Major Features of the RADEON	1-2
1.2.1 General and Interfacing Features	1-2
1.2.2 2D Acceleration Features	1-3
1.2.3 3D Acceleration Features	1-3
1.2.4 TCL (Transform, Clip, and Lighting) Features	1-5
1.2.5 Motion Video Acceleration Features	1-6
1.2.6 Video Port Features	1-7
1.2.7 Display Features	1-7
1.2.8 Bus Support Features	1-9
1.2.9 Memory Support Features	1-9
1.2.10 Power Management Features	1-9
1.3 A Chapter Summary of this Manual	1-10
1.4 Nomenclature and Conventions	1-11
1.4.1 Register and Field Names	1-11
1.4.2 Numeric Representations	1-11
1.4.3 Sample Codes	1-12
1.4.4 Register Description	1-12

Chapter 2: Programming Basics

2.1 Scope	2-1
2.2 Overview	2-2
2.2.1 Transform, Clip and Lighting (TCL) Engine	2-4
2.2.2 3D Graphics Coprocessor	2-5
2.2.3 2D Engine	2-6
2.2.4 Display System	2-7
2.2.5 Accelerated Graphics Display	2-8
2.2.6 Video Scaler/Overlay	2-9
2.2.7 Display Output	2-10
2.2.8 Video Interface Port (VIP)	2-11
2.2.9 AGP/PCI Host Bus Interface	2-12

2.3	Operation Modes.....	2-13
2.3.1	VGA Mode.....	2-13
2.3.2	Accelerator Mode	2-14
2.4	Drawing Modes in Acceleration-operation Mode	2-14
2.5	Review of Imaging Terminology.....	2-18
2.5.1	Raster Image.....	2-18
2.5.2	True RGB Color	2-19
2.5.3	Representing Pixels.....	2-19
2.5.4	Pixels.....	2-22
2.5.5	Pitch	2-23
2.5.6	Video Memory.....	2-24
2.6	Memory Apertures.....	2-26
2.6.1	The Address Spaces	2-27
2.6.2	Apertures of the System Address Space.....	2-27
2.6.3	The Linear Apertures	2-27
2.6.4	Setting the Linear Aperture Sizes	2-29
2.6.5	The Register Apertures.....	2-30
2.6.6	RADEON Internal Address Space.....	2-30
2.6.7	AGP Addressing	2-31
2.6.8	The PCI Remapper.....	2-32
2.6.9	Recommended Configuration of the RADEON Memory Space and Base Registers.....	2-33
2.6.10	VGA Memory Aperture	2-34
2.6.11	Video BIOS	2-34
2.7	Display Mode and Mode Switching.....	2-35
2.8	Engine Discipline.....	2-35
2.9	BIOS Services.....	2-36

Chapter 3: Accelerator Operation Mode

3.1	Scope.....	3-1
3.2	Chip ID Registers.....	3-2
3.3	Step 1: Detect the RADEON.....	3-3
3.3.1	Using the PCI Configuration Space	3-3
3.3.2	Scanning the BIOS Segment	3-4
3.3.3	Scratch Register Test	3-5
3.4	Step 2: Obtain the Configuration Information.....	3-5
3.5	Step 3: Set a Display Mode	3-7
3.5.1	Passing a CRT Parameter Table to Set a Display Mode	3-11

3.5.2	Manually Setting a Display Mode.....	3-13
3.5.3	Calculating the PLL Register Values	3-16
3.5.4	Determining the Post and Feedback Dividers.....	3-17
3.6	Step 4: Initialize the GUI Engine	3-20

Chapter 4: Programming

4.1	Scope	4-1
4.2	Engine Command Queue Maintenance	4-1
4.3	Programmed I/O Drawing Operations.....	4-3
4.3.1	Drawing Rectangles.....	4-3
4.3.2	Drawing Lines.....	4-10
4.4	Hardware Cursor	4-14

Chapter 5: CCE Initialization and Usage

5.1	Scope	5-1
5.2	Starting the CCE Microengine.....	5-3
5.2.1	Wait for Engine Idle.....	5-3
5.2.2	Load the Microcode into the Microengine.....	5-3
5.2.3	Command Stream Queue PIO Mode.....	5-4
5.2.4	Cautions When Programming RADEON in CCE Mode.....	5-7
5.3	Ring Buffer Management	5-7
5.3.1	The Ring Buffer Concept.....	5-7
5.3.2	Queue Server.....	5-9
5.3.3	Indirect Buffer.....	5-9

Chapter 6: CCE Packets

6.1	Scope	6-1
6.2	2D Coordinate System.....	6-1
6.2.1	Essentials of 2D Drawing Operations.....	6-2
6.3	Drawing Objects.....	6-3
6.3.1	Drawing Rectangles.....	6-4
6.3.2	Drawing Polylines	6-7
6.3.3	Drawing Polyscanlines	6-10
6.4	Block Transfers.....	6-13
6.4.1	Bit Block Transfer	6-13

6.4.2	Transparent Bit Block Transfer	6-16
6.5	Drawing TextDrawing Text	6-20
6.5.1	Drawing Text in Small Font	6-22
6.5.2	Drawing Text in Large Font	6-25

Chapter 7: 3D Programming

7.1	Scope.....	7-1
7.2	3D Context Setup And Initialization.....	7-1
7.3	Setting Render Target	7-2
7.4	Drawing 3D Primitives	7-2
7.4.1	Flexible Vertex Format	7-3
7.4.2	Setup Engine Fetcher Control.....	7-4
7.4.3	Drawing Primitives With Immediate Vertices From CCE Packets	7-6
7.4.4	Using Vertex Buffers	7-7
7.5	Depth and Stencil Buffers	7-15
7.5.1	Creating Depth/Stencil Buffer	7-16
7.5.2	Depth Buffer Operation.....	7-16
7.5.3	Stencil Buffer Operation	7-17
7.6	Setting 3D Render States.....	7-19
7.6.1	Alpha Blending.....	7-19
7.6.2	Alpha Testing	7-22
7.6.3	Culling.....	7-22
7.6.4	Dithering and Antialiasing	7-23
7.6.5	Fog	7-24
7.6.6	Raster Operations.....	7-24
7.6.7	Scissor Testing.....	7-25
7.6.8	Shading.....	7-26
7.6.9	Specular Color	7-27
7.6.10	Stipple Patterns	7-27
7.6.11	Wide Lines	7-28
7.7	Texture Mapping.....	7-29
7.7.1	Texture Loading.....	7-29
7.7.2	Enabling Texture Mapping.....	7-30
7.7.3	Texture Filtering	7-31
7.7.4	Texture Addressing Modes.....	7-31
7.7.5	Texture Combining	7-32
7.7.6	Texture Coordinate Selection	7-34
7.7.7	Mipmaps.....	7-35
7.7.8	3D Textures	7-35

7.7.9	Cubic Environment Mapping.....	7-36
7.7.10	Bump Mapping.....	7-38
7.7.11	Texture Color Space Conversion	7-40
7.8	TCL (Transform/Clip/Lighting) Engine	7-40
7.8.1	Overview.....	7-40
7.8.2	Transformations, Coordinate Systems and Matrices.....	7-41
7.8.3	Loading Matrices.....	7-43
7.8.4	Viewport Transform	7-44
7.8.5	TCL State Data.....	7-45
7.8.6	Setting up TCL Engine.....	7-51
7.8.7	Vertex Format.....	7-51
7.8.8	Vertex Processing.....	7-52
7.8.9	TCL Engine Culling	7-55
7.8.10	Clipping	7-55
7.8.11	Texture Coordinate Generation and Transformation.....	7-58
7.8.12	Vertex Blending	7-60
7.8.13	Lighting	7-61
7.8.14	Vertex Fog	7-75

Appendix A: VGA Functions

A.1	VGA Functions Summary.....	A-1
A.2	AH = 0 - Set Video Mode (AL = Video Mode).....	A-2
A.3	AH = 1 - Set Cursor Type	A-3
A.4	AH = 2 - Set Current Cursor Position.....	A-3
A.5	AH = 3 - Read Current Cursor Position At The Specified Page	A-3
A.6	AH = 5 - Select Active Display Page	A-3
A.7	AH = 6 - Scroll Active Page Up.....	A-3
A.8	AH = 7 - Scroll Active Page Down	A-3
A.9	AH = 8 - Read Character/Attribute At Current Active Cursor Position	A-4
A.10	AH = 9 - Write Character/Attribute At Current Cursor Position Of A Specified Page.....	A-4
A.11	AH = 0Ah - Write Character At Current Cursor Position Of A Specified Page	A-4
A.12	AH = 0Bh - Set Color Palette, Valid For Modes 4 And 5 Only.....	A-4
A.13	AH = 0Ch - Write Dot (Graphics Mode).....	A-4
A.14	AH = 0Dh - Read Dot (Graphics Mode).....	A-4
A.15	AH = 0Eh - Write Teletype To Active Page.....	A-5
A.16	AH = 0Fh - Return Current Video Setting	A-5

A.17 AH = 10h - Set Palette Registers	A-5
A.18 AH=11h - Character Generator Routines	A-7
A.19 AH = 12h - Return Current EGA Settings/Print Screen Routine Selection.....	A-9
A.20 AH=13h – Write String to Specified Page	A-10
A.21 AH=1Ah - Display Combination Code	A-11
A.22 AH=1Bh - Return VGA Functionality And State Information.....	A-11
A.23 AH=1Ch - Save And Restore Video State.....	A-15

Appendix B: VESA Functions

B.1 VESA Functions	B-1
B.1.1 Introduction to VBE.....	B-1
B.1.2 Function 00h - Return Super VGA Information.....	B-2
B.1.3 Function 01h - Return Super VGA Mode Information.....	B-5
B.1.4 Function 02h - Set Super VGA Video Mode.....	B-11
B.1.5 Function 03h - Return Current Video Mode.....	B-12
B.1.6 Function 04h - Save/Restore State.....	B-12
B.1.7 Function 05h - Display Window Control.....	B-13
B.1.8 Function 06h - Set/Get Logical Scan Line Length	B-14
B.1.9 Function 07h - Set/Get Display Start	B-15
B.1.10 Function 08h - Set/Get AC Palette Format.....	B-16
B.1.11 Function 09h - Set/Get AC Palette Data	B-17
B.2 Power Management Services.....	B-18
B.2.1 VBE/PM Function 0 - Report VBE/PM Capabilities.....	B-18
B.2.2 VBE/PM Function 1 - Set Display Power State	B-18
B.2.3 VBE/PM Function 2 - Get Display Power State.....	B-19
B.3 Display Identification Extensions	B-19
B.3.1 VBE/DDC Function 0 - Report VBE/DDC Capabilities	B-19
B.3.2 VBE/DDC Function 1 - Read EDID.....	B-20

Appendix C: Extended Bios Functions

C.1 Extended Functions.....	C-1
C.1.1 AL = 00h - Set Display Mode	C-2
C.1.2 AL = 01h - Set Display Controller State	C-3
C.1.3 AL = 02h - Set DAC State.....	C-3
C.1.4 AL = 03h - Program Specified Clock Entry	C-3
C.1.5 AL = 04h - Short Query Function 0.....	C-4
C.1.6 AL = 05h - Short Query Function 1	C-4

C.1.7	AL = 06h - Short Query Function 2	C-4
C.1.8	AL = 07h - Query Graphics Hardware Capability and Capture Width Info	C-5
C.1.9	AL = 08h - Query Installed Modes	C-7
C.1.10	AL = 09h - Query Supported Mode	C-7
C.1.11	AL = 0Ah - Display Power Management Service (DPMS)	C-7
C.1.12	AL = 0Bh - Display Data Channel (DDC) Service	C-8
C.1.13	AL = 0Ch - Save and Restore Graphics Controller Data	C-9
C.1.14	AL = 0Dh - Get/Set Refresh Rate (CRT only)	C-10
C.1.15	AL = 13h - Extended FP related functions	C-11
C.1.16	AL = 14h - Detect CRT/ TV /DFP/LCD	C-11
C.1.17	AL = 15h - Get/Set Active Display(s)	C-12
C.1.18	AL = 16h - Get/Set TV Standard	C-13
C.1.19	AL = 17h - Get TV Out Info	C-13

Appendix D: BIOS Header Description

D.1	BIOS Header	D-1
D.1.1	Initialization Table Description	D-2
D.1.2	Initialization Block	D-4
D.1.3	PLL Programming Block	D-5
D.1.4	Memory Configuration Block and Reset Sequence Table	D-6

Appendix E: CCE Command Packets

E.1	Scope	E-1
E.1.1	Notation used this Section	E-1
E.1.2	Type-0 CCE Packet	E-2
E.1.3	Type 1 CCE Packet	E-3
E.1.4	Type 2 CCE Packet	E-4
E.1.5	Type 3 CCE Packet	E-5
E.2	Summary of the CEE Packets	E-7
E.3	2D Packets	E-9
E.3.1	SETTINGS	E-9
E.3.2	NOP	E-16
E.3.3	PAINT	E-16
E.3.4	SMALL_TEXT	E-17
E.3.5	HOSTDATA_BLT	E-20
E.3.6	POLYLINE	E-21
E.3.7	POLYSCANLINES	E-22
E.3.8	NEXTCHAR	E-23
E.3.9	PAINT_MULTI	E-24

E.3.10	BITBLT.....	E-25
E.3.11	BITBLT_MULTI.....	E-25
E.3.12	TRANS_BITBLT	E-27
E.3.13	LOAD_MICROCODE.....	E-29
E.3.14	PLY_NEXTSCAN.....	E-29
E.3.15	LOAD_PALETTE.....	E-30
E.3.16	SET_SCISSORS.....	E-30
E.4	3D Packets	E-31
E.4.1	3D_DRAW_VBUF.....	E-31
E.4.2	3D_DRAW_IMMD	E-31
E.4.3	3D_DRAW_INDX	E-32
E.4.4	3D_LOAD_VBPNTR.....	E-32
E.4.5	3D_CLEAR_ZMASK.....	E-33
E.4.6	3D_RNDR_GEN_PRIM.....	E-34
E.4.7	3D_RNDR_GEN_INDX_PRIM.....	E-43
E.5	Miscellaneous Packets.....	E-45
E.5.1	WAIT_FOR_IDLE	E-45

Appendix F: Revision History

F.1	SDK-215R6-00-01 (SDK-215R6-00-01.pdf).....	H-1
-----	--	-----

1.1 Scope

This manual is a programming guide for the RADEON graphics controller and its derivatives (Radeon 7500, Mobility Radeon 7500, Radeon VE and Mobility Radeon). Unless specifically indicated otherwise, the term RADEON in this manual refers to all members of the RADEON family. The examples provided show how to program typical 2D and 3D drawing operations. This manual also provides details about various multimedia concepts.

For details about programming older generations of ATI graphics controller, refer to the *mach64 Programmer's Guide* and *RAGE 128 Programmer's Guide*. To request this information, contact ATI Developer Relations.

Background

The RADEON is a feature rich, high performance graphics and multimedia accelerator providing an unsurpassed single chip geometry and graphics engine. It combines astoundingly fast 3D and 2D acceleration, with advanced multimedia, geometry and multitexturing capabilities. This accelerator incorporates new technologies such as:

Charisma Engine: Provides dedicated geometry acceleration that makes 3D characters and environments look and behave believably. It includes high performance, versatile support in hardware for transformation, clipping and lighting (TCL) calculations. This reduces CPU workload and allows 3D scenes to contain more polygons and more complex lighting than any competing technology. The Charisma Engine is also the only available engine that accelerates advanced features such as 4-matrix vertex skinning and keyframe interpolation.

Pixel Tapestry Architecture: The Pixel Tapestry Architecture includes features that add detail to an image, such as reflections and shadows, without compromising performance. This unique architecture supports a wide array of versatile features like single pass multitexturing with 3 textures, 3D textures, bump mapping (emboss, dot product 3 and environment mapped), texture transformations, priority buffer support, shadow mapping, and range-based fog. Since these features can be enabled without impacting frame rates, they can be used more extensively to increase the realism of 3D scenes.

Comprehensive Digital Video Support: The RADEON integrates industry leading digital video features highlighted by integrated digital TV decode capability. Combined with a tuner and demodulator, the RADEON provides an all-format DTV/HDTV solution,

including 1920 pixel wide 1080i format. Coupled with the RAGE THEATER analog encoder/decoder chip, the RADEON provides a complete convergence experience.

Concurrent Command Engine (CCE): CCE uses the RADEON's bus mastering capabilities to deliver excellent drawing performance, as well as simplifying the programming effort.

1.2 Major Features of the RADEON

1.2.1 General and Interfacing Features

- 32-bit PCI bus (Rev 2.2), 3.3 V with bus mastering support.
- AGP with 1X (66 MHz), 2X (133 MHz), 4X (266MHz) transfer, full sideband addressing, no support for PIPE#.
- Two independent 64-bit memory buses can be operated as single-channel (64 bit) or dual-channel (128 bit) using SGRAM or SDRAM to build 8/16/32/64/128 MB configurations. Operating frequency is 125MHz minimum to 200MHz maximum, SDR or DDR.
- Fully compliant with expected PC 2001 requirements
- Full ACPI 1.0b, OnNow, and IAPC (Instantly Available PC) power management, including PCI Power Management registers
- Bi-endian support for compliance on a variety of processor platforms
- CCE high-speed pull architecture software interface optimized for Pentium III and Athlon systems:
 - Bus mastering of 2D & 3D display lists
 - Direct walk of Direct3D/OpenGL vertex list
 - Ultra-thin driver layer
 - Maximizes concurrency between RADEON and host
- Supports optional RAGE Theater companion chip for NTSC/PAL TV out and NTSC/PAL/SECAM analog video capture
- Support for ROM or Flash RAM parallel or serial video BIOS.
- VIP 2.0 capture port with 2 bit VIP host port. Includes 8-bit capture port running up to 75MHz, and independent I2C interface.
- Video capture port also directly interfaces to MPEG transport stream decoders.
- Independent TV out interface for connection to Rage Theater NTSC/PAL encoder.
- Integrated DAC for CRT with stereoscopic display support. Also integrated ability to

drive YPbPr component video for SD/HDTV connection.

- Integrated TMDS transmitter running up to 165MHz for support up to 1600x1200 at 60Hz. Fully compliant with DVI and DFP connection standards.
- Independent DDC lines for DAC and TMDS connections (separate from I2C). Also full AppleSense support on DAC connection.

1.2.2 2D Acceleration Features

- Highly-optimized 128-bit engine, capable of processing multiple pixels/clock. Higher speed compared to the RAGE 128 VR / GL chip due to various improvements.
- Hardware acceleration of Bitblt, Line Draw, Polygon / Rectangle Fill, Bit Masking, Monochrome Expansion, Panning/Scrolling, Scissoring, and full ROP support (including ROP3).
- Optimized handling of fonts and text using ATI proprietary techniques.
- Game acceleration including support for Microsoft's DirectDraw: Double Buffering, Virtual Sprites, Transparent Blit, and Masked Blit.
- Acceleration in 8/15/16/32 bpp modes.
- Setup of 2D polygons and lines.

1.2.3 3D Acceleration Features

- Primitive (Triangle) set up rates of up to 32 Million triangles per second.
- Multi-texturing via three texture blending units per 3D pipe, allowing 12 texel reads per pixel in a single tick and single pass.
- 3D Texture support.
- Cubic Environment Mapping.
- Perturbation bump mapping (Tritech method).
- Hierarchical Z-buffer. HyperZ technology dramatically improves performance by maximizing Z-buffer efficiency.
- On-chip texture cache dramatically improves large triangle performance
- On-chip vertex cache eliminates unnecessary vertex reads
- Comprehensive enhanced 3D feature set:
 - Improved precision in Anisotropic filtering and Bilinear filtering.
 - Complete 3D primitive support: points, lines, triangles, lists, strips and quadrilaterals and BLTs with Z compare.

- Improved texture compositing, with no limitations on texture formats or location (i.e., one texture may reside in AGP memory and the other in frame buffer memory).
- Full screen or window double buffering for smooth animation.
- Support of special effects such as simultaneous alpha blending and fog (vertex and z-based), video textures, texture lighting, reflections, shadows, spotlights, LOD biasing and texture morphing. Filtered texture alpha channel, and hardware support for D3D 'MODULATEALPHA' texture blending mode.
- Hidden surface removal using 16, 24, or 32-bit Z-buffering (maximum Z-buffer depth is 24 bits when stencil buffer enabled).
- 8-bit stencil buffer.
- Line and Edge anti-aliasing.
- 4 bits of subpixel and subtexel accuracy.
- Gouraud and specular shaded polygons.
- Perspectively correct per pixel mip-mapped texturing with chroma-key support.
- Bilinear and trilinear texture filtering.
- Full support of Direct3D texture lighting.
- Dithering support in 16 bpp for near 24 bpp quality in less memory.
- Extensive 3D mode support:
 - Draw in RGBA32, RGBA16, & RGB16.
 - Texture map modes: RGBA32, RGBA16, RGB16, RGB8, ARGB4444, YCrCb444.
 - Compressed texture modes: YCrCb422, CLUT4 (CI4), CLUT8 (CI8), VQ, and DX6 mode.
- Full scene sort independent anti-aliasing.
- DX6 Color Compression.
- Improved support for DX6 Blend mode.
- Control bit for Vertex Walker zero area.
- Support for 32 bpp Palette (similar to the RAGE PRO LCD).
- Improvement to Chroma Key functionality (similar to the RAGE PRO LCD).
- Support for OpenGL format for Indirect Vertices in Vertex Walker.
- Support for DirectX 6.0 compressed texture scheme (i.e., color cell compression) that allows for higher benchmark scores.
- Hardware Z-buffer clear and hierarchical Z support.

- Maximum performance on AGP 4X platforms.
- Improved sub-textel accuracy to improve the appearance of highly magnified textures).
- Improved performance in Raw Rendering and Geometry due to higher engine speed.

1.2.4 TCL (Transform, Clip, and Lighting) Features

Note: TCL features are not supported in Mobility Radeon and Radeon VE.

- HW Transformation, clipping and lighting rates of over 25 million triangles per second.
- Transforms homogeneous vertices from Object / Model Coordinates to Clip Coordinates
- Performs Frustum Clipping on the Point / Line and Triangle Primitives in Clip Coordinates
- Performs Texture Coordinate Transformation
- Supports DirectX skinning, with a 4-matrix / 2 vertex Vertex Blending (Skinning / Morphing (Tweening))
- Supports Guard-Band Clipping for clipping performance optimization
- Supports DirectX / OpenGL Texture Coordinate Generation
- Supports 6 User-Defined Clip Planes. Performance Optimizations Based on Coordinate Systems
- Supports OpenGL / DirectX Directional (Infinite) Lighting with Infinite & Local Viewer. Up to 8 Lights. Includes DX7 color-per-vertex support.
- Supports OpenGL / DirectX Local Lighting with Infinite & Local Viewer. Up to 8 lights. Includes Range and Spot Attenuation, Includes DX7 color-per-vertex support.
- Supports Vertex Reuse capability for performance optimization of triangle lists
- Supports Back-Face Culling for performance optimization
- Supports polygon-id based shadow processing.
- Supports DirectX dual cone spot lights for DirectX lighting model.
- Supports normal re-normalization and constant rescale for DirectX/OpenGL lighting model.
- Supports separate and combined accumulation of diffuse and specular colors for DirectX/OpenGL lighting models.
- Supports derivation

1.2.5 Motion Video Acceleration Features

- Video scaling and fully programmable YCrCb to RGB color space conversion for full-screen / full-speed video playback and fully adjustable color controls.
- Front and back end scalers plus capture port scaler to support multi-stream video for video conferencing and other applications.
- Front end scaler support for 8, 15, 16, and 32 bpp color depths.
- Back end overlay/scaler supports up to 8x4 tap filtering, and always ensures at least 4x2 tap filtering even in extreme cases. 4x4 tap is typical
- Expanded line buffer allowing vertical filtering of native standard definition and high definition images. The overlay and capture system includes support for all proposed ATSC resolutions upto 1920x1080.
- Enhanced MPEG-2 hardware decode acceleration, including support for both motion compensation and IDCT, to provide dramatically reduced CPU utilization without incurring the cost of a full MPEG-2 decoder. This is superior to competing motion compensation solutions, including those marketed as '9-bit' motion compensation.
- Integrated high performance iDCT and motion compensation allows MPEG-2 decode of all 18 ATSC formats including 1280x720p and 1920x1080i on standard speed CPUs. It also allows "timeshifting" of standard definition television in standard systems by ensuring the CPU has enough remaining power to perform real-time MPEG-2 compression while the RADEON handles most of the MPEG-2 decompression load.
- Hardware DVD subpicture decoder with interpolating scaler and alpha compositor to provide optimal DVD subpicture quality in all display bit depths.
- Adaptive de-interlacing filter eliminates video artifacts caused by displaying interlaced video on non-interlaced displays, by analyzing image and using optimal de-interlacing function on a per-pixel basis. Also can be programmed to other methods such as bob, weave, and ATI enhanced weave.
- Bi-directional bus mastering engine with full YCrCb planar mode support for superior MPEG-2 decode and video conferencing.
- Enhanced support for range based graphics and video keying for effective overlay of video and graphics.
- Ability to genlock to any broadcast video signal, eliminating synchronization problems.
- YCrCb to RGB color space converter with support for both packed and planar YCrCb.
- YCrCb422, YCrCb410, YCrCb420, RGB32, RGB16/15 in back end scaler/overlay.
- Ability to reconstruct frames of field-based content when software indicates that

content was originally progressive and has been converted via 3:2 pull down, provided that the decoder gives appropriate cues.

1.2.6 Video Port Features

- VIP 1.1 with extensions.
- VIP 2.0 compatible 8-bit video capture port (i.e. capture port supports higher frequencies and new flags defined in VIP 2.0).
- Optional downscaling in capture port.
- 2-bit VIP host port.
- Supports hardware MPEG-2 decoders.
- Support for HDTV transport stream capture via capture port.
- Digital Broadcast Satellite receiver support.
- Supports RAGE THEATER video decoder.
- Dedicated I2C capability for multimedia devices (not shared with either DAC or TMDS DDC lines).

1.2.7 Display Features

General

- Triple 10-bit palette DAC with gamma correction for true WYSIWYG color. Pixel rates up to 350 MHz standard.
- Supports two new graphics formats: 16bpp aRGB 4444 and 16bpp alpha and index 88. Also supports 8/15/16/24/32 bpp graphics formats with gamma correction in all modes.
- Stereoscopic display capability.
- Support for auxiliary window signal.
- Support for up to 4k x 4k resolution display.
- Support for HDTV display using component video (YPbPr).
- Support for DDC1 and DDC2B+ for plug and play monitors, and AppleSense monitor detection support.
- 8-bit alpha blending of graphics and video overlay.
- Hardware cursor up to 64x64 pixels in 2bpp, full color AND/XOR mix, and full color 8-bit alpha blend.

- Virtual desktop support.
- Support for external TV Out (see section on TV Out below for details).
- Support for flat panel displays (see section on Fixed Resolution Display below for details).

TV Out

Note: RADEON does not incorporate integrated TV Out capability. It relies on the RAGE THEATER family through a glueless interface to provide this functionality.

Note: Rage Theater option not available in Mobility Radeon and Radeon VE.

- Support of RAGE THEATER uses the VIP host port for commands. Data formats supported are RGB 888 (triple clocked), RGB 565, YUV4:2:2, YUV4:4:4, and ITU-R 656 (each double clocked).
- If overlay source data is YCbCr, then YCbCr format is used for TV output data. Scaler and sub-picture data that have YCbCr values that do not convert to normal RGB space are reproduced correctly on YCbCr or YPbPr outputs.
- If overlay source data is RGB, RGB format can be used for TV output data. RADEON or RAGE THEATER can convert to YUV space.
- Support for TV data output encode in ITU-656 format includes embedded syncs, interlaced and non-interlaced video indication and odd or even field indication. There is no support of task bit.

Features for Fixed Resolution (e.g. Flat Panel or Digital CRT) Display

- Supports DVI, DFP and VESA P&D interfaces with integrated TMDS transmitter.
- TMDS transmitter operates to 165 MHz and fully supports reduced blanking.
- Support for fixed resolution displays (e.g. panels) from VGA (640x480) to wide UXGA (1600x1200) resolution with full ratiometric expansion ability for source modes up to 1280x1024. Higher resolution panels and digital CRTs possibly supported.
- Programmable dithering logic in ratiometric expansion.
- Improved auto expansion.
- Optional auto-centering mode to display desktop at native size without ratiometric expansion.
- Support for VGA text modes in centering panel modes (up to approximately 165 MHz pixel frequency).

- Support for reduced blanking intervals, as defined by VESA.

1.2.8 Bus Support Features

- Comprehensive AGP support:
 - 2X and 4X mode operation.
 - Sideband Addressing.
 - AGP Texturing (Execute mode).
 - Both AGP reads and writes (without support for the 'fast write' capability defined in revision 2.0 of the AGP specification).
 - AGP 1X mode operation.
- PCI version 2.2 with full bus mastering and scatter / gather support.
- 3.3 V and 5 V PCI interfaces. 5 V PCI only compatible with systems that ensure bus signals do not rise above 3.3 V typical. This works with all current standard system chipsets from Intel and other major suppliers.

1.2.9 Memory Support Features

- Supports a variety of memory configurations for bandwidths of up to 6.4 GB/s:
 - SDRAM or SGRAM may be used for all memory configurations.
 - Single Data Rate (SDR), up to 200 MHz (3.2 GB/s at 128-bits).
 - Double Data Rate (DDR), up to 200 MHz (6.4 GB/s at 128-bits).
- Two separate 64-bit memory interface channels may be configured for one-channel or two-channel operation to optimize cost/performance.
- Flexible graphics memory configurations: 8MB up to 128MB SDR/DDR SGRAM or SDR SDRAM.
- Does not support SGRAM "block write" feature.
- Does not support memory interfaces less than 64 bits wide or less than 8MB memory configurations.

1.2.10 Power Management Features

- Support for version 1.0 of the ACPI Specification and version 1.1 of the PCI Bus Power Management Interface Specification (PCI PMI).

- The Chip Power Management Support logic supports four device power states - On, Standby, Suspend and Off - defined for the OnNow Architecture. Each power state can be achieved by software control bits.

1.3 A Chapter Summary of this Manual

Table 1-1 Chapter Summary

Chapter	Description
1 Overview	Scope of the manual. Overview of the contents. Feature summary of the RADEON. Display Modes
2 Using the RADEON	Basic programming guide. A general understanding of the features and functions.
3 Getting Started	Using the RADEON in accelerator mode: Card detection, setting a display mode, engine initialization, programming considerations.
4 Programmed I/O Operations	Issues covering the accelerator engine: Command FIFO queue Programmed I/O operations (such as bit block transfers, line, pattern, and rectangle drawing).
5 Concurrent Command Processor Initialization and Usage	Overview of the CCP programming model: Setup and initialization of the CCP in various operational modes.
6 CCE Packets	Description of the CCP packets. Programming examples for general engine operations (blts, rectangle and line draws, etc.).
7 Advanced Topics	Advanced topics covering special features and capabilities: Using the overlay scalar and front-end scalar. Using the bus mastering features.
Appendix A	BIOS Function Calls
Appendix B	Extended BIOS Function Calls
Appendix C	BIOS Header, Scratch Registers and Information Tables
Appendix D	VESA BIOS Extension
Appendix E	BIOS Hardware Configuration and Multimedia Tables
Appendix F	CCP Command Packets

1.4 Nomenclature and Conventions

These conventions below apply to the RADEON Register Reference Manual.

1.4.1 Register and Field Names

An upper-case mnemonic represents the name of a hardware register and field names. The naming conventions for registers and bit fields are as indicated below:

A mnemonic is used to identify the name of a hardware register. The naming conventions for registers and/or bit fields within a register are as follows:

- **Register_Mnemonic**
- **Register_Mnemonic[Bit_Numbers]**
- **Field_Name@Register_Mnemonic**

The following example is the mnemonic for the Configuration Chip ID register:

- **CONFIG_CHIP_ID**

Continuing the above example, the Product Type Code field within the above register occupies bit positions [0] through [15]. The examples below describe this field in two ways:

- **CONFIG_CHIP_ID[15:0]**
- **CFG_CHIP_TYPE@CONFIG_CHIP_ID**

The second convention will be the preferred one, with the first convention used mostly for describing unnamed fields.

1.4.2 Numeric Representations

- Hexadecimal numbers are appended with “h” whenever there is a risk of ambiguity. Other numbers are assumed to be in decimal.
- Registers (or fields) of identical function are sometimes indicated by a single expression in which the part of the signal name that differs is enclosed in [] brackets. For example, the eight Host Data registers — **HOST_DATA0** through to **HOST_DATA7** — are represented by the single expression **HOST_DATA[7:0]**.

1.4.3 Sample Codes

Sample code and functions will be typeset in a `courier` font.

Example: performing an operation

```
// Sample Function
void Sample_function (void)
{
    printf ("This is a sample function\n");
} // Sample_function
```

1.4.4 Register Description

All registers in this document are described with the format of the self-explanatory sample table below. All offsets are in hexadecimal notation, while programmed bits are in either binomial or hexadecimal notation. (Note: sometimes not shown are the indirect type of byte offsets, e.g., CFG, PLL, VGA, etc., which will be indicated on the appropriate registers).

SE_TCL_PER_LIGHT_CTL_0		
Field Name	Bits	Description
LIGHT_ENA_0	0	Enables light.
AMBIENT_ENA_0	1	Enables ambient computations for light.
SPECULAR_ENA_0	2	Enables specular computations for light.
LOCAL_LIGHT_0	3	Specifies a local light instead of an infinite light.
SPOT_ENA_0	4	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_0	5	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_0	6	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_0	7	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
etc...		

Chapter 2

Programming Basics

2.1 Scope

This chapter details the basics of the RADEON's operation and drawing modes. The following topics are covered:

- Functional block diagram of the RADEON.
- Operation modes.
- Accelerator programming modes.
- Review of imaging terminology.
- Display modes and switching modes.

2.2 Overview

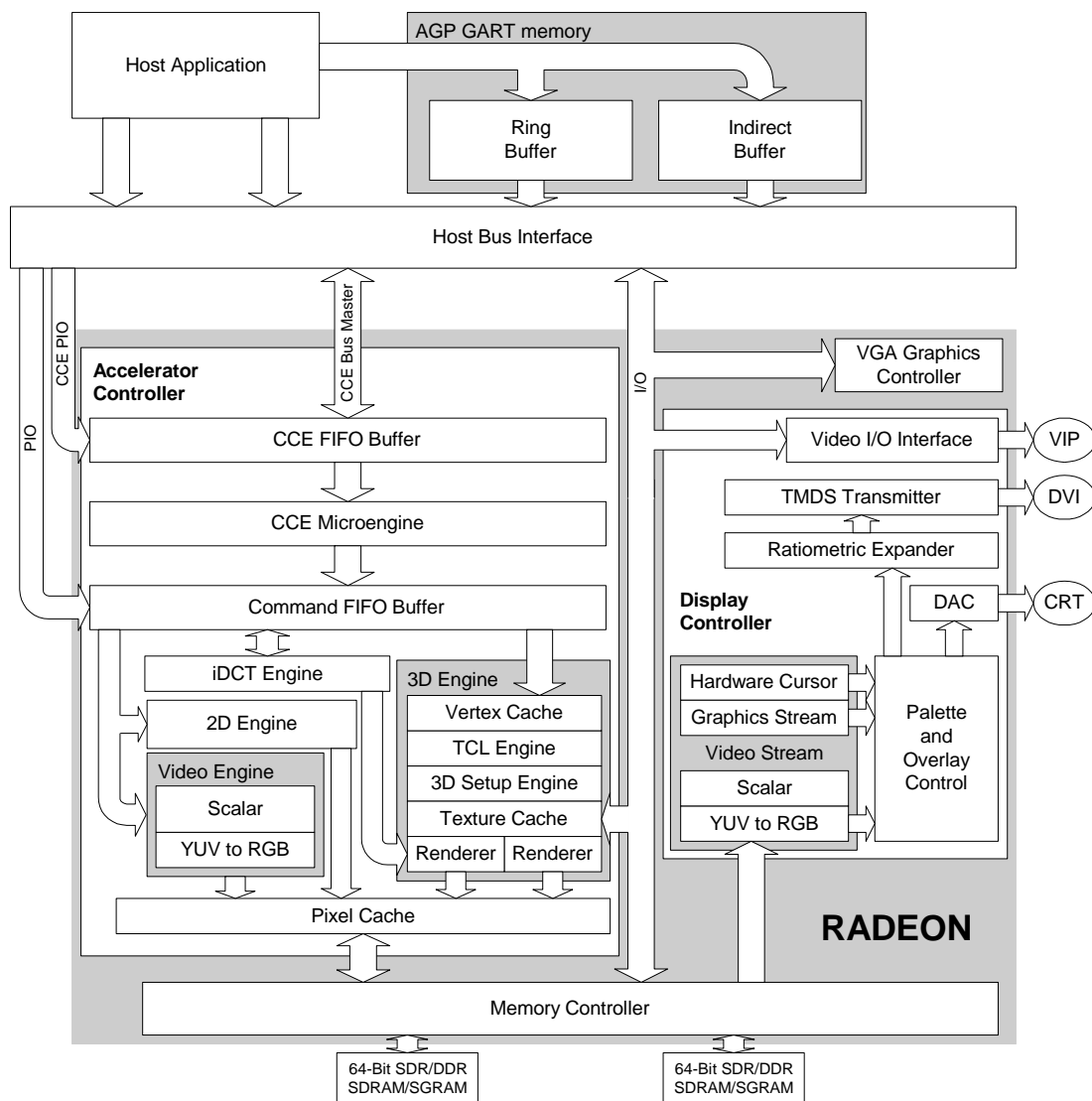


Figure 2-1. RADEON Structure and Data Flow

This chapter presents a basic description of the functional blocks in this diagram. Detailed descriptions are presented in subsequent chapters. For a summary of many of the RADEON's functional blocks, refer to [Table 2-1](#). For a summary of the RADEON's buffers, refer to [Table 2-2](#).

Table 2-1 RADEON Functional Blocks

Functional Unit	Purpose
Accelerated Graphics Port (AGP) Interface	Transfers data from the ring buffer (located in the system memory) to the RADEON's CCE FIFO buffer without direct involvement of the CPU.
CCE Microengine	Parses the command packets from the host application and places the results into the Command FIFO buffer.
2D Render Engine	Performs 2D primitive rasterization.
iDCT Engine	Performs iDCT and motion compensation. All format DTV/HDTV solution, including the 1920 pixel wide 1080i format.
Back-end Scaler	4-tap horizontal and vertical filtered up and downscaling of YUV and RGB content (RGB 15/16/32) to give top quality video playback.
Front-end Scaler	Front end scaler support for 8, 16 and 32 bpp color depths
3D Render Engine	Performs 3D primitive rasterization.
TCL Engine	Performs Transform, Clipping, and Lighting
3D Setup Engine	Performs 3D primitive setup operations.
Vertex Cache	Eliminates unnecessary vertex reads.
Texture Cache	Improves large triangle performance.
Pixel Cache	Reduces off-chip access to higher-latency frame buffer of AGP memory.
VGA Controller	Manages pixel operations under VGA mode.
Video I/O Interface	2-bit VIP host port. VIP 2.0 compliant 8-bit capture port. Compatible with Rage Theater video decoder.
DAC	Triple 10-bit palette DAC supports pixel rate to 350MHz.
TMDS Transmitter	DVI-compliant integrated 165MHz TMDS transmitter. Up to UXGA resolution. Supports VESA proposed reduced blanking timings. Ratiometric expansion.

Table 2-2 RADEON Buffers

Buffer Name	Size	Purpose
CCE FIFO Buffer	192 DWORDs	Contains command packet data queued for processing by the micro controller. Only used in CCE-programming mode.
Command FIFO Buffer	64 Entries	Contains register/data pairs for processing by the geometry, 3D-setup, 3D-render, and 2D-render engines (i.e., GUI engine). Data is written directly in PIO-programming mode, and streamed from the micro controller in CCE-programming mode.
Frame Buffer	Depends on the amount of video memory installed. Ranges from 16MB to 128MB.	Contains all on-screen and off-screen rendering buffers, such as: drawing, stencil and z buffers, bitmaps, and texture maps.

2.2.1 Transform, Clip and Lighting (TCL) Engine

The RADEON includes a new Transform, Clip, and Lighting Engine. The TCL engine significantly reduces CPU load and CPU memory access, by taking over the processing of row vertex data from the CPU.

The TCL engine can use state and vertex data along with primitive assembly data to perform one of the following combinations of operations

- Pass pre-transformed/clipped/lit vertex data without modification.
- Perform the required math necessary to transform and clip user provided lit vertices.
- Perform the required math necessary to transform, clip, and light user provided non-transformed, non-lit vertices.
- Perform texture coordinate transformation and generation.

When TCL is enabled, the transform process will rotate the incoming vertex positions from object to clip coordinates. The transform controller in the TCL block is also responsible for Vertex Blending and Frustum Clip code computations.

Vertex blending as a general process allows the user to combine up to 2 unique vertex positions and normals using up to 4 unique modelview matrices with 4 unique blending weights. The basic equation reads as:

$$V0/1 * MV0 * A + V0/1 * MV1 * B + V0/1 * MV2 * C + V0/1 * MV3 * D$$

where V0/1 is either vertex 0 or 1, MV0-3 are up to 4 unique modelview matrices, and A,B,C,D are blending weights.

Note that there is independent enabling of vertex blending for position and normal. There is no support in the RADEON design for blending of colors or texture coordinates across multiple vertices.

The general purpose of the clipping process is to detect any edges of a primitive which cross clip-plane boundaries, and compute the intersection point, in order to recreate primitives which lie completely within all clip planes. Frustum clipping clips to the view volume boundaries (typically a frustum) and the user-defined clipping allows the user to define additional clip planes that will delete portions of the 3-D environment for purposes such as cut-away views. Guard-Band clipping is an optimization for the X (left/right) and Y (top/bottom) clipping process.

Texture coordinate transformation (Tex Xform) provides the ability to rotate a texture vector (s,t,r,q) through a 4x4 matrix. For the RADEON implementation, the r-coordinate is not supported as an input texture coordinate, or as an output texture coordinate. This means that the input texture vector is (s,t,q) and the output vector is (s,t,q). The input q value is defaulted to 1.0 if it is not provided by the driver.

Texture coordinate generation (Tex Gen) is the process of creating texture coordinates from other data involved with the transform process. The generation of texture coordinates has the choices of the vertex position in model or eye coordinates, the vertex normal in eye coordinates, or the computed reflection vector in eye coordinates.

OpenGL also has the ability to supply an additional planar definition to modify the texture coordinates. The RADEON implementation assumes that the driver will pre-multiply these planar definitions into the texture transform matrix mentioned above, allowing the same level of functionality.

The process of vertex lighting takes some input color data and light definitions and calculates a new vertex color (diffuse and specular) based on how that vertex and normal interact with the lighting model. In general, the RADEON lighting implementation is the OpenGL lighting implementation. RADEON supports up to 8 lights with the complete OpenGL and DirectX set of functionality. The RADEON lighting calculations are all performed in 32-bit floating point with full-range support.

2.2.2 3D Graphics Coprocessor

The 3D graphics coprocessor offers a number of pixel processing features associated with the rendering of 3D images. These coprocessor functions were chosen to accelerate features of all of Microsoft's Direct 3D, OpenGL ICD, and Apple's QuickDraw 3D

RAVE interfaces. Drivers including OpenGL ICD will be available for all major operating systems and APIs.

RADEON includes a triangle setup engine, which needs only color, alpha, Z, U and V information at vertices of triangles to successfully draw Gouraud shaded, or perspective correct texture mapped triangles. The setup engine significantly reduces the CPU load in 3D graphics applications, giving applications more CPU time to perform non-setup related tasks.

Pixels to be displayed can be further modified by alpha blending with pixels in the destination, by fogging pixels with a fog color, and in the case of texture mapping, by lighting them and by LOD biasing. Depth buffering is achieved by associating a 16, 24, or 32-bit Z value with each pixel. The Z, alpha, and fog color for each new pixel is supplied from interpolators within the 3D coprocessor. In case of texture mapping, the alpha factor may even be stored in the texture map on a pixel-by-pixel basis.

Pixels in the 3D coprocessor are always operated on as 24-bit entities (8 bits each of Red, Green, and Blue). Other pixel sizes, i.e., 8-bit and 16-bit, are dithered by the 3D graphics subsystem to output at the desired pixel size.

The 3D coprocessor contains a powerful texture mapping engine. This engine takes a series of precomputed maps (mip maps) and selects texels from these maps in a way that allows them to look perspective correct. Texels can be filtered in a number of ways, and then lit (lightened or darkened). Once the texel is formed by filtering and lighting, any of the pixel processing modes mentioned previously can be applied to the texel.

A texture cache greatly reduces the memory bandwidth needed to support texture mapping.

The texture engine fully supports 3D texture, and bump mapping, with LOD calculation.

With AGP support, texture maps can be stored in system memory and pulled into the local texture cache as needed. This rids the system of the need to add significant amount of local frame buffer memory in order to support multiple detailed texture maps, and allows applications to support a richer and more realistic environment.

2.2.3 2D Engine

The RADEON 2D engine is a fixed-function subunit that runs concurrently with the host processor. It incorporates full ROP3 (Pattern/Src/Dst) support and is capable of drawing both rectangle and line draw primitives. A sophisticated pixel datapath (128-bits wide drawing multiple pixels per clock) allows monochrome to two-color expansion, fast solid color fills (via block writes), patterned fills, and host-to-screen transfers.

2D Engine features summary:

- Capable of addressing anywhere in local frame buffer, AGP system memory, or general system memory using 4GB address generation.
- Full ROP3 support (Pattern/Src/Dst).
- Patterns (8x8 color/mono brushes, 32x1 mono pens for lines).
- Rectangle and line trajectories.
- 128-bit datapath (multiple pixels generated per clock), 8/15/16/32 bpp support.
- Colorcmp on both src/dst channels.
- Colorcmp flip.
- Interrupt support.
- Multimedia event triggers.
- Quick engine setup.

A four-function simultaneous source/destination color compare for transparent blits, bit masking, and scissoring.

2.2.4 Display System

The RADEON display system supports VGA, VESA super VGA, and accelerator mode graphics display. The full features are outlined below:

Extended VGA Graphics Controller

- Fully register compatible with VGA standard.
- BIOS compatible with VESA super VGA.
- Includes support for VGA text modes up to 132 columns.
- Enhanced VGA refresh rates up to 85 Hz.

CRT Controller (CRTC)

- The CRTC subsystem supports standard display resolutions up to 2048x1536. For higher resolutions up to 4k by 4k, or non-standard modes, contact ATI.
- Internal filtered upscaling (ratiometric expansion) for fixed resolution displays (e.g. panels or digital CRTs). Scales source images up to 1280x1024 up to any panel size. Fully supports all VGA modes (including mode X, and all Asian DOS modes), as well as all accelerated graphics modes.
- Support for auto-centering for fixed resolution displays. This fully supports all VGA and accelerated display modes, on all panel sizes up to 1600x1200.

- Auto-centering and ratiometric expansion can be used together to allow expansion of the image while maintaining the correct aspect ratio of the image.
- Support for reduced blank flat panels meeting the VESA proposed standard. Ratiometric expansion also supported on reduced blank panels.
- Produces synchronization signals to allow the drawing engines to synchronize with the display raster or frame.
- Display buffer flips (front and back buffer) are synchronized with the drawing engines, and can occur on either vertical (normal) or horizontal retraces.
- Frame rate monitoring and adjustment logic to allow frame rate locking to video broadcasts (GEN-locking).
- Support for both interlaced and non-interlaced display.
- Up to 200 Hz vertical refresh rate.
- Support for overscan and double scan.
- Separate or composite horizontal and vertical sync generation.
- Supports stereoscopic display with generation of synchronization signal and automatic flipping of left/right eye images.
- Optional hardware interrupt generation on CRTC events.

2.2.5 Accelerated Graphics Display

- Supports 8 bpp indexed color modes, and 15, 16, 24 and 32 bpp true color modes.
- Also enhanced with 4444 aRGB 16 bit mode and 88 alpha/index 16 bit mode.
- Enhanced gamma correction in all modes (triple 256 entry by 10 bit palettes).
- 8 bit alpha blending of graphics and overlay surfaces using either per-pixel alpha (when in appropriate graphics bit depth) or global alpha.
- Per-pixel range based keying for video overlay, and multiple key mix functions. Key mix results can optionally use global graphics/overlay alpha blender for fade-in/out effects.
- Hardware cursor up to 64x64 pixels operates in 4 modes:
 - 2 bpp XGA compatible, with two solid colors, transparent, and inverse transparent.
 - Full color AND/XOR mixing, used in Windows 9x.
 - Full color alpha blending, pre-multiplied alpha. Used in Windows 2000.
 - Full color alpha blending, non pre-multiplied alpha.
- Support for 128MB local frame buffer, or display from system memory via SMA, in

either linear or tiled addressing modes.

2.2.6 Video Scaler/Overlay

- Surface formats supported: RGB1555, RGB565, RGB8888, Planar YUV9, YUV12, Packed YUYV, UYVY.
- YUV upscaling is four-tap horizontal by four-tap vertical on all color components for sources less than 920 pixels wide. For wider sources it reduces to two-tap vertical.
- YUV downscaling is up to eight-tap horizontal by four-tap vertical on all color components. The horizontal downscaling is always a minimum of four-taps, and increases to eight-taps as the horizontal scale ratio decreases. The vertical downscaling is normally four-taps, and only reduces to fewer taps if memory bandwidth limitations require. Vertical downscaling is always a minimum of two-tap.
- RGB upscaling and downscaling is four-tap horizontal by two-tap vertical. For vertical upscaling ratios greater than or equal to 1.5, the vertical filtering is four-tap. For horizontal downscaling below 0.5, the horizontal filtering increases to up to eight-taps.
- The scaler can add black borders for DVD letterboxing and then composite the subpicture on top of the black.
- The scaler can zoom in with subpixel windowing accuracy.
- The scaler can pan-and-scan for DVD.
- When the video window is cropped, resized or moved on the desktop, it can be updated (a register locking mechanism allows autonomous updates of the overlay characteristics) without any artifacts thanks to sufficient double buffering of scaler control register fields. Double buffering can be disabled.
- Supports either capture port or an application as a video provider.
- Supports standard de-interlacing methods of bob (with vertical shift on either field) or weave two fields together in a variety of styles designed to eliminate motion artifacts, including styles optimal for films provided via NTSC and PAL video standards, and viewing freeze frame on a VCR.
- Features adaptive de-interlacing mode that optimally de-interlaces image on a per-pixel basis. Provides sharp images where no motion in source material, and no temporal de-interlacing artifacts in motion areas.
- Quad 960-pixel line buffer for up to 4 tap vertical scaling.
- Double wide source mode to allow all ATSC widths, including 1920 x1080. Vertical filtering is 2-tap when source more than 960 pixels wide.
- Fully programmable YUV to RGB conversion. Supports all variations of YUV used in DVD, HDTV, NTSC, PAL, MPEG-2 and so on.

- Full color controls on video data, including brightness, contrast, saturation and hue.
- Adjustable gamma correction on overlay to improve brightness and contrast of video on CRT and flat panel displays.
- Optional per-pixel video keying for mixing with graphics.
- 30bpp accuracy in scaled and color converted overlay data. Ability to overlay 30 bpp video in any graphics bit depth (8/15/16/24/32).
- Features ability to mix YUV inputs with RGB graphics and cursor, and keep all original YUV video colors intact for standard definition TV output or HDTV output. This applies even to YUV colors that can not be represented in the normal RGB color space.
- Window controller for per-pixel alignment and scaling of video window.

2.2.7 Display Output

- Integrated triple 10-bit DACs with built-in reference circuit.
- Internal DACs can operate in VGA mode to support standard RGB monitors with up to 350 MHz pixel clock. Compliant with VESA VSIS proposal on VGA analog signal quality.
- Internal DACs can also operate in YPbPr mode to directly drive component video signals to standard or high definition televisions with up to 175 MHz pixel clock. Programmable embedded synchronization signal generator supports all YPbPr dual and tri-level sync formats. Internal RGB to YPbPr converter supports all versions of YPbPr used in standard and high definition television.
- Analog monitor detection with integrated DAC comparators.
- Support for DDC1 and DDC2B+. Also AppleSense compatible. Independent DDC lines for both VGA and DFP/DVI connector. These are also separate from the VIP I2C lines.
- Integrated TMDS transmitter operating up to 165 MHz. Supports standard single channel DVI, or DFP connections. Includes integrated conversion of 30bpp data to either 24bpp or 18bpp, depending on attached panel type.
- Digital interface to external video encoder for NTSC or PAL signal generation. Works with ATI RAGE THEATER. Supports either RGB or YUV data interface to RAGE THEATER. Using YUV improves overlay video image quality. Also supports independent brightness, contrast and saturation on graphics and video overlay pixels to account for graphics pixels normally being fully saturated colors and video not being fully saturated.
- Digital frame based CRC generator for testing of display system output.

2.2.8 Video Interface Port (VIP)

VESA Video Interface Port is a multimedia bus consisting of 3 components:

- Video capture bus
- I2C bus
- VIP host data bus

VIP is used to communicate with multimedia devices such as video decoders, MPEG decoders, analog or digital tuners, audio chips and so forth.

Video Capture Bus

The video capture port supports a direct connection to MPEG decoders such as the IBM-CD11, and video decoders such as Brooktree BT829, BT827, BT819, BT817, or BT815, and Philips SAA7111/2.

The video capture port has the following features:

- Support of video stream format of BT ByteStream, MPEG transport stream, CCIR656 and VIP 2.0.
- Capable of video data capture rate up to 75MHz.
- Hardware support for “bob and weave” and single field video capture.
- Hardware support for 3:2 pull down.
- Hardware support for VBI data capture.
- Hardware support for ANC data block capture.
- Hardware support of video mirroring during capture.
- Filtered horizontal 7-tap downscaling at 2:1 and 4:1 ratios. This allows high definition sources to be downscaled horizontally on capture with high quality while saving memory footprint and bandwidth. Note that with the 1920 wide backend overlay line buffers, this downscaling is optional for high definition capture.
- Vertical downscaling by decimation at 2:1 and 4:1 ratios.
- Hardware support for single, double, triple, and quadruple buffering for video capture.
- Hardware support for double, and quadruple buffering for VBI and/or ANC data.
- Hardware support for VBI horizontal downscaling.
- Hardware support of Signed UV format.
- Hardware interrupt support for video capture.

I²C Bus

The I²C is an industry standard 2-bit serial bus. The I²C interface allows programming of peripherals such as video decoders, MPEG decoders, analog or digital TV tuners, teletext decoders, audio decoders and volume control.

Two dedicated signals, SCL and SDA, are used to drive the I²C pins. The I²C data transfer can be done in two modes, the software mode, or the hardware-assisted mode. In general, software controlled I²C has higher flexibility, but yields a lower throughput because of the time involved in polling and programming the I/O signals from the CPU.

Since the I²C clock and data are open-collector signals and rely on external pull-up resistors, noise can be a factor during transition from low to high. The I²C in RADEON can be programmed such that I²C signals can either be driven high directly, or be pulled up by the external pullup resistor as before. Note that the driving of SCL and SDA signals is controlled separately, as this eliminates any potential problem when only one of the signals is desired to be driven high.

VIP Host Data Bus

The VIP host data bus is capable of supporting a maximum of four slave devices with the graphics chip being the sole master. The bus can be configured to use 2-bit data width and run up to a maximum of 33 MHz clock speed.

In RADEON, up to four busmaster channels can be supported and each channel is set to run in either direction, i.e., system or frame memory to VIP devices or VIP devices to system or frame memory. During FIFO burst mode, it is possible for the VIP driver to do a VIP register access without having to change the operating mode. Arbitration between register and FIFO transfer is hidden in the hardware. In addition, the VIP controller is capable of doing byte-aligned and dword aligned transfers, automatic interrupt polling, automatic FIFO port polling, and detect VIP slave timeout.

2.2.9 AGP/PCI Host Bus Interface

The system host bus interface supports both AGP 2.0 and PCI 2.2 standards. Support for AGP is by sideband addressing mode. Texture map data for 3D objects can be obtained directly from system memory. Similarly the AGP can be used to accumulate MPEG-2 playback. In these applications, AGP fetches are execute-mode fetches.

The AGP function supports pipelined reads, long reads, and writes of varying lengths. All throttling is done in a way to prevent the bus from being waitstated. The AGP function supports 1x, 2x, and 4x data transfers.

With PCI 2.2, functions such as bus control, data flow control, address/data signal generation, signal timings, and address decoding are supported. Data flow control is enhanced by a 6x64-bit write-through FIFO available in both VGA and direct memory modes. The controller achieves zero wait-state memory read/write burst cycles with burst access. It also supports Block I/O decoding and DAC palette snooping, and a separate aperture for accessing registers.

Bus mastering between: (1) system memory and frame buffer, (2) system memory and VIP, (3) frame buffer and VIP, (4) system memory and GUI engine, and (5) frame buffer and GUI engine, allows all data transfer operations to be off-loaded from the host processor onto the motherboard chip set. The bus mastering function concurrently performs transfers on nine buffered channels, moving one "programmable amount" at a time from each of them, to prevent the other channels from starving. It also selects the highest available transfer protocol (PCI or AGP) to be used for each of the channels, and arbitrates between them.

2.3 Operation Modes

The RADEON operates in two distinct modes:

- VGA mode.
- Accelerator mode.

These modes are mutually exclusive. However, they share the same frame-buffer memory and I/O ports. They are described in the following sections.

2.3.1 VGA Mode

VGA (Video Graphics Adapter) is an established industry standard created by IBM. When operating in VGA mode, the host application draws directly into the frame buffer using the VGA controller. The accelerator controller is disabled and no rendering operations are accelerated. The VGA controller and the data path from the host application to the frame buffer are shown in the figure ([Figure 2-1. on page 2-2](#)). The VGA Controller registers are programmed using conventional I/O.

There are many published texts that describe VGA programming. Consequently, this manual does not cover programming the VGA controller. For a comprehensive, informative source on this subject, refer to *Programmer's Guide to the EGA, VGA, and Super VGA Cards* by Richard F. Ferraro.

For SuperVGA programming, the RADEON supports the Video Electronics Standard Association (VESA) Video BIOS Extension (VBE) 2.0 programming interface. This

interface was created by VESA to provide a standard, hardware independent method for using Super VGA display modes. Contact VESA for more information about VBE.

2.3.2 Accelerator Mode

When operating in accelerator mode, rendering operations are performed by the RADEON's accelerator controller. The VGA controller is disabled. The host application is limited to setting up the accelerator controller, and the controller renders directly to the frame buffer.

The accelerator controller contains the following engines:

- 2D Rendering Engine that performs 2D rasterization.
- TCL engine that performs transform, clipping and lighting operations.
- 3D Setup engine that performs 3D primitive setup operations.
- 3D Render Engine that performs 3D rasterization.
- Video Engine

The engines are collectively referred to as the Graphical User Interface (GUI) engine ([Figure 2-1. on page 2-2](#)).

The following two modes are used to program the GUI engine:

- Programmable Input and Output (PIO) mode.
- Concurrent Command Execution (CCE) mode.

These programming modes are described in the following sections.

2.4 Drawing Modes in Acceleration-operation Mode

Programmable I/O (PIO) Mode

In this mode, the host application programs the GUI engine by writing directly to the RADEON's memory-mapped registers. The registers are written through one of the RADEON's two register apertures over the bus interface. The register writes are queued in the RADEON's internal 64 entry Command FIFO buffer as register-datum pairs. These Command FIFO buffer entries are processed by the GUI engine to draw into the frame buffer.

To see the data path from the host application to the Command FIFO, [Figure 2-1. on page 2-2](#)

For more details about the PIO-mode programming, [Chapter 4](#).

For more details about the RADEON's register apertures, refer to [“Memory Apertures” on page 2-26](#).

Concurrent Command Execution (CCE) Programming Mode

In this mode, the host sends commands to the RADEON in the form of *command packets*. A command packet is a data block that consists of a header followed by a variable size data body. Within the RADEON, the packets are queued in the 192 entry CCE FIFO buffer. A micro controller processes the packets, produces the conventional register data, and feeds this data to the Command FIFO buffer. The Command FIFO buffer data is processed (as it is in PIO mode) to render into the frame buffer.

The host application transfers packets to the CCE FIFO buffer using the following two methods:

- Write them directly into the CCE FIFO buffer through memory-mapped register writes over the bus interface.
- Queue them in system memory buffers and bus-master them to the CCE FIFO buffer.

The second method is by far the most efficient for programming the RADEON. ATI highly recommends using the bus-mastered CCE programming mode as the primary programming method. Streaming packets in this manner enables significant concurrency between the host and the RADEON. In addition, there are several predefined single-purpose packets that greatly simplify the programming of common drawing operations.

The RADEON uses the following two mechanisms for bus-mastering packets to the CCE FIFO buffer:

- Ring buffer
- Indirect buffer

These mechanisms are described in the following sections.

Ring Buffer

The ring buffer is a continuous block of memory allocated by the host application in AGP GART memory. The host and RADEON treat this buffer as a circular buffer by wrapping

back to the starting address when they reach the end. The starting address and the size of the buffer are passed to the RADEON when initializing the CCE bus-mastering mode.

The application copies packets into the ring buffer in consecutive order starting at the top. It instructs the RADEON where to read the next packet by writing to a CCE write-pointer register. The RADEON triggers bus-mastering operations to transfer packets from the ring buffer to its CCE FIFO buffer according to watermarks set during CCE initialization. After completing the transfer, the RADEON uses bus-mastering to update a host application read-pointer to indicate where it has read to in the ring buffer. The physical address of this pointer is passed to the RADEON during CCE initialization.

For more details about programming the ring buffer, [Chapter 5](#).

To view a diagram of the ring buffer, refer to [Figure 2-2. on page 2-16](#).

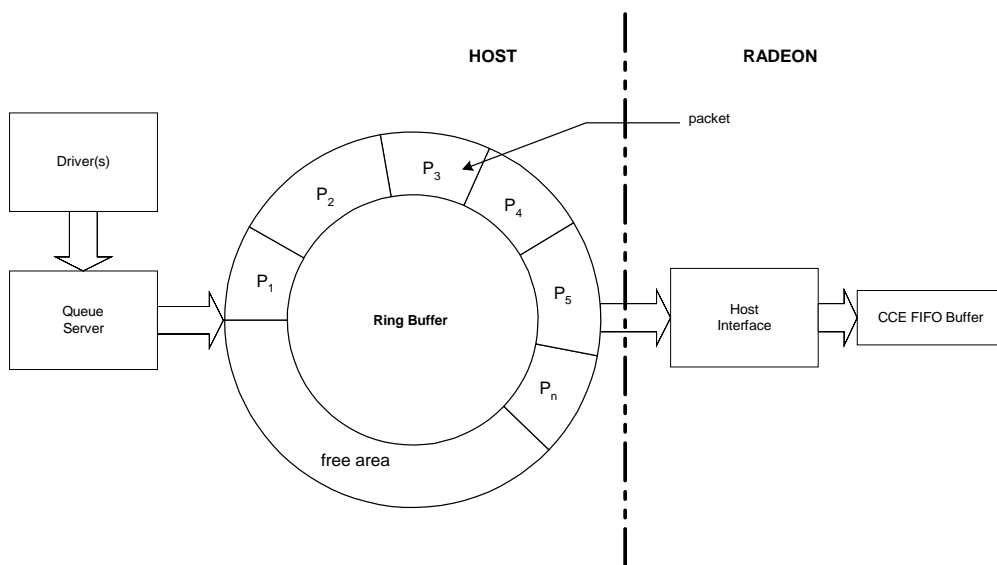


Figure 2-2. The Ring Buffer

Ring Buffer Queue Server

For multitasking operating systems where multiple clients may require synchronized access to the graphics resources, it may be beneficial to employ a queue server mechanism to arbitrate and control access to the ring buffer. This mechanism could enumerate clients and use semaphores to synchronize and protect access.

For an example of how to submit packets using such a mechanism, [Chapter 5](#).

Indirect Buffer

The indirect buffer is a contiguous block of memory allocated by the host application in AGP memory. The host and RADEON treat this as a linear buffer. They do not employ any buffer wrapping mechanisms for the indirect buffer.

The indirect buffer is also used for streaming command packets to the RADEON. In fact, we recommend using the indirect buffer as the primary means for streaming commands, especially for multimedia drivers such as 3D, DVD, etc. The indirect buffer allows long command sequences to be built up in parallel by different contexts or threads and then submitted to a queue server mechanism without the overhead of copies. The concept of queue servers is discussed in chapter 5. For more details, refer to [“Queue Server” on page 5-9](#). A packet transfer is initiated by writing the offset from the start of the indirect buffer and the size of the packet to specific registers. For more details about this procedure, refer to [“Ring Buffer Management” on page 5-7](#).

Packets to write these registers can be queued in the ring buffer, allowing the ring buffer to queue indirect buffer packet transfers to the RADEON CCE FIFO.

If the ring buffer is not used, these registers may be written through PIO. As a final note, packets may be placed and accessed from the indirect buffer in arbitrary order.

For a conceptual diagram of the indirect buffer, refer to [Figure 2-3](#).

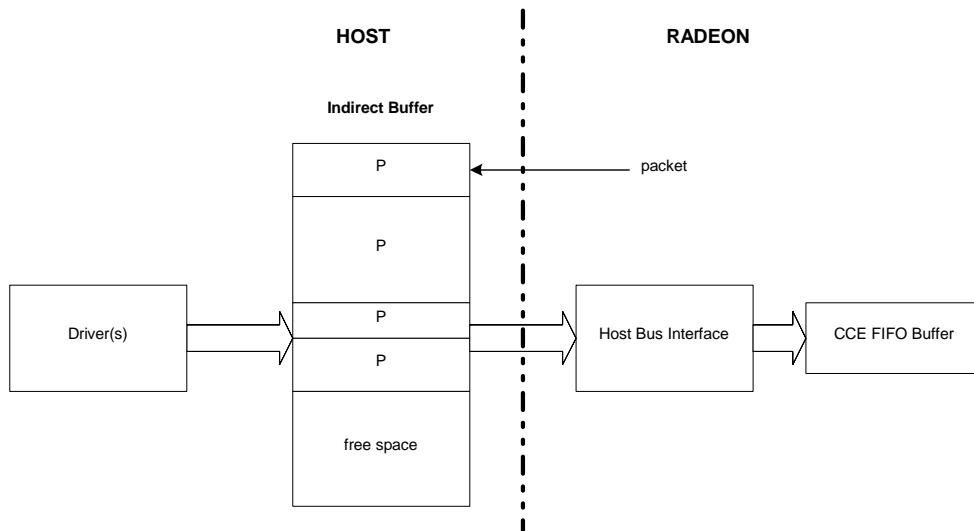


Figure 2-3. The Indirect Buffer

2.5 Review of Imaging Terminology

This section describes some background and terms about computer imaging.

2.5.1 Raster Image

Due to CRT (*Cathode Ray Tube*) technology, an image is broken into a number of equally spaced *scanlines*. Each scanline may be further broken into a number of smallest-viewable elements, called *pixels*. This type of image is commonly referred to as *raster image*.

The process of breaking an ordinary image into a raster image is called *rasterization*. This process allows an M -by- N array to represent an image, where:

- M represents the width of the image.
- N represents the height of the image (with its x-coordinate pointing to the right and y-coordinate pointing downwards).

The value of an element in this array represents the pixel's **color intensity**. This setup allows the video memory to contain the features of an image (i.e. image dimensions and color depth).

The **screen image** is the case where the raster image covers the entire CRT screen. The origin of the coordinate system is at top-left corner of the screen, where:

- The width of the image equals the number of pixels per scanline of the screen.
- The height is the number of scanlines that the screen has.

2.5.2 True RGB Color

Color in the real world is called **natural color** and it is represented as an analog quantity. Color from a CRT screen is called **digitized color** and it is represented as a digital quantity. The digitized-color value is an approximate of the natural-color value. The analog value can represent by an infinite number of color values. However, the digital value is limited in the number of unique (i.e. distinctive) colors.

For example, the maximum number of distinctive colors currently defined for this approximation is about 16 million (i.e. 2^{24}).

Each digitized color is represented by a combination of three **color components** (*Red*, *Green*, and *Blue*). The intensity of each component is divided into 256 levels. Zero intensity represents the lowest value and '255' represents the highest. Each component needs 8 bits. Therefore, to represent a color made up of a R, G, and B component, 24 bits are needed. This representation of digitized color is referred to as the **True RGB color**.

2.5.3 Representing Pixels

A RADEON can display monochrome and color images.

- Monochrome images refer to text.
- Color images refer to digitized color photographs, movies, and computer-generated color images.

Monochrome Images

Monochrome images are composed of pixels that can have just one of two digitized colors (i.e. black and white). Each pixel's color is represented by one bit (i.e. '0' for black and '1' for white).

The depth of the pixel is one *bit per pixel* (bpp). Monochrome pixels may be assigned with any two digitized colors, one representing the *foreground color*, such as *white*, and the other representing the *background color*, such as *black*.

To display blue-colored text on a background of white, a ‘0’ represents the color *blue* and a ‘1’ represents the color *white*. This type of treatment to monochrome images is termed *color expansion*. In fact, the realization of pixels in a mono image is done by mapping 0’s and 1’s onto background and foreground colors represented in the RGB format, although the memory representation of the pixels is one bpp.

Formats for Various Color Images

Color images may be represented in 8-, 15-, 16-, and 32-bpps formats (i.e. 2^8 , 2^{15} , 2^{16} , 2^{24} colors respectively).

The number of colors that can be displayed in the 32-bpp format is 2^{24} because the most significant eight bits of the 32 bits are not used. Using the byte as a measure of memory:

- One byte to represent a pixel for the 8-bpp format.
- Two bytes for the 15- and 16-bpp formats.
- Four bytes for the 32-bpp format.

1-bpp Format

Table 2-3 1-bpp Format (left-to-right)

1-bpp, BYTE_PIX_ORDER = 0 (left-to-right), Draw Engine Only																															
Structure of the Drawing Data as Used by the RADEON																															
19	1A	1B	1C	1D	1E	1F	20	11	12	13	14	15	16	17	18	9	A	B	C	D	E	F	10	1	2	3	4	5	6	7	8
Drawing Data Placed in Video Memory																															
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20
LSB																															
MSB																															

Table 2-4 1-bpp Format (right-to-left)

1-bpp Format, BYTE_PIX_ORDER = 1 (right-to-left), Draw Engine Only																															
Structure of the Drawing Data as Used by the RADEON																															
20	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
Drawing Data Placed in Video Memory																															
8	7	6	5	4	3	2	1	10	F	E	D	C	B	A	9	18	17	16	15	14	13	12	11	20	1F	1E	1D	1C	1B	1A	19

8-bpp Format

The value of a pixel does not represent the intensity of a color. Instead, it represents the index of the color table, called the *color palette*. The palette stores all of the possible colors that could be used. The host application uses this value to point to a specific color in the palette. Color represented in the 8-bpp format is known as *pseudo-color*.

Table 2-5 8-bpp Pseudo-color Format

8-bpp Pseudo-color Format			
Structure of the Drawing Data as Used by the RADEON			
4 (MSB)	3	2	1 (LSB)
Drawing Data Placed in Video Memory			
1 (LSB)	2	3	4 (MSB)

15-bpp, aRGB, or 1555 Format

This format uses two bytes to represent the three color components (Red, Green, and Blue). Each component uses five bits to represent its intensity.

- Bit [15] is not used (shown in the table as ‘a’).
- Bits [14:10] represent red.
- Bits [9:5] represent green.
- Bits [4:0] represent blue.

Table 2-6 15-bpp, aRGB, or 1555 Format

15-bpp, aRGB, 1555 Format			
Structure of the Drawing Data as Used by the RADEON			
Pixel #2 aRRRRRGGGGGBBBBB		Pixel #1 aRRRRRGGGGGBBBBB	
Drawing Data Placed in Video Memory			
Pixel #1 low GGBBBBB	Pixel #1 high aRRRRRG	Pixel #2 low GGBBBBB	Pixel #2 high aRRRRRG

Note: This format is similar to the 16-bpp format. But this format uses one alpha bit, the dummy bit (i.e. ‘a’). Sometimes this dummy bit maybe used for 3D rendering. For typical applications, this bit not used.

16-bpp, RGB, or 565 format

This format is similar to the 15-bpp format:

- Bits [15:10] represent red
- Bits [10:5] represent green
- Bits [4:0] represent blue.

Table 2-7 16-bpp, RGB, 565 Format

16-bpp, RGB, 565 Format			
Structure of the Drawing Data as Used by the RADEON			
Pixel #2		Pixel #1	
RRRRRGGGGGGBBBBB		RRRRRGGGGGGBBBBB	
Drawing Data Placed in Video Memory			
Pixel #1 low	Pixel #1 high	Pixel #2 low	Pixel #2 high
GGGBBBBB	RRRRRGGG	GGGBBBBB	RRRRRGGG

32-bpp, RGBa, or 8888 Format

This format is similar to the 24-bpp format with the addition of a dummy byte.

Table 2-8 32-bpp, RGBa, or 8888 Format

32-bpp, RGBa, or 8888 Format			
Structure of the Drawing Data as Used by the RADEON			
A	R	G	B
Drawing Data Placed in Video Memory			
B	G	R	a

2.5.4 Pixels

The RADEON supports pixel depths of 1, 8, 15, 16, and 32 bits per pixel.

The pixels are consumed from the most significant bit (MSB) to the least significant bit (LSB) (or vice versa, depending on the RADEON's configuration).

The following shows the bit definitions of the pixel formats in BYTE and DWORD representations (i.e., this is the 'little endian' representation):

- The ordinal values represent the ordering of the pixels in memory for a left to right pixel trajectory beginning on a DWORD boundary.

- The ordinal value '1' represents the position in memory of the left-most pixel in the DWORD.
- The color components are denoted as R, G, and B.

2.5.5 Pitch

In ATI terminology, ***pitch*** measures the size of memory for representing a scanline of pixels. Due to the RADEON's design, this measure must satisfy the following two requirements:

- Pitch must be an integer multiple of eight pixels. If the number of pixels per scanline does not meet this requirement, add the required number of dummy pixels to the scanline.
- The memory size of a pitch must be a multiple of 16 bytes.

If we denote the number of pixels per scanline by m , the number of added dummy pixels by n , and the number of bytes used to representing a pixel by l , the two requirements can be written as:

$$(m + n) \text{ MOD } 8 = 0 \quad \text{Equation 2.1}$$

Since l is restricted to values 1/8 for monochrome images, and 1, 2, 3, and 4 for color images, it is easy to show that Equation 2.1 is implied by Equation 2.2.

$$l \times (m + n) \text{ MOD } 16 = 0 \quad \text{Equation 2.2}$$

Using Equation 2.2, the number of dummy pixels can be calculated by the following equation:

$$n = \begin{cases} 128 - m \text{ MOD } 128 & l = 1/8 \\ 16 - m \text{ MOD } 16 & l = 1, 3 \\ 8 - m \text{ MOD } 8 & l = 2 \\ 4 - m \text{ MOD } 4 & l = 4 \end{cases} \quad \text{Equation 2.3}$$

Using Equation 2.3, the pitch can be written as:

$$Pitch = \frac{m+n}{8} \quad \text{Equation 2.4}$$

The size of memory for the pitch is:

$$PitchMemSize = l \times Pitch \times 8 \quad \text{Equation 2.5}$$

Equation 2.3, Equation 2.4, and Equation 2.5 are also applicable to the pitch of the bitmap, where m corresponds to the width of the bitmap. According to these equations, the pitch of an 800x600 screen can be 100 units of eight pixels, and the corresponding memory size is 1600 bytes ($2 \times 100 \times 8$) provided each pixel is represented by 16-bit color. To enable the block-write capability of the RADEON, the second requirement on defining a pitch must be changed to 128-byte alignment.

This leads to a modification of Equation 2.2, which can be rewritten as:

$$l \times (m+n) \text{ MOD } 128 = 0 \quad \text{Equation 2.6}$$

In addition, a corresponding modification to the calculation of dummy pixels has to be made; this effort is left for you.

2.5.6 Video Memory

The RADEON uses the **video memory** (i.e. the **frame buffer**) to display geometrical images on the CRT's screen. The frame buffer is further divided into the following areas:

- The **on-screen area** represents the entire screen image.
- The **dummy area** makes up the pitch of screen due to the hardware requirement.
- The **off-screen area** stores information about the image (e.g. bitmaps). There are some conditions when there is no off-screen area (e.g. the video memory may contain the on-screen data and any unused areas contain data about the depth of the pixels).

As a result of the equations developed in the previous section, it is easy to calculate the allocation of the video memory when a display configuration is given.

For example, the required display configuration is 800x600 pixels in the 16-bpp format mode. Consequently, the calculated memory size for the on-screen area is 960,000 bytes.

Therefore, the minimum size for the video memory must be 1MB. If the size of the video memory is 4MB, there will be more than 3MB left over for the off-screen area. The RADEON can support up to a maximum of 32MB of video memory.

Video Memory Addressing

Conventionally, the lowest address of the video memory corresponds to the top-left corner of the on-screen area, and the highest address to the bottom-right corner of the off-screen area.

For example, the video memory is 16MB. Then, the following conditions are true:

- The top-left corner of screen corresponds to address 0.
- The bottom-right corner of the off-screen area corresponds to 0xFFFFF0.

The RADEON addresses the video memory from zero to the upper bound. The video memory address seen by RADEON is called the ***Physical Memory Address***.

To begin drawing to the top-left corner of the CRT screen, set registers SRC_OFFSET and DST_OFFSET to zero. These two registers can be set to any value within the address space of video memory.

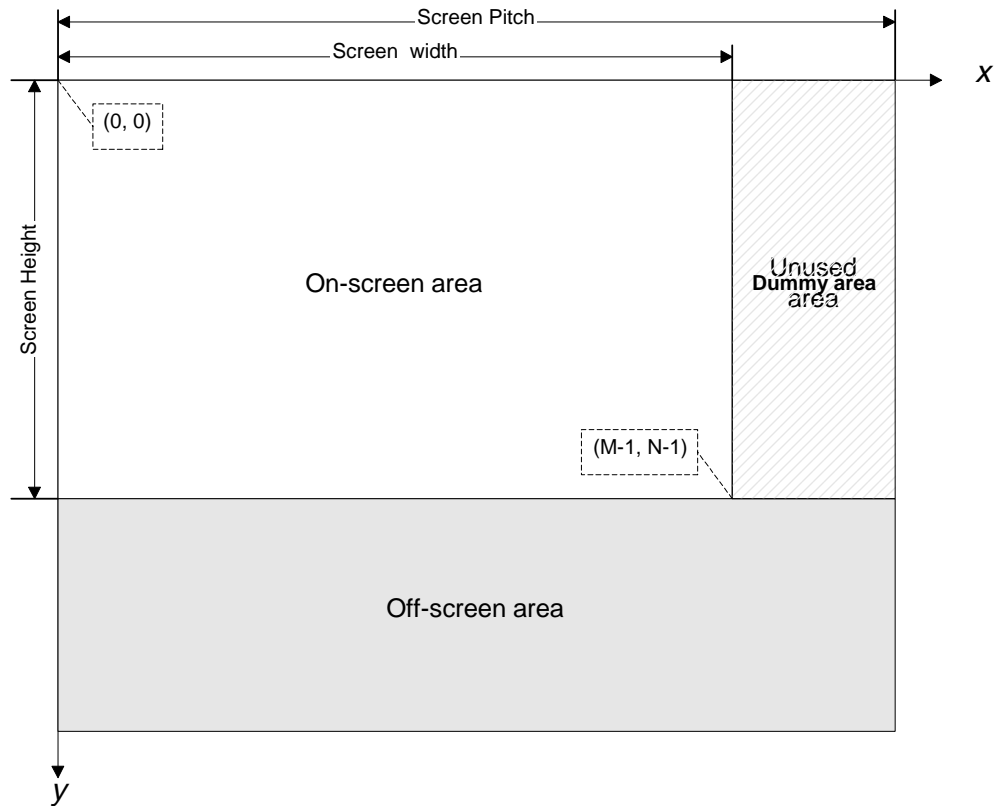


Figure 2-4. Video Memory

2.6 Memory Apertures

The RADEON requires memory apertures from the system. These apertures map the video memory and registers onto the host's memory space. By using this mapping, the host application can access the frame buffer and the memory-mapped registers as if they were part of the system memory.

The following are the types of apertures that exist within the system space:

- The **register aperture** is used for the memory-mapped registers that are related to the RADEON.
- The **video aperture**.

Normally, the apertures are located somewhere within the 4GB address space where it does not conflict with the system (host) memory.

2.6.1 The Address Spaces

The programmer needs to understand two distinct 4GB (i.e. 32 bit) address spaces when dealing with the RADEON.

The system address space represents the real address space seen by the CPU. These are the addresses that the RADEON chip sees on the PCI/AGP interface.

The RADEON internal address space (or MC address space) is what all internal clients to the RADEON memory controller (MC) use. This is also a 32 bit address space, but its mapping may or may not be the same as the system address space.

2.6.2 Apertures of the System Address Space

In the system address space, there are several areas of interest to the RADEON programmer. These are:

1. The primary linear aperture (APER_0 or AP0).
2. The secondary linear aperture (APER_1 or AP1).
3. The memory mapped register aperture.
4. The AGP space of system memory.
5. The ROM aperture.
6. The VGA aperture.

The ROM & VGA apertures are not covered here. They are the same in RADEON as any past ATI chips.

2.6.3 The Linear Apertures

The primary and secondary linear apertures are for direct CPU read/write access to/from the RADEON local frame buffer.

The base addresses in the system address space are found in these registers:

CONFIG_APER_0_BASE.APER_0_BASE

CONFIG_APER_1_BASE.APER_1_BASE

The size of these apertures are both the same, the size can be found in the CONFIG_APER_SIZE.APER_SIZE register. These registers are the same as for the RAGE 128 series, but the sizes may be different.

For each linear aperture, there is independent control of the big/little endian swapping done on the data reads or writes. Control of the endian swapping has changed from the Rage 128, and is now located in the SURFACEx_INFO.SURFx_APy_SWP and SURFACE_CNTL.NONSURF_APy_SWP register fields. If a transaction falls into a surface, then it uses the respective SURFx_APy_SWP setting. If the transaction misses all surfaces, then the swapping is done as set by the appropriate NONSURF_APy_SWP register.

The RADEON supports eight sets of surface registers for tiling data to/from the frame buffer. If the SURFACEx_INFO.SURFx_PITCHSEL=0, then tiling is disabled for this surface (but endian swapping settings still apply). When the pitch is non-zero, then the specific tiling method is set by the SURFACEx_INFO.SURFx_TILE_MODE register. The choices are as follows:

- 00 = Color macrotiling with no microtiling
- 01 = Color macrotiling and microtiling
- 10 = 32 bit Z tiling
- 11 = 16 bit Z tiling

RADEON offers two methods for how the two linear apertures map into the internal MC address space. This is controlled by HOST_PATH_CNTL.HDP_APER_CNTL.

When HDP_APER_CNTL=0, the two linear apertures both map into the same piece of the internal RADEON MC address space. When HDP_APER_CNTL=1, the second aperture maps into the internal address space directly above the primary aperture's mapping.

This is illustrated below:

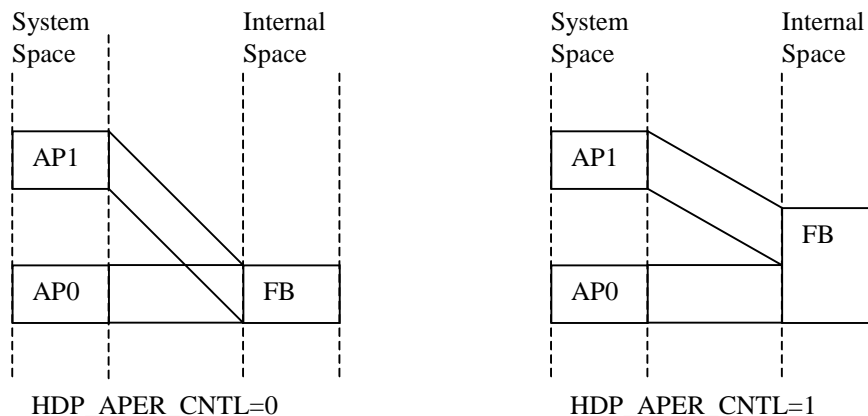


Figure 2-5.

Which method is used also affects how the SURFACE tiling registers operate. When both apertures map to an overlapped area, then the surface registers are limited to the range of `CONFIG_APER_SIZE.APER_SIZE`, and the surface ranges appear in the same locations in both apertures. When the two apertures map end-to-end, the surface ranges can cover double the value of `APER_SIZE`, and ranges can span across the boundary between the two apertures.

2.6.4 Setting the Linear Aperture Sizes

The size of the linear apertures (i.e. the value of `CONFIG_APER_SIZE.APER_SIZE`) is controlled by straps read into RADEON during the system reset. When a ROM is attached to RADEON, the size is taken from the `AP_SIZE` strap in the ROM. When no ROM is attached (e.g. on a motherboard implementation) the `AP_SIZE` defaults to 00. The current setting of `AP_SIZE` can be read from `CONFIG_XSTRAP.AP_SIZE`. The settings are:

00 = 2 x 64 MB
 01 = 2 x 128 MB
 10 = 2 x 32 MB
 11 = reserved

On a motherboard or other case with no ROM attached (e.g. Aurora) the `AP_SIZE` can not be changed. When a ROM is present, `AP_SIZE` can only be changed by flashing a new setting into the ROM and re-booting.

2.6.5 The Register Apertures

The memory mapped registers are repeated twice in their PCI aperture. The physical base address of the memory mapped register aperture can be found either in the PCI configuration space (REG_BASE @ 0x18), or by using MM_INDEX and MM_DATA in the block I/O register aperture (MM_INDEX=0xF18).

The size of each copy of the registers is in CONFIG_REG_APER_SIZE.REG_APER_SIZE. For the initial RADEON this is hardwired to 256 Kbytes (0x40000).

The location of the start of the second half (i.e. second copy) of the memory mapped register aperture is in CONFIG_REG_1_BASE. Read this whole register as one field to get the right value (i.e. it is not two separate fields as shown in the web based register spec).

Each half (or copy) of the memory mapped register aperture can have endian swapping applied as controlled by CONFIG_CNTL.APER_REG_ENDIAN as shown below:

- 00 = neither register aperture swaps
- 01 = both register apertures swap
- 10 = only register aperture 0 swaps
- 11 = only register aperture 1 swaps

2.6.6 RADEON Internal Address Space

Internally to the RADEON, all memory requests are made in a 32 bit address space. When the memory controller receives a read or write request from any internal client, the first thing it does is examine the address to determine which one of three classes it falls into, and therefore where to send the request as follows:

1. If in the range MC_FB_START...MC_FB_TOP then send to local frame buffer controller.
2. Else in the range MC_AGP_START... MC_AGP_TOP then send to AGP controller to generate bus master cycle to system chipset, where GART remapping will occur.
3. Else send to PCI master controller
 - a In PCI remapper space, then remap and do PCI bus master cycle,
 - b Else use original address for PCI bus master cycle.

This is illustrated below:

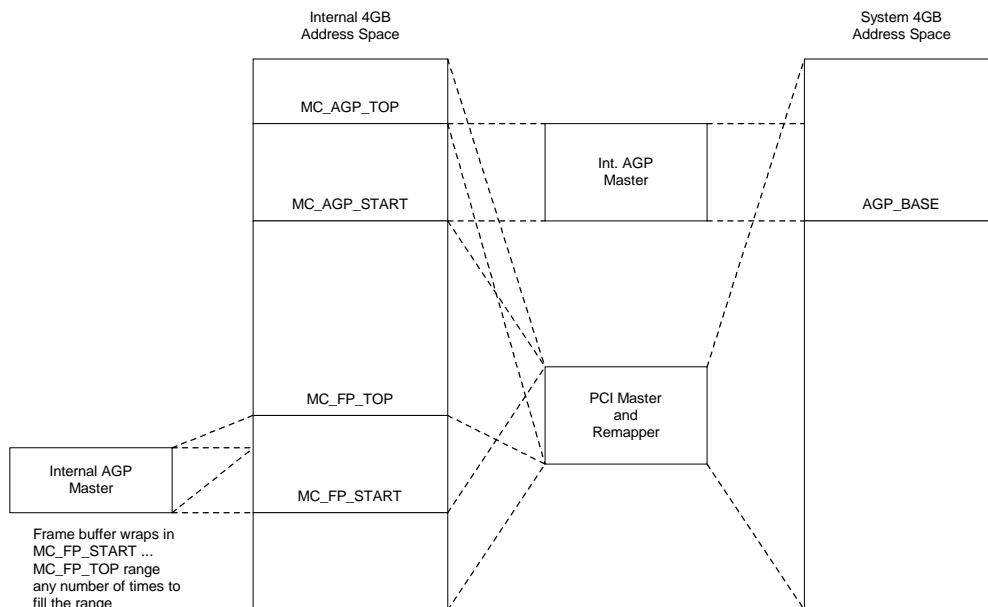


Figure 2-6. RADEON Internal Address Space

2.6.7 AGP Addressing

AGP Addressing in RADEON differs from RAGE 128 only in what address range of the internal clients maps into the AGP address range. In RADEON the AGP addressing range for internal clients is set by the `MC_AGP_LOCATION` register.

If the MC sees an address between `MC_AGP_START` and `MC_AGP_TOP`, it subtracts `MC_AGP_START` and gives it to the internal RADEON AGP interface controller. This then adds `AGP_BASE_ADDR` and generates an AGP cycle with that address to the system chipset. The system chipset then looks this physical address up in the GART to perform the physical-to-physical AGP remapping. This is then the final system memory address where the cycle is performed.

2.6.8 The PCI Remapper

Many clients in the graphics controller need to be able to access system memory linearly. An example of this kind of client is the CCE reader. The rest of this section will use CCE as the client, but the discussion applies to the other clients also.

The CCE reader wants to read linearly for the length of the buffer, which is in the order of many MB. When the CCE reader reads through a physical 4KB boundary, it will continue generating contiguous and sequential address. In other words, a series of CCE addresses around a 4KB boundary will look something like 03F0, 03F4, 03F8, 03FC, 0400, 0404. When the addresses pass from 03FC to 0400 a 4KB boundary is crossed. It is possible (likely) that the 2 physical pages that hold these two 4KB chunks of the CCE buffer are not contiguous. So a physical to physical address translation mechanism has been implemented that takes the sequential addresses that the CCE reader is generating, and translates them into the addresses where the CCE data physically resides in the system memory.

This translation is only done on accesses that are sent to the PCI bus master. Addresses that are within the AGP aperture are already translated via the AGP GART.

The PCI remapper translates addresses within the range specified by the AIC_LO_ADDR and AIC_HI_ADDR register values only. Addresses to the PCI bus master outside of this range are not translated at all. There is a TRANSLATE_EN bit at AIC_CTRL(0) which enables the translation when set to '1'. When this bit is '0', it not only disables the translation, but clears the TLB also.

Operation

Address translation happens in the following manner:

- 1 The AIC_LO_ADDR value is subtracted from the address that is presented to the PCI bus master. This gives an offset into the translation range
- 2 This offset is added to the AIC_PT_BASE value (this is a 4KB aligned value).
- 3 A read from system memory is made, and the data returned. This data is loaded into a TLB, along with bits 31:12 of the original request's address.
- 4 Finally the memory cycle for the original request is made. Bits 31:12 of the original address are substituted with bits 31:12 of the TLB read in #3.
- 5 Any subsequent requests made to the PCI bus master will follow the same #1 to #4 flow. If the new request matches the address stored in the TLB, the TLB contents will be used directly, and there is no need to do steps 2 and 3.

Initialization

The following needs to be set up for PCI remapper operation:

AIC_PT_BASE - this is the base of the PCI remapper table

AIC_LO_ADDR - this is the lower limit of the internal space that will be translated by the PCI remapper. (This assumes that the address goes to the PCI bus master.) Bits 11:0 of the HI and LO address values are wired to '0'. This implies that the HI and LO limits of the translation must be 4KB aligned. This should be obvious, of course, but is worth pointing out.

AIC_HI_ADDR - this is the upper limit of the internal space that will be translated by the PCI remapper. (This also assumes that the address goes to the PCI bus master.)

The PCI remapper table itself must be set up. Every DW in the table is an entry that points to 4KB blocks. These need to be set up for as many 4 KB blocks as the CCE reader will be using. So, if a 4MB ring buffer is being used, it will be necessary to initialize 1K entries. The PCI remapper table will be 4KB in this case. If CCE is using a 16MB ring buffer, the table has 4K entries, and is 16KB in size. These 16 KB must be linear in physical memory.

In order to have CCE reading properly, the CP_RB_BASE register must be set. This is the base, and the CCE read will add an offset to that as it is reading around the ring buffer. It is likely that CP_RB_BASE will have the same value in it as AIC_LO_ADDR. It is likely that this value is the same as the base of the ring buffer in system memory.

2.6.9 Recommended Configuration of the RADEON Memory Space and Base Registers

The safest way to configure the RADEON internal memory space is to have it look the same as the system 4GB address space. This ensures there are no conflicts where a RADEON internal block can not access a piece of system memory because the address conflicts with another location in the RADEON address space.

After the system BIOS enumerates the PCI/AGP busses, the ATI video BIOS or driver should program the RADEON as outlined below as the recommended default configuration of the RADEON address space and base registers.

First the frame buffer should be located at the same place in both address spaces:

```
MC_FB_LOCATION.MC_FB_START <=
(CONFIG_APER_0_BASE.APER_0_BASE) shr 16
```

```
MC_FB_LOCATION.MC_FB_TOP <= (CONFIG_APER_0_BASE.APER_0_BASE +
```

`CONFIG_APER_SIZE.APER_SIZE - 1) shr 16`

Also set the host data path to the typical aperture mapping:

`HOST_PATH_CNTL.HDP_APER_CNTL <= 0x0`

Next, if AGP is supported in the system, the AGP memory should be placed to correspond to the system. The `AGP_BASE.AGP_BASE_ADDR` register is programmed as for RAGE 128 chips. RADEON has no `AGP_SIZE` field as in RAGE 128. Instead the AGP size is set by the difference between `MC_AGP_TOP` and `MC_AGP_START`. Therefore:

`MC_AGP_LOCATION.MC_AGP_START <= AGP_BASE.AGP_BASE_ADDR shr 16`

`MC_AGP_LOCATION.MC_AGP_TOP <= (AGP_BASE.AGP_BASE_ADDR +
AGP_SIZE - 1) shr 16`

If PCI remapping is to be used, it is setup the same as for RAGE 128. This was discussed in the previous chapter.

Finally setup the default locations of the display clients to point to the local frame buffer:

`OV0_BASE_ADDR.OV0_BASE_ADDR <= MC_FB_LOCATION.MC_FB_START shl
16`

2.6.10 VGA Memory Aperture

When enabled for VGA, the RADEON claims the standard VGA resources. For most VGA graphics modes, the aperture is 128KB and starts at segment 0xA000.

2.6.11 Video BIOS

To relocate the RADEON's video BIOS, using the PCI configuration space. The system BIOS will normally shadow the entire BIOS image to the area starting at segment 0xC000 during system initialization.

2.7 Display Mode and Mode Switching

A *display mode*, also referred to as *video mode*, defines the following parameters:

- The type of display content.
- The screen resolution.
- The color depth of the pixels.

This implies that setting up a display mode is dependent on the available video memory. Once an operating mode is determined, a display mode must also be set for the RADEON according to the RADEON's capability and the available memory resource.

The RADEON supports the following display modes in the VGA operating mode:

- **VGA-alphanumeric mode** (also known as the *text mode*)
The text mode may further be classified into a number of sub-modes with variation in the size of character and in the color of text.
- **VGA-graphics mode**
This mode can also be further divided into sub-modes according to the screen resolution and the depth of color used to represent a pixel.

In the accelerator-operation mode, the RADEON supports the graphics mode with screen resolutions (from 320x200 to 2048x1536 pixels), and with depths of color (8, 16 and 32-bpp formats).

To switch from one display mode to another, call the BIOS service Set Display Mode (i.e. function AL = 1), and/or Coprocessor CRTC Parameters (i.e. function AL = 0).

2.8 Engine Discipline

In the accelerator-operation mode, the RADEON's GUI engine may use the PIO drawing mode or the CCE drawing mode.

If switching between these two operation modes is not handled properly, the RADEON may hang (i.e. stop operating). To avoid hanging the RADEON, follow these pointers:

- To safely switch from one mode to another, make sure the Command FIFO buffer is empty and it is in the idle state. This requires the program to check that the 31st bit of register RBBM_STATUS is set to zero.
- When the RADEON operates in the PIO drawing mode, the program must check if there are sufficient entries in the Command FIFO buffer before writing any data to it.

2.9 BIOS Services

A number of BIOS services are available. These services help to avoid problems of incorrectly setting up the RADEON or configuring the display mode of the system.

For details about *BIOS Services*, see [Appendix A](#) and [Appendix B](#).

Chapter 3

Accelerator Operation Mode

3.1 Scope

This chapter contains information about setting up the RADEON for accelerator operation mode. The intended audience for this information is X-type OS driver developers.

This chapter shows how to detect the RADEON without using the BIOS functions. The majority of the necessary information can be retrieved from the PCI Configuration Space, which is set at POST.

The following information can be retrieved through the PCI Configuration Space:

- PCI Vendor ID
- Device ID
- Revision ID
- BIOS segment
- Base address of the register
- Memory and I/O apertures

For host applications to access the registers and memories through the apertures, the initialization program needs to configure the RADEON for accelerator-operation mode, and convert the aperture addresses from physical space to linear space.

The initialization stage consists of the following four major steps:

- Step 1: Detecting the RADEON.
- Step 2: Obtaining the configuration information about the physical and linear (i.e. virtual) addresses of apertures.
- Step 3: Setting up a display mode.
- Step 4: Initializing the GUI engine.

3.2 Chip ID Registers

PCI_POS Registers

The `VENDOR_ID`, `DEVICE_ID` and `REVISION_ID` of an ASIC can be read through the `PCI_POS` registers in PCI configuration space. For convenience, copies of these registers are also available in the memory mapped register aperture. The memory-mapped copies are easily available without having to perform PCI configuration reads. These registers allow software to uniquely identify each family, type and variant of ATI chips.

PCI_POS Registers					
PCI Config Space Byte Address	Memory Mapped Byte Offset	Bits	R/W	Function	Power-Up Default
1:0h	F01:F00h	15:0	R	<code>VENDOR_ID</code>	1002h
3:2h	F03:F02h	15:0	R	<code>DEVICE_ID</code>	Indicates ATI ASIC family generation and which member of that generation
8h	F08h	3:0	R	<code>MINOR_REV_ID</code>	Indicates ASIC minor revision
		7:4	R	<code>MAJOR_REV_ID</code>	Indicates ASIC major revision

Figure 3-1. PCI_POS Registers

The `DEVICE_ID` is represented by a double ASCII character stored in little endian format that uniquely identifies the ASIC. For example, the default `DEVICE_ID` of the RADEON is "QD", or 5144h.

Table 3-1 RADEON Chips and Revision Identification

R100/RV100/M6/RV200/M7 ASIC (chip) PCI Device ID List	
R100 – RADEON 0x5144 0x5145 0x5146 0x5147 RV100 – RADEON VE (low-cost RADEON, dual CRT, no TCL) 0x5159 0x515A M6 – MOBILITY RADEON (RADEON VE based) 0x4C59 0x4C5A	RV200 – RADEON 7500 (RADEON based 2D & 3D pipe, dual-CRT) 0x5157 0x5158 M7 – MOBILITY RADEON 7500 (Mobility RADEON 2D pipe & RADEON 3D pipe) 0x4C57 0x4C58

3.3 Step 1: Detect the RADEON

This step determines the following:

- The presence of a RADEON within the system
- The various aperture addresses (memory, register, and I/O).

To determine much of this information, use the PCI configuration space. For the purposes of this document, the following lists several assumptions:

- The system uses the PCI host bus (since the RADEON is only available in PCI and AGP bus types).
- The OS being used provides an interface for querying the PCI configuration space. If this is not the case, the programmer must gain access to this information.

3.3.1 Using the PCI Configuration Space

To use the PCI configuration space, follow these steps:

1. Detect if a device is installed that contains the ATI PCI Vendor ID (0x1002). As per the PCI specification, offset '0' of the configuration space for a given device contains the PCI Vendor ID.
2. After identifying a device that has the ATI PCI Vendor ID, determine the Device ID.

3. Check to if the device ID matches the list of known RADEON device IDs. See Appendix B for device ID listings.
4. Note that the Device ID is located at offset 0x02 of the configuration space.
5. After obtaining the value of the Device ID, compare it against the above list. If there is a match, we can continue. Otherwise, we have two options:
 - Return an error, indicating that a RADEON device was not found; OR
 - Scan the BIOS segment to see if the 'RG6', 'M6' or 'RV100' signature string is found (note: a way to detect a RADEON revision that may not be in the list; RG6, M6, and RV100 refer to RADEON, Mobility Radeon and Radeon VE respectively). This protects against a driver not detecting new or revised RADEONs. However, this also has the potential for problems in that the new revision may require some modifications to the driver to work properly. This point should be considered before implementation.

For the latest list of Device IDs for the RADEON, contact Developer Relations at ATI (www.ati.com).

3.3.2 Scanning the BIOS Segment

By scanning the BIOS segment, the following information can be found:

- ROM ID
The ROM ID is defined as 'AA55' in the first two bytes of the BIOS segment.
- ATI product signature
The ATI product signature is '761295520'.
- RADEON string
The RADEON string is 'RG6', 'M6', 'RV100', 'RV200' or 'M7'.

For a successful installation of the RADEON, all three of these items must be present. They should all be present within the first 512 bytes of the BIOS segment.

3.3.3 Scratch Register Test

To confirm the presence of a RADEON board on the PCI bus, perform a read-and-write test on register BIOS_0_SCRATCH. Perform this test through the I/O port. Use the following steps:

1. Read and save the contents of register BIOS_0_SCRATCH.
2. Write the value (e.g. 0x55555555) to BIOS_0_SCRATCH.
3. Read back BIOS_0_SCRATCH. If the value is not the same as what was written, a RADEON is not present
4. Repeat steps 2 and 3, using the compliment of the previous value (e.g. 0xAAAAAAAA).
5. Restore the saved value of BIOS_0_SCRATCH.

3.4 Step 2: Obtain the Configuration Information

After locating the PCI configuration space for a RADEON, some additional configuration information can be retrieved, such as:

- Memory aperture base address (PCI configuration space offset 0x10).
- Register aperture base address (PCI configuration space offset 0x18).
- I/O base address (PCI configuration space offset 0x14).
- BIOS segment address (PCI configuration space offset 0x30).

The memory aperture base address value at offset 0x10 within the PCI configuration space is in bits [31:26] of its DWORD. Therefore, to isolate the proper bits, the value should be logically ANDed with 0xFC000000.

For the I/O base aperture, the actual value is within bits [31:8] of its DWORD (at offset 0x14). Therefore, to isolate the proper bits, the value should be logically ANDed with 0xFFFFF00.

The register aperture base value resides in bits [31:14] of its DWORD (at offset 0x18). Therefore, to isolate the proper bits, the value should be logically ANDed with 0xFFFFC000.

The BIOS segment, at offset 0x30, is in the upper WORD of this value (bits [31:17]), then shifted right one bit.

After obtaining these physical memory addresses for the memory and register apertures, convert them to virtual or linear addresses, so that the host application may use them.

Example Code: Converting the physical addresses to a usable virtual address

```
DWORD phys_to_virt (DWORD physical, DWORD size)
{
    union REGS r;
    struct SREGS sr;
    DWORD retval=0;

    memset (&r, 0, sizeof (r));
    memset (&sr, 0, sizeof (sr));
    r.w.ax = 0x0800;
    r.w.bx = physical >> 16;
    r.w.cx = physical & 0xFFFF;
    r.w.si = size >> 16;
    r.w.di = size & 0xFFFF;
    int386x (0x31, &r, &r, &sr);
    if ((r.w.cflag & INTR_CF) == 0)
    {
        retval = (long) (((long) r.w.bx << 16) | r.w.cx);
    } // if

    return (retval);
} // phys_to_virt
```

At this point, you have successfully detected that a RADEON-based graphics adapter is installed.

The following lists the configuration information about the adapter has been revealed:

- ASIC version (Device ID)
- BIOS segment
- Memory aperture address (both physical and virtual)
- Register aperture address (both physical and virtual)
- I/O base address

This gives sufficient information to begin the next step: setting up a display mode, and initializing the graphics engine (GUI).

3.5 Step 3: Set a Display Mode

This section covers how to set a display mode. To select a display mode, use one of the following methods:

- Use the BIOS function (i.e. the easy method).
- Manually set up the display mode (i.e. the hard method).

Easy Method

To set the display mode using an easy method, use the BIOS function 0x00. Supply parameters for the mode number and the color depth in the appropriate CPU registers. Then, call the function. A variant of this method also allows you to pass a CRT parameter table to supply custom CRT values, even custom resolutions.

Difficult Method

To set the display mode using a hard method, manually program the PLL and CRT to achieve the desired mode. Typically, protected-mode operating systems (i.e. usually X type OSs) must use this method (since they are unable to execute the BIOS functions within their OS). If the programmer has any possibility of using the BIOS to set the mode, this would be much preferred.

Using the BIOS Function

The RADEON can be set up in a particular display mode by calling the extended BIOS function 00h, **Set Display Mode**. Here are the inputs required for this function:

Table 3-2 Inputs for the Set Display Mode BIOS Function

Code	Purpose
di	Display Device Mask. This determines what display will be affected by this call. Default is '0', which affects all displays.
cl[0:3]	Color depth.
ch	Resolution.
dx:bx	Pointer to parameter table (if we choose to set the mode from a parameter table).

If you choose to use the BIOS installed modes, leave the simple set-register CH to the appropriate resolution. For the appropriate values, refer to the Video BIOS appendix. To pass a CRT parameter table, set CH = 0x81 and point to the parameter table using DX:BX (this is covered in the next section).

The BIOS functions can be called in two different manners. A far call to offset 0x64 of the BIOS Segment can be used, and the DOS interrupt 0x10 with AH = 0x10 is also supported. The following code uses the latter.

Example Code: Setting the Mode

```
BYTE Radeon_SetMode (WORD xres, WORD yres, BYTE bpp)
{
    union REGS r;
    CRTCIInfoBlock *mtable;
    BYTE retval = 0;
    WORD test = 0;

    if (Radeon_AdapterInfo.FLAGS & Radeon_USE_BIOS)
    {
        memset (&r, 0, sizeof (r));
        r.w.ax = 0xA000;    // Function 00h: Set Mode.

        // Determine requested resolution.
        if (xres == 320)
        {
            switch (yres)
            {
                case 200:    r.h.ch = 0xE2;
                            break;
                case 240:    r.h.ch = 0xE3;
                            break;
                default:     // Unsupported X resolution!
                            return (0);
                            break;
            }
        }
        else if (xres == 640)
        {
            switch (yres)
            {
                case 350:    r.h.ch = 0xE6;
                            break;
                case 400:    r.h.ch = 0xE1;
                            break;
                case 480:    r.h.ch = 0x12;
                            break;
                default:     // Unsupported Y resolution!
                            return (0);
                            break;
            }
        }
    }
}
```

```
else
{
    switch (xres)
    {
        case 400:    r.h.ch = 0xE5;
                    yres = 300;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        case 512:    r.h.ch = 0xE4;
                    yres = 384;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        case 640:    r.h.ch = 0x12;
                    break;

        case 800:    r.h.ch = 0x6A;
                    yres = 600;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        case 1024:   r.h.ch = 0x55;
                    yres = 768;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        case 1280:   r.h.ch = 0x83;
                    yres = 1024;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        case 1600:   r.h.ch = 0x84;
                    yres = 1200;
                    Radeon_AdapterInfo.xres = xres;
                    Radeon_AdapterInfo.yres = yres;
                    break;

        default:     // Unsupported X resolution!
                    return (0);
                    break;
    }
} // if/else

Radeon_AdapterInfo.xres = xres;
Radeon_AdapterInfo.yres = yres;

// Determine requested pixel depth
switch (bpp)
{
    case 8:         r.h.cl = 0x02;
                    break;
```

```
        case 15:    r.h.cl = 0x03;
                   break;
        case 16:    r.h.cl = 0x04;
                   break;
        case 32:    r.h.cl = 0x06;
                   break;
        default:    // Unsupported pixel depth!
                   return (0);
                   break;
    } // switch

    Radeon_AdapterInfo.bpp = bpp;
    Radeon_AdapterInfo.pitch = xres/8;
    Radeon_AdapterInfo.pitch = ((xres * Radeon_AdapterInfo.bytepp +
0x3f)
                               & ~(0x3f)) / 64;
    if (xres == 800 && bpp == 8)
        Radeon_AdapterInfo.pitch *=2;

    // Call the BIOS to set the mode.
    int386 (0x10, &r, &r);
    if (r.h.ah)
    {
        return (0);    // Error setting mode.
    }
    else
    {
        return (1);
    } // if
}
else
{
    Radeon_AdapterInfo.xres = xres;
    Radeon_AdapterInfo.yres = yres;
    Radeon_AdapterInfo.bpp = bpp;
    Radeon_AdapterInfo.pitch = xres/8;
    Radeon_AdapterInfo.pitch = xres * Radeon_AdapterInfo.bytepp / 64;
    Radeon_AdapterInfo.pitch = ((xres * Radeon_AdapterInfo.bytepp +
0x3f)
                               & ~(0x3f)) / 64;

    switch (Radeon_AdapterInfo.xres)
    {
        case 320:    mtable = &mode320_60;
                     break;
        case 400:    mtable = &mode400_75;
                     break;
        case 512:    mtable = &mode512;
                     break;
```

```
        case 640:    mtable = &mode640_60;
                    break;
        case 720:    mtable = &mode720_60;
                    break;
        case 800:    mtable = &mode800_60;
                    break;
        case 848:    mtable = &mode848_88;
                    break;
        case 864:    mtable = &mode864_60;
                    break;
        case 1024:   mtable = &mode1024_60;
                    break;
        case 1152:   mtable = &mode1152_60;
                    break;
        case 1280:   mtable = &mode1280_60;
                    break;
        case 1600:   mtable = &mode1600_60;
                    break;
        case 1920:   mtable = &mode1920_60;
                    break;
        default:     mtable = &mode640_60;
                    break;
    }

    test = (WORD)Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    retval = Radeon_SetModeNB (xres, yres, test, mtable);

    return (retval);
} // else

} // Radeon_SetMode
```

3.5.1 Passing a CRT Parameter Table to Set a Display Mode

While using the BIOS to set a display mode is straight forward, it does have some limitations. The only modes that can be set are:

- Those that are directly supported by the BIOS.
- Those whose refresh rate that is supported by the BIOS, which is typically 60 Hz for most modes.

In cases where a custom mode or refresh rate is required, the BIOS allows for passing a CRT parameter table, from which the BIOS will derive the appropriate CRT values, and program the CRT accordingly. For a full description of the structure of the CRT Parameter table, refer to the Video BIOS appendix.

Example Code: Setting the display mode

```
BYTE Radeon_SetDisplayModeFromTable (CRTParameterTable table)
{
    union REGS r;
    DWORD psize, segment, selector;
    char *data;
    int x, y;

    // We need to allocate some memory for the mode table, so we can
    // pass the BIOS a real mode address.
    psize = 2; // require 28 bytes of memory.
    if (DPMI_allocdosmem( psize, &segment, &selector) == 0)
    {
        /* can't allocate memory for mode table, shut down */
        Radeon_ShutDown ();
        printf ("\nUnable to allocate system memory for mode table!");
        exit (1);
    }

    memset (&r, 0, sizeof (r));
    r.w.ax = 0xA000;                // Function 00h: Set Mode.
    r.w.di = 0x0000;                // Set CRT only

    // Set DX equal to the segment that was allocated.
    // The offset (BX) will be 0.
    r.w.dx = segment;
    r.w.bx = 0;
    data = (char *) (segment << 4);

    // Copy the CRT Parameter Table data to the location pointed
    // to by DX:BX
    memcpy (data, &table, sizeof(CRTParameterTable));

    // Set BIOS to load resolution from specified table and set depth
    // to the requested pixel depth.
    r.w.cx = 0x8100 | Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    // Call the BIOS to set the mode.
    int386 (0x10, &r, &r);
    if (r.h.ah)
    {
        // We have encountered an error setting the display mode.
        return (0);
    }
    else
    {
        // Success!
        return (1);
    }
}
```

```
} // Radeon_SetDisplayModeFromTable ()...
```

3.5.2 Manually Setting a Display Mode

In cases where the video BIOS cannot be executed, the display mode must be manually programmed. This includes calculating the required CRTC and PLL values, and programming the appropriate registers.

Programming the CRTC Registers

To set up the RADEON for a display mode, the CRTC registers must be programmed so that they correspond with the requested display mode dimensions.

- While programming the CRTC registers, it is strongly recommended to disable the display. This can be accomplished by setting the following bits in register

CRTC_EXT_CNTL:

- **CRTC_HSYNC_DIS**
- **CRTC_VSYNC_DIS**
- **CRTC_DISPLAY_DIS**

After setting the display mode, enable the display.

First, start by clearing some “common” registers that, if active, may interfere with the CRTC settings. These registers are:

- **OVR_CLR** - disable the overscan color.
- **OVR_WID_LEFT_RIGHT** - no overscan border.
- **OVR_WID_TOP_BOTTOM** - no overscan border.
- **OV0_SCALE_CNTL** - disable the overlay.
- **SUBPIC_CNTL** - disable subpicture decoding (for MPEG/DVD).
- **VIPH_CONTROL** - disable VIP transfers.
- **I2C_CNTL_1** - disable the I2C bus.
- **GEN_INT_CNTL** - disable interrupts.
- **CAP0_TRIG_CNTL** - disable capture buffer 0.

The next step is to program the following CRTC related registers:

- **CRTC_GEN_CNTL** - this register is used to:
 - Enable the extended display mode (accelerator).
 - Enable the CRTC.
 - Disable the cursor.
 - Set the pixel width (i.e. the color depth).
 - Disable composite sync.
- **CRTC_EXT_CNTL**
 - Perform a READ-MODIFY-WRITE to preserve some power-up settings:
 - **CRTC_HSYNC_DIS**
 - **CRTC_VSYNC_DIS**
 - **CRTC_DISPLAY_DIS**

In addition, enable **VGA_ATI_LINEAR**, and **VGA_XCRT_CNT_EN**.

- **DAC_CNTL**
 - Perform a READ-MODIFY-WRITE to preserve the lower 3 bits (and 0x7).
 - Set **DAC_8BIT_EN**, disable **DAC_TVO_EN** and **DAC_VGA_ADR_EN**.
 - Set the **DAC_MASK** to 0xFF (enable all palette index bits).
- **CRTC_H_TOTAL_DISP** - set following two fields in this register:
 - **CRTC_H_DISP** contains the amount of visible horizontal 'characters'. This value is determined by taking the visible pixels (x-resolution), dividing by 8 (8 pixels = 1 'character'), then subtracting 1. This field occupies bits [0:8] of this register.
 - **CRTC_H_TOTAL** contains the total horizontal 'characters', which includes overscan right, front porch, sync width, back porch and overscan left. The value for this field is expressed in 'characters' as well, then subtract 1. **CRTC_H_TOTAL** resides in bits [16:23] of **CRTC_H_TOTAL_DISP**.
- **CRTC_H_SYNC_STRT_WID** - the starting horizontal position and width, as well as the sync polarity are written to this register:
 - Bits [0:2] of **CRTC_H_SYNC_STRT_PIX** allows for pixel accurate starting

positioning by delaying the start (in pixels) within the character value of bits [3:11] contained in **CRTC_H_SYNC_STRT_CHAR**.

- The horizontal sync start is typically part of the parameter table that is passed to the mode setting routine.
- Bits [16:21] of **CRTC_H_SYNC_WID** is calculated by taking the horizontal sync end subtracted by the horizontal sync start, then converting that to characters (divide by 8).
- Bit [23] of **CRTC_H_SYNC_POL** is '0' for positive sync, and '1' for negative sync.
- **CRTC_V_TOTAL_DISP** - set following two fields in this register:
 - Bits [16:26] of **CRTC_V_DISP** determines the amount of visible lines (not including overscan).
 - Bits [0:10] of **CRTC_V_TOTAL** is the vertical line total. This includes the display height, overscan bottom, front porch, sync width, back porch and overscan top.
- **CRTC_V_SYNC_STRT_WID** - set the following three fields for this register:
 - Bits [0:10] of **CRTC_V_SYNC_STRT** is the sum of display height, overscan bottom and front porch.
 - **CRTC_V_SYNC_WID** is the vertical sync width. This is typically dependent on the monitor. However, most modern monitors have a fair tolerance for this value.
 - **CRTC_V_SYNC_POL** is the polarity of the vertical sync. '0' is positive, and '1' is negative.
- **CRTC_OFFSET**
This register determines the start of displayable video memory. In most cases, this will be set to '0'. To set up some kind of virtual desktop, a non-zero value may be appropriate for this value.
- **CRTC_OFFSET_CNTL**
Clear this register. There are various functions related to the **CRTC_OFFSET** that can be enabled in this register. For the purposes of setting a display mode, initialize the value (i.e. set it to '0'). For more details, refer to the RADEON Register Reference manual.
- **CRTC_PITCH**
The display pitch is set in this register. Bits [0:9] hold the pitch value, expressed in pixels*8 (characters). For 24-bpp format modes, the CRTC uses pixels*8 for the

pitch, but the rendering engine uses bytes*8 for the pitch.

3.5.3 Calculating the PLL Register Values

To manually set a display mode, first determine the following parameters:

- Dot-clock reference frequency.
- Dot-clock reference divider.
- Minimum and maximum PLL output values for the dot clock for the installed adapter.

These values are used for reference to obtain the necessary CRT timing parameters for the requested display mode.

A RADEON-based graphics adapter may use one of several base or reference frequencies, depending on the features supported by the installed card. Common values for the reference frequencies are:

- 29.50 MHz
- 28.63 MHz
- 14.32 MHz

The RADEON's BIOS uses these values expressed in kHz/10. Therefore, 29.50 MHz would actually be 2950. To reliably determine this frequency, extract the value from the BIOS by looking at the appropriate tables within the BIOS.

- The BIOS header is located at offset 0x48 from the BIOS segment address.
- The PLL information block pointer is located at offset 0x30 to 0x31 within the BIOS header.

BIOS Header Pointer = BIOS segment address + 0x48

PLL Information Block Pointer = BIOS header pointer + 0x30

The dot clock reference frequency is located at offset 0x0E within the PLL information block.

Dot Clock Reference Frequency = PLL Information Block + 0x0E

The dot clock reference frequency is located offset 0x0E (word) within the PLL information block. Use the reference frequency to determine what post and feedback

divider values will be required to provide the proper dot clock frequency for a given display mode.

The value of the reference feedback divider is also required. This value is found at offset 0x10 (word) within the PLL information block.

Dot Clock Reference Divider = PLL Information Block + 0x10

Obtain the minimum and maximum output frequencies for the PLL.

Make sure that the desired output frequency can be provided given these values. The maximum post-divider value is 12, so the desired output frequency multiplied by 12 must be equal to or greater than the minimum PLL output frequency.

Also, the desired output frequency cannot be greater than the maximum PLL output frequency. The minimum dot clock PLL output frequency is located offset 0x18 (dword).

Dot Clock Minimum PLL Output Frequency = PLL Information Block + 0x12

The maximum dot clock PLL output frequency is located at offset 0x16 (dword) within the PLL information block.

Dot Clock Maximum PLL Output Frequency = PLL Information Block + 0x16

Regarding the output frequencies, two different output frequencies are discussed within this section. The ***Requested Output Frequency*** is the dot clock frequency for a given display mode.

For example, for a 640x480, 60 Hz refresh, the requested output frequency is 25.18 MHz.

The ***PLL Output Frequency*** is the frequency that the PLL will output, which is then divided down by the feedback divider. It is important to distinguish these two output frequencies. They are not the same and in the majority of cases, they are not equal. The Requested Output Frequency (dot clock) is in fact a result of the PLL Output Frequency divided down by the feedback divider.

3.5.4 Determining the Post and Feedback Dividers

The internal clock generator uses a PLL feedback system to produce the desired frequency output according to the following equation:

**Dot Clock Frequency =
(Reference Frequency * Feedback Divider) / (Reference Divider * Post Divider)**

The Feedback Divider must be from 128 to 255 inclusive, and the Post Divider can be one of 1, 2, 3, 4, 6, 8, or 12.

To easily determine the post divider, multiply the required dot clock frequency by one of the possible post divider values (i.e. 1, 2, 3, 4, 6, 8, 12) until it falls between the minimum and maximum PLL output frequencies. Therefore:

PLL Output Frequency = Post Divider * Dot Clock Frequency

After calculating the post divider and PLL output frequency, use the following equation to determine the feedback divider:

**Feedback Divider =
Post Divider * Reference Divider * PLL Output Frequency) / (Reference Frequency)**

At this point, all the required values to program the PLL to obtain the dot clock frequency required for a given display mode are known.

Example Code: Finding the post and feedback divider for a given dot clock frequency

```
void Radeon_PLLGetDividers (WORD Frequency)
{
    DWORD FeedbackDivider;           // Feedback divider value
    DWORD OutputFrequency;           // Desired output frequency
    BYTE PostDivider = 0;             // Post Divider

    for VCLK

        //
        // The internal clock generator uses a PLL feedback system to produce
        // the desired frequency output according to the following equation:
        //
        // OutputFrequency = (REF_FREQ * FeedbackDivider) / (REF_DIVIDER *
        PostDivider)
        //
        // Where REF_FREQ is the reference crystal frequency, FeedbackDivider is
        the
        // feedback divider (from 128 to 255 inclusive), and REF_DIVIDER is the
        reference
        // frequency divider. PostDivider is the post-divider value (1, 2, 3,
        4, 6, 8, or 12).
        //
        // The required feedback divider can be calculated as:
        //
        // FeedbackDivider = (PostDivider * REF_DIVIDER * OutputFrequency) /
        REF_FREQ
        //

    if (Frequency > PLL_BLOCK.PCLK_max_freq)
```

```
{
    Frequency = (WORD)PLL_BLOCK.PCLK_max_freq;
}

if (Frequency * 12 < PLL_BLOCK.PCLK_min_freq)
{
    Frequency = (WORD)PLL_BLOCK.PCLK_min_freq / 12;
}

OutputFrequency = 1 * Frequency;

if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
(OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
{
    PostDivider = 1;
    goto _PLLGetDividers_OK;
}

OutputFrequency = 2 * Frequency;

if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
(OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
{
    PostDivider = 2;
    goto _PLLGetDividers_OK;
}

OutputFrequency = 3 * Frequency;

if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
(OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
{
    PostDivider = 3;
    goto _PLLGetDividers_OK;
}

OutputFrequency = 4 * Frequency;

if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
(OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
{
    PostDivider = 4;
    goto _PLLGetDividers_OK;
}

OutputFrequency = 6 * Frequency;

if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
(OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
{
```

```
        PostDivider = 6;
        goto _PLLGetDividers_OK;
    }

    OutputFrequency = 8 * Frequency;

    if ((OutputFrequency >= PLL_BLOCK.PCLK_min_freq) &&
        (OutputFrequency <= PLL_BLOCK.PCLK_max_freq))
    {
        PostDivider = 8;
        goto _PLLGetDividers_OK;
    }

    OutputFrequency = 12 * Frequency;
    PostDivider = 12;

_PLLGetDividers_OK:

    //
    // OutputFrequency now contains a value which the PLL is capable of
    // generating.
    // Find the feedback divider needed to produce this frequency.
    //

    FeedbackDivider = RoundDiv (PLL_BLOCK.PCLK_ref_divider *
        OutputFrequency,
        PLL_BLOCK.PCLK_ref_freq);

    PLL_INFO.fb_div    = (WORD)FeedbackDivider;
    PLL_INFO.post_div  = (BYTE)PostDivider;

    return;
} // Radeon_PLLGetDividers()
```

We now have the necessary values to program the PLL to set the necessary pixel clock frequency. The dot clock uses PLL 3 on the RADEON. See the source code file “r128pll.c” for the steps required to program the actual PLL registers.

3.6 Step 4: Initialize the GUI Engine

After setting a display mode, this step involves using the acceleration capabilities of the RADEON to initialize the GUI engine. This consists of setting up the GUI to a known drawing context. To initialize the GUI engine, follow these steps:

1. Set the destination, source and default offset registers to equal the memory aperture address.
2. Program the engine pitch registers
3. Observe the following characteristics of the engine pitch:
 - It must be divisible by eight.
 - The value written to the pitch registers is expressed in bytes per line, not pixels.
4. Program the source, destination and default pitch registers to the appropriate values for the current display mode.
5. To enable a drawing area on the visible screen, program the scissors registers.
 - Generally, when initially configuring the GUI, program the scissors registers to the maximum values allowable, so that any part of display memory can be used as a source, and also to allow drawing anywhere in memory if needed.
 - The scissors can be set to the screen co-ordinates if required, however be careful when using off screen memory to store bitmaps and other data. The source scissors registers must be set to the appropriate dimensions in this case.

The RADEON contains the register **DP_GUI_MASTER_CNTL**, which can be used to set up the majority of the default drawing context registers in a single register write. A breakdown of the register and it's various fields follow:

Table 3-3 DP_GUI_MASTER_CNTL

Field Name	Purpose
GMC_SRC_PITCH_OFFSET_CNTL	This field allows setting the SRC_OFFSET = DEFAULT_OFFSET and SRC_PITCH = DEFAULT_PITCH (0) -OR- leave alone (1).
GMC_DST_PITCH_OFFSET_CNTL	Same functionality as previous field, only relating to the destination pitch and offset values.
GMC_SRC_CLIPPING	Determines whether the source scissor registers will equal the default scissor registers, or will not use the default value.
GMC_DST_CLIPPING	Same functionality as previous field, only relating to the destination scissor register default values.
GMC_BRUSH_DATATYPE	Determines what brush type will be used for drawing operations. Typically, a solid color brush would be used. Consult the register reference for the possible values for this field.

Table 3-3 DP_GUI_MASTER_CNTL

Field Name	Purpose
GMC_DST_DATATYPE	This field represents the destination pixel depth/format. Pixel depths of 8 to 32 are supported, as well as various YUV formats. Generally, the value for this field will equal the display-mode pixel depth. Consult the register reference for the complete list of available values.
GMC_SRC_DATATYPE	The source expansion value is initialized here. Values are: 0 = monochrome (source is expanded to foreground and background). 1 = for source expanded to foreground, and the background is left alone. 3 = color of the pixel is used (the pixel type is the same as the destination).
GMC_BYTE_PIX_ORDER	Allows for pixel ordering with respect to most significant byte (MSB) and least significant byte (LSB). Default = 0 (MSB->LSB).
GMC_DEFAULT_SEL	
GMC_ROP3	The default raster operation is set here. The RADEON supports all 256 ROPs as per the MS Win3.1 DDK. See appendix D regarding available ROP values.
GMC_DP_SRC_SOURCE	Determines the pixel source for source data. Possible sources are display memory and the host data registers.
GMC_SRC_DATATYPE2	0 = monochrome (source is expanded to foreground and background).
GMC_CLR_CMP_FCN_DIS	Enables or disables the color compare function.
GMC_WR_MSK_DIS	Enables or disables the write mask.

6. Initialize (i.e. clear out) the following additional registers (specifically the line drawing registers):
 - **DST_LINE_START**
 - **DST_LINE_END**
7. Set the desired default color values for both brush and source data, in the following registers:
 - **DP_BRUSH_FRGD_CLR**
 - **DP_BRUSH_BKGD_CLR**

- DP_SRC_FRGD_CLR
- DP_SRC_FRGD_CLR

Typically, the foreground color would be white (i.e. 0xFFFFFFFF) and the background color would be black (i.e. 0x00000000).

Example Code: Initializing the GUI engine

```
void Radeon_InitEngine (void)
{
    DWORD temp;

    // insure 3D is disabled
    regw(RB3D_CNTL, 0);

    // do an Engine Reset, just in case it's hung right now
    Radeon_ResetEngine ();

    // setup engine offset registers
    Radeon_WaitForFifo (1);

    // setup engine pitch registers
    temp = regr(DEFAULT_PITCH_OFFSET);
    regw (DEFAULT_PITCH_OFFSET, (temp & 0xC0000000)
        | (Radeon_AdapterInfo.pitch << 22));

    // set scissors to maximum size
    Radeon_WaitForFifo (1);
    regw (DEFAULT_SC_BOTTOM_RIGHT, (0x1FFF << 16) | 0x1FFF);

    // Set the drawing controls registers.
    Radeon_WaitForFifo (1);
    temp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    regw (DP_GUI_MASTER_CNTL, GMC_SRC_PITCH_OFFSET_DEFAULT |
        GMC_DST_PITCH_OFFSET_DEFAULT |
        GMC_SRC_CLIP_DEFAULT |
        GMC_DST_CLIP_DEFAULT |
        GMC_BRUSH_SOLIDCOLOR |
        (temp << 8) |
        GMC_SRC_DSTCOLOR |
        GMC_BYTE_ORDER_MSB_TO_LSB |
        ROP3_PATCOPY |
        GMC_DP_SRC_RECT |
        GMC_DST_CLR_CMP_FCN_CLEAR |
        GMC_WRITE_MASK_SET);
```

```
Radeon_WaitForFifo (7);

// Clear the line drawing registers.
regw (DST_LINE_START,0);
regw (DST_LINE_END,0);

// set brush color registers
regw (DP_BRUSH_FRGD_CLR, 0xFFFFFFFF);
regw (DP_BRUSH_BKGD_CLR, 0x00000000);

// set source color registers
regw (DP_SRC_FRGD_CLR, 0xFFFFFFFF);
regw (DP_SRC_BKGD_CLR, 0x00000000);

// default write mask
regw (DP_WRITE_MSK, 0xFFFFFFFF);

// Wait for all the writes to be completed before returning
Radeon_WaitForIdle ();

return;
} // Radeon_InitEngine ()
```

4.1 Scope

This chapter describes how to program the RADEON to perform drawing operations. This chapter also discusses some aspects of programming the RADEON using the Programmed I/O (PIO) drawing mode. The following topics are covered:

- Engine command queue maintenance
 - Engine Drawing Operations
- Rectangle Drawing
 - Bit Block Transfers
 - Line Drawing
 - Pattern Drawing
 - Compare Functionality
 - Monochrome Expansion
- Handling the Hardware Cursor

4.2 Engine Command Queue Maintenance

The command queue buffers the “FIFOed” register writes and reads to the engine. Generally, “FIFOed” registers are involved in the drawing operations. The file `REGDEF.H` specifically outlines the registers are “FIFOed”, and identifies the registers that can be read directly.

For the RADEON, the command queue consists of 64 DWORD entries. The **CMDFIFO_AVAIL@RBBM_STATUS** register field represents how many command queue entries are free at a given point in time. Before reading or writing a register that is “FIFOed”, check for the availability of a free queue entry. Once an entry is available, submit the read/write operation to the queue.

The following code polls the **GUI_STAT** register to ensure that the requested amount of FIFO entries are available. In addition, provisions are made for time-out errors, where the engine cannot provide a free queue entry (e.g. this may occur when the engine has locked up or hung to due improper programming).

Example Code: Waiting for the FIFO

```
void Radeon_WaitForFifo (DWORD entries)
{
    WORD starttick, endtick;

    starttick = *((WORD *) (DOS_TICK_ADDRESS));
    endtick = starttick;
    while ((reg (RBBM_STATUS) & RBBM_STATUS__CMDFIFO_AVAIL_MASK) < entries)
    {
        endtick = *((WORD *) (DOS_TICK_ADDRESS));
        if (abs (endtick - starttick) > FIFO_TIMEOUT)
        {
            Radeon_ResetEngine ();
        }
    }
    return;
}
```

In addition, some situations require the engine to become idle. For example, an idle engine is required in the following cases:

- Reading a status register.
- Getting a true status value.

In order to determine that the engine is idling, the following two conditions must be satisfied:

- There must be 64 free command queue entries.
- The engine (GUI) must be inactive.

NOTE: An empty command queue DOES NOT imply an idle engine. Code that satisfies these two conditions would do the following:

1. Poll (i.e. continually check) **CMDFIFO_AVAIL@RBBM_STATUS** until its contents equal 64.
2. Then, poll **CMDFIFO_AVAIL@RBBM_STATUS** until its contents equal '0'.

Under some conditions, the GUI engine may become unstable or lock. If the engine become unstable or locks, reset the engine. The code below handles locked up engines by checking for a time-out condition. The code calls the appropriate function to handle an engine time-out, thus allowing the program to continue to run after the engine has been reset.

Example Code: Waiting for idle

```
void Radeon_WaitForIdle (void)
{
    WORD starttick, endtick;

    // Insure FIFO is empty before waiting for engine idle.
    Radeon_WaitForFifo (64);

    starttick = *((WORD *) (DOS_TICK_ADDRESS));
    endtick = starttick;
    while ((regr (RBBM_STATUS) & GUI_ACTIVE) != ENGINE_IDLE)
    {
        endtick = *((WORD *) (DOS_TICK_ADDRESS));
        if (abs (endtick - starttick) > IDLE_TIMEOUT)
        {
            // We should reset the engine at this point.
            Radeon_ResetEngine ();
        } // if
    } // while

    // flush the pixel cache
    Radeon_FlushPixelCache ();

    return;
} // Radeon_WaitForIdle ()
```

4.3 Programmed I/O Drawing Operations

This section describes how to draw rectangles and lines.

- Methods for drawing rectangles:
 - Bit Block Transfer
 - BitBlt - Bit Block Transfer
 - Transparent BitBlt (Bit Block) Transfer
- Methods for drawing lines:
 - Drawing Patterned Lines
 - Monochrome Expansion

4.3.1 Drawing Rectangles

To draw a simple, solid-colored rectangle, the RADEON uses the following steps:

1. Set up the desired drawing context.

2. Program the destination registers to the desired values.

To set up the context for drawing, determine the screen location where to draw the rectangle.

For a clipped rectangle, program the scissor registers to the required parameters.

For a solid-filled rectangle, our data type is the current pixel depth, a solid brush is used, and the raster operation is a source copy.

The following code demonstrates how to draw a solid color rectangle. It is assumed that prior to calling this function, the engine has been initialized.

Example Code: Drawing a rectangle

```
void Radeon_DrawRectangle (DWORD x, DWORD y, DWORD width, DWORD height,
                          DWORD colour)
{
    DWORD save_dp_datatype, bppvalue;

    Radeon_WaitForFifo (6);

    // Save the current DP_DATATYPE register value
    save_dp_datatype = regr (DP_DATATYPE);

    bppvalue = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);
    regw (DP_DATATYPE, (bppvalue | BRUSH_SOLIDCOLOR | ROP3_SRCOPY));
    regw (DP_BRUSH_FRGD_CLR, Radeon_GetColourCode (colour));
    regw (DST_Y_X, (y << DST_Y_X_DST_Y_SHIFT) | x);
    regw (DST_WIDTH_HEIGHT, (width << DST_WIDTH_HEIGHT_DST_WIDTH_SHIFT)
        | height);

    // restore the DP_DATATYPE register
    regw (DP_DATATYPE, save_dp_datatype);

    return;
} // Radeon_DrawRectangle
```

Bit Block Transfer

One of the most widely used drawing features is the bit block transfer. This command transfers a bitmap or block of data from one area of video memory to another.

To transfer data within frame buffer from one location to another, and to transfer data from system memory to frame buffer, the RADEON uses hardware support.

Source - the location where the data is taken from.

Destination - the location where the data is transferred.

The size of the data transfer determines the size of a rectangular area on the screen. In this sense, *Block Transfer* means copying a group of pixels from one place to another with some manipulation of the pixels.

The following types of pixels are involved in a block transfer:

- Source
- Destination
- Brush pattern

The resulting destination is the combination of one, two, or all of three components. In this sense, all three are considered components of the source before the operation that combines them, and only the result of the combination is considered as the destination.

For block data transfers, specify the following:

- Location
- Dimension of the source
- Destination
- Setup parameters

The following types of data transfer can occur:

- ***BitBlt***
This is also called BitBlt or *source copy*. The source content is copied to the destination without any changes to its dimensions.
- ***Transparent BitBlt***
This transfer is similar to BitBlt except that it makes the background image at the destination shown through the image copied from the source (i.e. as if the source image is transparent).

BitBlt - Bit Block Transfer

This operation transfers pixels from a source rectangle to a destination. The dimension of the transferred rectangle remains the same as the source. The transfer is controlled by a ternary-raster operation code that specifies how the pixels from the source and the brush pattern are mixed with those of the destination to form the final pixels at the destination.

In addition to normal data transfer (i.e. the data transfer that does not change the format of the data taken from the source before placing it at the destination), the RADEON supports monochrome to color expansion when transferring a monochrome bitmap to the CRT

screen. For color expansion, specify the foreground and background colors for the bitmap. The RADEON will convert the white bit (1) to the foreground color of the corresponding pixel and the black bit (0) to the background color.

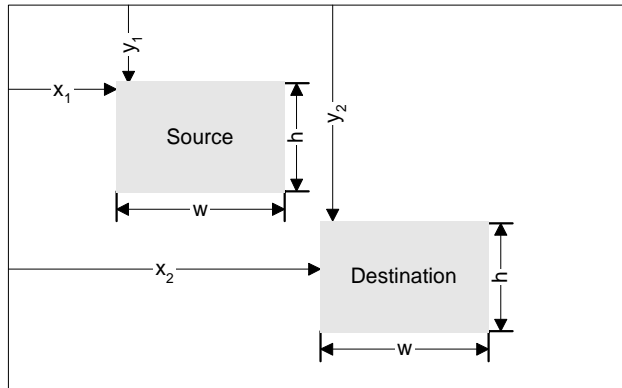


Figure 4-1. BitBlit - Bit Block Transfer Copying an Image from Source to Destination

Example Code: Copying an image from a source to a destination

```
void Radeon_Blt (WORD srcx, WORD srcy, WORD src_width, WORD src_height,
                WORD dst_x, WORD dst_y)
{
    WORD bpp;
    DWORD temp;
    img_info IMG_DATA;

    // clear the screen to yellow to start.
    Radeon_ClearScreen (YELLOW);

    // Set up the source data co-ordinates.
    srcx = 0;
    srcy = Radeon_AdapterInfo.yres;
    bpp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    // Load the source image file into offscreen memory.
    IMG_DATA = Load_Image (TRAJECTORY_RECTANGULAR, bpp, srcx, srcy);

    Radeon_WaitForFifo (4);

    // Tell the engine where the source data resides.
    regw (SRC_Y_X, (srcy << SRC_Y_X_SRC_Y_SHIFT ) | srcx);

    // set the mix to a ROP3_SRCCOPY, and a rectangular source
    regw (DP_MIX, ROP3_SRCCOPY | DP_SRC_RECT);
}
```



```

// Set the drawing direction to left->right, top->bottom
regw (DP_CNTL, DST_X_LEFT_TO_RIGHT | DST_Y_TOP_TO_BOTTOM);

// make the src pixel = dst pixel, in terms of pixel depth
temp = regw (DP_DATATYPE);
regw (DP_DATATYPE, temp | SRC_DSTCOLOR);

// blt image anywhere in the screen.

// calculate random destination x and y values
dstx = (rand()%(Radeon_AdapterInfo.xres - IMG_DATA.width));
dsty = (rand()%(Radeon_AdapterInfo.yres - IMG_DATA.height));

Radeon_WaitForFifo (2);
regw (DST_Y_X, (dsty << DST_Y_X_DST_Y_SHIFT ) | dstx);

// this is the blt initiator.
regw (DST_HEIGHT_WIDTH, IMG_DATA.width
      (IMG_DATA.height << DST_HEIGHT_WIDTH_DST_HEIGHT_SHIFT ));

// Batch command to restore old mode.
Radeon_ShutDown ();
Radeon_PrintInfoStruct();

exit (0); // No errors.
} // Radeon_Blt

```

Transparent BitBlt (Bit Block) Transfer

This operation conditionally copies pixels from the source to the destination with reference to a designated (reference) color (e.g. the background color).

If the color of a pixel is equal to (or not equal to according to the comparison criterion) the designated color, the pixel will not be copied to the destination. This operation filters out unwanted color from the source.

This operation is useful for:

- Copying odd-shaped objects onto a background with patterns (e.g. games).
- Making objects look transparent.

Since a transparent BitBlt operation is more complicated than a BitBlt operation, some terminology needs to be clarified before proceeding with an example.

For this operation, *source* means a color pixel that may come from one of the following sources:

- One of foreground or background colors used to expand a mono bitmap to a color bitmap.

- A color pixel from either the frame buffer or the host memory.
- A color pixel of a color pattern (brush).

The source pixel may be combined with the destination pixel according to a given raster operation code (e.g. AND operation) resulting in the *combined source pixel*.

To prevent certain colors of combined source pixels from being written to the destination, two color comparators are used for deciding whether to write a combined source pixel to the destination or to keep the original destination pixel. The comparators compare the source and destination pixels respectively against their reference colors (the source and destination references), and decide whether the combined source pixel can be written to the destination. The following lists the strategies for making such a decision:

Table 4-1 Source Comparator

Decision Code	Description
0	Combined pixels are always written to the destination, i.e. no comparison is performed.
1	No combined pixel is written to the destination, i.e. the destination pixel is unchanged.
4	The combined pixel is written to the destination if the color of the source pixel is equal to its reference color. Otherwise, the destination pixel is unchanged.
5	The combined pixel is written to the destination if the color of the source pixel is NOT equal to its reference color. Otherwise, the destination pixel is unchanged.
7	Only the source pixels whose color is equal to the reference color will be XORed with the foreground color of the source bitmap, and then written to the destination. That is, $\text{destPixel} = \text{srcPixel} \text{ XOR } \text{foreground Color}$ if srcPixel is equal to the foreground color of the source bitmap. This is sometimes referred to as flipping.

Table 4-2 Destination Comparator

Decision Code	Description
0	Combined pixels are always written to the destination, i.e. no comparison is performed.
1	No combined pixel is written to the destination, i.e. the destination pixel is unchanged.

Table 4-2 Destination Comparator (Continued)

Decision Code	Description
4	The destination is unchanged if the color of the destination pixel is equal to its reference color. Otherwise, the combined source pixel are written to the destination.
5	The destination is unchanged if the color of the destination pixel is NOT equal to its reference color. Otherwise, the combined source pixel are written to the destination.

The two tables give the decision strategy whenever either of the comparators is enabled.

- If both comparators are enabled, the final decision will depend on the agreement between the two decisions made separately.
- If both comparators decide that the combined source pixel should be written to the destination, the destination will be updated with the pixel (otherwise, the original destination pixel is preserved).

Example Code: Transparent BitBlt Operation

```
void Radeon_TransparentBlt (void)
{
    DWORD temp, save_dp_mix, save_dp_cntl, save_dp_datatype;

    Radeon_WaitForFifo (3);
    save_dp_mix = regr (DP_MIX);
    save_dp_cntl = regr (DP_CNTL);
    save_dp_datatype = regr (DP_DATATYPE);

    Radeon_WaitForFifo (3);
    // set up the engine trajectory registers for a blt.
    regw (DP_MIX, ROP3_SRCCOPY | DP_SRC_RECT);
    regw (DP_CNTL, DST_X_LEFT_TO_RIGHT | DST_Y_TOP_TO_BOTTOM);
    temp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);
    regw (DP_DATATYPE, temp | SRC_DSTCOLOR | BRUSH_SOLIDCOLOR);

    Radeon_WaitForFifo (4);
    // set up the transparency function in the colour compare circuitry
    regw (CLR_CMP_CLR_DST, Radeon_GetColourCode (TBLT.dst_clr));
    regw (CLR_CMP_CLR_SRC, Radeon_GetColourCode (TBLT.src_clr));
    regw (CLR_CMP_MSK, 0xFFFFFFFF);
    regw (CLR_CMP_CNTL, (TBLT.cmp_src << CLR_CMP_CNTL_CLR_CMP_SRC_SHIFT)
    | (TBLT.dst_cmp_fcnc << CLR_CMP_CNTL_CLR_CMP_FCNC_DST_SHIFT)
    | TBLT.src_cmp_fcnc);

    // Set up source and destination x and y values
    Radeon_WaitForFifo (3);
```

```

    regw (SRC_Y_X, (TBLT.src_y << SRC_Y_X_SRC_Y_SHIFT) | TBLT.src_x);
    regw (DST_Y_X, (TBLT.dst_y << DST_Y_X_DST_Y_SHIFT) | TBLT.dst_x);
    regw (DST_HEIGHT_WIDTH,
    (TBLT.src_height << DST_HEIGHT_WIDTH_DST_HEIGHT_SHIFT)
    | TBLT.src_width);

    // Restore the modified registers
    Radeon_WaitForFifo (3);
    regw (DP_MIX, save_dp_mix);
    regw (DP_CNTL, save_dp_cntl);
    regw (DP_DATATYPE, save_dp_datatype);

    return;

} // Radeon_TransparentBlt ()

```

4.3.2 Drawing Lines

To draw a simple, solid-colored line, RADEON uses the following steps:

1. Set up the desired drawing context.
2. Program the destination registers to the desired values.

To set up the context for drawing, determine the screen location where to draw the line.

The following code demonstrates how to draw a solid color line. It is assumed that prior to calling this function, the engine has been initialized.

Example Code: Accelerated line drawing

```

void Radeon_DrawLine (WORD x1, WORD y1, WORD x2, WORD y2, DWORD colour)
{
    int dx, dy;
    int small, large;
    int x_dir, y_dir, y_major;
    int err, inc, dec;
    DWORD save_dp_cntl, save_dp_datatype, bppvalue;

    // Determine x & y deltas and x & y direction bits.
    if (x1 < x2)
    {
        x_dir = 1 << 31;
    }
    else
    {
        x_dir = 0 << 31;
    } // if

    if (y1 < y2)
    {

```

```

        y_dir = 1 << 15;
    }
    else
    {
        y_dir = 0 << 15;
    } // if

    Radeon_WaitForFifo (9);

    save_dp_cntl = regr (DP_CNTL);
    save_dp_datatype = regr (DP_DATATYPE);

// Set DP_DATATYPE
    bppvalue = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    regw (DP_DATATYPE, (bppvalue | BRUSH_SOLIDCOLOR | ROP3_SRCCOPY));

// Draw line.
    regw (DP_BRUSH_FRGD_CLR, Radeon_GetColourCode(colour));
    regw (DST_Y_X, (y1 << DST_Y_X__DST_Y__SHIFT) | x1);

// Allow setting of last pel bit and polygon outline bit for line drawing.
    regw (DP_CNTL_XDIR_YDIR_YMAJOR, (y_dir | x_dir));
    regw (DST_LINE_START, (y1 << DST_LINE_START__DST_START_Y__SHIFT | x1));
    regw (DST_LINE_END, (y2 << DST_LINE_END__DST_END_Y__SHIFT | x2));
    regw (DP_CNTL, save_dp_cntl);
    regw (DP_DATATYPE, save_dp_datatype);

    return;
} // Radeon_DrawLine ()

```

Drawing Patterned Lines

The RADEON can also draw patterned lines. Pattern data is loaded into the brush data registers, and the appropriate brush is selected using

DP_BRUSH_DATATYPE@DP_DATATYPE.

Five brush types are suitable for patterned lines. They are:

- 8x1 mono pattern
- 8x1 mono pattern (leave background alone)
- 32x1 mono pattern
- 32x1 mono pattern (leave background alone)
- 8x1 color (pixel type is the same as the destination).

The following is some sample code to demonstrate drawing a patterned line:

Example Code: Drawing a patterned line

```

void Radeon_DrawPatternLine (WORD x1, WORD y1, WORD x2, WORD y2,
                             DWORD brushtype, DWORD *data)
{
    int dx, dy;
    int small, large;
    int x_dir, y_dir, y_major;
    int err, inc, dec;
    DWORD save_dp_cntl, save_dp_datatype, bppvalue;

    Radeon_LoadPatternData (brushtype, data);

    // Determine x & y deltas and x & y direction bits.

    if (x1 < x2)
    {
        x_dir = 1 << 31;
    }
    else
    {
        x_dir = 0 << 31;
    } // if

    if (y1 < y2)
    {
        y_dir = 1 << 15;
    }
    else
    {
        y_dir = 0 << 15;
    } // if

    Radeon_WaitForFifo (11);

    save_dp_cntl = regr (DP_CNTL);
    save_dp_datatype = regr (DP_DATATYPE);

    // Set DP_DATATYPE
    bppvalue = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);
    regw (DP_DATATYPE, (bppvalue | brushtype | ROP3_PATCOPY));

    // Draw line.
    regw (DST_Y_X, (y1 << DST_Y_X__DST_Y__SHIFT) | x1);

    // Allow setting of last pel bit and polygon outline bit for line drawing.
    regw (DP_CNTL_XDIR_YDIR_YMAJOR, (y_dir | x_dir));
    regw (DST_LINE_START, (y1 << DST_LINE_START__DST_START_Y__SHIFT | x1));
    regw (DST_LINE_END, (y2 << DST_LINE_END__DST_END_Y__SHIFT | x2));
    regw (DP_CNTL, save_dp_cntl);
    regw (DP_DATATYPE, save_dp_datatype);

    return;
} // Radeon_DrawPatternLine ()

```

Monochrome Expansion

This operation accepts monochrome data and expands this data into a two color bitmap. This is particularly useful for displaying text. The monochrome expansion circuitry on the RADEON allows for expanding both the foreground and background data, or just the foreground, leaving the background alone. The data must be sent to the controller via the host data registers.

The controller does not support monochrome expansion of data that resides in the frame buffer.

The following code shows how to perform a monochrome expansion blt using the host data registers to move the data to the engine.

Example Code: Monochrome expanded Blt operation

```
void MEBltThruHostData (DWORD *pSrc, WORD NumDWORDS, blt_data * pData)
{
    int loop;
    DWORD temp;

    // First write GUI_MASTER_CNTL.
    temp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    Radeon_WaitForFifo (5);
    regw (DP_GUI_MASTER_CNTL,
    // Use mmDEFAULT_OFFSET and mmDEFAULT_PITCH for SRC
        (0 << 0) |
    // Use mmDEFAULT_OFFSET and mmDEFAULT_PITCH for DST
        (0 << DP_GUI_MASTER_CNTL_GMC_DST_PITCH_OFFSET_CNTL_SHIFT) |
    // Use mmDEFAULT_SC_BOTTOM_RIGHT
        (0 << DP_GUI_MASTER_CNTL_GMC_SRC_CLIPPING_SHIFT) |
    // Use mmSC_TOP_LEFT and mmSC_BOTTOM_RIGHT for DST
        (0 << DP_GUI_MASTER_CNTL_GMC_DST_CLIPPING_SHIFT) |
    // NO_BRUSH needed.
        (0xF << DP_GUI_MASTER_CNTL_GMC_BRUSH_DATATYPE_SHIFT) |
    // Set DST_DATATYPE to current bpp
        (temp << DP_GUI_MASTER_CNTL_GMC_DST_DATATYPE_SHIFT) |
    // Expand to foreground and background.
        (0 << DP_GUI_MASTER_CNTL_GMC_SRC_DATATYPE_SHIFT) |
    // Consume monochrome data from Lsbit to Msbit
        (1 << DP_GUI_MASTER_CNTL_GMC_BYTE_PIX_ORDER_SHIFT) |
    // Set ROP3 to SRC_COPY
        (0xCC << DP_GUI_MASTER_CNTL_GMC_ROP3_SHIFT) |
    // Source Data is from HOSTDATA registers.
        (3 << DP_GUI_MASTER_CNTL_GMC_DP_SRC_SOURCE_SHIFT) |
        (0 << DP_GUI_MASTER_CNTL_GMC_SRC_DATATYPE2_SHIFT) |
    // Clear CLR_CMP_CNTL (Disable Color Compare)
        (1 << DP_GUI_MASTER_CNTL_GMC_CLR_CMP_FCN_DIS_SHIFT) |
    // Set DP_WRITE_MASK to 0xFFFFFFFF
```

```

        (1 << DP_GUI_MASTER_CNTL__GMC_WR_MSK_DIS__SHIFT)
    );
    // Set the colors to expand the data to.

    temp = Radeon_GetColourCode (pData->frgd);

    regw (DP_SRC_FRGD_CLR, temp);

    // use the complimentary colour for the background.
    temp = Radeon_GetColourCode (pData->bkgd);

    regw (DP_SRC_BKGD_CLR, temp);

    // Setup the destination trajectory.
    regw (DST_X_Y, ((pData->x << DST_X_Y__DST_X__SHIFT) | pData->y ));

    regw (DST_WIDTH_HEIGHT,
        ((pData->w << DST_WIDTH_HEIGHT__DST_WIDTH__SHIFT) | pData->h ));

    // Write the data out to the HostData registers.
    // We write the number of DWords

    for (loop = 0; loop < NumDWORDS; loop++ )
    {
        Radeon_WaitForFifo (1);

        regw (HOST_DATA0, *pSrc );

        pSrc += 1;
    }

} // Radeon_MEBltThruHostData ()

```

4.4 Hardware Cursor

The RADEON supports a hardware cursor. The cursor is represented by a bitmap of 64x64 pixels. This map normally resides in the off-screen area of frame buffer.

Register **CUR_OFFSET** points to the memory location of the bitmap, with reference to the beginning of frame buffer.

The cursor actually seen on the screen may be smaller than the bitmap, and occupies the top right corner of the bitmap. Therefore, the reduction in the bitmap's horizontal and vertical dimensions needs to be specified in the **CUR_HORZ_VERT_OFF** register. The coordinate (i.e. screen location) of the displayed cursor is determined by the **CUR_HORZ_VERT_POSN** register.

The hotspot (i.e. the “sensor” of the cursor) is inside the displayed cursor. The hotspot of the RADEON's cursor is at the top-left corner of the display cursor.

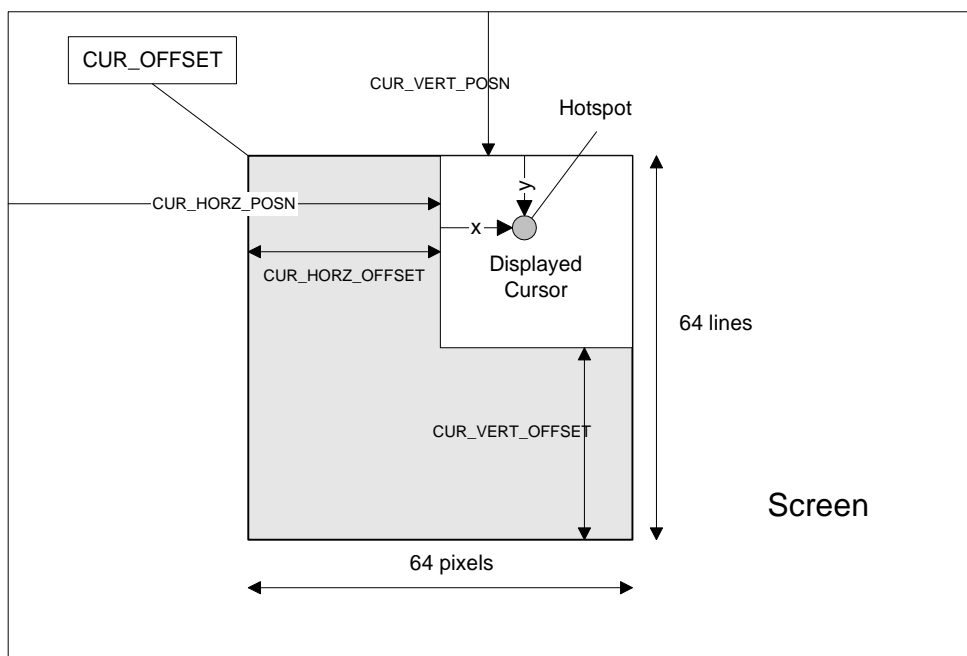


Figure 4-2. Cursor Related Parameters

The cursor bitmap consists of 64 rows, and each row has 64 pixels. Each pixel is represented by two bits. One is called the AND bit and the other is the XOR bit. Therefore, each row of the bitmap is represented by 128 bits. The first 64 bits represent the AND bits of the 64 pixels, and the remaining bits represent the XOR bits. The memory organization of the bitmap is shown as follows. In the table, entries Pixel, Bit and Byte denote the pixel, bit and byte positions of a pixel in a row.

- Row_x_A denote the positions of AND bits.
- Row_x_X denote the positions of XOR bits.

Table 4-3 Pixel Location in Memory

Pixel	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	56	57	58	59	60	61	62	63
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
Row_0_A	byte 0							byte 1									byte 7								
Row_0_X	byte 8							byte 9									byte 15								

Table 4-3 Pixel Location in Memory (Continued)

Pixel	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	56	57	58	59	60	61	62	63	
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0	
Row_1_A	byte 16								byte 17										byte 23							
Row_1_X	byte 24								byte 25										byte 31							
...							
Row_63_A	byte 1008								byte 1009										byte 1015							
Row_63_X	byte 1016								byte 1017										byte 1023							

The hardware cursor is specified by the following parameters:

- Cursor Pixel
- Cursor Pitch
- Cursor Position

Cursor Pixel

This pixel is represented by two bits. The following table shows the possible values and their meanings. The colors stored in registers CUR_CLR0 and CUR_CLR1 contain color codes in the 24-bit RGB format (i.e. the true-color format), regardless of the current pixel depth.

Table 4-4 Cursor Pixel

AND	XOR	Resulting Pixel
0	0	Cursor color 0 that is given in register CUR_CLR0.
0	1	Cursor color 1 that is given in register CUR_CLR1.
1	0	Transparent
1	1	Compliment of the current display pixel value.

Cursor Pitch

This is always 64 pixels. That is, each scan line of the hardware cursor definition is defined with 64*2 bits (16 bytes) of data, regardless of the actual cursor width. The pixel definition is specified in the Intel order.

Cursor Position

This specifies the coordinate of the top-left corner of the cursor on the screen. The coordinate is stored in register **CUR_HORZ_VERT_POSN**, which tells the current coordinate as the cursor moves around. When the cursor goes outside the screen, either its horizontal or vertical coordinate may become negative.

In such a circumstance, RADEON will not display the cursor at all. However, the hot spot of the cursor, which is inside of the displayed cursor, may still be on the screen, but is ineffective since the left-to corner of the cursor falls outside the screen. Therefore, some adjustment to the cursor-related parameters has to be made to keep the cursor being display partially when the left-top corner of the cursor falls outside the screen.

Example Code: Initializing a hardware cursor

```
void Radeon_SetHWCursor (BYTE cursor)
{
    DWORD cur_offset, horz_offset, vert_offset;
    DWORD temp;

    // Check that cursor size is within limits
    if ((CURSORDATA[cursor].width < 1) || (CURSORDATA[cursor].width >
64))
return;
    if ((CURSORDATA[cursor].height < 1) || (CURSORDATA[cursor].height > 64))
return;

    // determine offsets within cursor bitmap
    horz_offset = 64 - CURSORDATA[cursor].width;
    vert_offset = 64 - CURSORDATA[cursor].height;

    CURSORDATA[cursor].cur_offset = Radeon_AdapterInfo.MEM_BASE +
CURSORDATA[cursor].cur_offset;

    // Set cursor size offsets.
    regw (CUR_HORZ_VERT_OFF,
(horz_offset << CUR_HORZ_VERT_POSN__CUR_HORZ_POSN__SHIFT )
| vert_offset);

    // Set cursor offset to cursor data region.
    regw (CUR_OFFSET, CURSORDATA[cursor].cur_offset);
} // Radeon_SetHWCursor ()
```

This page intentionally left blank.

Chapter 5

CCE Initialization and Usage

5.1 Scope

The Command Processor (CP), also known as Concurrent Command Engine (CCE) on the RADEON, provides a simple method of programming 2D drawing operations. Instead of making register writes directly as you would in programmed I/O mode (PIO), simply prepare register(s) write data in the form of packets on system memory and submit them to the hardware.

The CCE microengine automatically parses the packet and programs the necessary registers. This method of programming is particularly efficient when bus master is used to send the packets from system memory to the hardware's Command Stream Queue (CSQ) for the CCE to process. If bus master is not available, the packets can be written directly to the CSQ.

In the past, the CCE registers were known as the ProMo4 (PM4) registers. ProMo4 stood for 'Programming Model 4' (i.e., programming the hardware thorough the submission of packets).

The other three methods, collectively known as PIO modes, were register writes through:

- The I/O space.
- The small aperture in VGA space.
- The register aperture.

The following figure shows:

- The architecture of the RADEON
- How the CCE microengine relates to the rest of the controller.

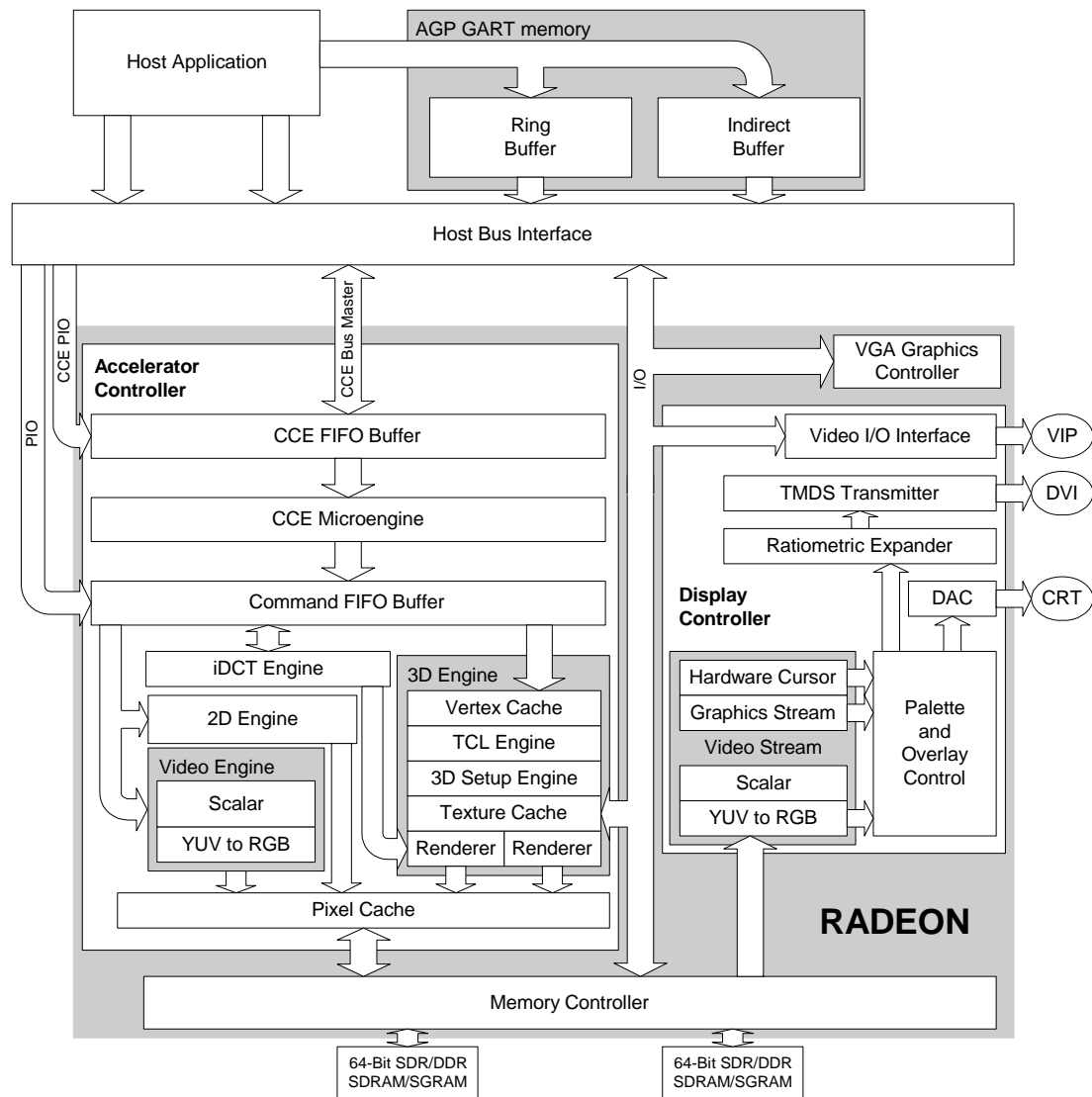


Figure 5-1. RADEON Structure and Data Flow

5.2 Starting the CCE Microengine

For the purpose of this explanation, it is assumed that the RADEON is working in PIO (i.e., programmable Input and Output) mode, and that the GUI engine is busy executing the commands in the command FIFO buffer.

5.2.1 Wait for Engine Idle

Prior to any writes to any CCE register, it is essential to check the state of the GUI engine to ensure that the contents of the command FIFO have been processed and the engine is in a state of idleness.

5.2.2 Load the Microcode into the Microengine

The microcode for the microengine is 256 QWORDS long, and can be loaded into the microengine through writing to the following registers:

- CP_ME_RAM_DATAH, and
- CP_ME_RAM_DATAH

The RADEON needs to be informed of the microcode's starting address in CP_ME_RAM_ADDR before loading begins.

Example Code: Loading the microcode into the microengine

```
DWORD CP_Microcode[256][2]={
    {low DWORD, high DWORD},
    ...
    {low DWORD, high DWORD}
};

void CPLoadMicroEngineRAMData (void)
{
    int i;

    // Wait for engine idle before loading the microcode.

    Radeon_WaitForIdle ();

    // Set starting address for writing the microcode.

    regw (CP_ME_RAM_ADDR, 0);

    for (i = 0; i < 256; i += 1)
    {
        // The microcode write address will automatically increment after
        // every microcode data high/low pair. Note that the high DWORD
```

```
        // must be written first for address autoincrement to work cor-
        rectly.

        regw (CP_ME_RAM_DATAH, CP_Microcode[i][1]);
        regw (CP_ME_RAM_DATAH, CP_Microcode[i][0]);
    } // for

    return;
} // CCELoadMicrocode
```

5.2.3 Command Stream Queue PIO Mode

There are two CSQ; Primary and Indirect. The CSQ_MODE in the CP_CSQ_CNTL register controls which CSQ is enable and whether it is PIO or Bus Master.

We would start with PrimaryPIO and IndirectPIO. When the Primary and/or Indirect PIO is/are enabled, the CSQ_CNT_PRIMARY and CSQ_CNT_INDIRECT in the CP_CSQ_CNTL register indicate the size of the queue in DWORDs. If only Primary is enable, the number of DWORDs would be 255. If both Primary and Indirect are enable, each would have the queue size of 127 DWORDs.

To write the PM4 packets to the CSQ in PIO mode, the registers CP_CSQ_APER_PRIMARYs from offset 0x1000 to 0x11FC for Primary Queue and CP_CSQ_APER_INDIRECTs from offset 0x1300 to 0x13FC for Indirect Queue may be used. The range of offset does not mean the size of the queues. It only provides you with different memory map addresses for writing to the queue. As a matter of fact, you can write to only the first offset at all times and it would still work. The method we recommend is to start at the first offset and move the end and then repeat the process.

Before writing to the queue, it is a good idea to check for available DWORDs in the queue. Each queue has a Read Pointer and a Write Pointer. The Read Pointer is the point at which the CP finished reading and the Write Pointer is the end of the submitted packets. When the Read Pointer is at the same location as the Write Pointer, it means that all the packets are executed by the CCE or there is nothing in the queue.

Before submitting packets to Indirect queue, the Primary CSQ must be idle; otherwise the hardware would hang. To ensure the Primary idle, check first for empty Primary queue and then check the CP_STAT register for CSQ_PRIMARY_BUSY bit.

Example code:

```
#define CP_CSQ_APER_PRIMARY_BEGIN 0x1000
#define CP_CSQ_APER_PRIMARY_END 0x11FC
static int CPSubmitPrimaryPIO (DWORD *ClientBuf, DWORD DataSize)
{
    static WORD count = CP_CSQ_APER_PRIMARY_BEGIN;
```



```

int size;
int i;

while (1)
{
    size = GetAvailPrimaryQueue();
    if(size == 0)
    {
        continue;
    }
    if(DataSize < size)
    {
        for(i = 0; i < DataSize; i++)
        {
            regw(count, *ClientBuf++);
            if(count == CP_CSQ_APER_PRIMARY__END)
                count = CP_CSQ_APER_PRIMARY__BEGIN;
            else
                count += 4;
        }
        break;
    }
    else
    {
        for(i = 0; i < size; i++)
        {
            regw(count, *ClientBuf++);
            if(count == CP_CSQ_APER_PRIMARY__END)
                count = CP_CSQ_APER_PRIMARY__BEGIN;
            else
                count += 4;
        }
        DataSize -= size;
    }
} // while
}
return (CCE_SUCCESS);
} // CPSubmitPrimaryPIO
#define CP_CSQ_APER_INDIRECT__BEGIN 0x1300
#define CP_CSQ_APER_INDIRECT__END 0x13FC

static int CPSubmitIndirectPIO (DWORD *ClientBuf, DWORD DataSize)
{
    static WORD count = CP_CSQ_APER_INDIRECT__BEGIN;
    int size;
    int i;

    CSQPrimaryIdle();
    while (1)
    {
        size = GetAvailIndirectQueue();
        if(size == 0)
        {
            continue;
        }
    }
}

```

```
    if(DataSize < size)
    {
        regw(CP_IB_BUFSZ, DataSize);
        for(i = 0; i < DataSize; i++)
        {
            regw(count, *ClientBuf++);
            if(count == CP_CSQ_APER_INDIRECT__END)
                count = CP_CSQ_APER_INDIRECT__BEGIN;
            else
                count += 4;
        }
        break;
    }
    else
    {
        regw(CP_IB_BUFSZ, size);
        for(i = 0; i < size; i++)
        {
            regw(count, *ClientBuf++);
            if(count == CP_CSQ_APER_INDIRECT__END)
                count = CP_CSQ_APER_INDIRECT__BEGIN;
            else
                count += 4;
        }
        DataSize -= size;
    } // while
}
return (CCE_SUCCESS);
} // CPSubmitIndirectPIO

int CSQPrimaryIdle(void)
{
    int data;

    if(CPmode[CSQmodeIndex].primaryCSQ == TRUE)
    {
        while(1)
        { // Ensure Queue is empty before waiting for engine idle.
            data = GetAvailPrimaryQueue();
            if(data >= (CSQPrimary - 1))
                break;
        }
        while(1)
        { //check for CSQ busy
            data = regr(CP_STAT);
            if(!(data & CP_STAT__CSQ_PRIMARY_BUSY))
                break;
        }
    }
    // flush the pixel cache
    Radeon_FlushPixelCache ();
    return (CCE_SUCCESS);
} // CSQPrimaryIdle
```

```

int CSQIndirectIdle(void)
{
    int data;

    if(CPmode[CSQmodeIndex].indirectCSQ == TRUE)
    {
        while(1)
        {
            // Ensure Queue is empty before waiting for engine idle.
            data = GetAvailIndirectQueue();
            if(data >= (CSQIndirect - 1))
                break;
        }
        while(1)
        {
            //check for CSQ busy and CP busy
            data = regr(CP_STAT);
            if(!(data & CP_STAT__CSQ_INDIRECT_BUSY))
                break;
        }
    }
    // flush the pixel cache
    Radeon_FlushPixelCache ();
    return (CCE_SUCCESS);
} // CSQIndirectIdle

```

5.2.4 Cautions When Programming RADEON in CCE Mode

All packets must be checked for proper formatting prior to submission to the server. Incorrectly-formatted packets will cause the RADEON to hang.

5.3 Ring Buffer Management

5.3.1 The Ring Buffer Concept

When operating in CCE mode, the RADEON receives commands from the host through the CCE command packets. A command packet is a data block that consists of a header followed by a data body of variable size. When operating in bus-mastering mode, command packets are sent to the RADEON through a ring buffer and/or an indirect buffer.

The *ring buffer* is a contiguous block of system memory allocated by the host application in AGP or PCI GART memory. For more details about PCI GART memory, [see “AGP Addressing” on page 2-31](#).

The RADEON treats this buffer as a ring by wrapping back to the starting address when it reaches the end. The starting address and the size of the buffer are passed to the RADEON when initializing it for CCE bus-mastering mode.

The host application copies packets into the ring buffer in consecutive order starting at the top. The packets are bus-mastered to the RADEON's on-chip CCE CSQ buffer, where they are processed by the microengine in the order they are received. The microengine places its output into the command FIFO as register-datum pairs. When the host reaches the end of the block, it starts copying from the top again. The following figure shows a conceptual representation of the ring buffer.

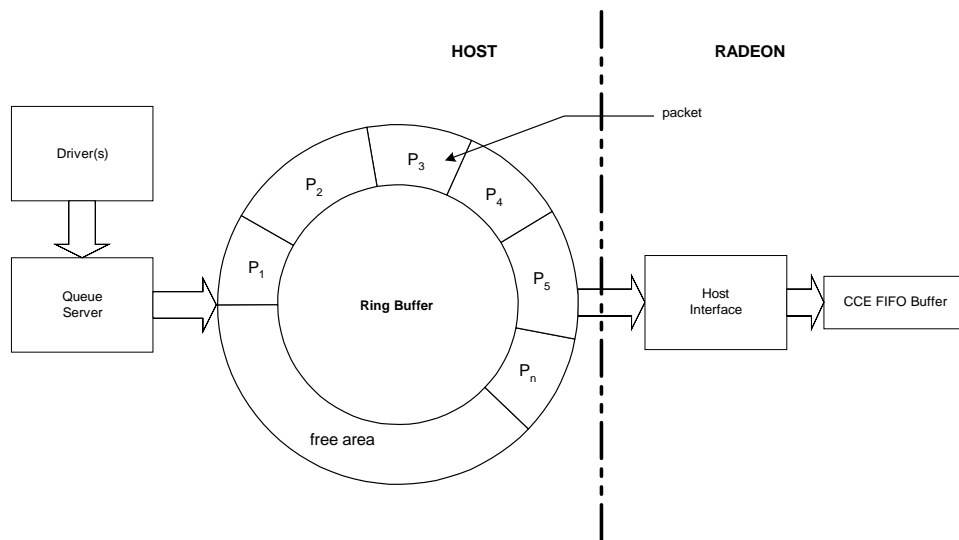


Figure 5-2. Ring Buffer and its Control Structure

The above figure shows that the command packets are placed into the buffer in clock-wise order, forming a packet queue. The first packet in the queue is denoted by P₁, and the last by P_n. The start of the queue, P₁, is pointed to by both a packet pointer maintained by the application, and the RADEON's **CP_RB_RPTR** register. The memory portion that is not occupied by packets is called the free area. It is pointed to by both a free buffer pointer maintained by the application and the RADEON's **CP_RB_WPTR** register. All incoming packets should be placed into this area.

Initially, both the packet and free area pointers point at the start of the memory block. Thereafter, whenever the two pointers meet it implies that the ring buffer is either completely empty or completely full. It is assumed that the data processing speed of the

RADEON is faster than the speed of data transfer from the ring buffer to the RADEON. Therefore, this condition is generally interpreted as the ring buffer being empty.

As packets are put into and taken out of the ring buffer, the packet and free area pointers must be updated to keep track. The updates should be kept synchronized between the host application and the RADEON. On the host side, the application places a command packet at the location pointed to by the current free area pointer, updates the free area pointer to point just beyond this new packet, and updates the RADEON free area pointer by writing the new free area address to the **CP_RB_WPTR** register.

On the RADEON side, packets are read one at a time from the head of the packet queue pointed to by the **CP_RB_RPTR** register, and sent to the CP CSQ buffer. The **CP_RB_RPTR** register is updated automatically after each packet transfer. The updated packet pointer must be sent back to the host application so that it may keep track of the available free space. The RADEON accomplishes this by updating the application's packet pointer through a bus-mastering operation. The address of the application's packet pointer must be written to the RADEON's **CP_RB_RPTR_ADDR** register during CP initialization. Note that the AGP interface stops data transfer once pointer **CP_RB_RPTR** meets **CP_RB_WPTR**.

5.3.2 Queue Server

In an operating system environment, there may be a need to share the ring buffer among several clients, such as a 2D display driver and a 3D driver. In this circumstance, a method is required to arbitrate the use of the ring buffer. One method is to grant clients exclusive access to the ring buffer through a queue server. Under this scheme, all clients submit packets to the queue server, and the server mediates and schedules delivery of the packets to the ring buffer. The queue server may also be used to mediate use of the indirect buffer, which is covered in the next section.

The following is a sample function defined for the server. The function needs two entrance parameters. The address of client's packet buffer **ClientBuf*, and the size of the data *dwDataSize* are submitted to the ring buffer. As the head of the packet queue is updated by RADEON through bus-mastering, the function keeps track of the updated queue head by keeping a copy for its own record, and updates the size of the available space at the same time.

5.3.3 Indirect Buffer

In addition to transferring packets through the ring buffer, the host application may transfer them through an *indirect buffer* when using CCE bus-mastering mode. Similar to the ring buffer, the indirect buffer is a contiguous block of memory allocated by the host

application in AGP or PCI GART space. However, unlike the ring buffer, the indirect buffer is linear. There are no wrapping mechanisms governing its use and operation.

To view a diagram of the indirect buffer [refer to Figure 5-3. on page 5-11.](#)

The indirect buffer allows long command sequences to be built up in parallel by different contexts or threads and submitted to a queue server without the overhead of copies. For this reason, the recommendation is to use the indirect buffer as the primary means for streaming packets, especially for multimedia drivers such as 3D, DVD, etc. One possible scheme for arbitrating the use of the indirect buffer among such drivers is to give ownership of this resource to a queue server. The queue server could allocate blocks of indirect buffer memory to the drivers, which the drivers could use to build their command sequences. Blocks used in this manner may be recycled and reused by the queue server. The queue server could use a time stamping mechanism to ensure that the hardware has completed using an indirect buffer block before recycling it.

The indirect buffer should be 4K page aligned.

The packet byte offset from the base of the indirect buffer is specified in the **CP_IB_BASE** register. The size in even number of DWORDs is specified in the **CP_IB_BUFSZ** register. If a packet's size is an odd count of DWORDs, it should be padded with a single type-2 NOP packet. Writing **CP_IB_BUFSZ** initiates the packet transfer.

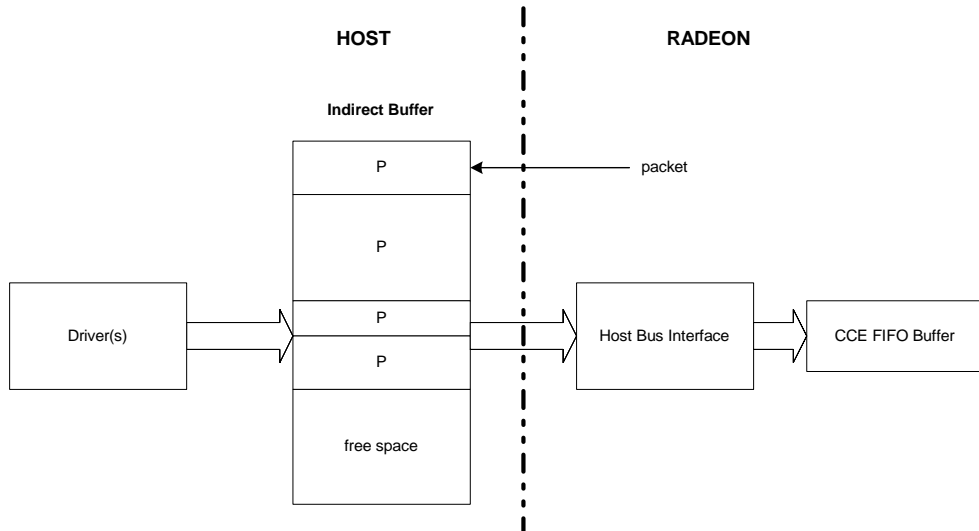


Figure 5-3. The Indirect Buffer

The most efficient programming model for the RADEON is to use both the ring buffer and the indirect buffer. The ring buffer enables concurrency and command streaming, whereas the indirect buffer allows long command sequence to be built up in parallel for different contexts without the overhead of ring buffer copies. The packet transfers out of the indirect buffer may be streamed by writing CP_IB_BASE and CP_IB_BUFSZ through a type-0 packet submitted to the ring buffer. If the ring buffer is not used, indirect buffer transfers may still be executed by writing these two registers through conventional PIO.

This page intentionally left blank.

6.1 Scope

This section describes how to use the CCE packets and provides programming examples for various engine operations (e.g. blts, rectangle and line draws, etc.). CCE packets are used to draw two-dimensional (2D) images, such as:

- Lines
- Rectangles
- Polygons
- Text
- Moving pixels

The targeted operation area is the entire CRT screen, not just a limited screen area such as a window. For all the 2D operations, this discussion will refer to a coordinate system that is based on the entire CRT screen.

Programming examples will demonstrate how to use the CCE packets to draw 2D images. For a detailed discussion about these packets, refer to Appendix F.

6.2 2D Coordinate System

The coordinate system used in 2-D operations is shown in *Figure 6-1*.

- x-axis points to the right.
- y-axis points downwards.
- Origin is located at the screen's top-left corner.
- Scales are integer intervals (a coordinate represents the position of a pixel).

For any objects to be drawn on the screen, the values of the x- and y-coordinates are limited to positive integers. They range from zero to M-1, and from zero to N-1, respectively.

For negative coordinates or coordinates beyond (M-1, N-1), the objects may not be entirely drawn on the screen, but could still be drawn into frame buffer.

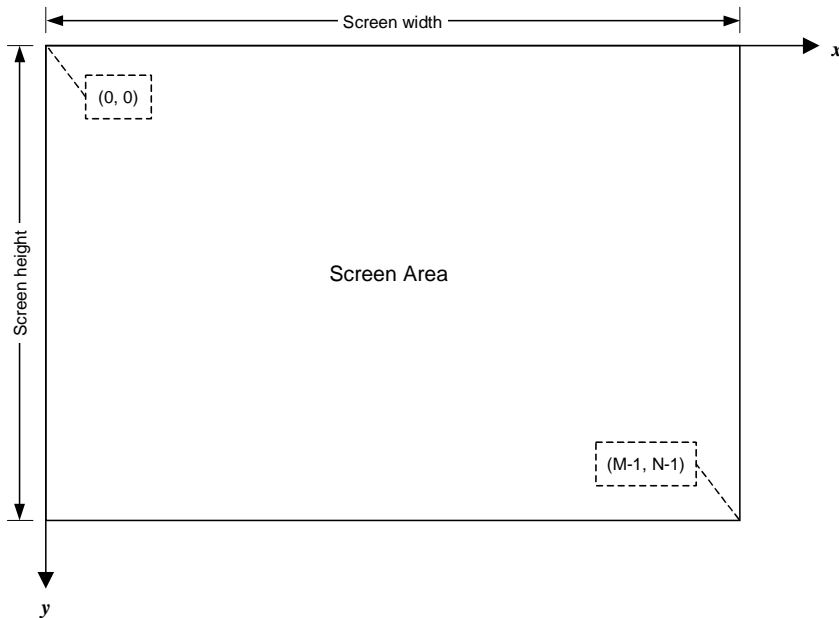


Figure 6-1. 2D Coordinate System

6.2.1 Essentials of 2D Drawing Operations

The term **rendering** describes general drawing operations to the screen or to the frame buffer. This operation manipulates pixels from a number of sources. It is more complex than a simple drawing operation such as drawing an object to the screen or copying data from one location to another. Rendering 2-D images actually involves manipulating pixels from different sources and placing the resulting pixels at a desired location.

When rendering, three types of source pixels are manipulated, such as:

- **Source pixels** are taken from a location in the frame buffer or supplied by the host. These pixels will not be modified after rendering.
- **Brush** are patterns are stored either in the relevant RADEON registers or in system memory. These pixels will not be modified after rendering.
- **Destination pixels** are taken from the frame buffer as source data before rendering, and they will be replaced by new pixels written to their position.

Generally, pixels that participate in the pixel manipulation are called the **source components**. The manipulated data is written to a location called the **destination area** or **destination**.

In the following discussion, a **destination pixel** is a source component that comes from the **destination**, unless specified otherwise. Rendering may involve one, two or all of the source components. The operation that manipulates the source components will be referred to as a **raster operation** (ROP). The RADEON supports all 256 ROP3 raster operations.

As rendering operations occur in a specific display mode, the program must specify the following parameters to the RADEON with respect to a specified operation. These parameters are referred to as **rendering parameters**. They are:

- The type of destination pixels (one of 8, 16 and 32 BPP).
- If there is a source involved, the type of source pixels.
- The brush type selected for the rendering operation.
- If the brush is involved, the color of the selected brush represented in the destination pixel type.
- The source where the source pixels will be loaded from (system memory or frame buffer).
- The drawing order of pixels (from left to right or from right to left).

If required, the source clipping rectangle that restricts the area where data are taken from.

- If required, the destination clipping rectangle that will specify the area where the rendering operation is carried out.
- The source offset and pitch that specify the source's start location and pitch. If not specified, the default offset and pitch (the screen offset and screen pitch) are assumed. This is only applicable to the source loaded from frame buffer.
- The destination offset and pitch that specify the destination's start location in the frame buffer and pitch. If not specified, the default offset and pitch (the screen offset and screen pitch) are assumed.
- The raster operation type carried out in combining the source, brush pattern and destination pixels.
- The location and geometry of the objects to be drawn.

6.3 Drawing Objects

RADEON provides hardware assistance for drawing the following:

- Polylines.
- Polyscanlines.
- Rectangles.

The RADEON does not support drawing the following:

- Circles.
- Ellipses.

While drawing the objects, the source pixels involved are the brush and destination components. The source component is not involved. In this case, the brush pattern selected for drawing is considered as a source, and the pixels of the object being drawn are considered as the destination. In addition to specifying the rendering parameters, also specify the location and geometry of the intended object.

6.3.1 Drawing Rectangles

To draw a rectangle, specify the:

- Rendering parameters.
- Location of the rectangle.
- Geometry of the rectangles.

If the rectangle is to be filled with a pattern, specify where the source pattern is loaded from for the brush. If the pattern is not stored in the brush registers, load the pattern from system memory by supplying the raster data of the pattern to the packet.

The rectangle's location is specified by the coordinate of its left-top corner.

The rectangle's geometry is specified by either: its height and width, or by the coordinate at its bottom-right corner (from which the height and width can be calculated). When the coordinate of the rectangle's right-bottom corner is specified, the bottom and right edges of the rectangle will not be drawn.

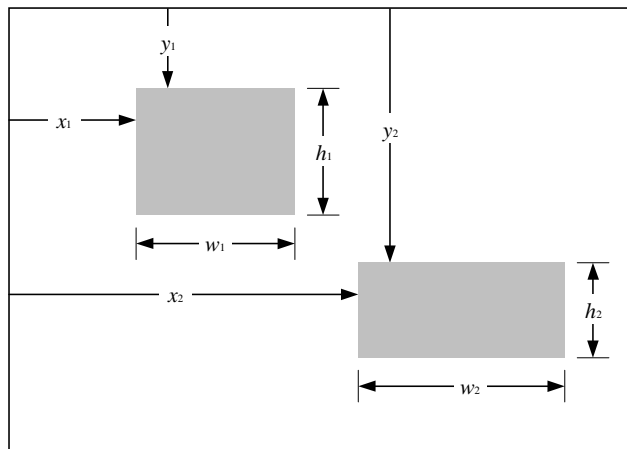


Figure 6-2. Rectangles

Figure 6-2. shows two rectangles to be drawn on the screen.

The destination-pixel type is aRGB (one of the 16 BPP modes). The dimensions of the rectangles are specified by parameters x_i , y_i , w_i , and h_i , for $i = 1, 2$.

The PAINT packet can be used to draw these rectangles. Assume the rectangles are drawn in the clipping rectangle specified by its top-left corner (x_1, y_1) and bottom-right corner (x_2+w_2, y_2+h_2) with a brush in the type of solid pen. The other parameters are similar to those of drawing polyscanlines except that a clipping rectangle is specified.

The following programming code shows how to draw rectangles.

Example Code: Drawing rectangles

```
#define CCE_PACKET3_CNTL_PAINT      0xC0009100
void Radeon_paint (int argc, char *argv[])
{
    int test;
    DWORD Buf[20];
    DWORD x, y, width, height, colour;

    // First, run StartUp function to set up the application
    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);
```

```

// Initialize the CCE microengine.
if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
{
    Radeon_ShutDown ();
    printf ("Radeon_CPInit failed!!\n");
    exit (1);
} // if

//Draw random sized rectangle
while (!kbhit ())
{
    x = (rand () % (Radeon_AdapterInfo.xres));
    y = (rand () % (Radeon_AdapterInfo.yres));
    width = rand () % (Radeon_AdapterInfo.xres - x);
    height = rand () % (Radeon_AdapterInfo.yres - y);
    colour = Radeon_GetColourCode (rand () % NUM_COLOURS);

    // Set up a rectangle packet and fill the header with packet size
    // Note that packet size is two less than total number of packets sent
    Buf[0] = CCE_PACKET3_CNTL_PAINT | (3 << 16);

    // Compose GUI_CONTROL
    Buf[1] = CCE_GC_BRUSH_SOLIDCOLOR | CCE_GC_SRC_DSTCOLOR
            | ROP3_PATCOPY
            | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

    // Colour used to draw the rectangle due to CCE_GC_BRUSH_SOLIDCOLOR
    Buf[2] = colour;

    // Fill rectangle data into packet

    Buf[3] = x + (y << 16);
    Buf[4] = (x + width) + ((y + height) << 16);

    test = Radeon_CPSubmitPackets (Buf, 5);
} // while

getch ();

// Shut down the microengine.
Radeon_CPEnd (CCE_END_WAIT);
Radeon_ShutDown ();

return;
} // Radeon_paint

```

6.3.2 Drawing Polylines

A polyline consists of a number of line segments that are connected at their end-points. The ending point of the first segment is the starting point of the second segment, etc. Therefore, if a polyline is composed of n line segments, it can be represented by $n + 1$ points.

For example, the polyline in Figure 6-3. is composed of four line segments. It may be represented by points p_1, p_2, \dots, p_5 , where each p_i denotes a coordinate (x_i, y_i) of point i on the screen. It is obvious that a line is just a special case of polyline, which is composed of one line segment. The RADEON draws a line from the start-point to the end-point. The last point of the line may or may not be drawn (this depends on how the GUI engine was set at initialization). In Figure 6-3., the drawing of the first line segment starts at p_1 , and ends at the point next to p_2 . The drawing of the second line segment starts at p_2 , and ends at the point next to p_3 . The remaining lines are drawn in a similar fashion. Point p_1 is part of the first line segment; point p_2 is part of the second line segment, etc.

To program the RADEON to draw a polyline with CCE packets, select the POLYLINE packet and specify the following rendering parameters:

- The type of destination pixels is aRGB (one of the 16 BPP formats).
- The type of source pixels is the same as the destination.
- The brush selected is a Solid Pen.
- The color of the brush is White.
- No source data is involved.
- The pixels are drawn from left to right.
- The source-clipping rectangle is not applicable.
- The destination clipping rectangle is set to default.
- The source offset and pitch are not applicable, and therefore use the default offset and pitch.
- The destination offset and pitch are the screen offset and pitch (default offset and pitch).
- The raster operation type is copying the brush pattern to the destination.
- The location and geometry of the object are specified by points p_1, p_2, \dots, p_5 .

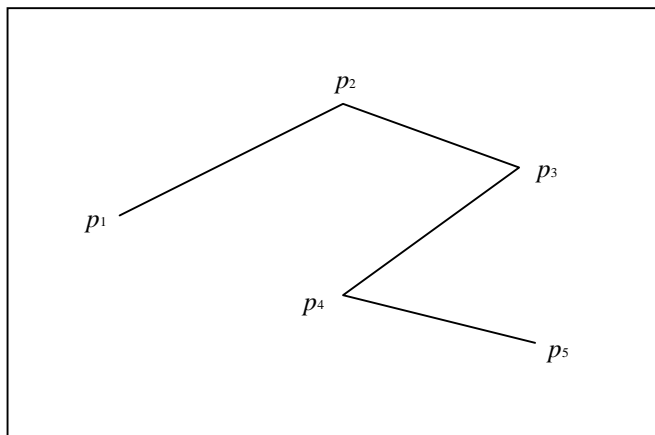


Figure 6-3. Polyline

Example Code: Drawing a polyline

```
#define CCE_PACKET3_CNTL_POLYLINE    0xC0009500
#define CSQ_MODE_PRIPIO_INDPIO      5

void Radeon_Polyline (int argc, char *argv[])
{
    DWORD Buf[20];
    DWORD x, y, colour;
    int test;
    int i, j;

    // First, run StartUp function to set up the application
    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);

    // Initialize the CCE microengine.
    if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
    {
        Radeon_ShutDown ();
        printf ("Radeon_CPInit failed!!\n");
        exit (1);
    } // if

    // Draw the outline of a rectangle using a 2D polyline packet:
    // Set up a polyline packet and fill the header with packet size
```



```

// Note that packet size is two less than total number of packets sent

i=0;
Buf[i++] = CCE_PACKET3_CNTL_POLYLINE;

// Compose GUI_CONTROL
Buf[i++] = CCE_GC_BRUSH_SOLIDCOLOR | CCE_GC_SRC_DSTCOLOR
          | ROP3_PATCOPY
          |(Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

// Colour used to draw the line due to CCE_GC_BRUSH_SOLIDCOLOR
Buf[i++] = Radeon_GetColourCode (WHITE);

// Fill rectangle data into packet:
// top, left
x = Radeon_AdapterInfo.xres * 0.2f;
y = Radeon_AdapterInfo.yres * 0.2f;

Buf[i++] = x + (y << 16);

// top, right
x = Radeon_AdapterInfo.xres * 0.8f;
y = Radeon_AdapterInfo.yres * 0.2f;

Buf[i++] = x + (y << 16);

// bottom, right
x = Radeon_AdapterInfo.xres * 0.8f;
y = Radeon_AdapterInfo.yres * 0.8f;

Buf[i++] = x + (y << 16);

// bottom, left
x = Radeon_AdapterInfo.xres * 0.2f;
y = Radeon_AdapterInfo.yres * 0.8f;

Buf[i++] = x + (y << 16);

// top, left again
x = Radeon_AdapterInfo.xres * 0.2f;
y = Radeon_AdapterInfo.yres * 0.2f;

Buf[i++] = x + (y << 16);

Buf[0] |= ((i - 2) << 16);

test = Radeon_CPSubmitPackets (Buf, i);

getch ();

```

```

// Shut down the microengine.
Radeon_CPEnd (CCE_END_WAIT);
Radeon_ShutDown ();

return;
} // Radeon_Polyline

```

6.3.3 Drawing Polyscanlines

A polyscanline is composed of a number of horizontal line segments. It is specified by its:

- Vertical position, y_i
- Line thickness, h_i (measured in number of pixels)
- Start-end positions of its segments (x_{ij}, x_{ij+1}) for $j = 0, 2, \dots, 2n$ where n_i denotes the number of segments of the i -th polyscanline

Figure 6-4. shows three polyscanlines. The first consists of three segments with thickness h_1 . The second and third consist of two segments and one segment, respectively (their thickness h_2 and h_3 are omitted from the figure).

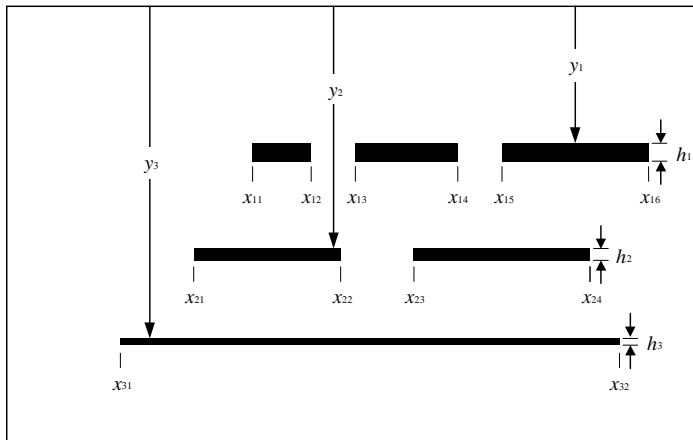


Figure 6-4. Polyscanlines

Assuming that the polyscanlines in *Figure 6-4.* are drawn completely without being clipped, in LightBlue of the 16 BPP format, select the CCE packet POLYSCANLINES to draw these images. Specify the related rendering parameters as follows:

- The type of destination pixels is aRGB (one of the 16 BPP format).
- The type of source pixels is the same as the destination.
- The brush selected is a Solid Pen.
- The color of the brush is Blue.
- No source data involved.
- The pixels are drawn from left to right.
- The source-clipping rectangle is not applicable.
- The destination-clipping rectangle is not specified.
- The source offset and pitch are not applicable, and therefore use the default offset and pitch.
- The destination offset and pitch are the screen offset and pitch (default offset and pitch).
- The raster operation type is copying the brush pattern to the destination.
- The location and geometry of the scanlines are specified by the variables y and (height + j).

Example Code: Drawing polyscanlines

```
#define CCE_PACKET3_CNTL_POLYSCANLINES  0xC0009800
#define CSQ_MODE_PRIPIO_INDPIO          5

void Radeon_Polyscanline (int argc, char *argv[])
{
    DWORD Buf[50];
    DWORD xs, xe, y, height;
    int test;
    int i, j, k, scan_count, num_segment, interval;

    // First, run StartUp function to set up the application
    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);

    // Initialize the CCE microengine.
    if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
    {
        Radeon_ShutDown ();
        printf ("Radeon_CPInit failed!!\n");
        exit (1);
    } // if

    i = 0;
```

```
// Set up a polyscanline packet and fill the header with packet size
// Note that packet size is two less than total number of packets sent
Buf[i++] = CCE_PACKET3_CNTL_POLYSCANLINES;

// Compose GUI_CONTROL
Buf[i++] = CCE_GC_BRUSH_SOLIDCOLOR | CCE_GC_SRC_DSTCOLOR
          | ROP3_PATCOPY
          | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

// Colour used to draw the line due to CCE_GC_BRUSH_SOLIDCOLOR
Buf[i++] = Radeon_GetColourCode (LIGHTBLUE);
Buf[i++] = 5;          //scan_count;

interval = Radeon_AdapterInfo.xres / 11;
height = 1;
y = Radeon_AdapterInfo.yres/6;
num_segment = 5;

for (j=0; j < 5; j++)
{
    Buf[i++] = num_segment;
    Buf[i++] = y | ((height + j) << 16);

    xs = interval;
    xe = xs + interval;

    for (k = 0; k < num_segment; k += 1)
    {
        // Fill rectangle data into packet

        Buf[i++] = xs | (xe << 16);

        xs += (interval << 1);
        xe += (interval << 1);

    } // for

    y += (Radeon_AdapterInfo.yres/6);

} // for

Buf[0] |= ((i - 2) << 16);
test = Radeon_CPSubmitPackets (Buf, i);

getch ();

// Shut down the microengine.
```

```
Radeon_CPEnd (CCE_END_WAIT);  
Radeon_ShutDown ();  
  
return;  
} // Radeon_Polyscanline
```

6.4 Block Transfers

The RADEON provides hardware support for transferring data within the frame buffer (from one location to another), and for transferring data from the system memory to the frame buffer.

The location where the data is taken from is referred to as the source, and the location where the data is transferred to is referred to as the destination. The size of the data transfer determines the size of a rectangular area on the screen. In this sense, Block Transfer means copying pixels from one place to another with some pixel manipulation. If a data transfer from system memory to frame buffer is required, the host must supply the raster data as part of a CCE packet.

Three types of pixels (source, destination, and brush pattern) may get involved in a block transfer. The resulting destination is the combination of one, two, or all three components. In this sense, all three components are considered as the components of the source before the operation that combines them, and only the result of the combination is considered as the destination. In a block data transfer, specify the location and dimension of the source and destination in addition to the setup parameters.

The following types of data transfer may occur:

- BitBlt, also known as source copy, where the content of the source is copied to the destination without any changes of its dimensions.

6.4.1 Bit Block Transfer

BitBlt operation transfers pixels from a source rectangle to a destination. The dimension of the transferred rectangle remains the same as the source. The transfer is controlled by a ternary-raster operation code that specifies how the pixels from the source and the brush pattern are mixed with those of the destination to form the final pixels at the destination.

The RADEON supports:

- Normal data transfer (i.e., the data transfer that does not change the format of the data taken from the source before placing it at the destination).
- Monochrome to color expansion when transferring a monochrome bitmap to the

screen.

For color expansion, specify the bitmap's foreground and background colors. RADEON will convert the white bit ('1') to the foreground color of the corresponding pixel and the black bit ('0') to the background color.

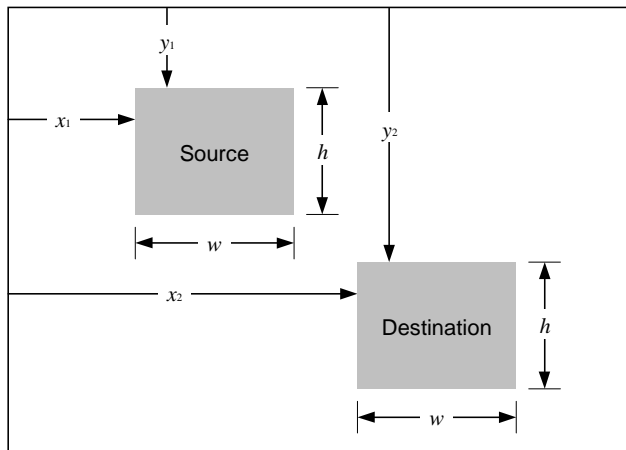


Figure 6-5. Copy an Image from Source to Destination

To copy the screen area named **Source** to the area named **Destination** in [Figure 6-5](#). It is obvious that the pixel types for the source and the destination are the same, say in the aRGB format.

If the resulting destination matches the source, choose the CCE packet BITBLT to perform the operation.

Specify the following parameters:

- The type of the destination pixels is aRGB.
- The type of the source pixels is the same as the destination.
- No brush is selected.
- The color of the brush is not applicable.

The source pixels are loaded from the video memory.

- The pixels are drawn from left to right.
- The source-clipping rectangle is not applicable.
- No destination-clipping rectangle is required.
- Use the default source pitch and offset as this is a screen-to-screen data transfer.
- Use the default destination pitch and offset as this is a screen-to-screen data transfer.
- The raster operation type is Source Copy (code 0xCC).
- The location and dimension of the source and destination are (x_1, y_1) , (h, w) and (x_2, y_2) , (h, w) , respectively, as shown in [Figure 6-5](#).

Example Code: Copying an image from a source to a destination

```
#define CCE_PACKET3_CNTL_BITBLT                                0xC0009200
void Radeon_bitblt (int argc, char *argv[])
{
    DWORD Buf[20];
    WORD srcx, srcy, dstx, dsty, bpp;
    int test;
    int i, j;
    _img_info IMG_DATA;

    // Set 32 BPP colour depth.

    BPPOverride = 32;

    // First, run StartUp function to set up the application

    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);

    // Set up the source data co-ordinates.
    srcx = 0;
    srcy = Radeon_AdapterInfo.yres;
    bpp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    // Load the source image file into offscreen memory.

    IMG_DATA = Load_Image (TRAJECTORY_RECTANGULAR, bpp, srcx, srcy);

    // Initialize the CCE microengine.
    if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
    {
        Radeon_ShutDown ();
        printf ("Radeon_CPInit failed!!\n");
        exit (1);
    }
}
```

```
    } // if

    i = 0;

    // Set up a rectangle packet and fill the header with packet size
    // Note that packet size is two less than total number of packets sent
    Buf[i++] = CCE_PACKET3_CNTL_BITBLT;

    // Compose GUI_CONTROL
    Buf[i++] = CCE_GC_BRUSH_NONE | CCE_GC_SRC_DSTCOLOR | ROP3_SRC_COPY
              | CCE_GC_DP_SRC_RECT
              | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

    dstx = 0;
    dsty = 0;

    // Fill rectangle data into packet

    Buf[i++] = (srcx << 16) + srcy;
    Buf[i++] = (dstx << 16) + dsty;
    Buf[i++] = (IMG_DATA.width << 16) + IMG_DATA.height;

    Buf[0] |= ((i - 2) << 16);
    test = Radeon_CPSubmitPackets (Buf, i);

    getch ();

    // Shut down the microengine.

    Radeon_CPEnd (CCE_END_WAIT);
    Radeon_ShutDown ();

    return;
} // Radeon_bitblt
```

The above code can be extended to copy a number of source areas to corresponding destinations respectively, provided that all the source areas share the same properties and so do the destination areas. For example, the source areas refer to the same offset and pitch as the starting memory address and the memory size of a scanline across the screen, as well as the destinations. In other words, the settings specified for the field GUI_CONTROL should be applicable to all the block transfers.

6.4.2 Transparent Bit Block Transfer

The Transparent Bit Block Transfer is also known as *Transparent BitBlt*. This operation conditionally copies pixels from the source to the destination with reference to a

designated (reference) color (e.g., the background color). If the color of a pixel is equal to (or not equal to according to the comparison criterion), the designated color, the pixel will not be copied to the destination. This operation filters out unwanted color from the source, and is very useful in copying odd-shaped objects onto a background with patterns (e.g., games), making the objects look transparent. Since a transparent BitBlt operation is more complicated than a BitBlt operation, the following discussion will clarify some terminology before proceeding with an example.

The **source** means a color pixel, which may come from one of the following sources:

- One of foreground or background colors used to expand a mono bitmap to a color bitmap
- A color pixel from either the frame buffer or the host memory
- A color pixel of a specific color pattern (brush).

The source pixel may be combined with the destination pixel according to a given raster operation code (e.g., the AND operation), resulting the *combined source pixel*. To prevent certain colors of combined source pixels from being written to the destination, two color comparators are used for deciding whether to write a combined source pixel to the destination or to keep the original destination pixel. The comparators compare the source and destination pixels respectively against their reference colors (the source and destination references), and decide whether the combined source pixel can be written to the destination. The following is a number of strategies for making such a decision:

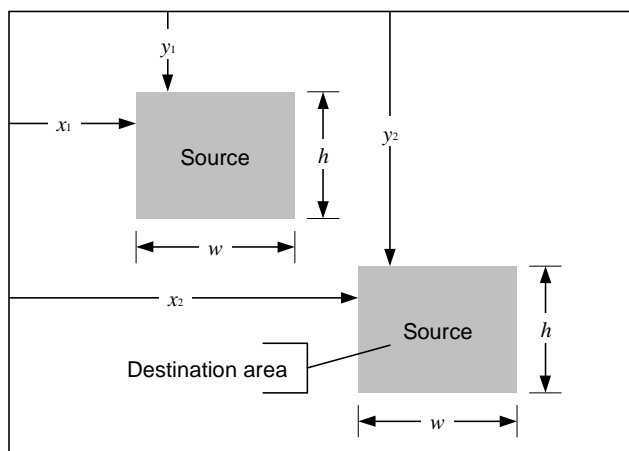
Table 6-1 Source Comparator

Decision Code	Description
0	Combined pixels are always written to the destination (i.e., no comparison is performed).
1	No combined pixel is written to the destination, i.e. the destination pixel is unchanged.
4	The combined pixel is written to the destination if the color of the source pixel is equal to its reference color. Otherwise, the destination pixel is unchanged.
5	The combined pixel is written to the destination if the color of the source pixel is NOT equal to its reference color. Otherwise, the destination pixel is unchanged.
7	Only the source pixels whose color is equal to the reference color will be XORed with the foreground color of a mono bitmap, and then written to the destination. That is, $\text{destPixel} = \text{srcPixel} \text{ XOR } \text{foregroundColor}$ if srcPixel is equal to the foreground color of a monochrome bitmap, specifically text. This is referred to as flipping sometimes.

Table 6-2 Destination Comparator

Decision Code	Description
0	Combined pixels are always written to the destination (i.e., no comparison is performed).
1	No combined pixel is written to the destination, i.e. the destination pixel is unchanged.
4	The destination is unchanged if the color of the destination pixel is equal to its reference color. Otherwise, the combined source pixel are written to the destination.
5	The destination is unchanged if the color of the destination pixel is NOT equal to its reference color. Otherwise, the combined source pixel are written to the destination.

The two tables give the decision strategy whenever either of the comparators is enabled. If both comparators are enabled, the final decision will depend on the agreement between the two decisions made separately. If both comparators decide that the combined source pixel should be written to the destination, the destination will be updated with the pixel. Otherwise, the original destination pixel is preserved.

**Figure 6-6. Transparent Bit-Block Transfer**

To perform a transparent BitBlt, as shown in [Figure 6-6.](#), the source area is the top-left rectangle, and the destination area is the bottom-right rectangle.

In the data transfer, remove the background pattern of the source and allow the word **Source** to be copied. Therefore, the pattern of the destination is preserved after the data transfer. Assume the text color is blue at the source, which is the desired color at the destination.

For this operation, select the CCE packet **TRANS_BITBLT**. The rendering parameters for this operation are the same as previous example, and are omitted here.

The combined source pixel will be the same as the source as the raster operation code is called Source Copy (**SRCCOPY**). For the CCE Transparent Bitblt Data Block, Supply data for fields **CLR_CMP_CNTL**, **SRC_REF_CLR**, **DST_REF_CLR**, **SRC_X_Y**, **DST_X_Y**, and **SRC_W_H**. As this operation only needs to compare the source pixels with the reference color, only the source comparator is enabled. Therefore, the destination reference color is not required. However, always supply this dummy data to the packet to satisfy its format requirement.

Example Code: Transparent BitBlt

```
#define CCE_PACKET3_CNTL_TRANS_BITBLT  0xC0009C00
void Radeon_Tblt (int argc, char *argv[])
{
    WORD srcx, srcy, dstx, dsty, bpp;
    DWORD Buf[20];
    int test;
    int i;
    img_info IMG_DATA;

    // Set 32 BPP colour depth.
    BPPOverride = 32;

    // First, run StartUp function to set up the application
    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);

    // Set up the source data co-ordinates.
    srcx = 0;
    srcy = Radeon_AdapterInfo.yres;
    bpp = Radeon_GetBPPValue (Radeon_AdapterInfo.bpp);

    // Load the source image file into offscreen memory.
    IMG_DATA = Load_Image (TRAJECTORY_RECTANGULAR, bpp, srcx, srcy);
```

```

// Initialize the CCE microengine.
if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
{
    Radeon_ShutDown ();
    printf ("Radeon_CPInit failed!!\n");
    exit (1);
} // if

// Do a transparent blt
i = 0;

// Set up a rectangle packet and fill the header with packet size
// Note that packet size is two less than total number of packets sent
Buf[i++] = CCE_PACKET3_CNTL_TRANS_BITBLT;

// Compose GUI_CONTROL
Buf[i++] = CCE_GC_BRUSH_NONE | CCE_GC_SRC_DSTCOLOR | ROP3_SRC_COPY
          | CCE_GC_DP_SRC_RECT
          | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);
dstx = 0;
dsty = 0;

// Fill rectangle data into packet
Buf[i++] = 0x01000004;
Buf[i++] = Radeon_GetColourCode (LIGHTMAGENTA);
Buf[i++] = 0x00000000;
Buf[i++] = (srcx << 16) + srcy;
Buf[i++] = (dstx << 16) + dsty;
Buf[i++] = (IMG_DATA.width << 16) + IMG_DATA.height;

Buf[0] |= ((i - 2) << 16);
test = Radeon_CPSubmitPackets (Buf, i);

getch ();

// Shut down the microengine.
Radeon_CPEnd (CCE_END_WAIT);
Radeon_ShutDown ();

return;
} // Radeon_Tblt

```

6.5 Drawing TextDrawing Text

Text is composed of a number of words which in turn is composed of characters. A character is represented by its raster image, and normally stored as a monochrome bitmap. RADEON supports printing characters whose raster images are stored in the format of

bit-packed monochrome bitmaps. This format is illustrated by the example shown in [Figure 6-7](#).

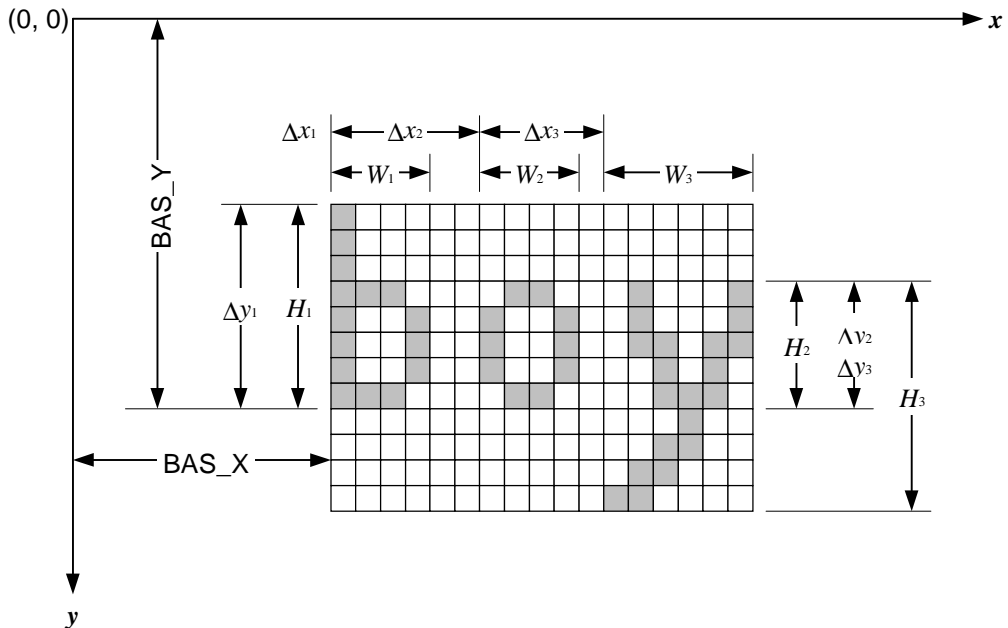


Figure 6-7. Parameters of Text

In this figure, the raster images of characters ‘b’, ‘o’ and ‘y’ are represented by 4x8, 4x5 and 6x9 arrays, respectively. Each cell of array is represented by a bit. If the cells in the array are scanned from left to right and top to bottom, and each cell is marked with a ordinal number according to its precedence in the scanning, the cells would form a queue.

The first eight cells (bits) are taken from the queue and are placed into the first byte of an array (referred to as the bitmap). The first cell is at the most significant bit and the 8th cell is at the least significant bit. Then, the next eight cells are taken from the queue and placed into the second byte of the bitmap in a same manner. This process is repeated until all the cells in the queue are taken out and placed into the bitmap. It is not necessary that the number of cells in an array must be a multiple of eight. This means that remaining bits of the last byte in the bitmap are undefined, and normally filled with 0’s. The monochrome bitmap created in this manner is said to be in the bit-packed format.

If the black cells in the arrays are coded as 1's, and the white cells are coded as 0's, the bit-packed codes for the raster images of characters in this example will be:

- 0x88, 0x8E, 0x99 and 0x9E for character 'b'.
- 0x69, 0x99 and 0x60 for 'o'.
- 0x45, 0x16, 0x0CA, 0x38, 0x43, 0x18 and 0x0C0 for 'y'.

To print the word "boy", specify the reference location of the text. For this example, this reference location is given by coordinates (*bas_x*, *bas_y*). In addition, specify the space between two adjacent bitmaps. These are denoted as Δx_i and Δy_i . Note that the values of Δx_i and Δy_i can be negative as they stand for deviations from a reference coordinate. For the case of [Figure 6-7](#), these parameters are:

- $H_1 = 8$, $W_1 = 4$, $\Delta x_1 = 0$, and $\Delta y_1 = 8$
- $H_2 = 5$, $W_2 = 4$, $\Delta x_2 = 6$, and $\Delta y_2 = 5$
- $H_3 = 9$, $W_3 = 6$, $\Delta x_3 = 5$, and $\Delta y_3 = 5$

The bitmap of a character may be categorized into two types: *Large Glyph* or *Small Glyph* according to its size. The difference between the two is the data type used to represent the location, dimensions, and the bitmap size of a category. This will be shown in the following description.

6.5.1 Drawing Text in Small Font

When both the height and width of a bitmap are limited to 255 pixels, the bitmap is stored in the format of Small Glyph. Therefore, each dimension can be represented by one byte. Now, use the packet **SMALL_TEXT** to print the text in [Figure 6-7](#) on the screen with the following setup parameters:

- The type of the destination pixels is aRGB.
- The type of the source pixels is monochrome with random foreground color (background color is the destination pixel color).
- No brush is selected.
- The color of the brush is not applicable.
- The source pixels are loaded from the host system memory (the data comes with the packet).
- The pixels are drawn from left to right.
- The source-clipping rectangle is not applicable.
- The destination-clipping rectangle is required.

- Use the default source pitch and offset.
- Use the default destination pitch and offset.
- The raster operation type is Source Copy (code 0xCC).
- The location and dimension of the source and destination are given in the sample program.

Example Code: Drawing text in small font

```
#define CCE_PACKET3_CNTL_SMALLTEXT 0xC0009300
DWORD Raster_ATI[6] = { 0xFF01FF01, 0xEEC1FF41, 0xE311E701,
                        0xE6CFED9D, 0xA073E0E7, 0x00008010 };
DWORD Raster_R[3] = { 0xC6C3C6FC, 0xC3C3C6FC, 0x0000C3C3 };
DWORD Raster_a[2] = { 0x6CF619F8, 0x0000C0CF };
DWORD Raster_d[3] = { 0x6730181C, 0xCC66B3D9, 0x000000FC };
DWORD Raster_e[2] = { 0xC1FE3C7B, 0x000000E0 };
DWORD Raster_o[2] = { 0x3C1E8F7D, 0x0000806F };
DWORD Raster_n[2] = { 0x369BCDDC, 0x0000C06C };

typedef struct tagSmallGlyph {
    BYTE dx, dy, w, h;
    int num_dwords;
    DWORD *raster;
} SmallGlyph;

SmallGlyph text[] = { { 0, 10, 17, 10, 6, Raster_ATI },
                      { 24, 10, 8, 10, 3, Raster_R },
                      { 10, 6, 7, 6, 2, Raster_a },
                      { 8, 10, 7, 10, 3, Raster_d },
                      { 8, 6, 6, 6, 2, Raster_e },
                      { 7, 6, 7, 6, 2, Raster_o },
                      { 8, 6, 7, 6, 2, Raster_n } };

void Radeon_smalltxt(int argc, char *argv[])
{
    DWORD Buf[50];
    DWORD x, y;
    int test;
    int i, j, k;

    // First, run StartUp function to set up the application
    Radeon_StartUp (argc, argv);

    // Clear the screen
    Radeon_ClearScreen (BLACK);

    // Initialize the CCE microengine.
    if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
```

```

{
    Radeon_ShutDown ();
    printf ("Radeon_CPInit failed!!\n");
    exit (1);
} // if

while (!kbhit ())
{
    i = 0;

    // Set up a rectangle packet and fill the header with packet size
    // Note that packet size is two less than total number of packets sent
    Buf[i++] = CCE_PACKET3_CNTL_SMALLTEXT;

    // Compose GUI_CONTROL
    Buf[i++] = CCE_GC_BRUSH_NONE | CCE_GC_SRC_MONO_LBKGD
              | ROP3_SRC_COPY | CCE_GC_DP_SRC_HOST
              | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

    Buf[i++] = Radeon_GetColourCode (rand () % NUM_COLOURS);

    x = (rand () % (Radeon_AdapterInfo.xres - 90));
    y = (rand () % (Radeon_AdapterInfo.yres - 10)) + 10;

    // Fill rectangle data into packet
    Buf[i++] = x + (y << 16);

    for (j = 0; j < (sizeof (text) / sizeof (SmallGlyph)); j += 1)
    {
        Buf[i++] = text[j].dx + (text[j].dy << 8) +
                  (text[j].w << 16) + (text[j].h << 24);
        for (k = 0; k < text[j].num_dwords; k += 1)
        {
            Buf[i++] = text[j].raster[k];
        } // for
    } // for

    Buf[0] |= ((i - 2) << 16);
    test = Radeon_CPSubmitPackets (Buf, i);

} // while

getch ();

// Shut down the microengine.
Radeon_CPEnd (CCE_END_WAIT);
Radeon_ShutDown ();

return;

```



```
} // Radeon_smalltxt
```

6.5.2 Drawing Text in Large Font

The format Large Glyph is defined for the bitmap whose height and width may exceed the limit of 255 pixels but are less than 65,535 pixels. The height and width of such a bitmap can be represented by 2 bytes (i.e., a 16-bit word).

Use packet **HOSTDATA_BLT** to print the text in [Figure 6-7](#). The parameter representation of this packet is slightly different from packet **SMALL_TEXT** in that it requires the coordinate of the left-top corner of each character, instead of the coordinate of text and delta values of each character.

The setup parameters for this drawing is the same as those in [“Drawing Text in Small Font” on page 6-22](#), except for the source type which requires both the foreground and background colors to be supplied in the packet.

Example Code: Drawing text in large font

```
#define CCE_PACKET3_CNTL_HOSTDATA_BLT    0xC0009400
DWORD Raster_ATI[6] = { 0xFF01FF01, 0xEEC1FF41, 0xE311E701,
                        0xE6CFED9D, 0xA073E0E7, 0x00008010 };
DWORD Raster_R[3] = { 0xC6C3C6FC, 0xC3C3C6FC, 0x0000C3C3 };
DWORD Raster_a[2] = { 0x6CF619F8, 0x0000C0CF };
DWORD Raster_d[3] = { 0x6730181C, 0xCC66B3D9, 0x000000FC };
DWORD Raster_e[2] = { 0xC1FE3C7B, 0x000000E0 };
DWORD Raster_o[2] = { 0x3C1E8F7D, 0x0000806F };
DWORD Raster_n[2] = { 0x369BCDDC, 0x0000C06C };

typedef struct tagLargeGlyph {
    WORD dx, dy, w, h;
    DWORD num_dwords;
    DWORD *raster;
} LargeGlyph;
LargeGlyph text[] = { { 0, 10, 17, 10, 6, Raster_ATI },
                      { 24, 10, 8, 10, 3, Raster_R },
                      { 34, 14, 7, 6, 2, Raster_a },
                      { 42, 10, 7, 10, 3, Raster_d },
                      { 50, 14, 6, 6, 2, Raster_e },
                      { 57, 14, 7, 6, 2, Raster_o },
                      { 65, 14, 7, 6, 2, Raster_n } };

void Radeon_Hostblt (int argc, char *argv[])
{
    DWORD Buf[50];
    DWORD x, y;
```

```

int test;
int i, j, k,n;

// First, run StartUp function to set up the application
Radeon_StartUp (argc, argv);

// Clear the screen
Radeon_ClearScreen (BLACK);

// Initialize the CCE microengine.
if (Radeon_CPInit (CSQ_MODE_PRIPIO_INDPIO) != CCE_SUCCESS)
{
    Radeon_ShutDown ();
    printf ("Radeon_CPInit failed!!\n");
    exit (1);
} // if

while (!kbhit ())
{
    i = 0;

    // Set up a rectangle packet and fill the header with packet size
    // Note that packet size is two less than total number of packets sent

    Buf[i++] = CCE_PACKET3_CNTL_HOSTDATA_BLT;

    // Compose GUI_CONTROL
    Buf[i++] = CCE_GC_BRUSH_NONE | CCE_GC_SRC_MONO_LBKGD
              | ROP3_SRC_COPY | CCE_GC_DP_SRC_HOST
              | (Radeon_GetBPPValue (Radeon_AdapterInfo.bpp) << 8);

    Buf[i++] = Radeon_GetColourCode (rand () % NUM_COLOURS);
    Buf[i++] = Radeon_GetColourCode (rand () % NUM_COLOURS);

    x = (rand () % Radeon_AdapterInfo.xres);
    y = (rand () % Radeon_AdapterInfo.yres);

    // Fill rectangle data into packet
    for (j = 0; j < (sizeof (text) / sizeof (LargeGlyph)); j += 1)
    {
        Buf[i++] = text[j].dx + x | ((text[j].dy + y) << 16);
        Buf[i++] = text[j].w | (text[j].h << 16);
        Buf[i++] = text[j].num_dwords;
        for (k = 0; k < text[j].num_dwords; k += 1)
        {
            Buf[i++] = text[j].raster[k];
        } // for
    } // for
}

```

```
Buf[0] |= ((i - 2) << 16);
test = Radeon_CPSubmitPackets (Buf, i);

// Slow things down a little so that the text can be read.
Radeon_Delay (1);
} // while

getch ();

// Shut down the microengine.
Radeon_CPEnd (CCE_END_WAIT);
Radeon_ShutDown ();

return;
} // Radeon_Hostblt
```

This page intentionally left blank.

Chapter 7

3D Programming

7.1 Scope

This chapter describes how to program the RADEON to set render states and draw 3D primitives, with or without TCL engine. It is assumed that you are familiar with 3D rendering concepts, so this text will not present an in-depth tutorial. Instead, it will focus on the implementation details.

7.2 3D Context Setup And Initialization

Prior to performing any of 3D operations, configure RADEON into a predefined 3D context.

1. Set render target.
2. Create and load depth/stencil buffer.
3. Set default rendering states.
4. If TCL engine is used, initialize TCL engine is described in *“TCL (Transform/Clip/Lighting) Engine” on page 7-40*.

Note: TCL is not supported in M6 and RV100. TCL_BYPASS@SE_CNTL_STATUS should be set to 1 for 3D functions to work.

The RADEON registers may be loaded through Type-0 CCE packets as shown below:

```
int i = 0;
DWORD Buf[BUF_SIZE];

Buf[i++] = CCE_PACKET0 | (register_address >> 2);
Buf[i++] = register_content_0;
. . .
Buf[i++] = register_content_n;
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
```

An example of how to setup 3D context can be found in the **context.c** file in the **libs\3D** directory of RADEON DDK.

7.3 Setting Render Target

Render target is a memory area anywhere in the frame buffer into which 3D primitives are rendered. To setup render target:

1. Load render target offset into RB3D_COLOROFFSET register.
2. Set render target width and height into RE_WIDTH_HEIGHT register.
3. Set render target pitch into RB3D_COLORPITCH register.
4. Set render target surface format into RB3D_CNTL:COLORFORMAT field.

The target surface formats supported by the RADEON are:

Table 7-1 Target Surface Formats Supported by the RADEON

Format	Description
3	ARGB1555
4	RGB565
6	ARGB8888
7	RGB332
8	Y8
9	RGB8
11	YUV422 packed (VYUY)
12	YUV422 packed (YVYU)
14	aYUV444
15	ARGB4444

7.4 Drawing 3D Primitives

The RADEON can render the following types of primitives using three Type-3 packets **3D_DRAW_IMMD**, **3D_DRAW_VBUF**, and **3D_DRAW_INDX**:

- Points (point lists).
- Independent lines (line lists).
- Polylines (line strips).
- Independent triangles (triangle lists).
- Triangle fans.
- Triangle strips.

- Rectangle lists.

3D_DRAW_IMMD packet draws primitives specified by the vertices contained in the body of the packet, while **3D_DRAW_VBUF** and **3D_DRAW_INDX** packets render data stored in vertex buffers.

RADEON also supports two Type-3 packets **3D_RNDR_GEN_INDX_PRIM** and **3D_RNDR_GEN_PRIM** for compatibility with Rage 128.

7.4.1 Flexible Vertex Format

The RADEON supports flexible vertex format allowing application to tailor the vertex format according to its specific requirements. Vertex format is specified by setting appropriate bits in **SE_VTX_FMT** field of the primitive packet. The table below describes **SE_VTX_FMT** field format.

[SE_VTX_FMT] Vertex Format Fields		
Field Name	Bits	Description
VTX_W0_PRESENT	0	Primary Vertex W value is present (1 float)
VTX_FPCOLOR_PRESENT	1	Floating Point Diffuse Color is Present (3 floats: RGB)
VTX_FPALPHA_PRESENT	2	Floating Point Alpha is Present (1 float)
VTX_PKCOLOR_PRESENT	3	Packed (8,8,8,8) ARGB Diffuse is Present. This is mutually exclusive with FP_DIFFUSE_PRESENT and FP_ALPHA_PRESENT
VTX_FPSPEC_PRESENT	4	Floating Point Specular Color is Present (3 floats: RGB)
VTX_FPFOG_PRESENT	5	Floating Point Fog is Present (1 float)
VTX_PKSPEC_PRESENT	6	Packed (8,8,8,8) FRGB Specular is Present. This is mutually exclusive with FPSPEC_PRESENT and FPCOLOR_PRESENT
VTX_ST0_PRESENT	7	Texture coordinate set 0 S,T values are present (2 floats)
VTX_ST1_PRESENT	8	Texture coordinate set 1 S,T values are present (2 floats)
VTX_Q1_PRESENT	9	non-Rage128 mode: Texture coordinate set 1 Q value is present (1 float) Rage128 mode: Texture coordinate set 1 ooW value is present (1 float)
VTX_ST2_PRESENT	10	Texture coordinate set 2 S,T values are present (2 floats)
VTX_Q2_PRESENT	11	Texture coordinate set 2 Q value is present (1 float)

[SE_VTX_FMT] Vertex Format Fields		
Field Name	Bits	Description
VTX_ST3_PRESENT	12	Reserved
VTX_Q3_PRESENT	13	Reserved
VTX_Q0_PRESENT	14	Texture coordinate set 0 Q value is present (1 float)
VTX_BLND_WEIGHT_CNT	17:15	Number of vertex blend weights present (0 to 4 floats)
VTX_N0_PRESENT	18	Primary Vertex Normal is present (3 floats: XYZ)
VTX_XY1_PRESENT	27	Second Vertex X,Y is present (2 floats) for vertex blending
VTX_Z1_PRESENT	28	Second Vertex Z is present (1 float) for vertex blending
VTX_W1_PRESENT	29	Second Vertex W is present (1 float) for vertex blending
VTX_N1_PRESENT	30	Second Vertex Normal is present (3 floats: XYZ) for vertex blending
VTX_Z_PRESENT	31	Primary vertex Z value is present (1 float)

The sample vertex definitions and format descriptors can be found in the **vertfmt.h** file in the **include\3D** directory of RADEON DDK.

7.4.2 Setup Engine Fetcher Control

To draw a 3D primitive, the setup engine needs to be instructed of the type of primitive being drawn and the method of passing vertex data. This setup engine control information is specified by setting the appropriate bit fields in the **SE_VF_CNTL** field of the primitive drawing packet. The table below describes the **SE_VF_CNTL** field format.

[SE_VF_CNTL] Setup Engine Fetcher Control Fields		
Field Name	Bits	Description
PRIM_TYPE	3:0	<p>Primitive Type</p> <p>0 : None (will not trigger Setup Engine to run)</p> <p>1 : Point List</p> <p>2 : Line List</p> <p>3 : Line Strip</p> <p>4 : Triangle List</p> <p>5 : Triangle Fan</p> <p>6 : Triangle Strip</p> <p>7 : Triangle with wFlags (aka, Rage128 'Type-2' triangles)*</p> <p>8 : Rectangle List</p> <p>9 : 3-Vertex Point List</p> <p>10 : 3-Vertex Line List</p> <p>*Note: Encoding 7 indicates whether a 16-bit word of wFlags is present in the stream of indices arriving when the PRIM_WALK is programmed as a '1'. The Setup Engine just steps over the wFlags word; ignoring it.</p> <p>Encoding 7 : Stream contains indices and wFlags:</p> <p>[Index1, Index0]</p> <p>[wFlags, Index2]</p> <p>[Index4, Index3]</p> <p>[wFlags, Index5] etc...</p> <p>Other encodings : Stream contains just indices, as:</p> <p>[Index1, Index0]</p> <p>[Index3, Index2]</p> <p>[Index5, Index4] etc...</p>
PRIM_WALK	5:4	<p>Method of Passing Vertex Data</p> <p>0 : Reserved</p> <p>1 : Indexes (Indices embedded in command stream; vertex data to be fetched from memory)</p> <p>2 : Vertex List (Vertex data to be fetched from memory)</p> <p>3 : Vertex Data (Vertex data embedded in command stream)</p>
COLOR_ORDER	6	<p>If diffuse/specular colors are in floating point format:</p> <p>Order of arrival of floating point diffuse and specular color parameters.</p> <p>0 : BGRA</p> <p>1 : RGBA</p> <p>If diffuse/specular colors are packed:</p> <p>Order of storage of the color components in the dword.</p> <p>0 : BGRA (B=[7:0] , G=[15:8], R=[23:16], A=[31:24])</p> <p>1 : RGBA (R=[7:0] , G=[15:8], B=[23:16], A=[31:24])</p>

[SE_VF_CNTL] Setup Engine Fetcher Control Fields		
Field Name	Bits	Description
EN_MAOS	7	Enable Multiple Arrays of Structures. 1 : Enable all Rage6 features; no fields are overridden. 0 : Only enable the Rage128 subset of features; for compatibility. The following fields get overridden: VTX_NUM_ARRAYS <- 1 VTX_AOS0_COUNT <- Number of dwords per-vertex, as indicated by the Flexible Vertex Format VTX_AOS0_STRIDE <- Number of dwords per-vertex, as indicated by the Flexible Vertex Format
VTX_FMT_MODE	8	Vertex Format Mode. This bit enables the Rage6 extended features for processing vertices. 1 : Enable all Rage6 features; no fields are overridden. 0 : Only enable the Rage128 subset of features; for compatibility. The following fields get overridden: VTX_Z_PRESENT <- 1 VTX_ST2_PRESENT <- 0 VTX_W2_PRESENT <- 0 VTX_XY_FMT <- 1 VTX_Z_FMT <- 1 VTX_ST0_NONPARAMETRIC <- 0 VTX_ST1_NONPARAMETRIC <- 0 VTX_ST2_NONPARAMETRIC <- 0 VTX_W0_FMT <- 0 VTX_W1_FMT <- 0 VTX_W2_FMT <- 0 VPORT_XY_XFEN <- 0 VPORT_Z_XFEN <- 0
TCL_ENABLE	9	Enable the Transform/Clip/Light Engine 1 : Enable the TCL features 0 : Disable all TCL features
NUM_VERTICES	31:16	Number of vertices in the command packet.

7.4.3 Drawing Primitives With Immediate Vertices From CCE Packets

The **3D_DRAW_IMMD** packet, used for drawing primitives with vertex data supplied in the packet, consists of these fields:

3D_DRAW_IMMD Packet Fields		
Ordinal	Field Name	Description
1	[HEADER]	Header of the packet

3D_DRAW_IMMD Packet Fields		
Ordinal	Field Name	Description
2	[SE_VTX_FMT]	Vertex format
3	[SE_VF_CNTL]	Primitive type and other
4 to end	Vertex data	Up to 64K – 12 bytes of vertex data

The following information should be specified in **SE_VF_CNTL** packet field:

- **PRIM_TYPE** field should contain primitive type.
- **PRIM_WALK** field should be 3.
- **EN_MAOS** field should be 1.
- **VTX_FMT_MODE** field should be 1.
- **TCL_ENABLE** field is 1 when TCL engine is enabled.
- **NUM_VERTICES** field should contain number of vertices to be draw.

The number of vertices required varies depending on the primitive type and should be:

- Any number greater than zero for point lists.
- Divisible by 2 for line lists.
- At least 2 for line strips.
- Divisible by 3 for triangle lists.
- At least 3 for triangle fans and strips.
- Divisible by 3 for rectangle lists.

Maximum number of vertices is limited by the size of the vertex data of the packet that cannot be greater than 64K – 12 bytes.

The example of how to draw primitives using vertices stored in packets can be found in the **prim.c** file in the **libs\3D** directory of RADEON DDK.

7.4.4 Using Vertex Buffers

Vertex buffers are simply memory buffers placed in local video memory, AGP memory space or PCI GART space. Using vertex buffers for drawing 3D primitives is an ideal method for drawing reusable geometry since it eliminates the need to repeatedly send vertex data in the primitive packets.

Before using vertex buffer with primitive drawing packet, its information should be loaded into the RADEON, indicating where to fetch vertices from and how.

Loading Vertex Buffers

The **3D_LOAD_VBPNT** Type-3 packet is used to load vertex buffer pointers into the RADEON.

The vertices can be either loaded from vertex buffer containing monolithic vertex information, or from several vertex buffers containing separate vertex components. To facilitate that functionality the RADEON has an entity called Array Of Structures (AOS) that allows so setup up to 12 pointers to vertex components. Each element in the AOS has a structure element size in DWORDs and stride in DWORDs associated with it. The RADEON has no sense of vertex components and for each vertex it just sequentially scans each of the AOS components fetching structure element and advancing structure pointer by stride. For monolithic vertices the AOS will contain only one array with element size and stride equal vertex size in DWORDs. The maximum number of vertices in vertex buffer should not exceed 64K vertices. The order and format of vertex components should be:

Table 7-2

Component	Parameter	Description	Format
Position0 XY	X0	The x coordinate of the vertex	IEEE floating point *
	Y0	The y coordinate of the vertex	IEEE floating point *
Position0 Z	Z0	The z coordinate of the vertex	IEEE floating point *
Position0 W	W0	The w coordinate of the vertex	IEEE floating point
Blend Weight(s)	BW0 BW1 BW2 BW3	0-4 Skinning Blend Weights (for vertex blending)	IEEE floating point
Vertex Normal 0	Nx0	The x coordinate of the vertex normal	IEEE floating point
	Ny0	The y coordinate of the vertex normal	IEEE floating point
	Nz0	The z coordinate of the vertex normal	IEEE floating point
Diffuse	ARGB	Diffuse color and alpha weight	Usually 8888, but can be three or four separate IEEE floating point values **
Specular	ARGB	Specular color and fog weight	Usually 8888, but can be three or four separate IEEE floating point values **

Table 7-2 (Continued)

Component	Parameter	Description	Format
Texture Coordinate Set n	Sn	The 1st coordinate for texture number n (usually the single dimension horizontal component S)	IEEE floating point
n= 0,1,2,3	Tn	The 2nd coordinate for texture number n (usually the two dimension vertical component T)	IEEE floating point
	Qn	The 3rd coordinate for texture number n (usually the homogeneous W value)	IEEE floating point
Position1 XY	X1	The x coordinate of the vertex for blending	IEEE floating point *
	Y1	The y coordinate of the vertex for blending	IEEE floating point *
Position1 Z	Z1	The z coordinate of the vertex for blending	IEEE floating point *
Position1 W	W1	The w coordinate of the vertex for blending	IEEE floating point
Vertex Normal 1	Nx1	The x coordinate of the vertex normal	IEEE floating point
	Ny1	The y coordinate of the vertex normal	IEEE floating point
	Nz1	The z coordinate of the vertex normal	IEEE floating point

* After viewport transformation and perspective divide, the X and Y values are clamped so they stay in our representable range, from -2047.0 to +2047.0, inclusive.

** If the ARGB or FRGB values are in IEEE floating point format, the Setup Engine expects them to be in the range from 0.0 to +1.0 inclusive, in accordance with the OpenGL specification. As a safety measure, these color values will be clamped to ensure that they are in this range.

The **3D_LOAD_VBPNT** packet structure:

3D_LOAD_VBPNT Packet Fields		
Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	VTX_NUM_ARRAYS	Number of arrays in AOS
3	VTX_AOS_ATTR01	Control for the first two arrays
4	VTX_AOS_ADDR0	Pointer to first array
5	VTX_AOS_ADDR1	Pointer to second array
6	VTX_AOS_ATTR23	And so on....
7	VTX_AOS_ADDR2	

3D_LOAD_VBPNTN Packet Fields		
Ordinal	Field Name	Description
8	VTX_AOS_ADDR3	
9	VTX_AOS_ATTR45	
10	VTX_AOS_ADDR4	
11	VTX_AOS_ADDR5	
12	VTX_AOS_ATTR67	
13	VTX_AOS_ADDR6	
14	VTX_AOS_ADDR7	
15	VTX_AOS_ATTR89	
16	VTX_AOS_ADDR8	
17	VTX_AOS_ADDR9	
18	VTX_AOS_ATTR1011	
19	VTX_AOS_ADDR10	
20	VTX_AOS_ADDR11	

VTX_NUM_ARRAYS field stores number of arrays with vertex components.

VTX_AOS_ADDR0 - VTX_AOS_ADDR11 fields contain array offsets in the AGP or PCI GART memory.

Attribute fields **VTX_AOS_ATTR01 - VTX_AOS_ATTR1011** have the following structure:

SE_VTX_AOS_ATTR01		
Field Name	Bits	Description
VTX_AOS_COUNT0	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE0	13:8	Number of dwords from one array element to the next.
VTX_AOS_COUNT1	21:16	Number of dwords in this structure.
VTX_AOS_STRIDE1	29:24	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR23		
Field Name	Bits	Description
VTX_AOS_COUNT2	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE2	13:8	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR23		
Field Name	Bits	Description
VTX_AOS_COUNT3	21:16	Number of dwords in this structure.
VTX_AOS_STRIDE3	29:24	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR45		
Field Name	Bits	Description
VTX_AOS_COUNT4	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE4	13:8	Number of dwords from one array element to the next.
VTX_AOS_COUNT5	21:16	Number of dwords in this structure.
VTX_AOS_STRIDE5	29:24	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR67		
Field Name	Bits	Description
VTX_AOS_COUNT6	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE6	13:8	Number of dwords from one array element to the next.
VTX_AOS_COUNT7	21:16	Number of dwords in this structure.
VTX_AOS_STRIDE7	29:24	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR89		
Field Name	Bits	Description
VTX_AOS_COUNT8	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE8	13:8	Number of dwords from one array element to the next.
VTX_AOS_COUNT9	21:16	Number of dwords in this structure.
VTX_AOS_STRIDE9	29:24	Number of dwords from one array element to the next.

SE_VTX_AOS_ATTR1011		
Field Name	Bits	Description
VTX_AOS_COUNT10	5:0	Number of dwords in this structure.
VTX_AOS_STRIDE10	13:8	Number of dwords from one array element to the next.
VTX_AOS_COUNT11	21:16	Number of dwords in this structure.

SE_VTX_AOS_ATTR1011		
Field Name	Bits	Description
VTX_AOS_STRIDE11	29:24	Number of dwords from one array element to the next.

Examples of loading vertex buffers.

Assume the vertices contain following components: XYZ vertex coordinates, normal and diffuse color.

Loading monolithic vertices from single vertex buffer

In case of a monolithic vertices our vertex buffer contains:

Vertex 0 XYZ
 Vertex 0 Normal
 Vertex 0 Diffuse
 Vertex 1 XYZ
 Vertex 1 Normal
 Vertex 1 Diffuse
 ...
 Vertex n XYZ
 Vertex n Normal
 Vertex n Diffuse

The code to load this vertex buffer pointer will be:

```
int i = 0;
DWORD Buf[BUF_SIZE];

Buf[i++] = CCE_PACKET3_3D_LOAD_VBPNTNR;
Buf[i++] = 1; // Number of arrays
// Control info for the array - structure size and stride
Buf[i++] = size << SE_VTX_AOS_ATTR01__VTX_AOS_COUNT0__SHIFT |
           stride << SE_VTX_AOS_ATTR01__VTX_AOS_STRIDE0__SHIFT);
Buf[i++] = vb_offset; // Pointer to an array in AGP or PCI GART
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
```

Loading strided vertices from multiple vertex buffers

In this case vertex buffers contain:

Vertex 0 XYZ
 Vertex 1 XYZ
 ...

Vertex n XYZ

Vertex 0 Normal

Vertex 1 Normal

...

Vertex n Normal

Vertex 0 Diffuse

Vertex 1 Diffuse

...

Vertex n Diffuse

The code to load these vertex buffers will be:

```
int i = 0;
DWORD Buf[BUF_SIZE];

Buf[i++] = CCE_PACKET3_3D_LOAD_VBPNTNTR;
Buf[i++] = 3; // Number of arrays
Buf[i++] = xyz_size << SE_VTX_AOS_ATTR01_VTX_AOS_COUNT0__SHIFT |
           xyz_stride << SE_VTX_AOS_ATTR01_VTX_AOS_STRIDE0__SHIFT |
           normal_size << SE_VTX_AOS_ATTR01_VTX_AOS_COUNT1__SHIFT |
           normal_stride << SE_VTX_AOS_ATTR01_VTX_AOS_STRIDE1__SHIFT);
Buf[i++] = xyz_vb_offset; // Pointer to an xyz in AGP or PCI GART
Buf[i++] = normal_vb_offset; // Pointer to an normals in AGP or PCI GART
Buf[i++] = diffuse_size << SE_VTX_AOS_ATTR23_VTX_AOS_COUNT2__SHIFT |
           diffuse_stride << SE_VTX_AOS_ATTR23_VTX_AOS_STRIDE2__SHIFT);
Buf[i++] = diffuse_vb_offset; // Pointer to an diffuse in AGP or PCI GART
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
```

For more examples of how to load vertex buffers see the **prim.c** file in the **libs\3D** directory of RADEON DDK.

Drawing Primitives With Vertex Buffers

The **3D_DRAW_VBUF** packet, used for drawing primitives with vertices fetched from vertex buffer pointed to by state data, consists of these fields:

3D_DRAW_VBUF Packet Fields		
Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[SE_VTX_FMT]	Vertex format
3	[SE_VF_CNTL]	Primitive type and other

The following information should be specified in **SE_VF_CNTL** packet field:

- **PRIM_TYPE** field should contain primitive type.
- **PRIM_WALK** field should be 2.
- **EN_MAO**s field should be 1.
- **VTX_FMT_MODE** field should be 1.
- **TCL_ENABLE** field is 1 when TCL engine is enabled.
- **NUM_VERTICES** field should contain number of vertices to be draw.

The number of vertices required varies depending on the primitive type and should be:

- Any number greater than zero for point lists.
- Divisible by 2 for line lists.
- At least 2 for line strips.
- Divisible by 3 for triangle lists.
- At least 3 for triangle fans and strips.
- Divisible by 3 for rectangle lists.

The maximum number of vertices in vertex buffer should not exceed 64K vertices.

The example of how to draw primitives from vertex buffers can be found in the **prim.c** file in the **libs\3D** directory of RADEON DDK.

Drawing Indexed Primitives With Vertex Buffers

The **3D_DRAW_INDX** packet allows drawing 3D primitives with vertices fetched from vertex buffer and indexed with indices stored in the primitive packet.

3D_DRAW_INDX Packet Fields		
Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[SE_VTX_FMT]	Vertex format
3	[SE_VF_CNTL]	Primitive type and other
4 to end	Index data	Up to 32K – 6 WORD size indices

Following information should be specified in **SE_VF_CNTL** packet field:

- **PRIM_TYPE** field should contain primitive type.
- **PRIM_WALK** field should be 1.
- **EN_MAOS** field should be 1.
- **VTX_FMT_MODE** field should be 1.
- **TCL_ENABLE** field is 1 when TCL engine is enabled.
- **NUM_VERTICES** field should contain number of vertices to be draw.

Indices are packed into DWORDs. The number of indices required varies depending on the primitive type and should be:

- Any number greater than zero for point lists.
- Divisible by 2 for line lists.
- At least 2 for line strips.
- Divisible by 3 for triangle lists.
- At least 3 for triangle fans and strips.
- Divisible by 3 for rectangle lists.

Maximum number of indices is limited by the size of the index data of the packet that cannot be greater than 32K – 6 indices.

The example of how to draw indexed primitives from vertex buffers can be found in the **prim.c** file in the **libs\3D** directory of RADEON DDK.

7.5 Depth and Stencil Buffers

The RADEON has support for both depth and stencil buffers. Depth buffer testing is used for hidden surface removal. The Z or W values for source primitives are compared against the Z or W values for destination pixels stored in a depth buffer, and a decision is made to accept or reject, or occlude the source pixel.

The stencil buffer is an auxiliary buffer used for performing special pixel by pixel operations. It may be used to stencil out specific shapes for operations such as applying shadow tones or masking out drawing regions. The RADEON supports an eight-bit stencil buffer. The stencil buffer is interleaved with a 24-bit depth buffer in a combined 32-bit buffer. The stencil buffer occupies the upper eight bits, and the depth buffer occupies the lower 24 bits.

The examples of how to setup and use depth and stencil buffers can be found in the **zbuffer.c** file in the **libs\3D** directory of RADEON DDK.

7.5.1 Creating Depth/Stencil Buffer

Depth buffer can be allocated anywhere in the frame buffer and pointed to by the RADEON state data.

1. Allocate depth buffer in the frame buffer
2. Load depth buffer offset into **RB3D_DEPTHOFFSET** register.
3. Load depth buffer format into **RB3D_ZSTENCILCNTL:DEPTHFORMAT** field.

RADEON supports following depth buffer formats:

Table 7-3 Supported Depth Buffer Formats

Format	Description
0	16-bit Integer Z
2	24-bit Integer Z
3	24-bit Floating Point Z
4	32-bit Integer Z
5	32-bit Floating Point Z
7	16-bit Floating Point W
9	24-bit Floating Point W
11	32-bit Floating Point W

The depth of the depth buffer need not be the same as the depth of the drawing surface. For instance, it is possible to use a 32-bit z-buffer with an RGB 565 drawing surface.

The depth buffer format for integer modes is self-explanatory. In 32-bit floating point format the depth values are stored as normal IEEE floating point numbers. In 24-bit floating point format the numbers resemble IEEE floating point numbers with the difference of 8 LSBs being dropped. The 16-bit floating point format is not IEEE floating point based. It has a 4-bit signed/biased exponent and a 12-bit unsigned mantissa.

7.5.2 Depth Buffer Operation

The z depth test is performed according to the following formula:

Decision = **DepthTestFunction** (SourceDepth, DestinationDepth)

The depth test function is set through the **RB3D_ZSTENCILCNTL:ZFUNC** field. It may be set to the following states:

Table 7-4 Depth Buffer Operation

Format	Description
0	Never
1	Less
2	Less or Equal
3	Equal
4	Greater or Equal
5	Greater Than
6	Not Equal
7	Always

The action to take following the depth test with respect to updating the depth buffer is controlled through the **RB3D_ZSTENCILCNTL:ZWRITEENABLE** field.

- '0' disables writes to the z buffer.
- '1' enables z writes.

Z testing is enabled and disabled through the **RB3D_CNTL:Z_ENABLE** field.

- '0' disables z testing
- '1' enables z testing.

7.5.3 Stencil Buffer Operation

The stencil buffer operates according to the following equation:

StencilCompareFunction ((StencilReference AND StencilMask), (StencilValue AND StencilMask))

The stencil compare function is selected by setting **RB3D_ZSTENCILCNTL:STENCILFUNC** field to one of the following values:

Table 7-5

Format	Description
0	Never
1	Less
2	Less or Equal

Table 7-5 (Continued)

Format	Description
3	Equal
4	Greater or Equal
5	Greater Than
6	Not Equal
7	Always

Different actions may be prescribed with respect to updating the stencil buffer based on a number of pass/fail criteria. Specifically, a set of stencil operations may be set based on whether the stencil test fails, both the stencil and z tests pass, or the stencil test passes but the z test fails.

These operations may be set by writing the following fields:

RB3D_ZSTENCILCNTL:STENCILFAIL
RB3D_ZSTENCILCNTL:STENCILZFAIL
RB3D_ZSTENCILCNTL:STENCILZPASS

Each may be set to one of the following states:

Table 7-6

Format	Description
0	Keep
1	Zero
2	Replace (write STENCILREF)
3	Increment
4	Decrement
5	Invert

Reference value for stencil buffer operations can be set by writing into **RB3D_STENCILREFMASK:STENCILREF** field.

Data written back to the stencil buffer is masked by the stencil write mask through the **RB3D_STENCILREFMASK:STENCILWRITEMASK** field.

Stencil testing is enabled and disabled through the **RB3D_CNTL:STENCIL_ENABLE** field.

- '0' disables stencil testing
- '1' enables stencil testing.

7.6 Setting 3D Render States

This section describes how to set rasterization render states for the 3D pipeline on the RADEON. The registers presented here may be modified through Type-0 CCE packets, as demonstrated by the following code:

```
int i = 0;
DWORD Buf[BUF_SIZE];

Buf[i++] = CCE_PACKET0 | (register_address >> 2);
Buf[i++] = register_content;
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
```

7.6.1 Alpha Blending

Alpha blending allows the source primitive color data to be combined with the destination color data in various ways to achieve special effects like translucency.

Alpha blending is enabled or disabled by setting **RB3D_CNTL:ALPHA_BLEND_ENABLE** field.

- '0' disables alpha blending
- '1' enables alpha blending

The alpha blend equation is:

PixelColor = **CombineFunc** (**SrcBlendFactor**(SrcData), **DestBlendingFactor**(dstData))

The source and destination alpha blending factors are set through the **RB3D_BLEND_CNTL:SRCBLEND** and **RB3D_BLEND_CNTL:DESTBLEND** fields. The following factors may be set:

Source blend factors:**Table 7-7 Source Blend Factors**

Source Blend Factor	Description
1	ZERO (Direct3D)
2	ONE (Direct3D)
3	SRCCOLOR (Direct3D)
4	INVSRCOLOR (Direct3D)
5	SRCALPHA (Direct3D)
6	INVSRCALPHA (Direct3D)
7	DESTALPHA (Direct3D)
8	INVDESTALPHA (Direct3D)
9	DESTCOLOR (Direct3D)
10	INVDESTCOLOR (Direct3D)
11	SRCALPHASAT (Direct3D)
12	BOTHSRCALPHA (Direct3D)
13	BOTHINVSRCALPHA (Direct3D)
32	ZERO (OpenGL)
33	ONE (OpenGL)
34	SRC_COLOR (OpenGL)
35	ONE_MINUS_SRC_COLOR (OpenGL)
36	DST_COLOR (OpenGL)
37	ONE_MINUS_DST_COLOR (OpenGL)
38	SRC_ALPHA (OpenGL)
39	ONE_MINUS_SRC_ALPHA (OpenGL)
40	DST_ALPHA (OpenGL)
41	ONE_MINUS_DST_ALPHA (OpenGL)
42	SRC_ALPHA_SATURATE (OpenGL)

Destination blend factors:**Table 7-8 Destination Blend Factors**

Destination Blend Factor	Description
1	ZERO (Direct3D)

Table 7-8 Destination Blend Factors (Continued)

Destination Blend Factor	Description
2	ONE (Direct3D)
3	SRCCOLOR (Direct3D)
4	INVSRCOLOR (Direct3D)
5	SRCALPHA (Direct3D)
6	INVSRCALPHA (Direct3D)
7	DESTALPHA (Direct3D)
8	INVDESTALPHA (Direct3D)
9	DESTCOLOR (Direct3D)
10	INVDESTCOLOR (Direct3D)
32	ZERO (OpenGL)
33	ONE (OpenGL)
34	SRC_COLOR (OpenGL)
35	ONE_MINUS_SRC_COLOR (OpenGL)
36	DST_COLOR (OpenGL)
37	ONE_MINUS_DST_COLOR (OpenGL)
38	SRC_ALPHA (OpenGL)
39	ONE_MINUS_SRC_ALPHA (OpenGL)
40	DST_ALPHA (OpenGL)
41	ONE_MINUS_DST_ALPHA (OpenGL)

Combine Function allows modification of how the SRCBLEND and DESTBLEND are combined. Alpha combining function can be set through the **RB3D_BLENDCTL:COMB_FCN** field. The following alpha-combination functions may be set:

Table 7-9

Combine Function	Description
0	Add and Clamp
1	Add but no Clamp
2	Subtract Dst from Src, and Clamp
3	Subtract Dst from Src, and don't Clamp

The examples of how to set alpha blending states can be found in the **blend.c** file in the **libs\3D** directory of RADEON DDK.

7.6.2 Alpha Testing

Alpha testing allows a pixel to be rejected based on a comparison of its alpha value to a reference alpha value.

The pass/fail decision is represented by the following formula:

Decision = **AlphaTestOperation** (Source Alpha, ReferenceAlpha))

The alpha reference is an 8-bit value ranging from zero to 255. It is set by writing to the **PP_MISC:REF_ALPHA** field.

The alpha test function is set through the **PP_MISC:ALPHA_TEST_OP** field. The following states may be set:

Table 7-10

Combine Function	Description
0	Always Fail
1	Src Alpha < Ref Alpha
2	Src Alpha <= Ref Alpha
3	Src Alpha == Ref Alpha
4	Src Alpha >= Ref Alpha
5	Src Alpha > Ref Alpha
6	Src Alpha != Ref Alpha
7	Always Pass

Alpha testing is enabled or disabled by setting **PP_CNTL:ALPHA_TEST_ENABLE** field.

- '0' disables alpha testing
- '1' enables alpha testing

7.6.3 Culling

Culling allows one to specify the triangles that are rendered based on their orientation. It is frequently used to eliminate back facing triangles. The culling capabilities of the RADEON are configured through the **SE_CNTL** register.

The **SE_CNTL:FFACE_CULL_DIR** field specifies triangle orientation:

- ‘0’ Front facing triangles are clockwise
- ‘1’ Front facing triangles are counter-clockwise

Fields **SE_CNTL:FFACE_CULL_FCN** and **SE_CNTL:BFACE_CULL_FCN** specify front-facing and back-facing triangle culling. Following values are allowed:

Table 7-11

Culling Function	Description
0	Don't draw triangles
3	Draw triangles

7.6.4 Dithering and Antialiasing

Dithering is a technique for reducing the banding artifacts that may appear when using a limited number of colors. Dithering is typically necessary when using 16-bpp and smaller display modes, such as RGB565, RGB1555, etc.

Dithering can be enabled by setting **RB3D_CNTL:DITHER_ENABLE** field:

- ‘0’ disables dithering
- ‘1’ enables dithering

Antialiasing is a technique that can be used to reduce line “jagginess” – the stair-step effect visible in any line that is not exactly horizontal or vertical. Antialiasing blends the pixels at the primitive boundaries to give rendered objects a more natural look. The RADEON supports both line and polygon antialiasing by modulating source alpha by the pixel area coverage prior to the SRC/DEST alpha blender. To set antialiasing write to the **PP_CNTL:ANTI_ALIAS_CTL** one of the following values:

Table 7-12

Culling Function	Description
0	No antialiasing
1	Line antialiasing
2	Polygon antialiasing
3	Both line and polygon antialiasing

The samples showing use of the dithering and antialiasing can be found in **raster.c** file located in **lib\3D** directory and also in **renderst.c** sample located in **chap7\renderst** directory of the RADEON DDK.

7.6.5 Fog

The fog is implemented by blending the color of 3D objects with a chosen fog color based on the depth of an object in a scene, or its distance from the “eye”.

Fog blending is performed according to the following equation:

Final color = fogFactor x Src + (1 – fogFactor) x fogColor

The RADEON uses the interpolated alpha component of the vertex specular color as the fog factor at each pixel. This vertex fog factor can be automatically computed by TCL engine when using TCL transformation pipeline. For more information on how to setup vertex fog processing with TCL engine refer to section [“Vertex Fog” on page 7-75](#).

Fog blending can be enabled or disabled by manipulating **PP_CNTL:FOG_ENABLE** field:

- ‘0’ disables fog blending
- ‘1’ enables fog blending

The fog color can be set in **PP_FOG_COLOR:FOG_COLOR** register in RGB888 format.

For examples on how to set fog blending refer to **fog.c** file located in **lib\3D** directory of the RADEON DDK.

7.6.6 Raster Operations

Raster operation (ROP) is a bit-wise operation that defines how the color data of the drawn primitive is to be combined with the destination color data.

Raster operations can be enabled by setting **RB3D_CNTL:ROP_ENABLE** field:

- ‘0’ disables raster operation for 3D primitives
- ‘1’ enables raster operation for 3D primitives

Following raster operation can be selected by writing to **RB3D_ROPCNTL:ROP** field:

Table 7-13

ROP	Description
0	Clear
1	NOR
2	AND inverted
3	Copy inverted
4	AND reverse
5	Invert
6	XOR
7	NAND
8	AND
9	Equivalent
10	No-op
11	OR inverted
12	Copy
13	OR reverse
14	OR
15	Set

The examples on how to set fog blending can be found in **raster.c** file located in **lib\3D** directory of the RADEON DDK.

7.6.7 Scissor Testing

Scissor testing allows to setup a rectangle area in screen coordinates called scissor box that occludes all pixels of drawn primitives that lie outside of that area.

The scissor box is defined by top-left and bottom-right corners specified in screen coordinates in **RE_TOP_LEFT** and **RE_WIDTH_HEIGHT** registers. Please, note the latter register specifies scissor box corner coordinates and not its dimensions.

Scissor box testing can be enabled and disabled by setting **PP_CNTL:SCISSOR_ENABLE** field:

- '0' disables scissor testing
- '1' enables scissor testing

The examples on how to work with scissor testing can be found in **scissor.c** file located in **lib\3D** directory of the RADEON DDK.

7.6.8 Shading

The shading mode determines the color or colors used to render the primitive and how the colors are applied. In shading model supported by RADEON, different shading modes can be separately applied to diffuse color RGB component, alpha component, specular color and fog factor by writing to **SE_CNTL:DIFFUSE_SHADE_FCN**, **SE_CNTL:ALPHA_SHADE_FCN**, **SE_CNTL:SPECULAR_SHADE_FCN** and **SE_CNTL:FOG_SHADE_FCN** fields appropriately. The following shading modes can be applied:

Table 7-14

Shade mode	Description
0	Solid shading
1	Flat shading
2	Gouraud shading

If solid shading mode is selected, the color components are obtained from color stored in **RE_SOLID_COLOR** register. In case of a flat shading RADEON can use color from one of the polygon vertices. Setting **SE_CNTL:FLAT_SHADE_VTX** field instructs which of the polygon vertices is used for shading:

Table 7-15

Shading vertex	Description
0	Use vertex 0
1	Use vertex 1
2	Use vertex 2
3	Use last vertex sent

The example of setting polygon shading can be found in **raster.c** file located in **lib\3D** directory of the RADEON DDK.

7.6.9 Specular Color

Specular color can be enabled or disabled in RADEON rasterizer by setting **PP_CNTL:SPECULAR_ENABLE** field:

- '0' disables specular color add
- '1' enables specular color add

7.6.10 Stipple Patterns

The RADEON has the ability to draw patterned primitives by masking out certain pixels in the rasterizer as defined by bit mask. There are two types of stipple patterns that can be used with different primitives. Line stippling works only with line primitives, while polygon stippling works with both lines and polygons.

Line stippling is defined by three parameters: 16-bit line stipple pattern, pixel repeat factor and pattern start. Rasterizer continually scans 16-bit line pattern and draws line segments for bits set to 1, while masking out line segments correlating to bits set to 0. The length of each line segment is defined in pixels by pixel repeat factor. Setting pixel repeat factor to zero will result in line segments 256 pixels long. Line stipple patterns can be set in **RE_LINE_PATTERN:LINE_PATTERN** field, while pixel repeat factor and pattern start can be set by writing to **RE_LINE_PATTERN:REPEAT_COUNT** and **RE_LINE_PATTERN:PATTERN_START** fields correspondingly.

The RADEON can be instructed to reset pattern counter after each primitive by setting **RE_LINE_PATTERN:AUTO_RESET_ENABLE**:

- '0' don't reset line pattern counter
- '1' reset line pattern counter after each primitive

Line stippling can be enabled and disabled by controlling **PP_CNTL:PATTERN_ENABLE** field:

- '0' disables line stippling
- '1' enables line stippling

Polygon stipple pattern is defined as 32x32 bit mask applied on a per-pixel basis to rasterized primitives. The polygon stipple pattern is loaded through **RE_STIPPLE_ADDR** and **RE_STIPPLE_DATA** registers. **RE_STIPPLE_ADDR** register defines a 5-bit index into pattern memory area. It is self-incremented with every access to **RE_STIPPLE_DATA** pattern data register. When updating multiple DWORDS, the CCE packet bit which prevents auto-incrementation should be used so that

all words are written to the data register. Polygon stipple pattern loading with Type-0 CCE packets would look like:

```
int j, i;
DWORD Buf[BUF_SIZE];

// Load pattern memory access counter
i = 0;
Buf[i++] = CCE_PACKET0 | (RE_STIPPLE_ADDR >> 2);
Buf[i++] = 0x00000000; // Initial counter value
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
// Load stipple pattern
i = 0;
Buf[i++] = CCE_PACKET0 | CCE_PACKET_0_ONE_REG_WR | (RE_STIPPLE_DATA >> 2);
for (j = 0; j < 32; j++)
    Buf[i++] = *pattern++;
Buf[0] |= ((i - 2) << 16);
Radeon_CPSubmitPackets (Buf, i);
```

Polygon stipple initial offset is controlled by **RE_MISC:STIPPLE_X_OFFSET** and **RE_MISC:STIPPLE_Y_OFFSET** fields.

Polygon stipple pattern can be enabled and disabled by setting **PP_CNTL:STIPPLE_ENABLE** field:

- '0' disables polygon stippling
- '1' enables polygon stippling

The example of setting up stipple patterns can be found in **raster.c** file located in **lib\3D** directory of the RADEON DDK.

7.6.11 Wide Lines

The line primitives drawn by RADEON can have width other than 1 pixel. To enable wide line drawing use **SE_CNTL:WIDELINE_EN** field:

- '0' disables wide lines
- '1' enables wide lines

The actual line width is specified in unsigned 6.4 fixed point format in **SE_LINE_WIDTH:LINE_WIDTH** field.

The example of setting line width can be found in **raster.c** file located in **lib\3D** directory of the RADEON DDK.

7.7 Texture Mapping

The RADEON contains powerful texture-combining units that can execute complex multi-texturing operations involving up to tree textures in a single pass. Textures may reside in both local video and AGP memory. The RADEON also supports mipmaps, 3D textures, perturbation bump mapping, cubic environment maps and texture compression. The textures supported can be power of two or non-power of two with the maximum size of 2048x2048.

The samples of how to use textures can be found in **textr.c** file located in **lib\3D** directory of the RADEON DDK.

7.7.1 Texture Loading

The texture can be loaded in any of three texturing units by setting pointers to texture location in local video or AGP memory and specifying texture properties. When textures are loaded into memory their pitch must be at least 32 bytes.

Texture memory location is specified by loading its offset from the beginning of the frame buffer into **PP_TXOFFSET_x** register, where *x* denotes texture stage 0-2.

The texture dimensions are specified in **PP_TXFORMAT_x:TXWIDTH** and **PP_TXFORMAT_x:TXHEIGHT** fields as Log_2 of texture width and height. For non-power 2 textures the texture size in texels should be loaded in **PP_TEX_SIZE_x:TEX_USIZE** and **PP_TEX_SIZE_x:TEX_VSIZE** fields and texture pitch should be set in the **PP_TXPITCH_x** register. The non-power of two textures are enabled by setting **PP_TXFORMAT_x:NON_POWER2** field:

- '0' texture size is power of two
- '1' texture size is non-power of two

Texture format is set though the **PP_TXFORMAT_x:TXFORMAT** field. Following texture formats are supported:

Table 7-16

Shading vertex	Description
0	8bpp I
1	16bpp AI (8:8)

Table 7-16 (Continued)

Shading vertex	Description
2	8bpp RGB (3:3:2)
3	16bpp ARGB (1:5:5:5)
4	16bpp RGB (5:6:5)
5	16bpp ARGB (4:4:4:4)
6	32bpp ARGB (8:8:8:8)
7	32bpp RGBA (8:8:8:8)
8	8bpp Y
9	AYUV 444 (8:8:8:8)
10	YUV 422 (V:Y0:U:Y1) (8:8:8:8)
11	YUV 422 (Y0:V:Y1:U) (8:8:8:8)
12	Color Cell Compression, NO alpha or 1-bit alpha
14	Color Cell Compression, explicit alpha
15	Color Cell Compression, compressed alpha
20	du8:dv8 bump map
21	du5:dv5:l6 bump map
22	A8:du8:dv8:l8 bump map

If texture format contains alpha values, the alpha can be disabled by setting **PP_TXFORMAT_x:ALPHA_ENABLE** field.

- '0' no alpha in the map
- '1' alpha in the map

When alpha is disabled, alpha values are assumed to be 0xFF regardless of the alpha stored in the texture. For formats that do not contain alpha information, the alpha is also assumed to be 0xFF regardless of this bit.

7.7.2 Enabling Texture Mapping

The texture mapping can be enabled through **PP_CNTL:TEX_0_ENABLE**, **PP_CNTL:TEX_1_ENABLE** and **PP_CNTL:TEX_2_ENABLE** fields for texture stages from 0 to 2 correspondingly:

- '0' disables texture mapping
- '1' enables texture mapping

7.7.3 Texture Filtering

Texture filtering is set by specifying minification and magnification filtering modes. The magnification filtering mode is set to the **PP_TXFILTER_x:MAG_FILTER** field and can be one of the following values:

Table 7-17

Mag Filter	Description
0	Nearest
1	Linear

To set minification filtering mode write to the **PP_TXFILTER_x:MIN_FILTER** field one of the following values:

Table 7-18

Min Filter	Description
0	Nearest
1	Linear
2	Nearest mipmap Nearest
3	Nearest mipmap Linear
6	Linear mipmap Nearest
7	Linear mipmap Linear
8	Anisotropy Nearest
9	Anisotropy Linear
10	Anisotropy Nearest mipmap Nearest
11	Anisotropy Nearest mipmap Linear

The anisotropic filtering is valid only for the texture stage 0.

7.7.4 Texture Addressing Modes

Texture addressing modes control how the texture is applied on the target primitive when the vertex S or T coordinates are greater than 1.0 or less than 0.0. Clamp mode governs how texels are replicated beyond 0.0-1.0 to the edges of the primitive. Mirror mode ‘flips’ the texture about the 0.0 or 1.0 coordinate to produce a mirror image. Border color mode fills the remaining pixels beyond 0.0-1.0 range with the texture's border color. Wrap addressing mode repeats, or tiles the texture along the texture coordinate axis.

Texture clamping is set through **PP_TXFILTER_x:CLAMP_S** field for texture S coordinate and through **PP_TXFILTER_x:CLAMP_T** field for T coordinate. Allowed clamping modes are:

Table 7-19

Clamp Mode	Description
0	Wrap
1	Mirror
2	Clamp Last
3	Mirror Clamp Last
6	Clamp Border
7	Mirror Clamp Border

Border colors used with ‘clamp border’ mode for each of the texture stages are set by writing ARGB8888 packed colors into **PP_BORDER_COLOR_x** registers. Border colors can be applied in one of two modes as defined by **PP_TXFILTER_x:BORDER_MODE** fields:

- ‘0’ OpenGL compatible border mode
- ‘1’ Direct3D compatible border mode

7.7.5 Texture Combining

The RADEON contains three texture combining units, each able to apply a color combining function for the RGB channels and an alpha combining function for the alpha channel. Both the color and alpha combining functions take two arguments as input, which can be diffuse value, texture, texture factor or output of the other texture combining unit.

Color combining function is specified in **PP_TXCBLEND_x** registers, while alpha combining function is specified in **PP_TXABLEND_x** registers.

Following tables show sample combining function parameters that match some of the OpenGL blenders:

Table 7-20

OpenGL Blend	Stage 0 Color Combining Function	Stage 1 Color Combining Function	Stage 2 Color Combining Function
MODULATE	0x00800142	0x00800182	0x008001C2
DECAL	0x008C2D42	0x008C3582	0x008C3DC2
BLEND	0x008C2902	0x008C3102	0x008C3902
REPLACE	0x00802800	0x00803000	0x00803800
ADD	0x00812802	0x00813002	0x00813802
SUBTRACT	0x00852802	0x00853002	0x00853802

Table 7-21

OpenGL Blend	Stage 0 Alpha Combining Function	Stage 1 Alpha Combining Function	Stage 2 Alpha Combining Function
MODULATE	0x00800051	0x00800061	0x00800071
DECAL	0x00800100	0x00800100	0x00800100
BLEND	0x00800051	0x00800061	0x00800071
REPLACE	0x00800500	0x00800600	0x00800700
ADD	0x00800051	0x00800061	0x00800071
SUBTRACT	0x00840051	0x00840061	0x00840071

Texture blending factors specified per texture stage can be set through **PP_TFACTOR_x** registers in ARGB8888 format.

Texture blending for each of the stages can be enabled by setting **PP_CNTL:TEX_BLEND_x_ENABLE** field:

- '0' disables texture blending
- '1' enables texture blending

The **multex** sample located in **chap7\multex** directory of RADEON DDK can be used to experiment with different blending modes.

Dot Product Texture Blender

One of the texture blending operations supported by the RADEON is dot product operation, which modulates the components of each argument (as signed components), adds their products, then replicates the sum to all color channels, including alpha:

$$T_R = T_G = T_B = T_A = (Op1_R * Op2_R + Op1_G * Op2_G + Op1_B * Op2_B)$$

The dot-product operation can be used for implementing bump-mapping and other per-pixel lighting effects. The color and alpha blending functions necessary to perform dot product blending between texture and texture factor for each of the blending units are summarized in the following table:

Table 7-22

Blend Function	Stage 0 Combining Function	Stage 1 Combining Function	Stage 2 Combining Function
Color	0x00D0010A	0x00D0010C	0x00D0010E
Alpha	0x00001000	0x00001000	0x00001000

The example of using dot product blending operation for bump-mapping can be found in the **dot3** sample in the **chap7\dot3** directory of RADEON DDK.

7.7.6 Texture Coordinate Selection

The RADEON can have up to three texture coordinate sets fed in from the vertex data or generated by TCL engine. Each of the texture stages can be independently mapped to any of the available texture coordinate sets. This texture coordinates routing within texture units can be specified in **PP_TXFORMAT_x:ST_ROUTE** fields. The texture coordinate set constants are summarized in the table below:

Table 7-23

Texture Coordinate Set	Coordinate Set Description
0	S0, T0, Q0
1	S1, T1, Q1
2	S2, T2, Q2

7.7.7 Mipmaps

The RADEON supports mipmapped textures to increase 3D scene drawing efficiency and visual realism. The mipmaps supported by RADEON are monolithic, meaning that each of the subsequent mipmap levels in the chain is located in memory immediately after the previous level. Please, note that each mipmap level should have pitch of at least 32 bytes. Use of non-power of two textures requires that each map be padded to the next power of two resolution.

For mipmap textures the number of mipmap levels has to be specified in **PP_TXFILTER_x:MAX_MIP_LEVEL** fields. Only the dimensions of the base map have to be specified for the mipmap textures.

Mipmap LOD bias measured in mip levels can be loaded into **PP_TXFILTER_x:LOD_BIAS** fields. The values loaded should be signed 2's complements of bias values with the range $-1.0 \leq \text{Bias} < 4.0$.

7.7.8 3D Textures

The RADEON features support for 3D (volume) textures that can be used for volume visualization in many scientific and medical applications as well as volumetric fogging and lighting. The 3D texture coordinates have three components (S, T, Q) where Q coordinate describes the depth of the texture. The volume textures are monolithic, with Q=0 plane stored first, Q=1 plane stored as next map and so forth. Please, note that each plane map should have pitch of at least 32 bytes. When volume texture is enabled, mipmaps cannot be used and both minification and magnification filters have to be programmed identically to either nearest or linear interpolation mode. The RADEON uses two stages to accommodate 3D texture processing, supporting only one 3D texture at a time. Stage 0 will operate on width and height of the volume texture, while stage 2 will operate on the depth. The 3D textures have volume limitation of 512x256x256 with width and height required to be power of two, while the depth can be non-power of two. Both texture stages have to be programmed and enabled:

1. Set surface offset for both stages in **PP_TXOFFSET_0** and **PP_TXOFFSET_2** registers.
2. Load 3D texture width into **PP_TXFORMAT_0:TXWIDTH**, height into **PP_TXFORMAT_0:TXHEIGHT** and depth into **PP_TXFORMAT_2:TXWIDTH** fields.
3. Set **PP_TXFORMAT_0:NON_POWER2** field to be power of two. **PP_TXFORMAT_2:NON_POWER2** should be set according to the depth dimension.
4. Set minification and magnification filters for both stages to the same value.

5. Set mipmap levels in **PP_TXFILTER_0:MAX_MIP_LEVEL** and **PP_TXFILTER_2:MAX_MIP_LEVEL** to be zero.
6. Set border color and texture blend factor to be equal in both stages.
7. Disable stage 2 blending in **PP_CNTL:TEX_BLEND_2_ENABLE** field.
8. Enable stage 0 and stage 2 by setting **PP_CNTL:TEX_0_ENABLE** and **PP_CNTL:TEX_2_ENABLE** fields accordingly.

Refer to the **voltextr** sample located in **chap7\voltextr** of the RADEON DDK and **textr.c** file in **lib\3D** of the RADEON DDK for reference.

7.7.9 Cubic Environment Mapping

Cubic environment mapping is a technique that is used to simulate highly reflective surfaces with environment texture maps represented as six internal faces of a cube. The RADEON is capable of supporting cubic environment maps by performing cube face selection and cubic environment mapping calculation in the rasterizer. The idea behind cubic environment mapping is to map reflection vector specified by (S, T, Q) triplet to the face and (S, T) coordinates within that face. The example of the left hand environment cube constriction is shown in the following figure.

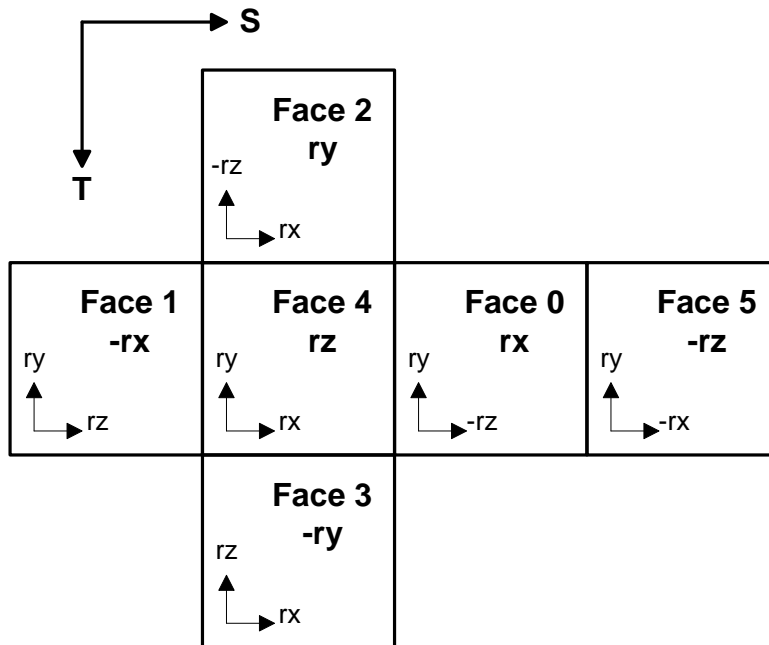


Figure 7-1. Cubic Environment Mapping

The cube faces are stored as a separate texture maps in the memory.

To set cubic environment map into a texture unit, face offsets and dimensions have to be specified:

1. Face 0 width and height specified as Log_2 of texture size are loaded into **PP_TXFORMAT_x:TXWIDTH** and **PP_TXFORMAT_x:TXHEIGHT** fields.
2. Face 1 width and height specified as Log_2 of texture size are loaded into **PP_CUBIC_FACES_x:FACE_WIDTH_1** and **PP_CUBIC_FACES_x:FACE_HEIGHT_1** fields.
3. Face 2 width and height specified as Log_2 of texture size are loaded into **PP_CUBIC_FACES_x:FACE_WIDTH_2** and **PP_CUBIC_FACES_x:FACE_HEIGHT_2** fields.
4. Face 3 width and height specified as Log_2 of texture size are loaded into **PP_CUBIC_FACES_x:FACE_WIDTH_3** and **PP_CUBIC_FACES_x:FACE_HEIGHT_3** fields.

5. Face 4 width and height specified as Log_2 of texture size are loaded into **PP_CUBIC_FACES_x:FACE_WIDTH_4** and **PP_CUBIC_FACES_x:FACE_HEIGHT_4** fields.
6. Face 5 width and height specified as Log_2 of texture size are loaded into **PP_TXFORMAT_x:FACE_WIDTH_5** and **PP_TXFORMAT_x:FACE_HEIGHT_5** fields.
7. Set face 0 offset to **PP_CUBIC_OFFSET_Tx_0**.
8. Set face 1 offset to **PP_CUBIC_OFFSET_Tx_1**.
9. Set face 2 offset to **PP_CUBIC_OFFSET_Tx_2**.
10. Set face 3 offset to **PP_CUBIC_OFFSET_Tx_3**.
11. Set face 4 offset to **PP_CUBIC_OFFSET_Tx_4**.
12. Set face 5 offset to **PP_TXOFFSET_x:TXOFFSET**.

Cubic environment maps are enabled by writing to the **PP_TXFORMAT_x:CUBIC_MAP_ENABLE** fields and can be set on per texture stage basis:

- '0' disables cubic environment mapping
- '1' enables cubic environment mapping

Texture coordinate generation feature of the RADEON TCL engine can be used to calculate reflection vector for the cubic environment mapping. Refer to section [“Texture Coordinate Generation For Cubic Environment Mapping” on page 7-59](#) for more information.

The example of how to perform cubic environment mapping can be found in the **cubemap** sample located in **chap7\cubemap** of the RADEON DDK.

7.7.10 Bump Mapping

Bump mapping is a technique used to simulate a reflection of the wrinkled or bumped surfaces without requiring highly tessellated objects. The bump mapping supported by the RADEON perturbs illumination of the object surface on the per-pixel basis by perturbing environment light map texture coordinates. This type of bump mapping involves two textures: bump map and environment map being bumped (bumpee). Instead of the color values the bump map contains du and dv values used to bump environment map coordinates. It could also contain luminance values to modulate colors of the environment map. The displacement values du and dv in the bump map can be calculated from the height maps based on the relative heights of the adjacent pixels. Before the displacement values are applied to the environment map, they are transformed by 2x2 rotation matrix:

$$du' = du * M_{00} + dv * M_{01}$$

$$dv' = du * M_{10} + dv * M_{11}$$

where du and dv are the values taken from the bump map and M is rotation matrix. The rotation matrix elements are loaded in **PP_ROT_MATRIX_0** and **PP_ROT_MATRIX_1** registers in a packed format.

The luminance is also transformed according to the formula:

$$L' = L * \text{Scale} + \text{Offset}$$

where Scale and Offset are specified by **PP_LUM_MATRIX:LSCALE** and **PP_LUM_MATRIX:LOFFSET** fields. The result of the luminance calculation is available in the alpha component of the stage 2 output.

The bump map is always fixed to the stage 2, while bumped map can be either in stage 0 or 1. This bumped map stage should be given in **PP_CNTL:BUMPED_MAP** field.

The bump mapping is enabled and disabled through **PP_CNTL:BUMP_MAP_ENABLE** field:

- '0' disables bump mapping
- '1' enables bump mapping

The following steps have to be taken to properly setup bump mapping:

1. Load bumped map (environment map) into stage 0 or 1.
2. Load bump map (perturbation map) into stage 2. Note, that bump map texture format should be one of the following: $du8:dv8$, $du5:dv5:l6$ or $A8:du8:dv8:l8$ (see [“Texture Loading” on page 7-29](#) for more information)
3. Load rotation matrix.
4. Load luminance scale and offset.
5. Enable stage 2 and the other stage containing environment map.
6. Disable stage 2 blending.
7. Enable bump mapping.

The samples of how to setup and use bump mapping can be found in **embm.c** file in **lib3D** directory of the RADEON DDK, and in the **bumpmap** sample located in **chap7\bumpmap** directory of the RADEON DDK.

7.7.11 Texture Color Space Conversion

The RADEON has the ability to automatically perform YUV to RGB color space conversion in the first texture unit, rendering YUV-encoded textures to RGB surfaces. To perform this color space conversion, specify one of the supported YUV texture formats in first texture unit (see section 7.7.1. Texture Loading for more information), and enable color space conversion by writing to **PP_TXFORMAT_0:YUV_TO_RGB**:

- '0' disables YUV to RGB color space conversion
- '1' enables YUV to RGB color space conversion

The color space conversion can be controlled by selecting one of the preset color temperatures in **PP_TXFORMAT_0:YUV_TEMPERATURE** field:

- '0' cool setting (R – 6500K, GB – 9300K)
- '1' hot setting (RGB – 9300K)

The sample of how to program YUV to RGB color space conversion in the texture unit can be found in **yuv2rgb** sample in **chap7/yuv2rgb** directory of the RADEON DDK.

7.8 TCL (Transform/Clip/Lighting) Engine

The RADEON introduces first generation TCL engine to process non-transformed and unlit vertices. Using hardware transform and lighting can maintain high concurrency between graphics engine and host CPU while spreading computational load, making the whole system more efficient.

7.8.1 Overview

The RADEON TCL engine has support for following operations:

- Transform homogeneous vertices from Object/Model Coordinates to Clip Coordinates.
- Perform Frustum Clipping on the point, line and triangle primitives in Clip Coordinates.
- Perform texture coordinate transformation/generation.
- Support 4-matrix vertex blending for skinning and morphing (tweening).
- Support up to 8 local or distant lights. Includes range and spot attenuation.
- Include color-per-vertex support with lighting.
- Support guard-band clipping for clipping performance optimization.

- Support 6 user-defined clip planes.
- Support back-face culling for performance optimization.
- Support per-vertex fog computations.

In the subsequent sections the operation and setup of each of the above-mentioned features is discussed.

7.8.2 Transformations, Coordinate Systems and Matrices

The transform process is nothing more than rotating the incoming vertex positions from object to clip coordinates. When TCL is enabled, the transform process is always performed.

There are three primary coordinate systems, which affect the TCL engine, with an additional 2 coordinate systems that are part of the RADEON Setup Engine Transformation Engine. The TCL engine outputs clip coordinate positions to the RADEON Setup Engine and the Transform Engine within the Setup Engine performs the perspective divide and viewport transformation. The following diagram describes the RADEON transformation pipeline:

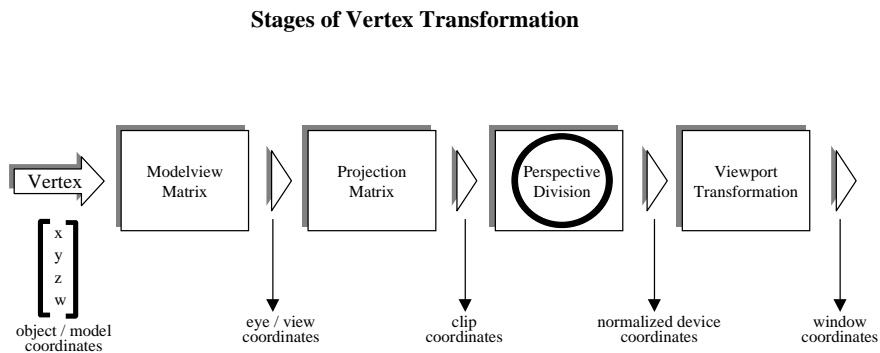


Figure 7-2. RADEON Transformation Pipeline

The coordinate systems used through the vertex transformation stages are:

- **Model/Object Coordinates** – This is, in general, the base coordinate system in which all vertices, user-defined clip planes, light positions, directions, etc. are specified. It is important to note that this coordinate system changes each time the ModelView

matrix changes.

- **Eye/View Coordinates** – This coordinate system is generally where all lighting, user defined clipping and vertex fog calculations should be performed. It is the one common ortho-normal coordinate system that all positions and normals can be rotated into
- **Clip Coordinates** – This coordinate system is always used for frustum clipping and generally used for user-defined clipping calculations. The advantage to clipping in this coordinate system is that the frustum of a perspective projection has actually been projected out to be a rectangular box. The box is defined by the following planes: $X = -W$ (LEFT), $X = W$ (RIGHT), $Y = -W$ (TOP/BOT), $Y = W$ (BOT/TOP), $Z = -W$ (NEAR), $Z = W$ (FAR). This allows the determination of distances from positions to clip planes to be based on only a single component of the vector instead of having to perform a 4-component dot product.
- **Normalized Device Coordinates** – This coordinate system results from the x/w , y/w , z/w “perspective” divide. The divide is only a perspective divide if the projection matrix was a perspective projection matrix. If the projection matrix was an orthographic matrix, the divide typically does nothing ($w = 1$). The divide should really be viewed as a conversion from homogenous coordinates to Cartesian coordinates. The x and y values range from -1.0 to 1.0 and represent the position of the vertex on the window screen. The z values range from -1.0 to 1.0 and represent (non-linearly) the distance from the near to the far clip plane.
- **Window Coordinates** - This coordinate system represents the typical geometry to rasterizer interface where x and y range from 0 to X_{max} , Y_{max} in pixel coordinates and Z typically ranges from 0 at the near plane to 1.0 at the far plane.

The transformations between coordinate systems in the 3D transformation pipeline are carried out by one of the following matrices:

- **ModelView Matrix** – This matrix is used to rotate (transform) positions from Model coordinates to Eye coordinates. Note that the inverse transpose of this matrix is used to rotate (transform) normals from Model to Eye coordinates.
- **Projection Matrix (Eye2Clip)** - This matrix is used to rotate positions from Eye to Clip coordinates. In this implementation, this matrix is only used in the special case of vertex blending.
- **Model2Clip Matrix** – This matrix is simply the concatenation of the ModelView and Projection Matrix. It is used to rotate positions from Model to Clip coordinates directly. This path is always used for vertex positions except in the case mentioned above in Projection Matrix description.

The viewport transform, which maps normalized device coordinates to desired screen coordinates and depth range is described in the section [“Viewport Transform” on page 7-44](#).

7.8.3 Loading Matrices

The RADEON’s 3D transformation pipeline implementation requires 16 matrices in the worst-case combination of processing. The base implementation requires a single ModelView matrix, an inverse-transpose ModelView matrix, and a Model2Clip matrix. Due to the 4-matrix vertex blending, there is actually a need for 4 of each of these matrices, accounting for 12 matrices. There are the additional 3 matrices needed for texture transform and a single matrix needed for the eye to clip projection transformation. All matrices are stored in the vector state memory described in [“Vector State Data” on page 7-45](#). The eye to clip matrix stored at the fixed offset in the memory, while all other matrices can occupy one of the 16 available matrix slots in the vector state memory. The intent here is that when not all matrices are needed for vertex blending and texture transform, the spare memory may be used as matrix storage for implementing a matrix stack via the driver. The mapping of the matrices to the matrix slots in the vector memory is achieved by writing matrix slot numbers into the fields on the **SE_TCL_MATRIX_SELECT_0** and **SE_TCL_MATRIX_SELECT_1** registers.

SE_TCL_MATRIX_SELECT_0		
Field Name	Bits	Description
MODELVIEW_MTX_0_SEL	3:0	Matrix Select for ModelView Matrix 0
MODELVIEW_MTX_1_SEL	7:4	Matrix Select for ModelView Matrix 1
MODELVIEW_MTX_2_SEL	11:8	Matrix Select for ModelView Matrix 2
MODELVIEW_MTX_3_SEL	15:12	Matrix Select for ModelView Matrix 3
IT_MODELVIEW_MTX_0_SEL	19:16	Matrix Select for Inverse Transpose ModelView Matrix 0
IT_MODELVIEW_MTX_1_SEL	23:20	Matrix Select for Inverse Transpose ModelView Matrix 1
IT_MODELVIEW_MTX_2_SEL	27:24	Matrix Select for Inverse Transpose ModelView Matrix 2
IT_MODELVIEW_MTX_3_SEL	31:28	Matrix Select for Inverse Transpose ModelView Matrix 3

SE_TCL_MATRIX_SELECT_1		
Field Name	Bits	Description
MODEL2CLIP_MTX_0_SEL	3:0	Matrix Select for Model To Clip Matrix 0
MODEL2CLIP_MTX_1_SEL	7:4	Matrix Select for Model To Clip Matrix 1
MODEL2CLIP_MTX_2_SEL	11:8	Matrix Select for Model To Clip Matrix 2
MODEL2CLIP_MTX_3_SEL	15:12	Matrix Select for Model To Clip Matrix 3
TEX_XFORM_MTX_0_SEL	19:16	Matrix Select for Texture Transform Matrix 0
TEX_XFORM_MTX_1_SEL	23:20	Matrix Select for Texture Transform Matrix 1
TEX_XFORM_MTX_2_SEL	27:24	Matrix Select for Texture Transform Matrix 2
TEX_XFORM_MTX_3_SEL	31:28	Reserved

The examples of setting up and loading matrices can be found in **tcutil.c** and **tcxform.c** files located in **lib3D** directory of the RADEON DDK.

7.8.4 Viewport Transform

The viewport transformation is a sub-set of a matrix which simply performs a scale and an offset operation on the x, y, and z normalized device coordinates to map them from a -1.0 to 1.0 range to the desired pixel or z range.

The viewport transform can be calculated as illustrated in the code snippet below:

```
Xscale = 0.5f * w;  
Xoffset = x + Xscale;  
Yscale = -0.5f * h;  
Yoffset = y - Yscale;  
Zscale = 0.5f * (maxz - minz);  
Zoffset = 0.5f * (maxz + minz);
```

where x, y define top left corner of the viewport rectangle in screen coordinates; w,h define viewport width and height in screen coordinates; and minz, maxz specifie the depth range.

These floating point viewport transformation parameters should be loaded into **SE_VPORT_XSCALE**, **SE_VPORT_XOFFSET**, **SE_VPORT_YSSCALE**, **SE_VPORT_YOFFSET**, and **SE_VPORT_ZSCALE**, **SE_VPORT_ZOFFSET** registers correspondingly.

The viewport transformations can be enabled and disabled by setting **SE_CNTL:VPORT_XY_XFEN** field for X, Y viewport transformation:

- '0' disables X, Y viewport transformation
- '1' enables X, Y viewport transformation

and **SE_CNTL:VPORT_Z_XFEN** field for Z viewport transformation:

- '0' disables Z viewport transformation
- '1' enables Z viewport transformation

Note, that the viewport transform is working independently from TCL engine even if the TCL engine is disabled.

For the viewport setup example refer to the **tcl.c** file in **lib\3D** directory of the RADEON DDK.

7.8.5 TCL State Data

The TCL engine state data is either stored in RADEON registers or in special on-chip memory area. This special memory area is further subdivided into vector data area and scalar data area, depending on the type of the information stored.

Vector State Data

The layout of the vector state data required for TCL engine operation is listed in the table below. Each entry will consist of 3 or 4 single precision *IEEE* floating-point vector values.

Table 7-24 Vector State Data

Offset	Parameter	Sub-Elements	Description	Format
0	Matrix_0	00,01,02,03	Matrix 0(4 elements from first row)	4 IEEE fp
1		10,11,12,13	Matrix 0(4 elements from second row)	4 IEEE fp
2		20,21,22,23	Matrix 0(4 elements from third row)	4 IEEE fp
3		30,31,32,33	Matrix 0(4 elements from fourth row)	4 IEEE fp
4	Matrix_1	00,01,02,03	Matrix 1(4 elements from first row)	4 IEEE fp
5		10,11,12,13	Matrix 1(4 elements from second row)	4 IEEE fp
6		20,21,22,23	Matrix 1(4 elements from third row)	4 IEEE fp
7		30,31,32,33	Matrix 1(4 elements from fourth row)	4 IEEE fp
	:	:	:	

Table 7-24 Vector State Data (Continued)

Offset	Parameter	Sub-Elements	Description	Format
	:	:	:	
	:	:	:	
	:	:	:	
60	Matrix_15	00,01,02,03	Matrix 15(4 elements from first row)	4 IEEE fp
61		10,11,12,13	Matrix 15(4 elements from second row)	4 IEEE fp
62		20,21,22,23	Matrix 15(4 elements from third row)	4 IEEE fp
63		30,31,32,33	Matrix 15(4 elements from fourth row)	4 IEEE fp
64	Light 0 Ambient	RGBA	Light 0 Ambient Red,Green,Blue, Alpha Color	4 IEEE fp
65	Light 1 Ambient	RGBA	Light 1 Ambient Red,Green,Blue, Alpha Color	4 IEEE fp
	:	:	:	
71	Light 7 Ambient	RGBA	Light 7 Ambient Red,Green,Blue, Alpha Color	4 IEEE fp
72	Light 0 Diffuse	RGBA	Light 0 Diffuse Red,Green,Blue, Alpha Color	4 IEEE fp
73	Light 1 Diffuse	RGBA	Light 1 Diffuse Red,Green,Blue, Alpha Color	4 IEEE fp
	:	:	:	
79	Light 7 Diffuse	RGBA	Light 7 Diffuse Red,Green,Blue, Alpha Color	4 IEEE fp
80	Light 0 Specular	RGBA	Light 0 Specular Red,Green,Blue, Alpha Color	4 IEEE fp
81	Light 1 Specular	RGBA	Light 1 Specular Red,Green,Blue, Alpha Color	4 IEEE fp
	:	:	:	
87	Light 7 Specular	RGBA	Light 7 Specular Red,Green,Blue, Alpha Color	4 IEEE fp
88	Light 0 Idir/LocPos	XYZW	Light 0 Infinite Light Direction or Local Spot Position	4 IEEE fp
89	Light 1 Idir/LocPos	XYZW	Light 1 Infinite Light Direction or Local Spot Position	4 IEEE fp
	:	:	:	
95	Light 7 Idir/LocPos	XYZW	Light 7 Infinite Light Direction or Local Spot Position	4 IEEE fp
96	Light 0 lhalfv/LocSpdir	XYZ	Light 0 Infinite Halfway Vector / Local Spot Direction	4 IEEE fp *

Table 7-24 Vector State Data (Continued)

Offset	Parameter	Sub-Elements	Description	Format
97	Light 1 lhalfv/LocSpdir	XYZ	Light 1 Infinite Halfway Vector / Local Spot Direction	4 IEEE fp *
	:	:	:	
103	Light 7 lhalfv/LocSpdir	XYZ	Light 7 Infinite Halfway Vector / Local Spot Direction	4 IEEE fp *
104	Light 0 Local Attenuation	KqKIKc	Light 0 Local Attenuation (Quadratic Kq , Linear KI, Constant Kc)	4 IEEE fp *
105	Light 1 Local Attenuation	KqKIKc	Light 1 Local Attenuation (Quadratic Kq , Linear KI, Constant Kc)	4 IEEE fp *
	:	:	:	
111	Light 7 Local Attenuation	KqKIKc	Light 7 Local Attenuation (Quadratic Kq , Linear KI, Constant Kc)	4 IEEE fp *
112	Eye2Clip Matrix	00,01,02,03	The Eye2Clip Matrix (4 elements from first row)	4 IEEE fp
113		10,11,12,13	The Eye2Clip Matrix (4 elements from second row)	4 IEEE fp
114		20,21,22,23	The Eye2Clip Matrix (4 elements from third row)	4 IEEE fp
115		30,31,32,33	The Eye2Clip Matrix (4 elements from fourth row)	4 IEEE fp
116	UCP0	XYZW	User clip plane 0	4 IEEE fp
117	UCP1	XYZW	User clip plane 1	4 IEEE fp
	:	:	:	
121	UCP5	XYZW	User clip plane 5	4 IEEE fp
122	GlbAmb/ EmGbAmb	RGBA	If Ambient Source and Emissive Source are PREMULT, then this field contains (MATemissive + (MATambient * LTglobal_ambient)), ELSE it contains the LTglobal_ambient color.	4 IEEE fp
123	Fog Parameters	RCDX	These values are used when computing vertex fog. See description below.	4 IEEE fp*
124	Eye Vector / Rescale Normal Constant	XYZ/K	XYZ Used as infinite viewer vector for specular light calculations W Term is the value used when the rescale_normal flag is set to normalize the normals based on a pre-defined scaling value.	4 IEEE fp*

* **Note:** Only the first 3 DWORDs are used for processing.

The entire vector memory is accessed via an index/data register pair. The description of these registers is shown below. When updating multiple DWORDS through this path, the CCE Type-0 packet bit, which prevents auto-incrementing, should be used so that all words are written to the data register.

SE_TCL_VECTOR_INDX_REG		
Field Name	Bits	Description
OCTWORD_OFFSET	6:0	Octword offset to begin writing
OCTWORD_STRIDE	22:16	Octword stride to increment each time moving to a new OCTWORD.
DWORD_COUNT	29:28	Dword count

The values received from **SE_TCL_VECTOR_DATA_REG** register will be written to the vector state memory starting with the vector, which offset is specified in **SE_TCL_VECTOR_INDX_REG:OCTWORD_OFFSET** field. The **SE_TCL_VECTOR_INDX_REG:DWORD_COUNT** field will be auto-incremented for each DWORD of the vector written. When all 4 DWORDs are sent, the offset for the next vector will be incremented by **SE_TCL_VECTOR_INDX_REG:OCTWORD_STRIDE** value.

SE_TCL_VECTOR_DATA_REG		
Field Name	Bits	Description
DATA REGISTER	31:0	32-bit data to write to Vector Memory

The specific values of the vector state data required for TCL operation will be discussed in detail in the subsequent sections.

The examples of how to write to the vector state data memory can be found in the **tblutil.c** file in the **libs\3D** directory of RADEON DDK.

Scalar State Data

The layout of the scalar state data required for TCL engine operation is listed in the table below. Each entry is a single precision *IEEE* floating-point number.

Table 7-25

Offset	Parameter	Sub-Elements	Description
0	Spot Dual Cone Delta	L0	Math Unit – Outer angle/[cos(Inner angle)-cos(Outer angle)]
1	Spot Dual Cone Delta	L1	Math Unit – Outer angle/[cos(Inner angle)-cos(Outer angle)]
	:	:	:
7	Spot Dual Cone Delta	L7	Math Unit – Outer angle/[cos(Inner angle)-cos(Outer angle)]
8	Spot Exponent	L0	Math Unit – $v^d \wedge \text{spot_exponent}$
9	Spot Exponent	L1	Math Unit – $v^d \wedge \text{spot_exponent}$
	:	:	:
15	Spot Exponent	L7	Math Unit – $v^d \wedge \text{spot_exponent}$
16	Spot Cutoff / Dual Cone Mult	L0	Spot Cutoff Angle for OpenGL-type spot. $1.0/[\cos(\text{Inner angle})-\cos(\text{Outer angle})]$ for Dual-Cone spot.
17	Spot Cutoff / Dual Cone Mult	L1	Spot Cutoff Angle for OpenGL-type spot. $1.0/[\cos(\text{Inner angle})-\cos(\text{Outer angle})]$ for Dual-Cone spot.
	:	:	:
23	Spot Cutoff / Dual Cone Mult	L7	Spot Cutoff Angle for OpenGL-type spot. $1.0/[\cos(\text{Inner angle})-\cos(\text{Outer angle})]$ for Dual-Cone spot.
24	Specular Theshold	L0	MAC Unit – $s^n > \text{spec_thres}$ – use $s^n \wedge \text{shininess}$ else 0
25	Specular Theshold	L1	MAC Unit – $s^n > \text{spec_thres}$ – use $s^n \wedge \text{shininess}$ else 0
	:	:	:
31	Specular Theshold	L7	MAC Unit – $s^n > \text{spec_thres}$ – use $s^n \wedge \text{shininess}$ else 0
32	Range Cutoff Squared	L0	Max range of light squared. If vertex->light distance squared is greater than range cutoff squared, the light attenuation is 0.0.
33	Range Cutoff Squared	L1	Max range of light squared. If vertex->light distance squared is greater than range cutoff squared, the light attenuation is 0.0.
	:	:	:
39	Range Cutoff Squared	L7	Max range of light squared. If vertex->light distance squared is greater than range cutoff squared, the light attenuation is 0.0.

Table 7-25 (Continued)

Offset	Parameter	Sub-Elements	Description
40	Range Attenuation Constant	L0	If Kl and Kq are 0.0 and Kc is not 1.0, the driver can pre-compute the 1/Kc value and place it here. Only used if both range_att_enable and range_att_const_enable are set.
41	Range Attenuation Constant	L1	If Kl and Kq are 0.0 and Kc is not 1.0, the driver can pre-compute the 1/Kc value and place it here. Only used if both range_att_enable and range_att_const_enable are set.
	:	:	:
47	Range Attenuation Constant	L7	If Kl and Kq are 0.0 and Kc is not 1.0, the driver can pre-compute the 1/Kc value and place it here. Only used if both range_att_enable and range_att_const_enable are set.
48	Guard Band		Vertical Guard Band Clip Adjust
49	Guard Band		Vertical Guard Band Discard Adjust
50	Guard Band		Horizontal Guard Band Clip Adjust
51	Guard Band		Horizontal Guard Band Discard Adjust
60	Shininess		Math Unit – $s^n \wedge \text{shininess}$

The scalar memory is accessed via an index/data register pair. The description of these registers is shown below. When updating multiple DWORDS through this path, the CCE Type-0 packet bit, which prevents auto-incrementing, should be used so that all words are written to the data register.

SE_TCL_SCALAR_INDXX_REG		
Field Name	Bits	Description
DWORD_OFFSET	6:0	Dword offset to begin writing
DWORD_STRIDE	22:16	Dword stride to increment each new data register write.

The writes to scalar memory will start from the DWORD offset specified by **SE_TCL_SCALAR_INDXX_REG:DWORD_OFFSET** field, incrementing the index counter by stride in **SE_TCL_SCALAR_INDXX_REG:DWORD_STRIDE** field upon each DWORD sent to **SE_TCL_SCALAR_DATA_REG** register.

SE_TCL_SCALAR_DATA_REG		
Field Name	Bits	Description
DATA REGISTER	31:0	32-bit data to write to Scalar Memory

The scalar memory entries will be discussed in the subsequent sections where appropriate.

The examples of how to write to the scalar state data memory can be found in the **tclutil.c** file in the **libs\3D** directory of RADEON DDK.

7.8.6 Setting up TCL Engine

The TCL engine is easy to setup and use. The proper TCL setup will include following steps:

1. Make sure TCL engine is not powered down by writing 0 to **SE_CNTL_STATUS:TCL_BYPASS** field.
2. Set and enable viewport transformation.
3. Set clipping guard bands.
4. Initialize transformation matrices.
5. Setup lighting parameters.
6. Instruct Setup Engine to receive W-coordinate from the TCL engine by writing appropriate value to the **SE_COORD_FMT** register.
7. When drawing primitives set **TCL_ENABLE** field of the primitive drawing packet to 1. See section *“Drawing 3D Primitives” on page 7-2* for more information.

The **tcl.c** file in **lib\3D** directory of the RADEON DDK contains sample code illustrating TCL engine configuration.

7.8.7 Vertex Format

The input to the TCL engine consists of the state data discussed above and vertex data. There are three basic different kinds of vertices supported and processed by hardware. If hardware transformation is used additional data operations may be performed, such as texture transformations or texture coordinate generation.

- **Untransformed, unlit vertices** – If an application/driver doesn't light or transform the vertices before rendering a scene, it can use vertices containing untransformed coordinates, normals and other required data. The application/driver specifies lighting

parameters, transformation matrices so the hardware can do all the math.

- **Untransformed, lit vertices** – If an application/driver doesn't transform the vertices before rendering a scene but it performs its own customized lighting, it can use vertices containing diffuse/specular components besides untransformed coordinates. The application/driver specifies transformation matrices and does its own customized lighting, and hardware does the transformation math, but uses the color per vertex for diffuse, specular, and fog.
- **Transformed, lit, vertices** – If an application/driver provides pre-transformed and pre-lit vertices before rendering a scene, it can use vertices defined containing position in screen coordinates and diffuse/specular lighting information. In this case, the TCL hardware will simply pass data to the Viewport Transformation hardware within the Setup Engine.

The flexible vertex format supported by TCL engine is identical to the vertex format described in the section *“Flexible Vertex Format” on page 7-3*. The RADEON allows to specify W coordinate for untransformed vertices, or have it set to default value. When TCL engine is in use and vertex components are missing in the input, RADEON will use following defaults: if Z is not specified it is defaulted to 0.0, W is defaulted to 1.0, and texture Q's are defaulted to 1.0. Also, if provided floating-point color data, the alpha and fog values are defaulted to 1.0.

7.8.8 Vertex Processing

When TCL engine is enabled and **TCL_ENABLE** field of the primitive drawing packet is set to 1, the vertices will be processed by the TCL engine before being passed to the viewport transformation in the Setup Engine. The input data to the TCL engine is defined by the **SE_VTX_FMT** field of the primitive packet. The output of the TCL engine needs to be defined by setting output vertex format and selecting output vertex component origin. The TCL engine output vertex format should be written to the **SE_TCL_OUTPUT_VTX_FMT** register as follows:

SE_TCL_OUTPUT_VTX_FMT		
Field Name	Bits	Description
VTX_W0_PRESENT	0	Primary Vertex W value is present (1 float)
VTX_FPCOLOR_PRESENT	1	Floating Point Diffuse Color is Present (3 floats: RGB)
VTX_FPALPHA_PRESENT	2	Floating Point Alpha is Present (1 float)
VTX_PKCOLOR_PRESENT	3	Packed (8,8,8,8) ARGB Diffuse is Present. This is mutually exclusive with FP_DIFFUSE_PRESENT and FP_ALPHA_PRESENT

SE_TCL_OUTPUT_VTX_FMT		
Field Name	Bits	Description
VTX_FPSPEC_PRESENT	4	Floating Point Specular Color is Present (3 floats: RGB)
VTX_FPFOG_PRESENT	5	Floating Point Fog is Present (1 float)
VTX_PKSPEC_PRESENT	6	Packed (8,8,8,8) FRGB Specular is Present. This is mutually exclusive with FPSPEC_PRESENT and FPFOG_PRESENT
VTX_ST0_PRESENT	7	Texture coordinate set 0 S,T values are present (2 floats)
VTX_ST1_PRESENT	8	Texture coordinate set 1 S,T values are present (2 floats)
VTX_Q1_PRESENT	9	non-Rage128 mode: Texture coordinate set 1 Q value is present (1 float) Rage128 mode: Texture coordinate set 1 ooW value is present (1 float)
VTX_ST2_PRESENT	10	Texture coordinate set 2 S,T values are present (2 floats)
VTX_Q2_PRESENT	11	Texture coordinate set 2 Q value is present (1 float)
VTX_ST3_PRESENT	12	Reserved
VTX_Q3_PRESENT	13	Reserved
VTX_Q0_PRESENT	14	Texture coordinate set 0 Q value is present (1 float)
VTX_BLND_WEIGHT_CNT	17:15	Reserved to match main vertex format register
VTX_N0_PRESENT	18	Reserved to match main vertex format register
VTX_XY1_PRESENT	27	Reserved to match main vertex format register
VTX_Z1_PRESENT	28	Reserved to match main vertex format register
VTX_W1_PRESENT	29	Reserved to match main vertex format register
VTX_N1_PRESENT	30	Reserved to match main vertex format register
VTX_Z_PRESENT	31	Primary vertex Z value is present (1 float)

The TCL output vertex component origin specifies where values should be taken from – from TCL engine output or from its input. Selecting input values allows to bypass TCL engine for separate vertex components. The output selection can be performed by programming **SE_TCL_OUTPUT_VTX_SEL** register.

SE_TCL_OUTPUT_VTX_SEL		
Field Name	Bits	Description
VTX_XYZW_SELECT	0	Select the computed XYZW values (0 means output the input data)
VTX_PKDIFFUSE_SELECT	1	Select the computed diffuse color/alpha value (0 means output the input data)
VTX_PKSPEC_SELECT	2	Select the computed specular color/fog value (0 means output the input data)
FORCE_NAN_IF_CCOLOR_NAN	3	Is set, instructs TCL to send NAN on output if NAN detected prior to Flt->Fix
FORCE_INORDER_PROC	4	If set, forces fixed order vertex output for lighting (@ slight performance penalty)
RSVD_1BIT_NUM0	5	Reserved 1 Bit Field
RSVD_3BIT_NUM0	8:6	Reserved 3-Bit Field
RSVD_3BIT_NUM1	11:9	Reserved 3-Bit Field
RSVD_4BIT_NUM0	15:12	Reserved 4-Bit Field
VTX_TEX0_SELECT	19:16	Select the computed S,T,Q values (See list below)
VTX_TEX1_SELECT	23:20	Select the computed S,T,Q values (See list below)
VTX_TEX2_SELECT	27:24	Select the computed S,T,Q values (See list below)
VTX_TEX3_SELECT	31:28	Reserved

The computed texture coordinates can be one of the following values:

Table 7-26

Texture Coordinate Selector	Description
0	Input texture coordinate set 0
1	Input texture coordinate set 1
2	Input texture coordinate set 2
3-7	Reserved
8	Computed texture coordinates 0
9	Computed texture coordinates 1
10	Computed texture coordinates 2
11-15	Reserved

The example of setting TCL engine output can be found in **tclutil.c** file in **lib\3D** directory of the RADEON DDK.

7.8.9 TCL Engine Culling

The RADEON TCL engine supports back-face culling, providing substantial performance benefits, especially in case of the expensive lighting. However, the benefits of the TCL culling are offset by some potential quality issues. The most important being that the culling is performed prior to the sub-pixel quantization of the rasterizer. Using TCL engine versus Setup Engine culling allows reaching a balance between image quality and performance.

The culling can be set by specifying face orientation in **SE_TCL_UCP_VERT_BLEND_CTL:CULLING_FF_DIR** field:

- '0' selects clockwise face ordering
- '1' selects counter-clockwise face ordering

and setting culling flags for front and back facing polygons in **SE_TCL_UCP_VERT_BLEND_CTL:CULL_FF_ENA** and **SE_TCL_UCP_VERT_BLEND_CTL:CULL_BF_ENA** fields:

- '0' disables culling
- '1' enables culling

The example of setting TCL engine culling parameters can be found in **tcl.c** file in the **lib\3D** directory of the RADEON DDK.

7.8.10 Clipping

The general purpose of the clipping process is to detect edges of primitives, which cross clip-plane boundaries and compute the intersection point, in order to recreate a primitive that lies completely within all clip planes. The frustum clipping clips to the view volume boundaries, typically a frustum. The user-defined clipping allows the user to define additional clip planes the will delete portions of the 3D environment for purposes such as cut-away views. Guard-band clipping is an optimization for the X (left/right) and Y (top/bottom) clipping process.

Each clip plane can add a new vertex to the primitive, so that with 6 frustum planes and an additional 6 user-defined planes, it is possible to create 15 new vertices. This would result in 12 primitives being created from a single primitive.

Frustum Clipping

Frustum clipping is the clipping that is performed to the view volume (which is a frustum for perspective projections). A frustum is basically a 4-sided pyramid with the top sliced off. The six planes that define the frustum are NEAR, FAR, LEFT, RIGHT, TOP and BOTTOM.

The basic mathematics of frustum clipping are as follows: the frustum planes are defined by x, y, z equaling the w value of the vertex where $x = -w$ is the left plane, $x = w$ is the right plane, $y = -w$ is top, $y = w$ is bottom, $z = -w$ is near, and $z = w$ is far.

A simple test is performed on all of the vertices of a given primitive to determine if the primitive can either be trivially discarded (completely outside the frustum), trivially accepted (completely inside the frustum) or must be clipped to the frustum planes. If the primitive must be clipped, new vertices are created at the intersection of the primitive edges and the clip planes and new primitive are created (if necessary) to output a “fan” of triangles, which represent the newly clipped primitive.

There is no performance penalty to compute the frustum clip codes, so it is always performed.

Guard-Band Clipping

Since clipping can be a computationally and control-intensive process, methods have been devised to reduce the amount of clipping that must be performed. Guard-band clipping is a method that allows the per-primitive clipping engine to clip (and trivially accept) to a region that is larger than the actual desired viewing volume in order to reduce the amount of clipping processing that must occur. As long as the rasterization device can support vertices that range outside the view space, this is a reasonable practice.

This process allows the user to trade off pixel processing in the rasterizer for primitive clip processing in the TCL engine. The guard bands only apply to the horizontal (left / right) and vertical (top / bottom) frustum clip planes. There is no guard band for the near and far planes, or for the user-defined clip planes.

The factor that is input by the user is a multiplier on the w term for the vertex. As described above, the comparison of $x < -w$ determines if a vertex is outside the left plane. The guard bands simply change this test to be $x < \text{HGB}(-w)$, where HGB is the horizontal guard-band modifier. A typical value would be 1.05 or 1.1. Note that the scaling is in clip coordinates, not screen coordinates, so pixel amounts must be converted to relative angle ratios.

Using guard-band clipping can introduce undesired visual artifacts caused by the by points and lines being discarded based on the vertices being outside the region, while the area of the primitive still overlaps the visible region. To solve this problem, another set of guard

bands has been devised. The discard guard band allows the user to modify the region in which primitives are trivially discarded. By slightly increasing the trivial discard region, the visual artifacts of guard-band clipping can be removed.

The guard bands are stored in the scalar state memory and should be set upon TCL setup. For the samples of guard-band loading refer to the code in **tcl.c** file of in **lib\3D** of the RADEON DDK.

User-Defined Clipping

The user-defined clip planes are an OpenGL concept that has also been adopted by Microsoft Direct3D. This function allows the user to create arbitrary clipping planes in the visual environment for functions like cut-away views and portals to other worlds.

The basic premise of user-clip planes is relatively simple. A 4-coordinate vector (x,y,z,w) is input which represents both the position and the normal to the plane. Any vertices which are found to obey the equation $X*x + Y*y + Z*z + W*w < 0$ are determined to be outside the clip plane (where X,Y,Z,W is vertex and x,y,z,w is clip plane).

The user-defined clip planes are typically specified in model / object coordinates by the user and are rotated into eye / view coordinates by the driver. Eye coordinates is the proper coordinate system in which to perform user-defined clipping based on the OpenGL definition.

There are two optimization paths provided in the RADEON TCL design. User clipping may be performed in clip space (most efficient) or model space (still better than eye space).

It is possible to perform user clipping in clip space when the projection matrix is non-singular (has an inverse). The driver is expected to detect this case and then rotate the user clip planes through the inverse transpose of the projection matrix. Use **SE_TCL_UCP_VERT_BLEND_CTL:UCP_IN_CLIP_SPACE** field to enable this mode:

- '0' disables user clipping in clip space
- '1' enables user clipping in clip space

If it is not possible to perform user clipping in clip space, it may be possible to perform user clipping in model space. The process can be performed in model space if the ModelView matrix preserves angles and distances. Again, it is a driver task to determine this situation and rotate the user clip planes from eye space to the current model space.

If neither of the above-mentioned optimization flags is set, user clipping will be performed in eye space. The field **SE_TCL_UCP_VERT_BLEND_CTL:UCP_IN_CLIP_SPACE** can be used to enable this mode:

- 0' disables user clipping in model space
- 1' enables user clipping in model space

The control of the user-clipping is setup by loading parameters of the clipping equation into the vector state memory and enabling individual clip planes by writing to **SE_TCL_UCP_VERT_BLEND_CTL:UCP_ENA_x** fields where **x** is 0-6 and corresponds to a clip plane number:

- '0' disables user clip plane
- '1' enables user clip plane

The example of working with user clip planes can be found in **tblucp.c** file in **lib\3D** directory of the RADEON DDK.

7.8.11 Texture Coordinate Generation and Transformation

Texture coordinate generation is the process of creating texture coordinates from other data involved with the transform process. The choices for the type of coordinate generation are the vertex position in model or eye coords, the vertex normal in eye coordinates, or the computed reflection vector in eye coordinates. This type of coordinate generation can be selected by writing to

SE_TCL_TEXTURE_PROC_CTL:TEX_CS_PROC_SRC_x fields where **x** denotes texture coordinate set 0-2. The valid types are:

Table 7-27

Texture Coordinate Generation Mode	Description
0	Input texture coordinate set 0
1	Input texture coordinate set 1
2	Input texture coordinate set 2
3	Reserved
4	Vertex Position in Object Coordinates
5	Vertex Position in Eye Coordinates
6	Vertex Normal in Eye Coordinates
7	Reflection Vector in Eye Coordinates

The texture coordinate generation is enabled or disabled by setting **SE_TCL_TEXTURE_PROC_CTL:TEX_CS_PROC_ENA_x** fields:

- ‘0’ disables texture coordinate generation for coordinate set *x*
- ‘1’ enables texture coordinate generation for coordinate set *x*

OpenGL has the ability to supply an additional planar definition to modify the texture coordinates. These additional texture coordinate transformations can be performed by loading appropriate texture transformation matrices into vector state data memory and enabling coordinate transformation by writing to

SE_TCL_TEXTURE_PROC_CTL:TEX_XFORM_ENA_x fields:

- ‘0’ disables texture coordinate transformation for coordinate set *x*
- ‘1’ enables texture coordinate transformation for coordinate set *x*

The sample uses for the texture coordinate generation are given in the next two sections: *“Texture Coordinate Generation For Spherical Environment Mapping” on page 7-59* and *“Texture Coordinate Generation For Cubic Environment Mapping” on page 7-59*.

The samples of setting up texture coordinate transformation and generation can be found in **tcmtxgen.c** file in **lib\3D** of the RADEON DDK.

Texture Coordinate Generation For Spherical Environment Mapping

Spherical environment mapping is a technique that simulates highly reflective surfaces by texturing objects with special sphere texture maps representing 360 degree view of the environment.

The texture coordinates that map sphere environment map to the object address the texture as a function of the reflective distortion created by the curvature of the surface. For that we could use texture coordinate generation based on the vertex normals in the eye coordinates.

The example of using texture coordinate generation for spherical environment mapping is located in **chap7\sphrmap** directory of the RADEON DDK.

Texture Coordinate Generation For Cubic Environment Mapping

The cubic environment mapping, discussed in detail in section *“Cubic Environment Mapping” on page 7-36* can also be used with texture coordinate generation. In this case the reflection vector in eye coordinates should be used for texture coordinate generation.

The example of how to use texture coordinate generation with cubic environment mapping can be found in the **cubemap** sample located in **chap7\cubemap** of the RADEON DDK.

7.8.12 Vertex Blending

Vertex blending as a general process allows the user to combine up to 2 unique vertex positions and normals using up to 4 unique ModelView matrices with 4 unique blending weights. The basic equation reads as

$$V_{0/1} * MV_0 * A + V_{0/1} * MV_1 * B + V_{0/1} * MV_2 * C + V_{0/1} * MV_3 * D$$

Where $V_{0/1}$ is either vertex 0 or 1 coordinates, MV_{0-3} are up to 4 unique ModelView matrices, and A,B,C,D are blending weights from the vertex.

The selection of the vertex coordinate sets for each of the parts of the equation above is performed by setting:

SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_SOURCE_0 field for the first blending weight,

SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_SOURCE_1 field for the second blending weight,

SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_SOURCE_2 field for the third blending weight, and

SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_SOURCE_3 field for the fourth blending weight:

- '0' selects primary vertex coordinate set
- '1' selects secondary vertex coordinate set

The ModelView matrices should be loaded as described in section [“Loading Matrices” on page 7-43](#).

Vertex blending can be enabled by setting

SE_TCL_UCP_VERT_BLEND_CTL:POSITION_BLEND_OPERATION and **SE_TCL_UCP_VERT_BLEND_CTL:NORMAL_BLEND_OPERATION** fields for vertex positions and vertex normals:

- '0' disables blending
- '1' enables blending

Note that there is an independent enabling of vertex blending for position and normal. There is no support in for blending of colors or texture coordinates across multiple vertices.

There is a control field called

SE_TCL_UCP_VERT_BLEND_CTL:BLEND_WEIGHT_MINUS_ONE which allows the user to provide one-less blend weight when the sum of the blend weights is 1.0. If this bit is set, the TCL engine will compute the last blend weight to be 1.0 minus the sum of the other blend weights, otherwise it will just use the weights provided.

The number of blend weights can be specified in one of two ways. If the **SE_TCL_UCP_VERT_BLEND_CTL:USE_ST_BLEND_OP_CNT** bit is asserted then TCL engine uses the number of blend weights specified in the **SE_TCL_UCP_VERT_BLEND_CTL:BLEND_OP_CNT** field. Otherwise the number of weights specified in the flexible vertex format field **SE_VTX_FMT** of the primitive drawing packet is used. Note that the number of blend weights in the flexible vertex format vertex data packet must equal the specified blend weight count field.

There are two optimization paths available for vertex blending. The first is set through the **SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_2_OPTIMIZE** field which allows the hardware to perform a slightly different order of operations when there are only 2 blend weights for higher performance. The risk here is that the answer will slightly differ from the result when more than 2 blend weights are used, but some are 0.0.

The second optimization path is governed by the

SE_TCL_UCP_VERT_BLEND_CTL:VERTEX_BLEND_USE_PROJ_MTX field which allows the TCL engine to rotate the single “blended” vertex from eye to clip space (instead of re-blending from model to clip) using the provided projection matrix. The risk here is as follows: if some blended vertices do not need to rotate to eye coords and others do, the first set will use the Model2Clip path for blending and the second set will use the Model2Eye path for blending and the Eye2Clip path for projection, yielding slightly different results. This is the only use for the projection matrix in the TCL vector memory.

Both of these optimization paths can result in significant performance increases and only create issues in extremely rare circumstances.

The example of using vertex blending can be found in the **tblblend.c** file of the **lib\3D** directory and the **tblvblnd** sample in the **chap7\tblvblnd** directory of the RADEON DDK.

7.8.13 Lighting

The process of vertex lighting basically takes some input color data and light definitions and calculates a new vertex color (diffuse and specular) based on how that vertex and

normal interact with the lighting model. In general, the RADEON lighting implementation is the OpenGL lighting implementation.

The RADEON lighting calculations are all performed in 32-bit floating point with full-range support. The final diffuse and/or specular value is converted from float to 8-bit fixed before being passed to Setup Engine. Negative values are valid during the lighting calculations, but final values that are negative will be turned into 0 during the float to fixed conversion.

The RADEON supports up to 8 infinite or local lights as well as global ambient lighting component.

Light Model Control

In the RADEON the lighting model common to all light sources is controlled through the **SE_TCL_LIGHT_MODEL_CTL** register:

SE_TCL_LIGHT_MODEL_CTL		
Field Name	Bits	Description
LIGHTING_ENA	0	Enables light processing. Must be set for any light processing to occur.
LIGHTING_IN_MODEL	1	Enables light processing in model. This is a performance optimization path. If the ModelView matrix is length and angle-preserving, it is possible to perform lighting computations in model (object) coordinates. This flag, currently, should only be enabled if all of the lights are infinite viewer / infinite light lights.
LOCAL_VIEWER	2	Selects local viewer processing for specular computation as opposed to infinite viewer.
NORMALIZE_NORMAL	3	If either the input vertex normals are not normalized, or the Modelview matrix changes the length of the normals, this flag should be set. It forces the renormalization of the vertex normal after the InvTranspose MV rotation.
RESCALE_NORMAL	4	If the input vertex normals are normalized, but the MV matrix changes the length of the normals, the renormalization (rescale) factor can be computed once from the matrix
SPECULAR_ENA	5	Enable specular lighting computations. If any of the 8 lights that are enabled desire specular computations, this bit needs to be set.
DIFFUSE_SPECULAR_COMBINE	6	If set, the lighting diffuse and specular values are combined and output in the diffuse field. If clear, the two color values are output independently.
ALPHA_LIGHTING	7	Perform lighting computations on the alpha channel.

SE_TCL_LIGHT_MODEL_CTL		
Field Name	Bits	Description
LOC_LIGHT_W_SCALE_SUB	8	If set, the local lighting vertex->light vector is computed using the w terms using the OpenGL cross-multiply method. If clear, the w value of the light and the vertex are ignored for computing the vertex->light vector.
NO_NORMAL_DO_AMB_ONLY	9	If set and no normal in the vertex, TCL only calculates ambient per light
RSVD_LT_1BIT_NUM0	10	Reserved
RSVD_LT_1BIT_NUM1	11	Reserved
RSVD_LT_1BIT_NUM2	12	Reserved
RSVD_LT_1BIT_NUM3	13	Reserved
RSVD_LT_2BIT_NUM0	15:14	Reserved
EMISSIVE_SOURCE	17:16	Input Selection for Emissive Value
AMBIENT_SOURCE	19:18	Input Selection for Ambient Value
DIFFUSE_SOURCE	21:20	Input Selection for Diffuse Value
SPECULAR_SOURCE	23:22	Input Selection for Specular Value
RSVD_LT_2BIT_NUM1	25:24	Reserved
RSVD_LT_3BIT_NUM0	28:26	Reserved
RSVD_LT_3BIT_NUM1	31:29	Reserved

The lighting portion of the TCL engine is enabled and disabled through the **SE_TCL_LIGHT_MODEL_CTL:LIGHTING_ENA** field, which overrides all other per-light enables:

- '0' disables lighting
- '1' enables lighting

In the RADEON implementation, positions and normals can be rotated into eye coordinates to perform lighting and user clipping, but only if required by the matrix definitions. The desired performance path is to perform lighting calculations in model coordinates (this prevents the cost of rotating the vertex positions and normals from model to eye and the potential requirement to renormalize the normals after the rotation). This method (lighting in object coordinates) can only be used if the ModelView matrix does not modify lengths or relative angles of vectors. It is up to the driver to make this determination and tell the H/W whether to perform the lighting calculations in eye or object.

For performance reasons, only infinite lighting is supported in model coordinates. There is a significant number of situations where TCL engine will not support lighting in model coordinates, but the base cases of infinite lights justifies the H/W support and driver support. Here are the situations where lighting in model cannot be used:

- Any of the lights are local lights. No H/W support for local lighting in model space.
- Vertex Blending is enabled. Since the result of the vertex blending is in eye, lighting is done in eye.
- Any form of texture coordinate generation requiring data in eye is enabled. If we must rotate the vertex position (Position in Eye) or vertex normal (Normal In Eye) or both (Reflection Vector in Eye), then lighting will be performed in eye coordinates.

The coordinates in which lighting is performed can be set through **SE_TCL_LIGHT_MODEL_CTL:LIGHTING_IN_MODEL** field:

- '0' disables lighting in model space
- '1' enables lighting in model space

The specular computations can be controlled by **SE_TCL_LIGHT_MODEL_CTL:LOCAL_VIEWER** field that selects local vs. infinite viewer model:

- '0' infinite viewer computations
- '1' local viewer computations

In most cases infinite viewer is very acceptable and much faster.

For the lighting to be computed correctly, the TCL engine might require to renormalize normals. This feature can be enabled by setting **SE_TCL_LIGHT_MODEL_CTL:NORMALIZE_NORMAL** field:

- '0' disable normal normalization
- '1' enable normal normalization

The optimization can be applied to normal normalization process, when vertex input normals are normalized, but ModelView matrix linearly scales them. In that case the normal rescale constant can be applied without performing time-consuming normalization computation. The rescale factor should be stored in the W component of the Eye Vector (at the offset 124) in vector state memory, and this mode should be set by setting **SE_TCL_LIGHT_MODEL_CTL:RESCALE_NORMAL** field.

When specular computations are required for any of the lights the **SE_TCL_LIGHT_MODEL_CTL:LIGHTING_ENA** field should be set, overriding all other per-light specular enables:

- '0' disables specular computations
- '1' enables specular computations

The RADEON lighting model allows to combine post-lighting diffuse and specular components into a single diffuse component for output. In this case the specular from vertex can be used for something else. This mode is enabled and disabled through **SE_TCL_LIGHT_MODEL_CTL:DIFFUSE_SPECULAR_COMBINE** field:

- '0' disables diffuse and specular combine
- '1' enables diffuse and specular combine

The RADEON has the ability to perform lighting calculations with diffuse alpha and fog (specular alpha) values, as if they are normal (R, G, B) color components. This alpha lighting mode is enabled by writing to

SE_TCL_LIGHT_MODEL_CTL:ALPHA_LIGHTING field:

- '0' disables alpha lighting
- '1' enables alpha lighting

The RADEON lighting model allows vertex color components (emissive, ambient, diffuse and specular) to come from 4 different sources:

Table 7-28

Color Source	Description
0	Pre-multiplied light color
1	Material color multiplied by lighting color in TCL
2	Diffuse color from vertex multiplied by lighting color in TCL
3	Specular color from vertex multiplied by lighting color in TCL

The vertex color components are selected by writing color source values to the following fields: **SE_TCL_LIGHT_MODEL_CTL:EMISSIVE_SOURCE**, **SE_TCL_LIGHT_MODEL_CTL:AMBIENT_SOURCE**, **SE_TCL_LIGHT_MODEL_CTL:SPECULAR_SOURCE** and **SE_TCL_LIGHT_MODEL_CTL:DIFFUSE_SOURCE**.

If the color does not change from vertex to vertex the pre-multiplied material color and light color should be used. If the ambient and emissive sources are both set to pre-multiply, the EmissiveGlobalAmbient term in the TCL vector memory is used as the pre-computed Emissive + (Global Ambient * Material Ambient). When alpha lighting is disabled, the output alpha will come from the alpha component of the field selected by the diffuse color source, and the fog component will come from the alpha component of the field selected by the specular color source.

The example of setting up lighting model can be found in the **tcllight.c** file located in the **lib3D** directory of the RADEON DDK.

Per Light Control

Some of the lighting parameters are unique to each light and are set separately from lighting model on the per-light basis. The RADEON packs per-light information for each of its 8 light sources in 4 registers:

SE_TCL_PER_LIGHT_CTL_0 for light 0 and 1,

SE_TCL_PER_LIGHT_CTL_1 for light 2 and 3,

SE_TCL_PER_LIGHT_CTL_2 for light 4 and 5,

SE_TCL_PER_LIGHT_CTL_3 for light 6 and 7.

SE_TCL_PER_LIGHT_CTL_0		
Field Name	Bits	Description
LIGHT_ENA_0	0	Enables light.
AMBIENT_ENA_0	1	Enables ambient computations for light.
SPECULAR_ENA_0	2	Enables specular computations for light.
LOCAL_LIGHT_0	3	Specifies a local light instead of an infinite light.
SPOT_ENA_0	4	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_0	5	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_0	6	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_0	7	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT0_1BIT_NUM0	8	Reserved 1-Bit Field
RSVD_LT0_1BIT_NUM1	9	Reserved 1-Bit Field
RSVD_LT0_1BIT_NUM2	10	Reserved 1-Bit Field
RSVD_LT0_1BIT_NUM3	11	Reserved 1-Bit Field
RSVD_LT0_2BIT_NUM0	13:12	Reserved 2-Bit Field

SE_TCL_PER_LIGHT_CTL_0		
Field Name	Bits	Description
RSVD_LT0_2BIT_NUM1	15:14	Reserved 2-Bit Field
LIGHT_ENA_1	16	Enables light.
AMBIENT_ENA_1	17	Enables ambient computations for light.
SPECULAR_ENA_1	18	Enables specular computations for light.
LOCAL_LIGHT_1	19	Specifies a local light instead of an infinite light.
SPOT_ENA_1	20	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_1	21	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_1	22	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_1	23	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT1_1BIT_NUM0	24	Reserved 1-Bit Field
RSVD_LT1_1BIT_NUM1	25	Reserved 1-Bit Field
RSVD_LT1_1BIT_NUM2	26	Reserved 1-Bit Field
RSVD_LT1_1BIT_NUM3	27	Reserved 1-Bit Field
RSVD_LT1_2BIT_NUM0	29:28	Reserved 2-Bit Field
RSVD_LT1_2BIT_NUM1	31:30	Reserved 2-Bit Field

SE_TCL_PER_LIGHT_CTL_1		
Field Name	Bits	Description
LIGHT_ENA_2	0	Enables light.
AMBIENT_ENA_2	1	Enables ambient computations for light.
SPECULAR_ENA_2	2	Enables specular computations for light.
LOCAL_LIGHT_2	3	Specifies a local light instead of an infinite light.
SPOT_ENA_2	4	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_2	5	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_2	6	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_2	7	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT2_1BIT_NUM0	8	Reserved 1-Bit Field
RSVD_LT2_1BIT_NUM1	9	Reserved 1-Bit Field

SE_TCL_PER_LIGHT_CTL_1

Field Name	Bits	Description
RSVD_LT2_1BIT_NUM2	10	Reserved 1-Bit Field
RSVD_LT2_1BIT_NUM3	11	Reserved 1-Bit Field
RSVD_LT2_2BIT_NUM0	13:12	Reserved 2-Bit Field
RSVD_LT2_2BIT_NUM1	15:14	Reserved 2-Bit Field
LIGHT_ENA_3	16	Enables light.
AMBIENT_ENA_3	17	Enables ambient computations for light.
SPECULAR_ENA_3	18	Enables specular computations for light.
LOCAL_LIGHT_3	19	Specifies a local light instead of an infinite light.
SPOT_ENA_3	20	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_3	21	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_3	22	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_3	23	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT3_1BIT_NUM0	24	Reserved 1-Bit Field
RSVD_LT3_1BIT_NUM1	25	Reserved 1-Bit Field
RSVD_LT3_1BIT_NUM2	26	Reserved 1-Bit Field
RSVD_LT3_1BIT_NUM3	27	Reserved 1-Bit Field
RSVD_LT3_2BIT_NUM0	29:28	Reserved 2-Bit Field
RSVD_LT3_2BIT_NUM1	31:30	Reserved 2-Bit Field

SE_TCL_PER_LIGHT_CTL_2

Field Name	Bits	Description
LIGHT_ENA_4	0	Enables light.
AMBIENT_ENA_4	1	Enables ambient computations for light.
SPECULAR_ENA_4	2	Enables specular computations for light.
LOCAL_LIGHT_4	3	Specifies a local light instead of an infinite light.
SPOT_ENA_4	4	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_4	5	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_4	6	Enables range attenuation processing for the light.

SE_TCL_PER_LIGHT_CTL_2		
Field Name	Bits	Description
RNG_ATT_CONSTANT_ENA_4	7	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT4_1BIT_NUM0	8	Reserved 1-Bit Field
RSVD_LT4_1BIT_NUM1	9	Reserved 1-Bit Field
RSVD_LT4_1BIT_NUM2	10	Reserved 1-Bit Field
RSVD_LT4_1BIT_NUM3	11	Reserved 1-Bit Field
RSVD_LT4_2BIT_NUM0	13:12	Reserved 2-Bit Field
RSVD_LT4_2BIT_NUM1	15:14	Reserved 2-Bit Field
LIGHT_ENA_5	16	Enables light.
AMBIENT_ENA_5	17	Enables ambient computations for light.
SPECULAR_ENA_5	18	Enables specular computations for light.
LOCAL_LIGHT_5	19	Specifies a local light instead of an infinite light.
SPOT_ENA_5	20	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_5	21	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_5	22	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_5	23	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT5_1BIT_NUM0	24	Reserved 1-Bit Field
RSVD_LT5_1BIT_NUM1	25	Reserved 1-Bit Field
RSVD_LT5_1BIT_NUM2	26	Reserved 1-Bit Field
RSVD_LT5_1BIT_NUM3	27	Reserved 1-Bit Field
RSVD_LT5_2BIT_NUM0	29:28	Reserved 2-Bit Field
RSVD_LT5_2BIT_NUM1	31:30	Reserved 2-Bit Field

SE_TCL_PER_LIGHT_CTL_3		
Field Name	Bits	Description
LIGHT_ENA_6	0	Enables light.
AMBIENT_ENA_6	1	Enables ambient computations for light.
SPECULAR_ENA_6	2	Enables specular computations for light.
LOCAL_LIGHT_6	3	Specifies a local light instead of an infinite light.
SPOT_ENA_6	4	Enables spotlight attenuation processing for light.

SE_TCL_PER_LIGHT_CTL_3		
Field Name	Bits	Description
SPOT_DUAL_CONE_6	5	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_6	6	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_6	7	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT6_1BIT_NUM0	8	Reserved 1-Bit Field
RSVD_LT6_1BIT_NUM1	9	Reserved 1-Bit Field
RSVD_LT6_1BIT_NUM2	10	Reserved 1-Bit Field
RSVD_LT6_1BIT_NUM3	11	Reserved 1-Bit Field
RSVD_LT6_2BIT_NUM0	13:12	Reserved 2-Bit Field
RSVD_LT6_2BIT_NUM1	15:14	Reserved 2-Bit Field
LIGHT_ENA_7	16	Enables light.
AMBIENT_ENA_7	17	Enables ambient computations for light.
SPECULAR_ENA_7	18	Enables specular computations for light.
LOCAL_LIGHT_7	19	Specifies a local light instead of an infinite light.
SPOT_ENA_7	20	Enables spotlight attenuation processing for light.
SPOT_DUAL_CONE_7	21	Specifies dual-cone (inner/outer) spot calculations (Direct X) instead of cutoff angle spot calculations (OpenGL).
RNG_ATT_ENA_7	22	Enables range attenuation processing for the light.
RNG_ATT_CONSTANT_ENA_7	23	Enables using a constant value for the range attenuation instead of computing the value. The RNG_ATT_ENA bit must also be set.
RSVD_LT7_1BIT_NUM0	24	Reserved 1-Bit Field
RSVD_LT7_1BIT_NUM1	25	Reserved 1-Bit Field
RSVD_LT7_1BIT_NUM2	26	Reserved 1-Bit Field
RSVD_LT7_1BIT_NUM3	27	Reserved 1-Bit Field
RSVD_LT7_2BIT_NUM0	29:28	Reserved 2-Bit Field
RSVD_LT7_2BIT_NUM1	31:30	Reserved 2-Bit Field

When lighting is globally enabled, each of the lights can be turned on by setting **SE_TCL_PER_LIGHT_CTL_x:LIGHT_ENA_y** field:

- '0' disables particular light
- '1' enables particular light

When light is enabled, the computation of the light's ambient and specular components can also be enabled by setting **SE_TCL_PER_LIGHT_CTL_x:AMBIENT_ENA_y** and **SE_TCL_PER_LIGHT_CTL_x:SPECULAR_ENA_y** fields correspondingly:

- '0' disables light component
- '1' enables light component

For the specular light computation to take place, the specular computation has to be globally enabled in the light model control register.

Each light source can be independently selected to be either local or infinite (directional). This selection is performed by writing to **SE_TCL_PER_LIGHT_CTL_x:LOCAL_LIGHT_y** field:

- '0' sets infinite light type
- '1' sets local light type

For infinite lights, the direction is specified in the TCL vector memory and if it is infinite viewer also, the pre-computed halfway vector for specular computations is stored in the TCL vector memory. For local lights, the light position and direction are stored in the TCL vector memory.

For local lights spot attenuation can be enabled by setting **SE_TCL_PER_LIGHT_CTL_x:SPOT_ENA_y** field:

- '0' enables spot attenuation
- '1' disables spot attenuation

There are two spotlight attenuation models supported: the OpenGL method using a cutoff angle and spot exponent, and the DirectX method using an inner and outer cone angle and spot exponent. The mode is selected by manipulating **SE_TCL_PER_LIGHT_CTL_x:SPOT_DUAL_CONE_y** field:

- '0' enables spot attenuation
- '1' disables spot attenuation

The cutoff and/or inner/outer cone angle values as well as the spot exponent are stored in the TCL scalar memory. Note that all angles are specified as $\cos(\text{angle})$ 0.0 – 1.0 range.

The **SE_TCL_PER_LIGHT_CTL_x:ATT_ENA_y** field should be used to enable range attenuation calculations:

- ‘0’ enables range attenuation
- ‘1’ disables range attenuation

The range attenuation constants Kq, Kl, Kc are stored in the TCL vector memory. When performing range attenuation calculations, there is a range-cutoff squared term (in the TCL scalar memory) that is used to discard light calculations when the light is too far away. In case of the constant range attenuation (when Kq and Kl are zero, the optimization can be performed by supplying the range attenuation constant in the TCL scalar memory and setting **SE_TCL_PER_LIGHT_CTL_x:ATT_CONSTANT_ENA_y** field:

- ‘0’ enables constant range attenuation calculations
- ‘1’ disables constant range attenuation calculation

The ambient, diffuse and specular light colors are stored in the vector state memory on a per-light basis. Please, refer to *“Vector State Data” on page 7-45* for information on vector offsets and their loading.

Material Properties

The material properties can be accessed through the following registers:

SE_TCL_MATERIAL_EMISSIVE_RED		
Field Name	Bits	Description
MATERIAL_EMISSIVE_RED	31:0	Material Emissive Color Red

SE_TCL_MATERIAL_EMISSIVE_GREEN		
Field Name	Bits	Description
MATERIAL_EMISSIVE_GREEN	31:0	Material Emissive Color Green

SE_TCL_MATERIAL_EMISSIVE_BLUE		
Field Name	Bits	Description
MATERIAL_EMISSIVE_BLUE	31:0	Material Emissive Color Blue

SE_TCL_MATERIAL_EMISSIVE_ALPHA		
Field Name	Bits	Description
MATERIAL_EMISSIVE_ALPHA	31:0	Material Emissive Color Alpha

SE_TCL_MATERIAL_AMBIENT_RED		
Field Name	Bits	Description
MATERIAL_AMBIENT_RED	31:0	Material Ambient Color Red

SE_TCL_MATERIAL_AMBIENT_GREEN		
Field Name	Bits	Description
MATERIAL_AMBIENT_GREEN	31:0	Material Ambient Color Green

SE_TCL_MATERIAL_AMBIENT_ALPHA		
Field Name	Bits	Description
MATERIAL_AMBIENT_ALPHA	31:0	Material Ambient Color Alpha

SE_TCL_MATERIAL_DIFFUSE_RED		
Field Name	Bits	Description
MATERIAL_DIFFUSE_RED	31:0	Material Diffuse Color Red

SE_TCL_MATERIAL_DIFFUSE_GREEN		
Field Name	Bits	Description
MATERIAL_DIFFUSE_GREEN	31:0	Material Diffuse Color Green

SE_TCL_MATERIAL_DIFFUSE_BLUE		
Field Name	Bits	Description
MATERIAL_DIFFUSE_BLUE	31:0	Material Diffuse Color Blue

SE_TCL_MATERIAL_DIFFUSE_ALPHA

Field Name	Bits	Description
MATERIAL_DIFFUSE_ALPHA	31:0	Material Diffuse Color Alpha

SE_TCL_MATERIAL_SPECULAR_RED

Field Name	Bits	Description
MATERIAL_SPECULAR_RED	31:0	Material Specular Color Red

SE_TCL_MATERIAL_SPECULAR_GREEN

Field Name	Bits	Description
MATERIAL_SPECULAR_GREEN	31:0	Material Specular Color Green

SE_TCL_MATERIAL_SPECULAR_BLUE

Field Name	Bits	Description
MATERIAL_SPECULAR_BLUE	31:0	Material Specular Color Blue

SE_TCL_MATERIAL_SPECULAR_ALPHA

Field Name	Bits	Description
MATERIAL_SPECULAR_ALPHA	31:0	Material Specular Color Alpha

SE_TCL_SHININESS

Field Name	Bits	Description
SE_TCL_SHININESS	31:0	Specular Shininess NOTE: This register maps to the same memory as the indexed register pair (SE_TCL_SCALAR_INDXX_REG and SE_TCL_SCALAR_DATA_REG) when the index register equals 60)

Generally speaking, there is very little need to specify material colors explicitly through the registers above. In most cases the material color should be pre-multiplied with light color and used as a color source.

7.8.14 Vertex Fog

Fog based on the depth is called depth or plane-based fog and fog based on the real distance to the vertex in the view space is called range fog. The RADEON supports both plane-based and range vertex fog.

Range based fog can be set by setting **SE_TCL_UCP_VERT_BLEND_CTL:RNG_BASED_FOG** field:

- '0' plane-based vertex fog
- '1' range-based vertex fog

The fog parameters are loaded in the vector state memory in the following format (R, C, D, 0.0). These parameters should be calculated based on the following formulas:

Linear Depth-Based Fog

R = 0.0

C = End / (End - Start)

D = 1.0 / (End - Start)

Linear Range-Based Fog

R = -1.0f / (End - Start)

C = End / (End - Start)

D = 0.0

Exp Depth-Based Fog

R = 0.0

C = 0.0

D = Density

Exp Range-Based Fog

R = -Density

C = 0.0

D = 0.0

Exp² Depth-Based Fog

R = 0.0

C = 0.0

D = -Density * Density

Exp² Range-Based Fog

R = -Density * Density

C = 0.0

D = 0.0

Appendix A

VGA Functions

A.1 VGA Functions Summary

Table A-1 VGA Functions

Function	Description	Reference	Compliance
AH = 0	Set Video Mode	see section A.2	Yes
AH = 1	Set Cursor Type	see section A.3	Yes
AH = 2	Set Current Cursor Position	see section A.4	Yes
AH = 3	Read Current Cursor at the Specified Page	see section A.5	Yes
AH=5, AL = Page # to be active	Select Active Display Page	see section A.6	Yes
AH = 6	Scroll Active Page Up	see section A.7	Yes
AH = 7	Scroll Active Page Down	see section A.8	Yes
AH = 8	Read Character/Attribute at Current Active Cursor Position	see section A.9	Yes
AH = 9	Write Character/Attribute at Current Cursor Position	see section A.10	Yes
AH = 0Ah	Write Character at Current Cursor position of a specified Page	see section A.11	Yes
AH = 0Bh	Set Color Palette	see section A.12	Yes
AH = 0Ch	Write Dot	see section A.13	Yes
AH = 0Dh	Read Dot	see section A.14	Yes
AH = 0Eh	Write Teletype to Active Page	see section A.15	Yes
AH = 0Fh	Return Current Video Setting	see section A.16	Yes
AH = 10h	Set Palette Registers	see section A.17	Yes
AH = 11h	Character Generation Routines	see section A.18	Yes
AH = 12h	Return Current EGA Settings/Print Screen Routine Selection	see section A.19	Yes
AH = 13h	Write String to Specified Page	see section A.20	Yes
AH = 1Ah	Display Combination Code	see section A.21	Yes

Table A-1 VGA Functions (Continued)

Function	Description	Reference	Compliance
AH = 1Bh	Return VGA Functionality and State Information	see section A.22	Yes
AH = 1Ch	Save and Restore Video State	see section A.23	Yes

A.2 AH = 0 - Set Video Mode (AL = Video Mode)

Table A-2 AH = 0 - Set Video Mode (AL = Video Mode)

AL	MODE/TYPE	RESOLUTION	DIM/COLOR	START ADDRESS
IBM Compatible Modes:				
00h	color/alpha	640x200	40x25/BW	B800h:0
01h	color/alpha	640x200	40x25/16	B800h:0
02h	color/alpha	640x200	80x25/BW	B800h:0
03h	color/alpha	640x200	80x25/16	B800h:0
04h	color/graphics	320x200	40x25/4	A000h:0
05h	color/graphics	320x200	40x25/BW	A000h:0
06h	color/graphics	320x200	80x25/BW	A000h:0
07h	mono/alpha	720x350	80x25/BW	B000h:0
0Dh	color/graphics	320x200	40x25/16	A000h:0
0Eh	color/graphics	640x200	80x25/16	A000h:0
0Fh	mono/graphics	640x350	80x25/BW	A000h:0
10h	color/graphics	640x350	80x25/16	A000h:0
11h	mono/graphics	640x480	80x30/BW	A000h:0
12h	color/graphics	640x480	80x30/16	A000h:0
13h	color/graphics	320x200	80x25/256	A000h:0
ATI Enhanced Modes:				
21h	color/alpha	800x400	100x25	B800h:0
22h	color/alpha	800x480	100x30	B800h:0
23h	color/alpha	1056x200	132x25/16	B800h:0
33h	color/alpha	1056x352	132x44/16	B800h:0

A.3 AH = 1 - Set Cursor Type

CH = start line of cursor

CL = end line of cursor

CX = 1F00h to turn off cursor

A.4 AH = 2 - Set Current Cursor Position

BH = page number of the desired page

DH, DL = row and column of cursor

A.5 AH = 3 - Read Current Cursor Position At The Specified Page

BH = page number of the desired page

On Exit:

CH, CL = cursor type

DH, DL = row, column of cursor at the specified page

A.6 AH = 5 - Select Active Display Page

AL = page number to be active

A.7 AH = 6 - Scroll Active Page Up

AL = number of lines to be scrolled

= 0; blanks the whole window

BH = attribute of blanked line

CH, CL = row, column of upper left hand corner of scrolling window

DH, DL = row, column of lower right hand corner of scrolling window

A.8 AH = 7 - Scroll Active Page Down

AL = number of lines to be scrolled

= 0; blanks the whole window

BH = attribute of blanked line

CH, CL = row, column of upper left hand corner of scrolling window

DH, DL = row, column of lower right hand corner of scrolling window

A.9 AH = 8 - Read Character/Attribute At Current Active Cursor Position

BH = page number of the desired page

On Exit:

AL = character

AH = attribute (for text mode only)

A.10 AH = 9 - Write Character/Attribute At Current Cursor Position Of A Specified Page

AL = character to be written

BL = attribute of character

BH = page number

CX = count of character to write

A.11 AH = 0Ah - Write Character At Current Cursor Position Of A Specified Page

AL = character to be written

BH = page number

CX = count of character to write

A.12 AH = 0Bh - Set Color Palette, Valid For Modes 4 And 5 Only

BH = 0; selects the background color

BL = color value used with that color id

BH = 1; selects the palette to be used

BL = 0; palette value is GREEN(1)/RED(2)/BROWN(3)

=1; palette value is CYAN(1)/MAGENTA(2)/WHITE(3)

A.13 AH = 0Ch - Write Dot (Graphics Mode)

BH = page number

DX, CX = row, column of dot position

AL = color value of dot (if bit 7 of AL is ON, the color value will XOR with the current value of the dot)

A.14 AH = 0Dh - Read Dot (Graphics Mode)

BH = page number

DX, CX = row, column of dot position

On Exit:

AL = color value of dot

A.15 AH = 0Eh - Write Teletype To Active Page

AL = character to write

BL = foreground color in graphics mode

A.16 AH = 0Fh - Return Current Video Setting

On Exit:

AL = current mode

AH = number of column (in characters) on screen

BH = current active display page

A.17 AH = 10h - Set Palette Registers

AL = 0; set individual palette register

BL = palette register

BH = palette value

AL = 1; set overscan register

BH = palette value

AL = 2; set all palette and overscan registers

ES:DX = pointer to palette value table (17 bytes long),
bytes 0 - 15 are palette values for 16 palette registers,
byte 16 is palette value for the overscan register

AL = 3; toggle between intensity/blinking bit

BL = 0; set intensity on

= 1; set blinking on

AL = 7; read individual palette register

BL = palette register

On Exit:

BH = palette value

AL = 8; read overscan register

On Exit:

BH = overscan value

AL = 9; read all palette and overscan registers

ES:DX = pointer to 17-byte buffer

On Exit:

ES:DX = pointer to palette value table (17 bytes long),

bytes 0 - 15 are palette values for 16 palette registers,
byte 16 is palette value for the overscan register

AL = 10h; set a color register
BX = color register
DH = red value
CH = green value
CL = blue value

AL = 12h; set a block of color registers
BX = first color register to be set
CX = total number of color registers to be set
ES:DX = pointer to table of color register values in red, green, blue,
red, green, blue,... format

AL = 13h; set color pages (only valid for 16 color modes)
BL = 0; select color page mode
BH = 0; select 4 pages of 64 color registers each
 = 1; select 16 pages of 16 color registers each
BL = 1; select color page
BH = color page number

AL = 15h; read a color register
BX = color register
On Exit:
DH = red value
CH = green value
CL = blue value

AL = 17h; read a block of color registers
BX = first color register to be set
CX = total number of color registers to be set
ES:DX = pointer to buffer to store the color register values
On Exit:
ES:DX = pointer to table of color register values in red, green, blue,
red, green, blue,..., format

AL = 18h; update DAC mask register
BL = new mask value

AL = 19h; read DAC mask register
BL = value read from DAC mask register

AL = 1Ah; read current color page information
BL = current color page mode
BH = current color page

AL = 1Bh; change color values to gray shades
BX = first color register to be changed
CX = total number of color registers to be changed

A.18 AH=11h - Character Generator Routines

AL = 00; load user specified character set
ES:BP = pointer to character table
CX = number of characters to be stored
DX = character of offset into current table
BL = block to load
BH = bytes per character

AL = 01; load 8x14 character set
BL = block to load

AL = 02; load 8x8 character set
BL = block to load

AL = 03; set block specifier
BL = character generator block specifier

AL = 04; load 8x16 character set
BL = block to load

Note: The following functions, AL = 1?h, are similar to the functions AL = 0?h, except that with AL=1?h, the number of rows on the screen is recalculated.

AL = 10h; load user specified character set
ES:BP = pointer to character table
CX = number of characters to be stored
DX = character of offset into current table
BL = block to load
BH = bytes per character

AL = 11h; load 8x14 character set
BL = block to load

AL = 12h; load 8x8 character set
BL = block to load

AL = 14h; load 8x16 character set
BL = block to load

AL = 20h; update alternative character generator pointer (INT 1F)
ES:BP = pointer to table

AL = 21h; update alternative character generator pointer (INT 43)
ES:BP = pointer to table
CX = bytes per character
BL = row specifier
 = 0; DL = rows
 = 1; rows = 14
 = 2; rows = 25
 = 3; rows = 43

AL = 22h; update alternative character generator pointer (INT 43)
 with the 8x14 character
 ; generator in ROM
BL = row specifier
 = 0; DL = rows
 = 1; rows = 14
 = 2; rows = 25
 = 3; rows = 43

AL = 23h; update alternative character generator pointer (INT 43)
 with the 8x8 character
 ;generator in ROM
BL = row specifier
 = 0; DL = rows
 = 1; rows = 14
 = 2; rows = 25
 = 3; rows = 43

AL = 24h; update alternative character generator pointer (INT 43)
 with the 8x16 character
 ; generator in ROM
BL = row specifier
 = 0; DL = rows
 = 1; rows = 14
 = 2; rows = 25
 = 3; rows = 43

AL = 30h; return EGA character generator information
BH = 0; return current INT 1F pointer

-
- = 1; return current INT 43 pointer
 - = 2; return pointer to 8x14 character generator
 - = 3; return pointer to 8x8 character generator (lower)
 - = 4; return pointer to 8x8 character generator (upper)
 - = 5; return pointer to alternate 9x14 alpha
 - = 6; return pointer to 8x16 character generator
 - = 7; return pointer to alternate 9x16 alpha

On Exit:

ES:BP = pointer to table as requested

CX = points (pixel column per char)

DL = rows (scan line per char)

A.19 AH = 12h - Return Current EGA Settings/Print Screen Routine Selection

BL = 10h; return EGA information

On Exit:

BH = 0; color mode in effect

= 1; monochrome mode in effect

BL = 3; 256k video memory installed (always return 3)

CH = simulated value of feature bits

CL = simulated EGA/VGA dip switch setting

BL = 20h; select alternate print screen routine for EGA graphics mode

On Exit:

AL = 12h; function supported

BL = 30h; select number of scan lines for alpha modes

AL = 0; 200 scan lines

= 1; 350 scan lines

= 2; 400 scan lines

On Exit:

AL = 12h; function supported

BL = 31h; default palette loading during mode set

AH = 0

AL = 0; enable

= 1; disable

On Exit:

AL = 12h; function supported

BL = 32h; video controller

AL = 0; enable video controller

= 1; disable video controller

On Exit:

AL = 12h; function supported

BL = 33h; summing of color registers to gray shades

AL = 0; enable summing

= 1; disable summing

On Exit:

AL = 12h; function supported

BL = 34h; cursor emulation

AL = 0; enable cursor emulation

= 1; disable cursor emulation

On Exit:

AL = 12h; function supported

BL = 35h; not supported currently

BL = 36h; video screen on/off

AL = 0; video screen on

= 1; video screen off

On Exit:

AL = 12h; function supported

A.20 AH=13h – Write String to Specified Page

Input:

ES:BP = pointer to string

CX = length of string

BH = page number

DH,DL = starting row and column of cursor in which the string is placed

AL = 0 ; cursor is not moved

BL = attribute

String = (char, char, char, char,...)

AL = 1 ; cursor is moved

BL = attribute

String = (char, char, char, char,...)

AL = 2 ; cursor is not moved

String = (char, attr, char, attr,...)

AL = 3 ; cursor is moved
String = (char, attr, char, attr,...)

A.21 AH=1Ah - Display Combination Code

AL = 0; read current display combination information

On Exit:

AL = 1Ah

BL = current active display code

BH = alternate display code

Display Codes (AH = 1Ah)

Table A-3 Display Combination Codes

Code	Function
00	No display
01	MDA mode
02	CGA mode
04	EGA in color mode
05	EGA in monochrome mode
07	VGA with analog monochrome monitor
08	VGA with analog color monitor

AL = 1 ; set display combination information

BL = active display

BH = inactive display

On Exit:

AL = 1Ah

A.22 AH=1Bh - Return VGA Functionality And State Information

BX = 0

ES:DI = pointer to buffer used to store the functionality and state information
(minimum 64 bytes)

On Exit:

AL = 1Bh

ES:DI = pointer to buffer with functionality and state information

Functionality and State Information (AH = 1Bh)

Table A-4 Functionality and State Information (AH = 1Bh)

Location	Information
[DI+00h] word	offset to static functionality information
[DI+02h] word	segment to static functionality information
[DI+04h] byte	current video mode
[DI+05h] word	character columns on screen
[DI+07h] word	page size in number of bytes
[DI+09h] word	starting address of current page
[DI+0Bh] word	cursor position for eight display pages
[DI+1Bh] word	current cursor type
[DI+1Dh] byte	current active page
[DI+1Eh] word	current CRTC address
[DI+20h] byte	current 3x8 register setting
[DI+21h] byte	current 3x9 register setting
[DI+22h] byte	number of character rows on screen
[DI+23h] word	= number of scan lines per character
[DI+25h] byte	= active display combination code
[DI+26h] byte	alternate display combination code
[DI+27h] word	number of colors supported in current mode
[DI+29h] byte	number of pages supported in current mode
[DI+2Ah] byte	0 ; 200 scan lines in current mode 1 ; 350 scan lines in current mode 2 ; 400 scan lines in current mode 3 ; 480 scan lines in current mode
[DI+2Bh] byte	Reserved
[DI+2Ch] byte	Reserved
[DI+2Dh] byte	miscellaneous state information bits 7, 6 = Reserved bit 5= 0; background intensity = 1; blinking bit 4= 1 ; cursor emulation active bit 3 = 1 ; mode set default palette loading disabled bit 2= 1 ; monochrome display attached bit 1= 1 ; summing active bit 0 = 1 ; all modes on all display active
[DI+2Eh] byte	Reserved
[DI+2Fh] byte	Reserved
[DI+30h] byte	Reserved

Table A-4 Functionality and State Information (AH = 1Bh) (Continued)

Location	Information
[DI+31h] byte	3; 256Kb of video memory available
[DI+32h] byte	save pointer information bits 7, 6 = Reserved bit 5 = 1; DCC extension active bit 4 = 1; palette override active bit 3 = 1; graphics font override active bit 2 = 1; alpha font override active bit 1 = 1; dynamic save area active bit 0 = 1; 512 character set active
[DI+33h] 13 bytes	Reserved

Table A-5 Static Functionality Table Format

static functionality table format: 0 – function not supported 1 – supported function	
Location	Information
[00h] byte	supported video mode bit 7 = mode 07h bit 6 = mode 06h bit 5 = mode 05h bit 4 = mode 04h bit 3 = mode 03h bit 2 = mode 02h bit 1 = mode 01h bit 0 = mode 00h
[01h] byte	supported video mode bit 7 = mode 0Fh bit 6 = mode 0Eh bit 5 = mode 0Dh bit 4 = mode 0Ch bit 3 = mode 0Bh bit 2 = mode 0Ah bit 1 = mode 09h bit 0 = mode 08h
[02h] byte	supported video mode bits 7 to 4 = Reserved bit 3 = mode 13h bit 2 = mode 12h bit 1 = mode 11h bit 0 = mode 10h

Table A-5 Static Functionality Table Format (Continued)

static functionality table format: 0 – function not supported 1 – supported function	
Location	Information
[03h] to [06h] bytes	Reserved
[07h] byte	Scan lines available in text modes bits 7 to 3 = Reserved bit 2= 400 scan lines bit 1 = 350 scan lines bit 0 = 200 scan lines
[08h] byte	Number of character fonts available in text modes
[09h] byte	Maximum number of character fonts that can be active in text modes
[0Ah] byte	Miscellaneous functions Bit 7= color paging bit 6= color palette (color register) bit 5 = EGA palette bit 4 = cursor emulation bit 3 = default palette loading when mode set bit 2 = character font loading bit 1 = color palette summing bit 0 = all modes supported on all displays
[0Bh] byte	scan lines available in text modes bits 7 to 4 = Reserved bit 3 = DCC supported bit 2= background intensity/blinking control bit 1= save/restore supported bit 0= light pen supported
[0Ch] to [0Dh] bytes	Reserved
[0Eh] byte	save pointer functions bits 7 to 6 = Reserved bit 5 = DCC extension supported bit 4 = palette override bit 3 = graphics font override bit 2 = alpha font override bit 1= dynamic save area bit 0 = 512-character set
[0Fh]	Reserved

A.23 AH=1Ch - Save And Restore Video State

AL = 0; return video save state buffer size requirement

CX = requested states

bit 0 = video hardware state

bit 1 = video BIOS data area

bit 2 = video DAC state and color registers

On Exit:

AL = 1Ch

BX = number of 64 bytes block required for the states requested in CX

AL = 1; save video state

CX = requested states (see AL=0)

ES:BX = pointer to buffer to store the video states information

On Exit:

AL = 1Ch

AL = 2; restore video state

CX = requested states (see AL=0)

ES:BX = pointer to buffer with previous saved video states information

On Exit:

AL = 1Ch

This page intentionally left blank.

Appendix B

VESA Functions

B.1 VESA Functions

B.1.1 Introduction to VBE

The VESA BIOS supports 16 color and HiColor modes through this VBE extension. A brief description of the VESA BIOS functions is included for completeness. For detailed information or any discrepancy, please refer to the original published documentation (VBE Core Functions Standard Ver. 2.0).

Status Information

Every function returns status information in the AX register. The format and description of the status word is as follows:

AL==4Fh: Function is supported
AL !=4Fh: Function is not supported
AH==00h: Function call successful
AH==01h: Function call failed
AH==02h: Function is not supported in the current hardware configuration
AH==03h: Function call invalid in current video mode

Software should treat a non-zero value in the AH register as a general failure condition.

Table B-1 VESA Functions

Function	Description	Reference	Compliance
AX=4F00h	Return Super VGA Information	see section B.1.2	Yes
AX=4F01h	Return Super VGA Mode Info	see section B.1.3	Yes
AX=4F02h	Set Super VGA Mode	see section B.1.4	Yes
AX=4F03h	Return Current Video Mode	see section B.1.5	Yes
AX=4F04h	Save/Restore States	see section B.1.6	Yes
AX=4F05h	Display Window Control	see section B.1.7	Yes

Table B-1 VESA Functions (Continued)

Function	Description	Reference	Compliance
AX=4F06h	Get/Set Logical Scan Line Length	see section B.1.8	Yes
AX=4F07h	Get/Set Display Start	see section B.1.9	Yes
AX=4F08h	Get/Set Palette Format	see section B.1.10	Yes
AX=4F09h	Get/Set Palette Data	see section B.1.11	Yes
AX=4F10h BL=0	Report VBE/PM Capabilities	see section B.2.1	Yes
AX=4F10h BL=1	VBE/PM Set Display Power State	see section B.2.2	Yes
AX=4F10h BL=2	VBE/PM Get Display Power State	see section B.2.3	Yes
AX=4F15h BL=0	Report DDC Capability	see section B.3.1	Yes
AX=4F15h BL=1	Read EDID	see section B.3.2	Yes

B.1.2 Function 00h - Return Super VGA Information

Input:

AH	=	4Fh	Super VGA support
AL	=	00h	Return Super VGA information
ES:DI	=	Pointer to 256-byte buffer	

Output:

AX	Status
----	--------

Comments: All other registers are preserved.

The information block has the following structure:

Table B-2 Information Block Structure

Field			Description
VgaInfoBlock struc			
VESASignature	db	'VESA'	;4 signature bytes
VESAVersion	db	200h	;VESA version number
OEMStringPtr	dd	?	;Pointer to OEM string
Capabilities	db	4 dup (?)	;Capabilities of the video;environment

Table B-2 Information Block Structure (Continued)

Field			Description
VideoModePtr	dd	?	;Pointer to supported Super VGA modes (see table below)
TotalMemory	dw	?	;Number of 64Kb memory blocks on board
OEMSoftwareRev	dw	?	; VBE implementation Software revision
OEMVendorNamePtr	dd	?	;Pointer to OEM Vendor Name String
OEMProductNamePtr	dd	?	;Pointer to OEM Product Name String
OEMProductRevPtr	dd	?	;Pointer to OEM Product Revision String
Reserved	db	222 dup (?)	;Reserved for VBE implementation scratch area
OemData	db	256 dup (?)	;Data Area for OEM Strings
VgaInfoBlock ends			

- The **VESASignature** field contains the characters VESA if this is a valid block. VBE 2.0 application should preset this field with the ASCII characters 'VBE2' to indicate to the VBE implementation that the VBE 2.0 extended information is desired, and the VBE InfoBlock is 512 bytes in size. Upon return from VBE Function 00h, this field should always be set to 'VESA' by the VBE implementation.
- **VESAVersion** is a binary field that specifies what level of the VESA standard the Super VGA BIOS conforms to.
- **OEMStringPtr** is a far pointer to a null-terminated, OEM-defined string that currently points to ATI MACH64. This pointer may point into either the ROM or RAM, depending on the specific implementation. VBE 2.0 BIOS implementations must place this string in the OemData area within the VbeInfoBlock if 'VBE2' is preset in the VbeSignature field on entry to Function 00h. This makes it possible to convert the RealMode address to an offset within the VbeInfoBlock for Protected mode applications.
- The **Capabilities** field describes the general features supported in the video environment. The bits are defined as follows:

Table B-3 Capabilities Field Description

Bit	Description
D0	DAC is switchable 0 = DAC is fixed-width, with 6 bits per primary color 1 = DAC width is switchable
D1	0 = Controller is VGA compatible 1 = Controller is not VGA compatible

Table B-3 Capabilities Field Description

Bit	Description
D2	0 = Normal RAMDAC operation 1 = When programming large blocks of information to the RAMDAC, use the blank bit in Function 09h.
D[3:31]	Reserved

- VGA compatibility is defined as supporting all standard IBM VGA modes, fonts and I/O ports; however, VGA compatibility doesn't guarantee that all modes which can be set are VGA compatible, or that the 8x14 font is available.
- The **VideoModePtr** points to a list of supported Super VGA (VESA-defined as well as OEM-specific) mode numbers. Each mode number occupies one word (16 bits). The list of mode numbers is terminated by a -1 (0FFFFh) The pointer could point into either the ROM or RAM, depending on the specific implementation. Either the list would be a static string stored in ROM, or the list would be generated at run-time in the information block (see above in RAM). It is the application's responsibility to verify the current availability of any mode returned by this function, through the **Return Super VGA mode information** (Function 1) call. Some returned modes may not be available, due to the video board's current memory and monitor configuration.
- The **TotalMemory** field indicates the maximum amount of memory physically installed and available to the frame buffer in 64KB units.
- The **OemSoftwareRev** field is a BCD value which specifies the OEM revision level of the VBE software.
- The **OemVendorNamePtr** is a pointer to a null-terminated string containing the name of the vendor which produced the display controller board product. This field is only filled in when 'VBE2' is preset in the VbeSignatur field on entry to Function 00h.
- The **OemProductNamePtr** is a pointer to a null-terminated string containing the product name of the display controller board. This field is only filled in when 'VBE2' is preset in the VbeSignatur field on entry to Function 00h.
- The **OemProductRevPtr** is a pointer to a null-terminated string containing the revision or manufacturing level of the display controller board product. This field is only filled in when 'VBE2' is preset in the VbeSignatur field on entry to Function 00h.
- The **OemData** field is a 256 byte data area that is used to return OEM information returned by VBE Function 00h when 'VBE2' is preset in the VbeSignatur field.

Table B-4 VESA Super VGA Modes

15-bit Mode Number	7-bit Mode Number	Resolution	Colors
100h	-	640x400	256
101h	-	640x480	256
102h	-	800x600	16
103h	-	800x600	256
104h	-	1024x768	16
105h	-	1024x768	256
107h	-	1280x1024	256
110h	-	640x480	32K (5:5:5)
111h	-	640x480	64K (5:6:5)
112h	-	640x480	16.8M (8:8:8)
113h	-	800x600	32K (5:5:5)
114h	-	800x600	64K (5:6:5)
115h	-	800x600	16.8M (8:8:8)
116h	-	1024x768	32K (5:5:5)
117h	-	1024x768	64K (5:6:5)
118h	-	1024x768	16.8M (8:8:8)
119h	-	1280x1024	32K (5:5:5)
11Ah	-	1280x1024	64K (5:6:5)
11Bh	-	1280x1024	16.8M (8:8:8)

The **Total Memory** field indicates the amount of memory installed on the VGA board. Its value represents the number of 64Kb blocks of memory currently installed.

B.1.3 Function 01h - Return Super VGA Mode Information

This function returns information about a specific Super VGA video mode.

Input:	AH	=	4Fh	Super VGA support
	AL	=	01h	Return Super VGA Mode Information
	CX	=	Super VGA video mode*	
	ES:DI	=	Pointer to 256-byte buffer	

Output: AX Status

Comments: All other registers are preserved.

* Mode number must be one of those returned by Function 0

The mode information block has the following structure:

Table B-5 Mode Information Block Structure

Field			Description
;mandatory information			
ModeAttributes	dw	?	;mode attributes
WinAAttributes	db	?	;window A attributes
WinBAttributes	db	?	;window B attributes
WinGranularity	dw	?	;window granularity
WinSize	dw	?	;window size
WinASegment	dw	?	;window A start segment
WinBSegment	dw	?	;window B start segment
WinFuncPtr	dd	?	;pointer to window function
BytesPerScanLine	dw	?	;bytes per scan line
;formerly optional information (now mandatory)			
XResolution	dw	?	;horizontal resolution
YResolution	dw	?	;vertical resolution
XCharSize	db	?	character cell width
YCharSize	db	?	character cell height
NumberOfPlanes	db	?	number of memory planes
BitsPerPixel	db	?	bits per pixel
NumberOfBanks	db	?	number of banks
MemoryModel	db	?	memory model type
BankSize	db	?	bank size, in KB
NumberOfImagePages	db	?	number of images
Reserved	db	1	Reserved for page function
;New Direct Color Fields			
RedMaskSize	db	?	;bit position of lsb of red mask

Table B-5 Mode Information Block Structure (Continued)

Field			Description
RedFieldPosition	db	?	;size of direct color green mask, in;bits
GreenMaskSize	db	?	;bit position of lsb of green mask
GreenFieldPosition	db	?	;size of direct color blue mask, in bits
BlueMaskSize	db	?	;bit position of lsb of blue mask
BlueFieldPosition	db	?	;size of direct color Reserved mask;;in bits
RsvdMaskSize	db	?	;bit position of lsb of Reserved mask
RsvdFieldPosition	db	?	;direct color mode attributes
DirectColorModeInfo	db	?	;bit position of lsb of red mask
;Mandatory information for VBE 2.0 and above			
PhysBasePtr	dd	?	;physical address for flat memory frame buffer
OffScreenMemOffset	dd	?	;pointer to start of off screen memory
OffScreenMem	dw	?	;amount of off screen memory in 1k units
Reserved	db	206 dup (?)	;remainder of ModeInfoBlock
ModeInfoBlock ends			

The **ModeAttributes** field describes certain important characteristics of the video mode. The field is defined as follows:

Table B-6 ModeAttributes Field

Bit	Description
D0	Mode supported in hardware: 0 = Mode is not supported in hardware 1 = Mode is supported in hardware
D1	1 (Reserved)
D2	Output functions supported by BIOS: 0 = Output functions not supported by BIOS 1 = Output functions supported by BIOS

Table B-6 ModeAttributes Field (Continued)

Bit	Description
D3	Monochrome/color mode (see note below): 0 = Monochrome mode 1 = Color mode
D4	Mode type: 0 = Text mode 1 = Graphics mode
D5	VGA compatible mode: 0 = Yes 1 = No
D6	VGA compatible windowed memory mode is available: 0 = Yes 1 = No
D7	Linear frame buffer mode is available: 0 = Yes 1 = No
D[8:15]	Reserved

- The **BytesPerScanline** field specifies the number of bytes in each logical scanline. The logical scanline could be equal to or larger than the displayed scanline.
- **WinAAttributes** and **WinBAttributes** describe the characteristics of the CPU windowing scheme, such as whether the windows exist and are read/writable, as follows:

Table B-7 CPU Windowing Scheme

Bit	Description
D0	Window supported: 0 = window is not supported 1 = window is supported
D1	Window readable: 0 = window is not readable 1 = window is readable
D2	Window writable: 0 = window is not writable 1 = window is writable
D[3:31]	Reserved

If windowing is not supported (bit **D0** = 0) for both Window A and Window B, an application can assume that the display memory buffer resides at the standard CPU address appropriate for the **MemoryModel** of the mode.

- **WinGranularity** specifies the smallest boundary, in KB, on which the window can be placed in the video memory. The value of this field is undefined if Bit D0 of the appropriate **WinAttributes** field is not set.
- **WinSize** specifies the size of the window, in KB.
- **WinASegment** and **WinBSegment** addresses specify the segment addresses where the windows are located in the CPU address space.
- **WinFuncAddr** specifies the address of the CPU video memory windowing function. The windowing function can be invoked either through **VESA BIOS function 05h** or by calling the function directly. A direct call will provide faster access to the hardware paging registers than using Int 10h, and is intended to be used by high-performance applications. If this field is Null, Function 05h must be used to set the memory window, if paging is supported.
- **XResolution** and **YResolution** specify the height and width of the video mode, in pixels.
- **XCharCellSize** and **YCharCellSize** specify the size of the character cell, in pixels.
- The **NumberOfPlanes** field specifies the number of memory planes available to software in that mode. For standard 16-color VGA graphics, this would be set to 4. For standard packed pixel modes, the field would be set to 1.
- The **BitsPerPixel** field specifies the total number of bits that define the color of one pixel. For example, a standard VGA 4-plane, 16-color graphics mode would have a 4 in this field, and a packed-pixel, 256-color graphics mode would specify 8 in this field. The number of bits per pixel *per plane* can normally be derived by dividing the **BitsPerPixel** field by the **NumberOfPlanes** field.
- The **MemoryModel** field specifies the general type of memory organization used in this mode. The following models have been defined:

Table B-8 MemoryModel Field

Bit	Description
00h	Text mode
01h	CGA graphics
02h	Hercules graphics
03h	4-plane planar
04h	Packed pixel

Table B-8 MemoryModel Field (Continued)

Bit	Description
05h	Non-chain 4, 256 color
06h	Direct Color
07h	YUV
08:0Fh	Reserved, to be defined by VESA
10:FFh	To be defined by OEM

In version 1.1 and earlier of the VESA Super VGA BIOS Extension, OEM-defined Direct Color video modes with pixel formats 1:5:5:5 and 8:8:8:8 were described as a **Packed Pixel** model with 16, 24, and 32 bits per pixel, respectively.

- **NumberOfBanks** is the number of banks in which the scan lines are grouped. This field is set to 1.
- The **BankSize** field specifies the size of a bank, in units of 1KB. This field is set to 0.
- The **NumberOfImagePages** field specifies the number of additional, complete display images that will fit into the memory, at one time, in this mode. The application may load more than one image into the memory if this field is non-zero, and flip the display between the images.
- The Reserved field has been defined to support a future VESA BIOS extension feature, and will always be set to 1 in this version.
- The **RedMaskSize**, **GreenMaskSize**, **BlueMaskSize**, and **RsvdMaskSize** fields define the size, in bits, of the red, green, and value components of a direct color pixel. A bit mask can be constructed from the MaskSize fields, using simple shift arithmetic. For example, the MaskSize values for a Direct Color 5:6:5 mode would be 5, 6, 5, and 0, for the red, green, blue, and Reserved fields, respectively.
- The **RedFieldPosition**, **GreenFieldPosition**, **BlueFieldPosition**, and **RsvdFieldPosition** fields define the bit position within the direct color pixel or YUV pixel of the lsb of the respective color component. A color value can be aligned with its pixel field by shifting the value left by the FieldPosition. For example, the FieldPosition values for a Direct Color 5:6:5 mode would be 11, 5, and 0, for the red, green, blue, and Reserved fields, respectively.
- The **DirectColorModeInfo** field describes important characteristics of direct color modes. **Bit D0** specifies whether the color ramp of the DAC is fixed or programmable. If the color ramp is fixed, it cannot be changed. If the color ramp is programmable, it is assumed that the red, green, and blue lookup tables can be loaded using a standard VGA DAC color registers BIOS call (AX=1012h). **Bit D1** specifies whether the bits in the **Rsvd** field of the direct color pixel can be used by the

application, or are Reserved, and thus unusable.

Table B-9 DirectColorModelInfo Field

Bit	Description
D0	Color ramp is fixed/programmable: 0 = color ramp is fixed 1 = color ramp is fixed
D1	Bits in Rsvd field are usable/Reserved: 0 = bits in Rsvd field are Reserved 1 = bits in Rsvd field are usable by the application

- The **PhysBasePtr** is a 32-bit physical address of the start of frame buffer memory when the controller is in flat frame buffer memory mode. If this mode is not available, then this fields will be zero.
- The **OffScreenMemOffset** is a 32-bit offset from the start of frame buffer memory. Extra off-screen memory that is needed by the controller may be located either before or after this off-screen memory, be sure to check OffscreenMemSize to determine the amount of off-screen memory which is available to the application.
- The **OffScreenMemSize** contains the amount of available, contiguous off-screen memory in 1k units, which can be used by the application.

B.1.4 Function 02h - Set Super VGA Video Mode

This function initializes a video mode. The BX register contains the mode to set.

Input:

AH	=	4Fh	Super VGA support
AL	=	02h	Set Super VGA video mode
BX	=		Video mode D[0:8] = Video mode D[9-13] = Reserved (must be 0) D14 = frame buffer model: 0 = use windowed frame buffer model 1 = use linear/flat frame buffer model D15 = Clear memory flag: 0 = clear video memory 1 = don't clear video memory
ES:DI	=		Pointer to 256-byte buffer

Output:

AX		Status
----	--	--------

Comments: All other registers are preserved.

B.1.5 Function 03h - Return Current Video Mode

This function returns the current video mode in BX.

Input:

AH	=	4Fh	Super VGA support
AL	=	03h	Return current video mode

Output:

AX	=	Status
BX	=	Current video mode
		D[0-13] = Video mode
		D14 = 0, use windowed frame buffer model
		= 1, use linear/flat frame buffer model
		D15 = 0, clear video memory
		= 1, don't clear video memory

Comments: All other registers are preserved.

B.1.6 Function 04h - Save/Restore State

This function provides a complete mechanism to save and restore the display controller hardware state.

Input:

AH	=	4Fh	Super VGA support
AL	=	04h	Save and restore state
DL	=	00h	Return Save/Restore state buffer size
		01h	Save state
		02h	Restore state
CX	=	Requested states	
		D0 = Save/Restore controller hardware state	
		D1 = Save/Restore BIOS state	
		D2 = Save/Restore DAC state	
		D3 = Save/Restore Register state	
ES:BX	=	Pointer to buffer (if DL <> 00h)	

Output:

AX	=	Status
BX	=	Number of 64-byte blocks to hold the state buffer (if DL = 00h)

Comments: All other registers are preserved.

B.1.7 Function 05h - Display Window Control

This function sets or gets the position of the specified display window or page in the frame buffer memory by adjusting the necessary hardware paging registers. To use this function properly, the software should use **VESA BIOS Function 01h** (Return Super VGA mode information) to determine the size, location, and granularity of the windows.

Input:

AH	=	4Fh	Super VGA support
AL	=	05h	Super VGA display window control
BH	=	00h	Set memory window
BL	=	Window number: 0 = Window A 1 = Window B	
DX	=	Window number in video memory (in window granularity units)	

Output: AX Status

Comments: See notes below.

Input:

AH	=	4Fh	Super VGA support
AL	=	05h	Super VGA display window control
BH	=	01h	Get memory window
BL	=	Window number: 0 = Window A 1 = Window B	

Output:

AX	Status
DX	= Window number in video memory (in window granularity units)

Comments: See notes below.

Notes:

- This function is also directly accessible through a far call from the application. The address of the BIOS function may be directly obtained by using VESA BIOS function 01h (return Super VGA mode information). Afield in the ModeInfoBlock contains the address of this function. Note that this function may be different among video modes in a particular BIOS implementation, so the function pointer should be obtained after each set mode.
- In the far call version, no status information is returned to the application. Also, in the far call version, the AX and DX registers will be destroyed. Therefore, if AX and/or DX must be preserved, the application must do so before making the call
- The application must load the input arguments in BH, BL, and DX (for set window), but does not need to load either AH or AL in order to use the far call version of this function.

B.1.8 Function 06h - Set/Get Logical Scan Line Length

This function sets or gets the length of a logical scan line. It allows an application to set up a logical video memory buffer that is wider than the displayed area. Function 07h then allows the application to set the starting position that is to be displayed.

Input:	AH	=	4Fh	Super VGA support
	AL	=	06h	Logical scan line length
	BL	=	00h	Set scan line length in Pixel
		=	02h	Set scan line length in Byte
	CX	=	Desired width, in pixels (if BL = 00h)	
		=	Desired width, in byte (if BL = 02h)	

Output:	AX	=	Status
	BX	=	Bytes per scan line
	CX	=	Actual pixels per scan line
	DX	=	Maximum number of scan lines

Comments: See notes below.

Input:

AH	=	4Fh	Super VGA support
AL	=	06h	Logical scan line length
BL	=	01h	Get scan line length
	=	03h	Get maximum scan line length

Output:

AX	=	Status
BX	=	Bytes per scan line
CX	=	Actual pixels per scan line
DX	=	Maximum number of scan lines

Comments: See notes below.

Notes:

- The desired width, in pixels, may not be achievable because of hardware limitations. The next-larger value that will accommodate the desired number of pixels will be selected, and the actual number of pixels will be returned in CX. BX returns a value, which when added to a pointer into video memory, will point to the next scan line.
- The *mach64* implementation only supports this function in 256 color mode and above.

B.1.9 Function 07h - Set/Get Display Start

This function selects the pixel to be displayed in the upper left corner of the display from the logical page. This function can be used to pan and scroll around logical screens that are larger than the displayed screen. This function can also be used to rapidly switch between two, different displayed screens for double-buffered animation effects.

Input:

AH	=	4Fh	Super VGA support
AL	=	07h	Display start control
BH	=	00h	Reserved, must be 0
BL	=	00h	Set display start
	=	80h	Set display start during vertical retrace
CX	=		First displayed pixel in scan line
DX	=		First displayed scan line

Output:

AX	=	Status
BX	=	Bytes per scan line

CX = Actual pixels per scan line
DX = Maximum number of scan lines

Comments: See a note below.

Input:

AH = 4Fh Super VGA support
AL = 07h Display start control
BH = 00h Reserved, **must be 0**
BL = 01h Get display start

Output:

AX = Status
BH = Reserved, **and will be 0**
CX = First displayed pixel in scan line
DX = First displayed scan line

Comments: See a note below.

Note: The *mach64* implementation only supports this function in 256 color mode and above.

B.1.10 Function 08h - Set/Get AC Palette Format

This function manipulates the operating mode or format of the DAC palette. Some DACs are configurable to provide 6 bits, 8 bits, or more of color definition per red, green, and blue primary colors. The DAC palette width is assumed to be reset to the standard VGA value of 6 bits per primary color during any mode set.

Subfunction 0 - Set AC Palette Format

Input:

AH = 4Fh VESA Extension
AL = 08h Set/Get AC Palette Format
BL = 00h Set AC Palette Format
BH = Desired bits of color per primary

Output:

AX = Status
BH = Current number of bits of color per primary

Subfunction 1 - Get AC Palette Format

Input:	AH	=	4Fh	VESA Extension
	AL	=	08h	Set/Get AC Palette Format
	BL	=	01h	Get AC Palette Format
Output:	AX	=	Status	
	BH	=	Current number of bits of color per primary	

B.1.11 Function 09h - Set/Get AC Palette Data

This required function is very important for any/all RAMDAC larger than a standard VGA RAMDAC. The standard INT 10h BIOS Palette function calls assume standard VGA ports and VGA palette widths. This function offers a palette interface that is independent of the VGA assumptions.

Input:	AH	=	4Fh	VESA Extension
	AL	=	09h	Set/Get AC Palette Format
	BL	=	00h	Set Palette Data
			01h	Get Palette Data
			02h	Set Secondary Palette Data
			03h	Get Secondary Palette Data
			03h	Set Palette Data during Vertical Retrace with Blank Bit on
			80h	
	CX	=	Number of palette registers to update (to a maximum of 256)	
	DX	=	First of the Palette registers to update (start)	
	ES:DI	=	Table of palette value	
Output:	AX	=	Status	

B.2 Power Management Services

B.2.1 VBE/PM Function 0 - Report VBE/PM Capabilities

Input:	AH	=	4Fh	VESA Extension
	AL	=	10h	VBE/PM Services
	BL	=	00h	Report VBE/PM Capabilities
	ES:DI	=	Null pointer; must be 0000:0000h in version 1.0 (Reserved for future use).	
Output:	AX	=	Status	
	BH	=	Power saving state signals supported by the controller: 1 = supported, 0 = not supported	
			bit 0 = STANDBY	
			bit 1 = SUSPEND	
			bit 2 = OFF	
			VBE/PM Version number (0001 0000b for this version)	
	BL	=	bits 0:3 = Minor Version number	
			bits 4:7 = Major Version number	
	ES:DI	=	Unchanged	

B.2.2 VBE/PM Function 1 - Set Display Power State

Input:	AH	=	4Fh	VESA Extension
	AL	=	10h	VBE/PM Services
	BL	=	01h	Set Display Power State
	BH	=	Requested Power state:	
			00h =	ON
			01h =	STANDBY
			02h =	SUSPEND
			04h =	OFF
Output:	AX	=	Status	
	BH	=	Unchanged	

B.2.3 VBE/PM Function 2 - Get Display Power State

Input:	AH	=	4Fh	VESA Extension
	AL	=	10h	VBE/PM Services
	BL	=	02h	Get Display Power State
Output:	AX	=	Status	
	BH	=	Power state currently requested by the controller:	
		00h =	ON	
		01h =	STANDBY	
		02h =	SUSPEND	
		04h =	OFF	

B.3 Display Identification Extensions

The VESA VBE sub-function 15h is used to implement the VBE/DDC services. The VBE/DDC services are defined below and are not included in the VBE Standard documentation.

B.3.1 VBE/DDC Function 0 - Report VBE/DDC Capabilities

Input:	AH	=	4Fh	VESA Extension
	AL	=	15h	VBE/DDC Services
	BL	=	00h	Report DDC Capabilities
	CX	=	00h	Controller unit number (00=primary controller)
	ES:DI	=	Null pointer, must be 0:0 in version 1.0 (Reserved for future use).	
Output:	AX	=	Status	
	BH	=	Approximate time in seconds, rounded up, to transfer one EDID block (128 byte)	
	BL	=	DDC level supported (*): bit0=0 DDC1 not supported; =1 DDC1 supported; bit1=0 DDC2 not supported; =1 DDC2 supported; bit2=0 Screen not blanked during data transfer (**); =1 Screen blanked during data transfer.	
	CX	=	Unchanged	

ES:DI = Unchanged

Comments
: All other registers may be destroyed.

(*) DDC level supported by both the display and the controller.

(**) This refers to the behavior of the controller and the VBE/DDC SW.

B.3.2 VBE/DDC Function 1 - Read EDID

Input:

AH	=	4Fh	VESA Extension
AL	=	15h	VBE/DDC Services
BL	=	01h	Read EDID
CX	=	00h	Controller unit number (00=primary controller)
DX	=	00h	EDID block number. Zero is only a valid value in version 1.0
ES:DI	=	Pointer to area in which the EDID block (128 bytes shall be returned).	

Output:

AX	=	Status(*)
BH	=	Unchanged
CX	=	Unchanged
ES:DI	=	Pointer to area in which the EDID block is returned.

Comments
: All other registers may be destroyed.

Appendix C

Extended Bios Functions

C.1 Extended Functions

BIOS_ADDR:64h

all functions return error code in AH

ah = 0 - no error
ah = 1 - function completed with error
ah = 2 - function is not supported

Definitions

DISPLAY DEVICE ID	= 0	-CRT
	= 1	-TV
	= 2	- DFP
DISPLAY DEVICE MASK [0]	=	CRT
	[1]	= TV
	[2]	= DFP
	[3]	= LCD
CRT STANDARD	= 0	- NO MONITOR
	= 1	- MONOCHROME MONITOR
	= 2	- COLOR MONITOR
DFP STANDARD [0]	= 0	- TFT
	[2] = 1	- scalable DFP
other bits and values are reserved		
TV STANDARD	= 1	- NTSC
	= 2	- PAL
	= 3	- PALM
	= 4	- PAL60
	= 5	- NTSC-J
	= 6	- SCART RGB

TVSTANDARDMASK	[0]	= 1	- NTSC
	[1]	= 1	- PAL
	[2]	= 1	- PALM
	[3]	= 1	- PAL60
	[4]	= 1	- NTSC-J
	[5]	= 1	- SCART RGB

C.1.1 AL = 00h - Set Display Mode

cl[3-0]	= color depth parameter
= 1	- 4 bpp
= 2	- 8 bpp
= 3	- 15 bpp (555)
= 4	- 16 bpp (565)
= 5	- 24 bpp in RGB format
= 6	- 32 bpp in xRGB format
= 7	- 16 bpp (4444)
= 8	- 16 bpp (Ind8)
ch	= mode number
= E1h	- 640x400
= E2h	- 320x200
= E3h	- 320x240
= E4h	- 512x384
= E5h	- 400x300
= E6h	- 640x350
= 12h	- 640x480
= 6Ah	- 800x600
= 55h	- 1024x768
= 81h	- load CRTC table from buffer pointed to by dx:bx
= 82h	- load CRTC table from frame buffer, 32-bit offset in dx:bx (supported in VGA disable products)
= 83h	- 1280x1024
= 84h	- 1600x1280
dx:bx	= pointer to internal parameter table if ch = 81h
dx:bx	= 32-bit linear address offset (in DWORD boundary) into the frame buffer if ch = 82h

C.1.2 AL = 01h - Set Display Controller State

This function is used to setup the pre-condition to allow the controller to go into VGA or Extended mode. This function does not actually program the CRT Controller. However, this function will program the DAC to the color depth required by the display. This function will be automatically invoked if a Set Mode is called through the BIOS.

Input

cl	= 0	- VGA
	= 1	- Extended

C.1.3 AL = 02h - Set DAC State

Input

cl	= 0	- set DAC to active mode (This function will not alter number of bits for the DAC)
	= 1	- set DAC to sleep mode
	= 2	- set DAC to 6 bit
	= 3	- set DAC to 8 bit

C.1.4 AL = 03h - Program Specified Clock Entry

Input

cl[2-0]	= 0	- CORE CLOCK, engine clock (currently disabled for Radeon)
	= 1	- MEMORY CLOCK, memory clock (currently disabled for Radeon)
	= 2	- PCLK, dot clock
ch	= entry in the frequency table for programming PCLK	
bx	= clock frequency in units of [MHz/100] (e.g. 25.18 MHz=2518 [MHz/100])	

Output

al	= clock chip type (4 = Radeon)
bx, cl	= programming word depending on type

C.1.5 AL = 04h - Short Query Function 0

Output

ch[3-0]	= DAC type (hardcoded to 0 for Radeon)
ch[7,6, 5, 4]	= sync on green, gamma correction, 8bit, sleep
cl	= Color depth support (bit definition):
[7] = 0	- 4 bpp
[4] = 1	- 32 bpp (unpack 24 bpp in xRGB, x is MSB)
[3] = 1	- RGB in 24 bpp
[2] = 1	- 15 bpp (555)
[1] = 1	- 16 bpp (565)
[0] = 1	- 8 bpp
dl[2-0] = 000b	- generic bios
= 011b	- fixed frequency monitor BIOS, should only use default CRTC in BIOS
= 010b	- fixed frequency monitor BIOS, and can use external CRTC values
bl[3-0]	= bus type (hardcoded 01 = IS_AGP_BUS)
si	= subsystem vendor id
di	= subsystem id

C.1.6 AL = 05h - Short Query Function 1

Output

cl[3-0]	= card id;
dx	= IO Base Address
si	= PCI bus/dev/func information
di	= BIOS segment address

C.1.7 AL = 06h - Short Query Function 2

Output

al	= revision id (currently hardcoded to 0)
bx	= aperture address (frame buffer address in Mbytes)
cl	= memory size, number of 512K blocks
dx	= PCI device id
di:si	= alternative aperture address (memory mapped registers in linear 32-bit aperture)

C.1.8 AL = 07h - Query Graphics Hardware Capability and Capture Width Info

Output

dx:di - pointer (seg:off) to a table specifying max dot clock information (see below); the table is terminated by a zero in the first column
dx:[di-1] = number of bytes per row
dx:[di-2] = format type
cl = support mask to be used

Note: There exists an assembly directive to disable this function. Currently this function is enabled.

H_DISP	SUPPORTMASK (use bit 7-4 only)	MEMREQ	MAX DOTCLOCK	PIXEL WIDTH
0(end of table)				

H_DISP = horizontal resolution in number of characters
SUPPORTMASK = a bit value to indicate the valid condition of the entry
MEMREQ = minimum memory required to support the specified resolution and color depth (number of 512K blocks)
MAX DOTCLOCK = max dot clock with the specified resolution and color depth in MHz
PIXEL WIDTH = color depth parameter

To determine if a video mode is supported, the following algorithm can be used

```
if(      ( H_DISP <= horizontal display (in char)&&
          ( SUPPORTMASK & cl )                      &&
          ( MEMREQ <= video memory size )            &&
          ( MAX DOTCLOCK >= dot clock of the requested mode )&&
          ( PIXEL WIDTH >= requested color depth )
      )
then
    the mode can be supported
else
    the mode cannot be supported.
```

dx:si = pointer (seg:off) to a table specifying maximum capture widths. The table is terminated by a zero in the first column. If si == 0 - no table is provided and drivers should use default settings.

dx:[si-1] = number of bytes per row

dx:[si-2] = format type

H_DISP	SCALER SOURCE	MEMREQ	MAX DOTCLOCK	PIXEL WIDTH	MAX CAPTURE SIZE
0(end of table)					

H_DISP = horizontal resolution in number characters

SCALER SOURCE [7] = 1 - scaler source format is in 32 bpp aRGB888

SCALER SOURCE [6] = 1 - scaler source format is in 15 bpp aRGB, 16 bpp RGB565, YUV12, VYUY422, or YVYU422

MEMREQ = minimum memory required to support the specified resolution and color depth (in number of 512K blocks)

MAX DOTCLOCK = max dot clock with the specified resolution and color depth in MHz

PIXEL WIDTH = color depth parameter

MAX CAPTURE SIZE = the max capture width in number of characters

To determine the max capture width for a video mode, the following algorithm can be used

```

if ( ( H_DISP >= horizontal disp( in char) &&
      ( SCALER SOURCE & scaler source ) &&
      ( MEMREQ <= current memory size ) &&
      ( MAX DOTCLOCK >= dot clock of the requested mode)&&
      ( PIXEL WIDTH >= requested color depth) )
then
    max capture width = MAX CAPTURE SIZE

```

C.1.9 AL = 08h - Query Installed Modes

Input

di = DISPLAY DEVICE ID (0 = CRT, 1 = TV, 2 = FP)
dx:bx = pointer to buffer (64 bytes)

Output

dx:bx = pointer to a list of supported modes terminated by a zero.

Note: There exists an assembly directive to disable this function. Currently, this function is supported on Radeon.

C.1.10 AL = 09h - Query Supported Mode

Input

di = DISPLAY DEVICE ID
cl = color depth parameter (see function 0)
ch = mode number as returned by Query Installed Modes (al=08h), or as specified in Set Display Mode (al=00h)
dx:bx = pointer to buffer (64 bytes)

Output

dx:bx = pointer to CRT parameter table (if the mode is supported)

Note: There exists an assembly directive to disable this function. Currently this function is enabled for Radeon.

C.1.11 AL = 0Ah - Display Power Management Service (DPMS)

Input

Ch	= 0	- set DPMS mode
cl[2-0]	= 0	- active
	= 1	- stand-by
	= 2	- suspend
	= 3	- off
	= 4	- blank the display (this is not a DPMS state)
ch	= 1	- return current DPMS state

Output

cl = current DPMS state (as above)

C.1.12 AL = 0Bh - Display Data Channel (DDC) Service

Input

bh = DISPLAY DEVICE ID
bl = 0 - return DDC format supported by Graphics Controller and Monitor

Output

bx = 0 - DDC not supported by monitor
[0] = 1 - DDC1 supported by monitor
[1] = 1 - DDC2B supported by monitor

al = 0 - DDC not supported by BIOS
[0] = 1 - DDC1 supported by BIOS
[1] = 1 - DDC2B supported by BIOS
[2] = 1 - DDC2AB supported by BIOS
[3] = 1 - DDC2Bi supported by BIOS

[6] = 1 - BIOS supports detailed EDID timing at power-up
[7] = 1 - BIOS can use/uses EDID to setup the board at power-up

bl = 1 - read EDID data (support DDC1/DDC2B only, first EDID block for DDC2B)

cx = buffer size

dx:di = pointer to buffer (not less than 256 bytes)

Output

dx:di = EDID data

bl = 2 - read from device to buffer (only supported by DDC2B/2AB/2Bi), master read

cx = buffer size

dx:di = pointer to buffer (monitor address in first byte of dx:di when calling)

Output

dx:di = pointer to buffer with data read

bl = 3 - write to device from buffer[/read from device to buffer] (only supported by DDC2B/2AB/2Bi), master write[/slave read]

cx = bytes to write

dx:di = pointer to buffer

dx:[di]...dx:[di + cx - 1] = data to write

dx:[di+cx] = max bytes to read after write (<= actual buffer size)

dx:[di+cx+1] = waiting limit for slave read in msec

Output

dx:di = pointer to buffer with data read (if required)

bl = 4 - return DDC format supported by BIOS

Output

Al = 0 - DDC not supported by BIOS

[0] = 1 - DDC1 supported by BIOS

[1] = 1 - DDC2B supported by BIOS

[2] = 1 - DDC2AB supported by BIOS

[3] = 1 - DDC2Bi supported by BIOS

[6] = 1 - BIOS supports detailed EDID timing at power-up

[7] = 1 - BIOS can use/uses EDID to setup the board at power-up

C.1.13AL = 0Ch - Save and Restore Graphics Controller Data

Input

ch[1-7] = 0 - reserved

cl = 0 - return buffer size required to fit saved data in number of bytes

Output

cx = buffer size

cl = 1 - save controller data

dx:di = pointer to buffer

Output

dx:di = saved data
cl = 2 - restore controller data
dx:di = pointer to buffer
cl = 3 - reinitialize controller from D3 power saving mode
bx = 0

Note (1) that this function only works if the base io address is the same as the default io address at POST.

Note (2): ch option to include GUI register not supported anymore. Do not include GUI registers.

C.1.14AL = 0Dh - Get/Set Refresh Rate (CRT only)

Input

bl = 0 - Get current refresh rate information
= 1 - Change current refresh rate information
= 2 - Save refresh rate information (not supported)
dx:di = pointer to buffer (min 20 bytes required and is terminated by 0FFFFh)

Table C-1

offset(word)	Content
0	12h(640x480),
1	12h(640x480) refresh mask bit 2 = 85Hz bit 1 = 75Hz bit 0 = 72Hz if bits = 0; 60Hz
2	6Ah(800x600)
3	6Ah(800x600) refresh mask bit 4 = 85Hz bit 3 = 56Hz bit 2 = 60Hz bit 1 = 72Hz bit 0 = 75Hz
4	55h(1024x768)

Table C-1 (Continued)

offset(word)	Content
5	55h(1024x768) refresh mask bit 4 = 85Hz bit 3 = 87Hz Interlaced bit 2 = 60Hz bit 1 = 70Hz bit 0 = 75Hz
6	83h(1280x1024)
7	83h(1280x1024) refresh mask bit 5 = 85Hz bit 4 = 43Hz bit 3 = 47Hz bit 2 = 60Hz bit 1 = 70Hz bit 0 = 75Hz
8	0FFFFh

= 3 - restore factory default refresh rate information

The following functions are available if TV or DFP is supported

C.1.15AL = 13h - Extended FP related functions

This function is not supported for RADEON

C.1.16AL = 14h - Detect CRT/ TV /DFP/LCD

Input

ch[0]	= 0	- return CRT information based on previous detection
	= 1	- return current CRT information by detection
ch[1]	= 0	- return TV information based on previous detection
	= 1	- return current TV information by detection
ch[2]	= 0	- return DFP information based on previous detection
	= 1	- return current DFP information by detection
ch[3]	= 0	- return LCD information based on previous detection
	= 1	- return current LCD information by detection
ch[4]	= 1	- force CRT to be detected

ch[5]		- reserved
ch[6]	= 1	- force DFP to be detected
ch[7]		- reserved

Output

bh	= TV STANDARD	
bl[0]	= 0	- no DFP attached
	= 1	- DFP is attached
bl[1]	= 0	- no LCD attached
	= 1	- LCD attached
ch	= 000b	- no TV attached
	= 001b	- TV attached to composite connector
	= 100b	- TV attached to S-Video connector
	= 101b	- TV sets attached to both composite and S-Video connectors
cl	= CRT STANDARD	

C.1.17 AL = 15h - Get/Set Active Display(s)

Input

ch = 0	- return data on display(s) currently set to active (will be on at next Set Mode)
--------	---

Output

cl	= active display(s)	
[0]	= 1	- CRT
[1]	= 1	- TV
[2]	= 1	- DFP/LCD
[3]	= 1	- Auto-switch
[6-4]	= 0000b	- reserved
ch = 1	- set active display(s) (will be on at next Set Mode)	
cl	= requested display	
[0]	= 1	- CRT
[1]	= 1	- TV
[2]	= 1	- DFP/LCD
[3]	= 1	- Auto-switch
[6-4]	= 0000b	- reserved

C.1.18AL = 16h - Get/Set TV Standard

Input

ch = 0 - return current TV standard

Output

ch = current active TV STANDARD value
cl = TV standard mask that can be supported on the fly
ch = 1 - set TV standard, will be on at next Set Mode
cl = TV STANDARD value

Note: This function will return an error is dynamic switching of TV standard is not supported.

C.1.19AL = 17h - Get TV Out Info

Input

di = 0 - get TV Out information

Output

ch = TV Out chip major revision code
bx = TV Out chip minor revision code
cl = Reference Frequency
= 0 - 29.49892 MHz
= 1 - 28.63636 MHz
= 2 - 14.31818 MHz
= 3 - 27.00000 MHz
dx = 0 - no TV Out capability is detected
= 1 - TV Out is detected but not supported in BIOS
= 3 - TV Out is detected and is supported in BIOS
di = 1 - reset Graphics Controller DSP value based on current setting

OUTPUT

None.

Note: This function will return an error code if TV Out is not supported.

This page intentionally left blank.

Appendix D

BIOS Header Description

D.1 BIOS Header

This table describes the content of the first 128bytes of the video BIOS binary file.

To maintain industry standard compatibility as well as consistence within ATI products, this information should not be altered.

Table D-1 Video BIOS Binary File Contents

Byte Offset	Content
0,1	055h, 0aah ROM signature
2	BIOS size in 512 byte blocks (this area must checksum to 0)
3,4	jump pointer to initialization entry point
7-17h	PCI reserved space
18h,19h	pointer to PCI Data Structure
1eh,1fh,20h	'IBM' for compatibility with old VGA programs
21h	checksum byte
30h-39h	' 761295520' ATI Product signature
40h,41h	'??' used by ATI for different purposes in pre-Rage ASIC days.
42h	hardware version
48h	pointer to the BIOS Header information table block
50h-5fh	reserved for time stamp (put in by CXCHKSUM)
62h	OEM_ID
64h,65h	jump pointer to entry point of ATI extended functions
68h,69h	jump pointer to entry point of VGA functions
70h-7fh	reserved for HW straps
80h-	bios install message ending with odh, 0ah and 0
	copyright notice
	bioskit version number
	configuration file name

D.1.1 Initialization Table Description

The BIOS header stores information for the BIOS and driver's use. The offset 48h has a pointer to the structure below. This information, called the BIOS header, is intended to provide the hardware specifications and settings to graphics and multimedia drivers.

There are pointers in this header that points to two types of data blocks: information and programming blocks. The information blocks contain the current settings of the hardware components. The programming blocks contain the data for programming registers. For example, PLL programming block contains PLL initialization programming table which can be used to initialize PLL.

It is recommended that the information header table type in the first byte be checked before using the information as the structure may be slightly different from one type to another.

Table D-2

Byte offset	Content	Rage 128	M128	RADEON
0	=4, type definition, 2=Rage128 & Rage128Pro, 3=M3, 4=Radeon, 5=Piglet	2	3	4
1	Extended function code, 0A0h,0A1h...etc.	x	x	x
2	OEM_ID1	x	x	x
3	OEM_ID2	x	x	x
4	BIOS_MAJOR_REV	x	x	x
5	BIOS_MINOR_REV	x	x	x
6-7	Size of structure in number of bytes	x	x	x
8-9	Pointer to SMI (BIOS entry + 1)	x	x	x
0Ah-0Bh	Pointer to PMID	x	x	x
0Ch-0Dh	Pointer to initialization table	x	x	x
0Eh-0Fh	Pointer to CRC checksum block	x	x	x
10h-11h	Pointer to configuration file name string	x	x	x
12h-13h	Pointer to logon message string	x	x	x
14h-15h	Pointer to misc. information string	x	x	x
16h-17h	PCI bus/dev/func code	x	x	x
18h-19h	BIOS runtime segment address	x	x	x
1Ah-1Bh	IO base address	x	x	x
1Ch-1Dh	Subsystem vendor id	x	x	x
1Eh-1Fh	Subsystem id	x	x	x
20h-21h	POST vendor id	x	x	x

Table D-2 (Continued)

Byte offset	Content	Rage 128	M128	RADEON
22h-23h	INT 10h offset, "coprocessor only" BIOS	x	x	x
24h-25h	INT 10h segment, "coprocessor only" BIOS	x	x	x
26h-27h	Monitor information, OEM specific	x	x	x
28h-29h	Pointer to configuration block (if non-zero)	x	x	x
2Ah-2Bh	Pointer to DAC pipeline delay information	x	x	x
2Ch-2Dh	Pointer to max. color depth table	x	x	x
2Eh-2Fh	Pointer to internal CRTC tables	x	x	x
30h-31h	Pointer to PLL information block	x	x	x
32h-33h	Pointer to TV information table (if non-zero)	x	x	x
34h-35h	Pointer to DFP information table (if non-zero)	x	x	x
36h-37h	Pointer to hardware configuration table	x	x	x
38h-39h	Pointer to multimedia configuration table (if non-zero)	x	x	x
3Ah-3Dh	TV standard patch table signature \$TVS (if Dynamic BootUp TV Standard is supported, else this field contains zeros)	x	x	x
3Eh-3Fh	Pointer to TV standard patch table (if non-zero and if offset 3Ah-3Dh is equal to \$TVS)	x	x	x
40h-41h	Pointer to Panel information table (if non-zero)		x	x
42h-43h	Pointer to Mobile information block (if non-zero)		x	x
44h-45h	Pointer to Aurora information block	x		x
46h-47h	Pointer to PLL init. Programming block			x
48h-49h	Pointer to supported memory configuration block			x
4ah-4bh	Pointer to save mask block			x
4ch-4dh	Pointer to hard coded EDID data (if non-zero)			x
4eh-4fh	Pointer to the second initialization block			x
50h-52h	Pointer to second information block (if non-zero)			x

Initialization using the BIOS Header Information Tables

The RADEON can be initialized and configured using the BIOS Header Information Tables. Use the tables in the following order to initialize the chip:

-
- 1 Use Initialization Block to initialize RADEON specific registers that need to be initialized right after power up.
 - 2 Use PLL Programming Block to program the PLLs.
 - 3 Use the Second Initialization Block to initialize the rest of the RADEON specific registers
 - 4 Use the Memory Configuration Block to configure and auto detect the memory size and type.

D.1.2 Initialization Block

These tables, which are labeled `rage_regs_1` and `rage_regs_2`, contains the registers to be programmed when the chip is powered up. These registers are programmed before anything else. The following applies to the first and second initialization block.

Each entry of the table can be one of the following forms:

- 1 Offset in word followed by dword programming value – total 6bytes long
- 2 Offset in word followed by dword AND mask and dword OR mask – total 10 bytes long
- 3 Delay indicator in word followed by delay time in word – total 4 bytes
- 4 End of table indicator in word – total 2 bytes: value 0.

The (offset-1) to the table contains the table revision number.

More specifically, the first (offset) word has the following flags:

Bit 15 : 1 if the following value is the delay in micro second

Bit 14 : 1 if the following value is four byte AND mask and four byte OR mask instead of 4 byte programming value

Bit 13 : 1 if using I/O instead of MM_INDEX. Next four byte indicates I/O Address.

Bits 12- 0 : register address

Example:

```
Db 0                ; revision number
Table_start:
Dw 02050h           ; use I/O to program dword
Dd 012345678h       ; dword programming value
```

Dw 04140h	; do AND and OR
Dd 0edffffffh	; AND word first
Dd 012000000h	; then OR word
Dw 02056h	; use I/O and do AND and OR
Dd 00ffffffh	; AND mask
Dd 0f0000000h	; OR mask
Dw 08000h	; delay in micro seconds
Dw 00050h	; delay 50 micro seconds
Dw 0000h	; end of the table

Second Initialization Block

The structure of this table is identical to the first table above. This table is normally used after clock is initialized.

D.1.3 PLL Programming Block

After the RADEON specific registers have been initialized in the first initialization block, the PLLs are initialized. The PLL programming block is a table pointed to at offset 46-47h of the BIOS header.

This programming block contains a table of data that can be used to initialize the PLLs.

The size of each entry varies. Each entry of the table can be one of the following forms:

- 1** Offset (bits 5-0) in byte followed by dword programming value – total 5 bytes long.
- 2** Offset (bits 5-0) and bit 6 ON in byte followed by 3 bytes AND/OR indication – total 4 bytes long. The 3 AND/OR bytes consist of:
 - a** Byte 0-3 (0 being least significant byte) on PLL dword value to change.
 - b** AND mask
 - c** OR mask.
- 3** Special command bit 7 ON Ored with Command number – total 1 byte. Currently 5 special commands are defined:
 - a** Delay 150 microseconds
 - b** Delay 15 millisecond
 - c** Wait for MC_BUSY = 0 in CLK_PWRMGT_CNTL register.
 - d** Wait for DLL_READY = 1 in CLK_PWRMGT_CNTL register.
 - e** Check & set bit 24 to 0 in CLK_PWRMGT_CNTL register.

4 End of table indicator in byte – total 1 byte. Byte value is 0.

More specifically the first (index) byte has the following flags:

Bit 5-0	- if Bit 8 = 0, indicates the PLL offset Otherwise indicates special command number
Bit 7: 1	- indicates following bytes are byte offset, AND mask and OR mask.
0	- indicates following dword to write to PLL offset.
Bit 8: 1	- indicates special command.

Example:

Db	1	; revision number
Table_start:		
Db	0dh	; PLL offset 0d (SCLK_CNTL)
Dd	1FFF0000	; programming value in dword
Db	4eh	; PLL offset (MPLL_CNTL) or'ed with 40h indicating AND/OR'ing
Db	0	; First byte
Db	fdh	; AND mask
Db	0	; OR mask
Db	82h	; Special command number 2
Db	00h	; end of table

The (offset-1) to the table contains the table revision number. The correct PLL offset is accessed by writing the offset to ioCLOCK_CNTL_INDEX (0x8) and then reading/writing ioCLOCK_CNTL_DATA (0xC).

D.1.4 Memory Configuration Block and Reset Sequence Table

Memory is first initialized in the first initialization block. The Memory Configuration Block, which is pointed by offset 48-49h in the BIOS header - label Memory_config_table for RADEON, is used to configure and auto detect the memory size and type.

Format:

Memory Configuration Block

The Memory Configuration Block is pointed to by offset 48-49h on the BIOS Header Information Tables. For RADEON, "Memory_config_table" is the label used for the start of this table. The following applies on a per channel basis.

Each entry contains 2 bytes:

-
- 1 The First byte is the size of memory in 2MByte units. So, for example 8 = 16Mbytes. A zero indicates the end of table. This value is used in programming register CONFIG_MEMSIZE.
 - 2 The Second byte is the memory address mapping value. This value is used in programming the MEM_CNTL register.

Auto memory detection works by scanning the table loading each table entry into MEM_CNTL and testing the memory. When the memory test passes, the correct table entry has been found. Fixed memory works by providing the “memory address mapping value” (second byte in table) into the FIXED_MEMORY constant and the “memory size” (first byte in table) into the FIXED_MEM_SIZE constant. These 2 values are used to program MEM_CNTL and CONFIG_MEMSIZE respectively.

The (offset-1) to the table contains the table revision number.

Example:

Db	0	; revision number
Table_start:		
Db	8	
Db	02eh	
Db	8	
Db	0adh	
Db	0	; end of the table

Memory reset is performed in function Init_vga after the initialization block and before memory sizing. Memory reset function is performed using memory register MEM_SDRAM_MODE_REG.

Each entry of the table is either one or three bytes long. There are two types of one byte entries:

- 1 A 0FH value is a delay telling the BIOS to wait until register MC_STATUS (159h) indicates MC_IDLE = idle.
- 2 A 0FFH value indicates the end of table.

There is one type of 3 byte entry:

- 1 The first byte indicates a reset/initialization bits ORed into register MEM_SDRAM_MODE_REG. These bits will tell the memory whether to: 1) Reset/normal or 2) initialization complete/not complete.
- 2 The next word also indicates what to write to register MEM_SDRAM_MODE_REG.

The least significant byte of the word on the table are written to bits 0-7 of MEM_SDRAM_MODE_REG followed by the most significant byte written to bits 24-31 of MEM_SDRAM_MODE_REG.

A typical sequence would be to do a Normal/not complete (0x0), then a Reset/not complete (0x80). The Normal/complete (0x10) is only performed at the very end.

Example:

memreset_table:

db	00	
dw	031h	; default value
db	80	
dw	031h	; default value
db	0Fh	; wait
db	10	
dw	031h	; default value
db	0FFh	; end of table

Appendix E

CCE Command Packets

E.1 Scope

This section provides a summary of the CCE command packets. In CCE mode, programming the RADEON does not require writing directly to the registers to draw 2D or 3D images. Instead, the data is prepared in the format of CCE *Command Packets* in system memory, and the hardware microengine does the work of drawing.

There are four types of CCE command packets:

- Type 0
- Type 1
- Type 2
- Type 3

A CCE command packet consists of:

- A *packet header*, identified by field HEADER. The packet header defines the operations to be carried out by the CCE microengine.
- An *information body*, identified by IT_BODY, that follows the header. The information body contains the data to be used by the engine in carrying out the operation.

E.1.1 Notation used this Section

- Brackets [] are used to denote a DWORD in a packet.
- Braces { } are used to denote a size-varying field that may consist of a number of DWORDs.
- If a DWORD is shared by more than one field, the fields are separated by '|'.
- The field that appears on the far left takes the most significant bits, and the field that appears on the far right takes the least significant bits.
- For example: DWORD [HI_WORD | LO_WORD] denotes that HI_WORD is defined on bits 31:16, and LO_WORD on bits 15:0.

- A C-style notation of referencing an element of a structure refers to a subfield of a main field.
- For example: MAIN_FIELD.SUBFIELD refers to the subfield SUBFIELD of MAIN_FIELD.

E.1.2 Type-0 CCE Packet

Purpose: For writing N DWORDs in the information body to the N consecutive registers (or to the register) that is pointed to by the BASE_INDEX field of the packet header. The use of this type of packet requires the complete understanding of the registers to be written.

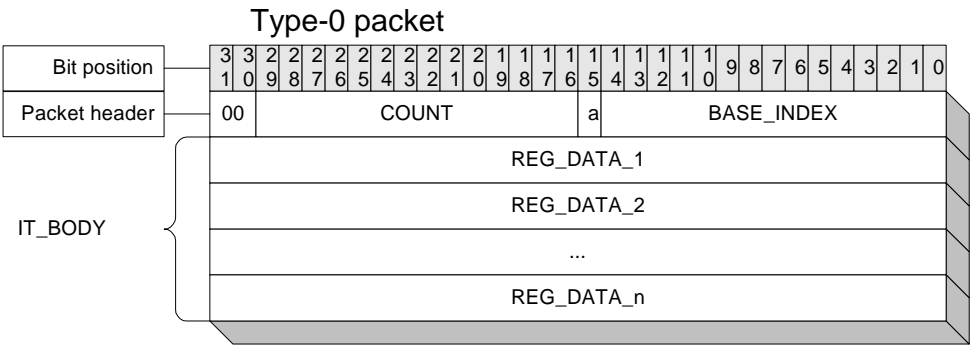


Figure E-15. Type 0 CCE Packet

Table E-1 Format for a Type-0 CCE Packet

Ordinal	Field Name
1	[HEADER]
2	[REG_DATA_1]
3	[REG_DATA_2]
	...
N+1	[REG_DATA_N]

Table E-2 Header Fields for a Type-0 CCE Packet

Bit(s)	Field Name	Description
10:0	BASE_INDEX	Memory-mapped address (in units of DWORD) of the first register to be written to.
14:11	Reserved	
15	ONE_REG_WR	0:- Write the data to N consecutive registers. 1:- Write all the data to the same register.
29:16	COUNT	Count of DWORDs in the information body. Its value should be N-1 if there are N DWORDs in the information body.
31:30	TYPE	Packet identifier. It should be zero.

Table E-3 Information Body for a Type-0 CCE Packet

Bit(s)	Field Name	Description
31:0	REG_DATA_x	The bits correspond to those defined for the relevant register. See Register Reference of RADEON. Note the suffix x of REG_DATA_x stands for an integer ranging from 1 to N.

Note: The use of this packet requires the complete understanding of the registers to be written.

E.1.3 Type 1 CCE Packet

Purpose: For writing REG_DATA_1 and REG_DATA_2 in the information body respectively to the registers pointed to by REG_INDEX1 and REG_INDEX2.

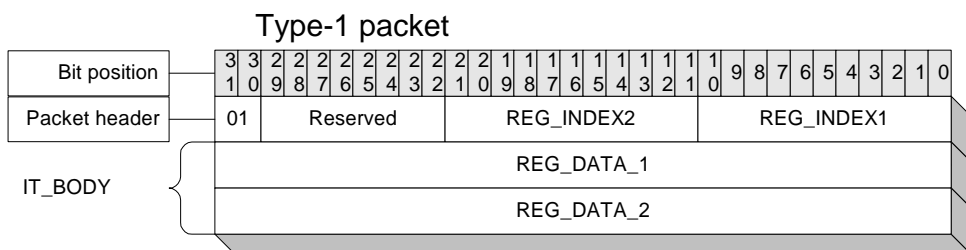
**Figure E-16. Type 1 CCE Packet**

Table E-4 Format for a Type 1 CCE Packet

Ordinal	Field Name
1	[HEADER]
2	[REG_DATA_1]
3	[REG_DATA_2]

Table E-5 Header Fields for a Type 1 CCE Packet

Bit(s)	Field Name	Description
10:0	REG_INDEX1	The field points to a memory-mapped register that REG_DATA_1 is written to.
21:11	REG_INDEX2	The field points to a memory-mapped register that REG_DATA_2 is written to.
29:22	Reserved	
31:30	TYPE	Packet identifier. It should be 1 (one).

Table E-6 Information Body for a Type 1 CCE Packet

Bit(s)	Field Name	Description
31:0	REG_DATA_x	The bits correspond to those defined for the relevant register. See Register Reference of RADEON.

E.1.4 Type 2 CCE Packet

Purpose: For filling up the trailing space left when the allocated buffer for a packet, or packets, is not fully filled.

This allows the microengine to skip the trailing space and to fetch the next packet. This is a filler packet. It has only the header. Its content is not important except for bits 30 and 31.

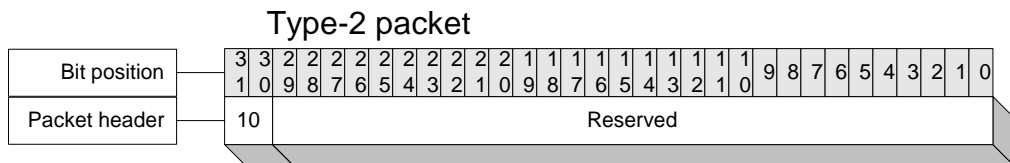


Figure E-17. Type 2 CCE Packet

Table E-7 Type 2 CCE Packet

Ordinal	Field Name
1	[HEADER]

Table E-8 Header Fields of a Type 2 CCE Packet

Bit(s)	Field Name	Description
29:0	reserved	
31:30	TYPE	Packet identifier. It should be 2.

E.1.5 Type 3 CCE Packet

Purpose: For carrying out the operation indicated by field IT_OPCODE.

Type-3 packets has a common format in their headers. However, the size of their information body may vary depending on the value of field IT_OPCODE. The size of the information body is indicated by field COUNT. If the size of the information is N DWORDs, the value of COUNT is N-1. In the following packet definitions, we will describe the field IT_BODY for each packet with respect to a given IT_OPCODE, and omit the header. The MSB of the IT_OPCODE identifies whether this packet requires the GUI_CONTROL field (described later). A 1 in the MSB of the IT_OPCODE indicates that GUI control is required. A 0 in the MSB of the IT_OPCODE indicates that the GUI_CONTROL should be omitted.

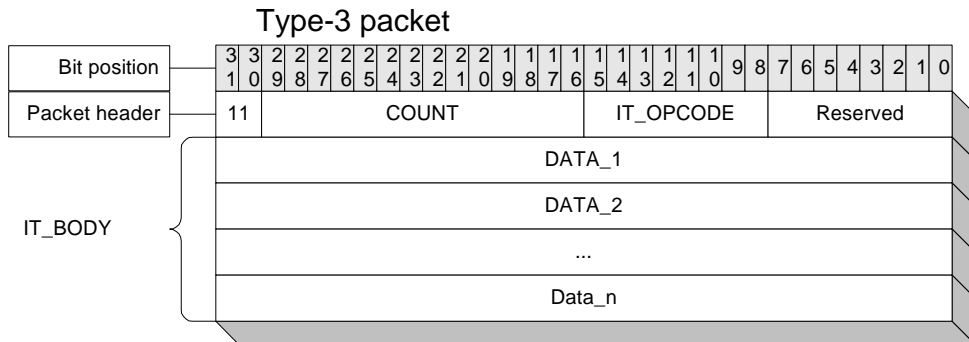


Figure E-18. Type 3 CCE Packet

Table E-9 Format for a Type 3 CCE Packet

Ordinal	Field Name
1	[HEADER]
2	{IT_BODY}

Table E-10 Header Fields for a Type 3 CCE Packet

Bit(s)	Field Name	Description
7:0	Reserved	This field is undefined, and is set to zero by default.
15:8	IT_OPCODE	Operation to be carried out. See section B.2 for details.
29:16	COUNT	Number of DWORDs -1 in the information body. It is N-1 if the information body contains N DWORDs.
31:30	TYPE	Packet identifier. It should be 3.

E.2 Summary of the CEE Packets

Table E-11 Summary of the CEE Packets

Packet Name	IT_OPCODE	Description	Status
NOP	0x10	Skip N DWORDs to get to the next packet.	Supported
PAINT	0x91	Paint a number of rectangles with a colour brush.	Supported
SMALL_TEXT	0x93	Draw a string of small characters on the screen.	Supported
BITBLT	0x92	Copy a source rectangle to a destination rectangle.	Supported
HOSTDATA_BLT	0x94	Draw a string of large characters on the screen, or copy a number of bitmaps to the video memory.	Supported
POLYLINE	0x95	Draw a polyline (lines connected with their ends).	Supported
SCALE	0x96	Scale the given rectangular screen area by a factor. This packet is used by both 2D and 3D operations.	Removed (comment 1)
TRANS_SCALE	0x97	A transparent scaling operation in which the information of the source rectangle mixes with the destination. This packet is actually used only by 3-D graphics.	Removed (comment 1)
POLYSCANLINES	0x98	Draw polyscanlines or scanlines.	Supported
NEXTCHAR	0x19	Print a character at a given screen location using the default foreground and background colours.	Supported
PLY_NEXTSCAN	0x1D	Draw polyscanlines using current settings.	Supported
SET_SCISSORS	0x1E	Set up scissors.	Supported
SET_MODE_24BPP	0x1F	Set the 24bpp mode flag.	Removed
PAINT_MULTI	0x9A	Paint a number of rectangles on the screen with one colour. The difference between this function and PAINT is the representation of parameters.	Supported
BITBLT_MULTI	0x9B	Copy a number of source rectangles to destination rectangles of the screen respectively.	Supported
TRANS_BITBLT	0x9C	2D transparent bitblt operation.	Supported
3D_SAVE_CONTEXT	0x20	Save the 3-D drawing context to RADEON's memory.	Removed

Table E-11 Summary of the CEE Packets (Continued)

Packet Name	IT_OPCODE	Description	Status
3D_PLAY_CONTEXT	0x21	Play back a previous saved 3-D context.	Removed
3D_RNDR_GEN_INDX_PRIM	0x23	Draw 3-D objects using the vertex walker.	Supported
LOAD_MICROCODE	0x24	Load the microcode into the microcode engine	Supported
3D_RNDR_GEN_PRIM	0x25	Draw 3-D points, lines, triangles, strips, fans using the ring buffer.	Supported
WAIT_FOR_IDLE	0x26	Wait for the IDLE state of the engine. This packet is for hardware debug only.	Supported
LOAD_PALETTE	0x2C	Load a palette onto RADEON for 2D scaling.	Supported
PURGE	0x2D	Purge the pixel cache.	Removed (replace with NOP)
NEXT_VERTEX_BUFFER	0x2E	Add more vertices to the end of a 3D_RNDR_GEN_INDX_PRIM packet.	Removed
3D_DRAW_VBUF	0x28	Draw primitives using vertex buffer	New
3D_DRAW_IMMD	0x29	Draw primitives using immediate vertices in this packet	New
3D_DRAW_INDX	0x2A	Draw primitives using vertex buffer and indices in this packet	New
3D_LOAD_VBPNT	0x2F	Load pointers to vertex buffers	New
3D_CLEAR_ZMASK	0x32	Clear portion of ZMASK memory	New

Comment 1: Scaling is implemented in the 3D pipe for the RADEON. The current scale packet is very dependent of the specifics of the combined 3D and 2D pipeline of the RADEON. With the separate pipelines of the RADEON we cannot support the 128 scale packet. Since we hope in a future version of the RADEON pipeline to implement 2D scaling in the 2D pipe, we chose not to define a new scale packet now, and instead expect the 2D driver to create a scale operation through type 0 state setting packets, and then the new 3D_DRAW_IMMD packet to initiate the scale. If a 2D ROP is needed it may be necessary to do the scale in two operations, a 3D scale followed by a 2D blit to the destination.

E.3 2D Packets

The information body IT_BODY of 2-D packets may have the following format:

Table E-12

Ordinal	Field Name
1	{SETTINGS}
2	{DATA_BLOCK}

E.3.1 SETTINGS

This field consists of 2 subfields, GUI_CONTROL and SETUP_BODY.

Table E-13

Ordinal	Field Name
1	[GUI_CONTROL]
2	{SETUP_BODY}

SETTINGS.GUI_CONTROL

This field will be used to setup the RADEON (register DP_GUI_MASTER_CNTL), and it also decides the content of SETTINGS.SETUP_BODY.

Bit(s)	Field Name	Description	RADEON
0	SRC_PITCH_OFF	The bit controls the pitch and offset of the blitting source. 0:- Use the default pitch and offset, and no datum [SRC_PITCH_OFFSET] is supplied in SETUP_BODY. 1:- Use the datum [SRC_PITCH_OFFSET] supplied in SETUP_BODY to set up a new pitch offset.	

Bit(s)	Field Name	Description	RADEON
1	DST_PITCH_OFF	The bit controls the pitch and offset of the blitting destination. 0:- Use the default pitch and offset, and no datum [DST_PITCH_OFFSET] is supplied in SETUP_BODY. 1:- Use the datum [DST_PITCH_OFFSET] supplied in SETUP_BODY. The pitch may mean the bitmap pitch and the offset may points the off-screen area of the video memory.	
2	SRC_CLIPPING	This bit controls the clipping parameters of the blitting source. 0:- Use the default clipping parameters, and no relevant clipping data supplied in SETUP_BODY. 1:- Use datum [SRC_SC_BOT_RITE] supplied in SETUP_BODY to set up the bottom and right edges of the clipping rectangle.	
3	DST_CLIPPING	This bit controls the clipping parameters of the blitting destination. 0:- Use the default clipping parameters, and no relevant clipping data supplied in SETUP_BODY. 1:- Use data [SC_TOP_LEFT] and [SC_BOTTOM_RIGHT] supplied in SETUP_BODY to set up a new clipping rectangle.	
7:4	BRUSH_TYPE	Types of brush used in drawing. The type code determines how to supply data to the subfield BRUSH_PACKET in SETUP_BODY. See detailed definition of BRUSH_TYPE in the following.	

Bit(s)	Field Name	Description	RADEON
11:8	DST_TYPE	<p>The pixel type of the destination.</p> <p>0-1 :- (reserved)</p> <p>2 :- 8 bpp pseudocolor</p> <p>3 :- 16 bpp aRGB 1555</p> <p>4 :- 16 bpp RGB 565</p> <p>5 :- reserved</p> <p>6 :- 32 bpp aRGB 8888</p> <p>7 :- 8 bpp RGB 332</p> <p>8 :- Y8 greyscale</p> <p>9 :- RGB8 greyscale (8 bit intensity, duplicated for all 3 channels. Green channel is used on writes)</p> <p>10 :- (reserved)</p> <p>11 :- YUV 422 packed (VYUY)</p> <p>12 :- YUV 422 packed (YVYU)</p> <p>13 :- (reserved)</p> <p>14 :- aYUV 444 (8:8:8:8)</p> <p>15 :- aRGB4444 (intermediate format only. Not understood by the Display Controller)</p> <p>Note: choices 7-15 only valid in 3D mode.</p>	7 through 15 not supported in 3D pipe
13:12	SRC_TYPE	<p>The field indicates the pixel type of blitting source.</p> <p>0:- The source data type is mono opaque, and the fore- and back-ground colours need to be redefined.</p> <p>1:- The source data type is mono transparent, and only the foreground colour needs to be redefined.</p> <p>2:- Reserved.</p> <p>3:- The source pixel type is the same as that given in field DST_TYPE.</p> <p>If bit 27 (SRC_TYPE) is one then the following new (RADEON) sources are available:</p> <p>4:- 4bpp source clut translation (May not be supported, value reserved)</p> <p>5:- 8bpp source clut translation</p> <p>6:- 32 bpp source clut translation (gamma correction)</p> <p>7:- 64 bpp Obuffer blit</p>	
14	PIX_ORDER	<p>The bit decides the order of bits (or pixels) in DWORD to be consumed. Only applicable to the monochrome mode.</p> <p>0 :- Bits to be consumed from the Most Significant Bit (MSB) to the Least Significant Bit (LSB).</p> <p>1 :- Bits to be consumed from LSB to MSB.</p>	
15	COLOR_CONVT	Reserved	Not supported in 2D pipe

Bit(s)	Field Name	Description	RADEON
23:16	WIN31_ROP	This field tells the GUI engine how the raster operation to be carried out. The code of this field follows the ROP3 code defined by Microsoft. See WIN31 DDK for reference.	
26:24	SRC_LOAD	The field indicates where the source data come from. 0,1 :- Reserved 2 :- loaded from the video memory (rectangular trajectory) 3 :- loaded through the HOSTDATA registers (linear trajectory) 4 :- loaded through the HOSTDATA registers (linear trajectory & byte-aligned) Note that during 3D/Scale Operations (whenever SCALE_3D_FCN@MISC_3D_STATE_REG is non-zero), this field is ignored and data is always loaded from the 3D/Scaler pipeline.	
27	SRC_TYPE	Third bit of SRC_TYPE	New for RADEON. Compatible 128 code must write zero to this register.
28	GMC_CLR_CMP_FCN_DIS	0 :- No change to CLR_CMP_FCN_SRC and CLR_CMP_FCN_DST 1 :- clear CLR_CMP_FCN_DST and CLR_CMP_FCN_SRC to 0	TBD
29	Reserved	Reserved	Reserved
30	GMC_WR_MSK_DIS	0 :- No Change to DP_WR_MSK/CLR_CMP_MSK 1 :- Set DP_WR_MSK/CLR_CMP_MSK to 0xffffffff	
31	BRUSH_FLAG	This field indicates whether there is a field BRUSH_Y_X field in the SETTINGS.SETUP_BODY. 0:- No such a field in SETTINGS.SETUP_BODY. 1:- There is a field in SETTINGS.SETUP_BODY.	

SETTINGS.SETUP_BODY

This field may contain the following subfields. Their presence depends on the bits 0-7 of **SETTINGS.GUI_CONTROL**.

Ordinal	Field Name	Description
1	[SRC_PITCH_OFFSET]	Bit 30: Select between untiled(0) and tiled (1) Bit 31: select between no microtiling(0) and microtiling(1) Bits 29:22 Pitch in units of 64 bytes, 64 to 16384 bytes across bits 21:0 Offset in units of 1KB, 0 to 4GB-1K
2	[DST_PITCH_OFFSET]	Bit 30: Select between untiled(0) and tiled (1) Bit 31: select between no microtiling(0) and microtiling(1) Bits 29:22 Pitch in units of 64 bytes, 64 to 16384 bytes across bits 21:0 Offset in units of 1KB, 0 to 4GB-1K
3	[SRC_SC_BOT_RITE]	The parameters are used to setup the clipping area of the source. The implied coordinates of the top-left corner of the clipping rectangle is the same as the source. [13:0] :- x-coordinate of the right edge of the clipping rectangle (in number of pixels). [29:16] :- y-coordinate of the bottom edge of the clipping rectangle (in number of scanlines).
4	[SC_TOP_LEFT] [SC_BOT_RITE]	The parameters are used to setup the clipping area of destination. SC_TOP_LEFT: [13:0] :- x-coordinate of the left edge of the clipping rectangle (in number of pixels). [29:16] :- y-coordinate of the top edge of the clipping rectangle (in number of scanlines). SC_BOT_RITE: [13:0] :- x-coordinate of the right edge of the clipping rectangle (in number of pixels). [29:16] :- y-coordinate of the bottom edge of the clipping rectangle (in number of scanlines).
5	{ BRUSH_PACKET }	The content of this field is determined by field SETTINGS.GUI_CONTROL.BRUSH_TYPE . See the following table for the possible content.
6	[BRUSH_Y_X]	[4:0] :- x-coordinate for brush alignment. [12:8] :- y-coordinate for brush alignment. [20:16] :- Initial value used for BRUSH_X pointer in drawing Lines. When POLY_LINE is off , it is reloaded from BRUSH_X at the end of the line. When POLY_LINE is on , it is reloaded from the current Brush pointer at the end of the line. Whenever BRUSH_X is updated, the field should be written with the same value.

SETTINGS.SETUP_BODY.BRUSH_PACKET

RADEON: note that all but 6 and 7 are not available for lines, and 6 and 7 are only usable for lines.

BRUSH_TYPE	Description of the brush	Packet size	Packet content
0	A 8 x 8 mono pattern with the foreground and background colours specified in the packet. Here the matrix is represented in the format <i>column-by-row</i> .	4 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
1	A 8 x 8 mono pattern with the foreground colour specified in the packet and the background colour the same as that of the area to be painted.	3 DWORDs	[FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
2	Reserved	not applicable	
3	Reserved	not applicable	
4	Reserved	not applicable	
5	Reserved	not applicable	
6	A 32 x 1 mono pattern with the foreground and background colours specified in the packet. This pattern corresponds to the PEN of Win95 DDK. And is only usable for lines.	3 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1]
7	A 32x1 mono pattern with the foreground colour specified in the packet and the background colour the same as that of the area to be painted. This is PEN as well. And is only usable for lines.	2 DWORDs	[FRGRD_COLOR] [MONO_BMP_1]
8	Removed, see 32x32 in 3D pipe	not applicable	
9	Removed, see 32x32 in 3D pipe	not applicable	
10	A 8x8 colour pattern. The pixel type is given by field SETTINGS.GUI_CONTROL.DST_TYPE.	16* N DWORDs, where N stands for the number of bytes per pixel with exception that a 24-BPP pixel is still represented by 4 bytes.	[COLOR_BMP_1] [COLOR_BMP_2] ... [COLOR_BMP_16*N]
11	Reserved	not applicable	
12	Reserved	not applicable	

BRUSH_TYPE	Description of the brush	Packet size	Packet content
13	Use the colour specified in the packet as the solid (plain) colour for the brush, i.e. a colour brush without pattern.	1 DWORD	[FRGRD_COLOR]
14	Use the colour specified in the packet as the solid (plain) colour for the brush, i.e. a colour brush without pattern.	1 DWORD	[FRGRD_COLOR]
15	No brush used.	0	

Pixel size in bytes

SETTINGS.GUI_CONTROL.DST_TYPE	N
0-1	not applicable
2	1
3	2
4	2
5	
6	4
7	1
8-15	not applicable

Brush packet content

Field Name	Description
[FRGRD_COLOR]	The foreground colour of the text in the RGBQUAD format. bits [7:0] :- intensity of Blue; bits [15:8] :- intensity of Green; and bits [23:16] :- intensity of Red. bits [31:25] :- reserved.
[BKGRD_COLOR]	The background colour of the text in the RGBQUAD format. bits [7:0] :- intensity of Blue; bits [15:8] :- intensity of Green; and bits [23:16] :- intensity of Red. bits [31:25] :- reserved.
[MONO_BMP_x]	Raster data of monochrome pixels. One bit represents one pixel. If the number of pixels for the field is less than 32, the pixels take the lower bits. The remaining bits should be filled with 0's.
[COLOR_BMP_x]	Raster data of colour pixels. The representation depends on the pixel type.

DATA_BLOCK

The composition of this field depends on the operation code `IT_OPCODE` given in the header. Section B.2 gives details of `DATA_BLOCK` with respect to `IT_OPCODE`. In the following, the field `SETTINGS` may appear in the definition of a packet, but will not be described further.

E.3.2 NOP

Functionality

Skip a number of DWORDs to get to the next packet.

Format

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

DATA_BLOCK

This field may consists of a number of DWORDs, and the content may be anything.

E.3.3 PAINT

Functionality

Paint a number of rectangles with a colour brush.

Format:

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[TOP_1 LEFT_1]	The coordinates of the top-left corner of the 1st rectangle to be painted. LEFT_1: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. TOP_1: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[BOTM_1 RITE_1]	The coordinates of the bottom-right corner of the 1st rectangle to be painted. RITE_1: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. BOTM_1: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
...		
2n-1	[TOP_n LEFT_n]	The coordinates of the top-left corner of the n-th rectangle to be painted.
2n	[BOTM_n RITE_n]	The coordinates of the bottom-right corner of the n-th rectangle to be painted.

E.3.4 SMALL_TEXT

Functionality

Print a string of characters on the screen in the format of the bit-packed Small Glyph.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOUR]	The foreground colour of the text in the RGBQUAD format. BLUE: [7:0] :- intensity of the blue component. GREEN: [15:8] :- intensity of the green component. RED: [23:16] :- intensity of the red component. bits [31:25] :- reserved.
2	[BAS_Y BAS_X]	The base coordinates of the text rectangle in the screen coordinate system. See the following illustration for details. BAS_X: [15:0] :- x-coordinate. BAS_Y: [31:16] :- y-coordinate.
3	{SMALLCHAR_1}	The 1st character of the text.
...		
n+2	{ SMALLCHAR _n}	The n-th character of the text, i.e. the last character.

DATA_BLOCK.SMALLCHAR_x

Ordinal	Field Name	Description
1	[H W ΔY ΔX]	The geometry of the bitmap and the deviation of its top-left corner from the base coordinates. ΔX: [7:0] :- deviation from the base x-coordinate of the preceding glyph ΔY: [15:8] :- deviation from the base y-coordinate. W: [23:16] :- width of the character bitmap H: [31:24] :- height of the character bitmap.
2	[RASTER_1]	The 1st DWORD of the mono bitmap data.
...		
m+1	[RASTER_m]	The m-th DWORD of the mono bitmap data.

Parameters H, W, ΔY and ΔX

The relationship between the parameters and the reference coordinates bas_x and bas_y is shown in the following figure. In the figure below, the starting position of text is at (bas_x, bas_y) . The actual sizes of characters 'b', 'o' and 'y' respectively are 4x8, 4x5, and 6x9. Therefore, the related parameters are:

$$H_1=8, W_1=4, \Delta x_1=0, \text{ and } \Delta y_1=8$$

$$H_2=5, W_2=4, \Delta x_2=6, \text{ and } \Delta y_2=5$$

$$H_3=9, W_3=6, \Delta x_3=5, \text{ and } \Delta y_3=5$$

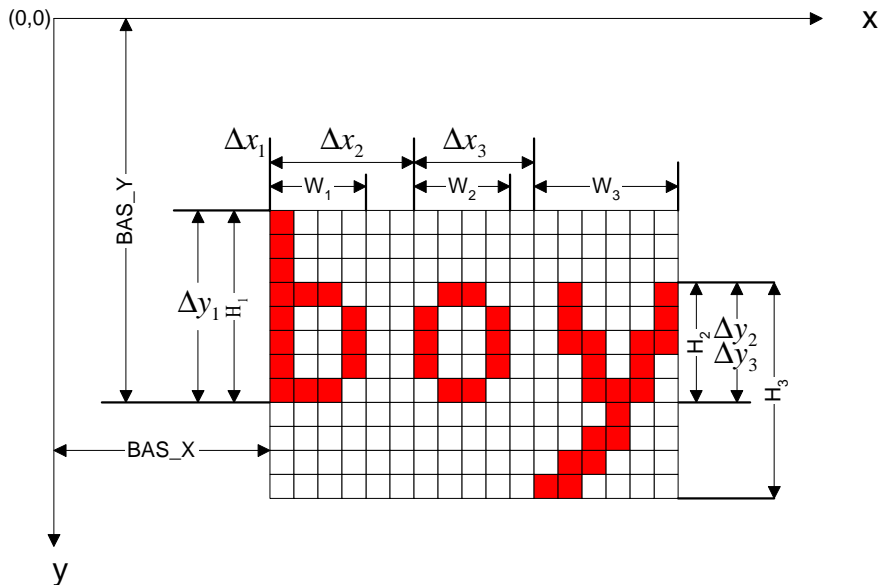


Figure E-19. H, W, ΔY and ΔX

RASTER_x

Raster_x represents the data block of a mono bitmap. The bitmap represents the raster image of a character. This data block corresponds to the bitmap data following structure SMALLBITGLYPH in Windows95' DDK.

E.3.5 HOSTDATA_BLT

Functionality

Copy a number of bit-packed bitmaps to the video memory. It can be used to print a string of large characters on the screen. In other words, the function supports the LARGEBITGLYPH structure of Windows95 DDK.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOUR]	Foreground colour in the RGBQUAD format. For mono-to colour expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
2	[BKGD_COLOUR]	Background colour in the RGBQUAD format. For mono-to colour expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
3	{BIGCHAR_1}	Data block of the 1st character.
...		
m+2	{BIGCHAR _m}	Data block of the m-th character.

DATA_BLOCK.BIGCHAR_x

Ordinal	Field Name	Description
1	[BaseY BaseX]	The coordinate of the top-left corner of the character's bitmap. BaseX: [15:0] :- x-coordinate. BaseY: [31:16] :- y-coordinate.
2	[HEIGHT WIDTH]	The geometry of the bitmap. WIDTH: [15:0] :- width of the bitmap. HEIGHT: [31:16] :- height of the bitmap.

Ordinal	Field Name	Description
3	[NUMBER]	The number of DWORDs in the bitmap. It should be m in this case.
4	[RASTER_1]	The 1st DWORD of the mono bitmap data.
...		
$m+3$	[RASTER_m]	The m -th DWORD of the mono bitmap data.

E.3.6 POLYLINE

Functionality

Draw a polyline specified by a set of coordinates $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, where coordinate (x_0, y_0) is the beginning of the polyline, and coordinate (x_n, y_n) is the end.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[Y0 X0]	The starting coordinate of the polyline. X0: [15:0] :- x-component of the coordinate. Y0: [31:16] :- y-component.
2	[Y1 X1]	The 2nd coordinate of the polyline. Definition of bits is the same as above.
...		
$n+1$	[Yn Xn]	The ending coordinate of the polyline. Definition of bits is the same as above.

E.3.7 POLYSCANLINES

Functionality

Draw a number of scanlines and polyscanlines. The number can be one. The difference between a scanline and a polyscanline is that a scanline has only one starting x-coordinate and one ending x-coordinate while a polyscanline has a number of starting-ending x-coordinate pairs .

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[SCAN_COUNT]	The number of scan subpackets identified by SCAN_x, where x denotes the ordinal number of a SCAN subpacket.
2	{ SCAN_1 }	The 1st scanline/polyscanline.
...		
n+1	{ SCAN_n }	The n-th scanline/polyscanline.

DATA_BLOCK.SCAN_x

Ordinal	Field Name	Description
1	[NUM_LINE]	The number of line segments in a polyscanline.
2	[HEIGHT TOP]	TOP: [15:0] :- y-coordinate of the polyscanline. HEIGHT: [31:16] :- The thickness of the line measured in pixels.
3	[END_1 START_1]	START_1: [15:0] :- the starting x-coordinate of the 1st line segment. END_1: [31:16]:- the ending x-coordinate of the 1st line segment.
...		

Ordinal	Field Name	Description
n+2	[END_n START_n]	START_n: [15:0] :- the starting x-coordinate of the n-th line segment. END_n: [31:16]:- the ending x-coordinate of the n-th line segment.

E.3.8 NEXTCHAR

Functionality

Print a character at a given screen location using the default foreground and background colours.

Format

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[DST_Y DST_X]	The coordinates of the top-left corner of the destination bitmap. DST_X: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_Y: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_H DST_W]	The width and height of the destination bitmap, expressed in unsigned integers. DST_W: [15:0]:- width. DST_H [31:16]:- height.
3	[BITMAP_DATA_1]	The 1st DWORD of the bitmap data.
...		
n+2	[BITMAP_DATA_n]	The n-th DWORD of the bitmap data.

E.3.9 PAINT_MULTI

Functionality

Paint a number of rectangles on the screen with one colour. The colour used is specified in field SETTINGS while the location and geometry of the rectangles are specified in field DATA_BLOCK.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[DST_X1 DST_Y1]	The coordinates of the top-left corner of the 1st rectangle. DST_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_W1 DST_H1]	The width and height of the 1st rectangle, expressed in unsigned integers. DST_H1: [15:0]:- height. DST_W1: [31:16]:- width.
...		
2n-1	[DST_Xn DST_Yn]	The coordinates of the top-left corner of the n-th rectangle. DST_Yn: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_Xn: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2n	[DST_Wn DST_Hn]	The width and height of the n-th rectangle, expressed in unsigned integers. DST_Hn: [15:0]:- height. DST_Wn: [31:16]:- width.

E.3.10 BITBLT

Functionality

Copy a source rectangle to a destination rectangle of the screen. It is assumed that the geometry of the destination is identical to its source.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[SRC_X1 SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_X1 DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
3	[SRC_W1 SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- height. SRC_W1: [29:16]:- width.

E.3.11 BITBLT_MULTI

Functionality

Copy a number of source rectangles to destination rectangles of the screen respectively. It is assumed that the geometry of the destination is identical to its source.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[SRC_X1 SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_X1 DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
3	[SRC_W1 SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- height. SRC_W1: [29:16]:- width.
...		
3n-1	[SRC_Xn SRC_Yn]	The coordinates of the top-left corner of the n-th source bitmap. SRC_Yn: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_Xn: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
3n-2	[DST_Xn DST_Yn]	The coordinates of the top-left corner of the n-th destination. The definition of bits is the same as SRC_Xn and SRC_Yn.
3n	[SRC_Wn SRC_Hn]	The width and height of the n-th source bitmap, expressed in unsigned integers. SRC_Hn: [13:0]:- height. SRC_Wn: [29:16]:- width.

E.3.12 TRANS_BITBLT

Functionality

Copy pixels from the source rectangle to the destination with transparency.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[CLR_CMP_CNTL]	This field decides how the transparent blitting is done. See following for details.
2	[SRC_REF_CLR]	Source reference colour in the RGBQUAD format. This is the colour to be stripped off from the source.
3	[DST_REF_CLR]	Destination reference colour in the RGBQUAD format. This is the colour to be preserved at the destination.
4	[SRC_X1 SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
5	[DST_X1 DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
6	[SRC_W1 SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- height. SRC_W1: [29:16]:- width.

DATA_BLOCK.CLR_CMP_CNTL

This field controls how the source pixels are written to the destination, depending on the source and destination reference colours and comparison settings. The source pixels may be filtered against the source reference colour, and the destination pixels with a specific colour may be preserved according to field CLR_CMP_DST.

Bit(s)	Bit-Field Name	Description
2:0	CLR_CMP_SRC	<p>Strip off the source reference colour from the source pixels.</p> <p>0 :- Do not strip off source pixels. All source pixels are written to the destination.</p> <p>1 :- Block the blitting source. No source pixel is written to the destination.</p> <p>2, 3 :- reserved.</p> <p>4 :- The source pixels whose colour is NOT equal to the reference colour are written to the destination.</p> <p>5 :- The source pixels whose colour is equal to the reference colour are written to the destination.</p> <p>6 :- Reserved.</p> <p>7 :- The source pixels whose colour is equal to the reference colour will be XORed with the foreground colour of a mono bitmap, and then written to the destination. That is, destPixel = srcPixel XOR foregrndColor if srcPixel is equal to the foreground colour of a mono bitmap, specifically text. This is referred to as flipping sometimes.</p>
7:3	Reserved	
10:8	CLR_CMP_DST	<p>Preserve pixels at the destination.</p> <p>0 :- Do not preserve the destination pixels. All pixels from the source are written to the destination.</p> <p>1 :- Preserve all the destination pixels. No source pixel is written to the destination.</p> <p>2, 3 :- Reserved.</p> <p>4 :- The destination pixels whose colour is equal to the reference colour are preserved. No source pixel is written on top of the pixels.</p> <p>5 :- The destination pixels whose colour is NOT equal to the reference colour are preserved.</p> <p>6, 7 :- Reserved.</p>
23:11	Reserved	
25:24	CMP_ENABLE	<p>The bits controls what type of operation to be carried out.</p> <p>0 :- Enable function CLR_CMP_DST.</p> <p>1 :- Enable function CLR_CMP_SRC</p> <p>2 :- Enable both CLR_CMP_SRC and CLR_CMP_DST. The final decision is based on the agreement between decisions made separately.</p> <p>3 :- Reserved.</p>
31:26	Reserved	

E.3.13 LOAD_MICROCODE

Functionality

Load the microcode into the PM4 microcode engine. This packet allows the user to replace the running microcode with a new one while the microcode engine is running. However, caution must be taken on the starting address and size of the microcode being loaded, as it may crash the engine if handled improperly.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[START_ADDR]	Start address of the microcode in the memory of microcode engine
3	[MICROCODE_1_H]	The high-order DWORD of the 1st instruction of the microcode.
4	[MICROCODE_1_L]	The low-order DWORD of the 1st instruction of the microcode.
...		
2n+1	[MICROCODE_n_H]	The high-order DWORD of the n-th instruction of the microcode.
2n+2	[MICROCODE_n_L]	The low-order DWORD of the n-th instruction of the microcode.

E.3.14 PLY_NEXTSCAN

Functionality

Draw a number of scanlines or polyscanlines using the current settings.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[HEIGHT TOP]	TOP: [15:0] :- y-coordinate of the scanline/polyscanline. HEIGHT: [31:16] :- The thickness of the line measured in pixels.
3	[END_1 START_1]	START_1: [15:0] :- the starting x-coordinate of the 1st dash. END_1: [31:16]:- the ending x-coordinate of the 1st dash.
...		
n+2	[END_n START_n]	START_n: [15:0] :- the starting x-coordinate of the 1st dash. END_n: [31:16]:- the ending x-coordinate of the 1st dash.

E.3.15LOAD_PALETTE

Functionality

Set up the 3D engine scaler and load a palette onto RADEON for a consequent 2D scaling operation.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[SCALE_DATATYPE]	1:- The palette has 16 entries (4 bpp palette). 2:- The palette has 256 entries (8 bpp palette).
3	[COLOUR_1]	The 1 st entry of the palette. Data is in destination format (i.e. ARGB8888, RGB565, RGB555,...)
4	[COLOUR_2]	The 2 nd entry of the palette. Bits are defined as above.
...		
n+2	[COLOUR_n]	The n-th entry of the palette. n = 16 (4bpp) or 256 (8bpp)

E.3.16SET_SCISSORS

Functionality

Set the scissors to the given parameters.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[TOP_LEFT]	[13:0] :- x-coordinate of the left edge of the clipping rectangle (in number of pixels). [29:16] :- y-coordinate of the top edge of the clipping rectangle (in number of scanlines).
3	[BOTTOM_RIGHT]	[13:0] :- x-coordinate of the right edge of the clipping rectangle (in number of pixels). [29:16] :- y-coordinate of the bottom edge of the clipping rectangle (in number of scanlines).

E.4 3D Packets

There are two sets of 3D packets. The first set (3D_RNDR_GEN_PRIM, 3D_RNDR_GEN_INDX_PRIM) is carried over from the R128 and is unchanged. There is a new set of packets (3D_DRAW, 3D_DRAW_IMMD, 3D_DRAW_INDX) which should be used for all new code, and provides access to the third texture, and the more flexible vertex buffers.

E.4.1 3D_DRAW_VBUF

Functionality

Draws a set of primitives using a vertex buffer(s) pointed to by state data.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[SE_VTX_FMT]	Vertex format (see SE_VTX_FMT register in register spec)
3	[SE_VF_CNTL]	Primitive type and other control (See SE_VF_CNTL register in register spec) 31:16 number of indices

E.4.2 3D_DRAW_IMMD

Functionality

Draws a set of primitives using vertices stored in packet.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[SE_VTX_FMT]	Vertex format (see SE_VTX_FMT register in register spec)
3	[SE_VF_CNTL]	Primitive type and other control (See SE_VF_CNTL register in register spec) 31:16 number of vertices
4 to end	Vertex data	Up to 64K – 12 bytes of vertex data

E.4.3 3D_DRAW_INDEX

Functionality

Draws a set of primitives using a vertex buffer(s) pointed to by state data, index from indices in packet.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[SE_VTX_FMT]	Vertex format (see SE_VTX_FMT register in register spec)
3	[SE_VF_CNTL]	Primitive type and other control (See SE_VF_CNTL register in register spec) 31:16 number of vertices
4 to end	[indx 2 indx 1]	Up to 32K – 6 indices to vertex data pointed to by state registers.

E.4.4 3D_LOAD_VBPNT

Functionality

Load the vertex arrays pointers. The size of the packet is variable and given in the (either NUM_ARRAYS or the packet size, whichever is easier). Valid lengths are: 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, and 20.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	VTX_NUM_ARRAYS	Number of arrays
3	VTX_AOS_ATTR01	Control for the first two arrays
4	VTX_AOS_ADDR0	Pointer to first array
5	VTX_AOS_ADDR1	Pointer to second array
6	VTX_AOS_ATTR23	And so on....
7	VTX_AOS_ADDR2	
8	VTX_AOS_ADDR3	
9	VTX_AOS_ATTR45	
10	VTX_AOS_ADDR4	
11	VTX_AOS_ADDR5	

Ordinal	Field Name	Description
12	VTX_AOS_ATTR67	
13	VTX_AOS_ADDR6	
14	VTX_AOS_ADDR7	
15	VTX_AOS_ATTR89	
16	VTX_AOS_ADDR8	
17	VTX_AOS_ADDR9	
18	VTX_AOS_ATTR101 1	
19	VTX_AOS_ADDR10	
20	VTX_AOS_ADDR11	

E.4.5 3D_CLEAR_ZMASK

Functionality

Clear ZMASK ram

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	START	Start
3	Count	Count (must be a multiple of 4)
4	Clear Value	The Value to write into the Zmask

Clears ZMASK ram.

Writes start address to ZMASK_WRINDX. Writes “count” copies of Clear Value to RB3D_ZMASK_DWORD. The count must be a multiple of 4.

E.4.6 3D_RNDR_GEN_PRIM

Functionality

Render 3-D primitives points, lines, triangles and rectangles through the ring buffer.

The general form of 3D_RNDR_GEN_PRIM packets is as follows. It consists of a header field HEADER, a flag field SE_VTX_FMT that indicates how the vertex data blocks should be interpreted, a control field SE_VF_CNTL that defines the type of primitive being drawn and the drawing method to be used, and a number of vertex data blocks that specify the coordinates and geometry of the primitive. As the vertex data blocks are arranged contiguously in memory, they may be referred to as *vertex array* or *vertex list*. The size of a vertex block may vary depending on the flag field SE_VTX_FMT. Therefore, such as vertex may be referred to as *flexible vertex*. However, for a specific packet, all the vertex blocks are of the same size. So, the vertices of the packet constitute a vertex array.

Format

Ordinal	Field Name
1	[HEADER]
2	[SE_VTX_FMT]
3	[SE_VF_CNTL]
4	{FTLVERTEX_1}
...	
n+3	{FTLVERTEX_n}

SE_VTX_FMT

This field is composed of a number of flags or subfields. Each flag determines the presence of a corresponding data field in the data block FTLVERTEX_x. If a flag is ON or one, the corresponding data field is present in FTLVERTEX_x. Otherwise, the data field is not.

SE_VTX_FMT Setup Engine Vertex Format		
Field Name	Bit(s)	Description
VTX_W0_PRESENT	0	Primary Vertex W value is present (1 float)
VTX_FPCOLOR_PRESENT	1	Floating Point Diffuse Color is Present (3 floats)

SE_VTX_FMT		
Setup Engine Vertex Format		
Field Name	Bit(s)	Description
VTX_FPALPHA_PRESENT	2	Floating Point Alpha is Present (1 float)
VTX_PKCOLOR_PRESENT	3	Packed (8,8,8,8) ARGB Diffuse is Present. Table for all combinations of COLOR/ALPHA PRESENT bits: PKCOLOR FPALPHA FPCOLOR Result 0 0 0 No color or alpha present 0 0 1 3 Floating Pt colors (RGB) present 0 1 0 Floating Pt alpha present 0 1 1 3 Floating Pt colors and alpha present 1 X X Packed color and alpha present
VTX_FPSPEC_PRESENT	4	Floating Point Specular Color is Present (3 floats)
VTX_FPFOG_PRESENT	5	Floating Point Fog Color is Present (1 float)
VTX_PKSPEC_PRESENT	6	Packed (8,8,8,8) FRGB Diffuse is Present. Table for all combinations of SPECULAR/FOG PRESENT bits: PKSPEC FPFOG FPSPEC Result 0 0 0 No color or fog present 0 0 1 3 Floating Pt colors (RGB) present 0 1 0 Floating Pt fog present 0 1 1 3 Floating Pt colors and fog present 1 X X Packed color and fog present
VTX_ST0_PRESENT	7	Texture coordinate set 0 S,T values are present (2 floats)
VTX_ST1_PRESENT	8	Texture coordinate set 1 S,T values are present (2 floats)
VTX_Q1_PRESENT	9	non- RADEON mode: Texture coordinate set 1 Q value is present (1 float) RADEON mode: Texture coordinate set 1 ooW value is present (1 float)
VTX_ST2_PRESENT	10	Texture coordinate set 2 S,T values are present (2 floats)
VTX_Q2_PRESENT	11	Texture coordinate set 2 Q value is present (1 float)
VTX_ST3_PRESENT	12	Texture coordinate set 3 S,T values are present (2 floats)
VTX_Q3_PRESENT	13	Texture coordinate set 3 Q value is present (1 float)
VTX_Q0_PRESENT	14	Texture coordinate set 0 Q value is present (1 float)
VTX_BLND_WEIGHT_CNT	17:15	Number of vertex blend weights present (0 to 4 floats)
VTX_N0_PRESENT	18	Primary Vertex Normal is present (3 floats: XYZ)
VTX_XY1_PRESENT	27	Second Vertex X,Y is present (2 floats) for vertex blending
VTX_Z1_PRESENT	28	Second Vertex Z is present (1 float) for vertex blending
VTX_W1_PRESENT	29	Second Vertex W is present (1 float) for vertex blending

SE_VTX_FMT Setup Engine Vertex Format		
Field Name	Bit(s)	Description
VTX_N1_PRESENT	30	Second Vertex Normal is present (3 floats: XYZ) for vertex blending
VTX_Z_PRESENT	31	Primary vertex Z value is present (1 float)

SE_VF_CNTL

This field corresponds to RADEON register SE_VF_CNTL. It selects the type of rendering primitive and the method of using the hardware.

SE_VF_CNTL Setup Engine Vertex Fetcher Control		
Field Name	Bit(s)	Description
NUM_VERTICES	31:16	Number of vertices in the command packet.
VTX_FMT_MODE	8	<p>Vertex Format Mode. This bit enables the RADEON extended features for processing vertices. 1 = Enable all RADEON features; no fields are overridden. 0 = Only enable the RADEON subset of features; for compatibility. The following fields get overridden: VTX_Z_PRESENT <- 1 VTX_ST2_PRESENT <- 0 VTX_W2_PRESENT <- 0 VTX_XY_FMT <- 1 VTX_Z_FMT <- 1 VTX_ST0_NONPARAMETRIC <- 0 VTX_ST1_NONPARAMETRIC <- 0 VTX_ST2_NONPARAMETRIC <- 0 VTX_W0_FMT <- 0 VTX_W1_FMT <- 0 VTX_W2_FMT <- 0 VPORT_XY_XFEN <- 0 VPORT_Z_XFEN <- 0</p>

SE_VF_CNTL Setup Engine Vertex Fetcher Control		
Field Name	Bit(s)	Description
EN_MAOS	7	<p>Enable Multiple Arrays of Structures. 1 = Enable all RADEON features; no fields are overridden. 0 = Only enable the RADEON subset of features; for compatibility.</p> <p>The following fields get overridden: VTX_NUM_ARRAYS <- 1 VTX_AOS0_COUNT <- Number of dwords per-vertex, as indicated by the Flexible Vertex Format VTX_AOS0_STRIDE <- Number of dwords per-vertex, as indicated by the Flexible Vertex Format</p>
COLOR_ORDER	6	<p>If diffuse/specular colors are in floating point format: Order of arrival of floating point diffuse and specular color parameters. 0 = BGRA 1 = RGBA</p> <p>If diffuse/specular colors are packed: Order of storage of the color components in the dword. 0 = BGRA (B=[7:0], G=[15:8], R=[23:16], A=[31:24]) 1 = RGBA (R=[7:0], G=[15:8], B=[23:16], A=[31:24])</p>
PRIM_WALK	5:4	<p>Method of Passing Vertex Data. 0 = Reserved 1 = Indexes (Indices embedded in command stream; vertex data to be fetched from memory) 2 = Vertex List (Vertex data to be fetched from memory) 3 = Vertex Data (Vertex data embedded in command stream)</p>

SE_VF_CNTL Setup Engine Vertex Fetcher Control		
Field Name	Bit(s)	Description
PRIM_TYPE	3:0	Primitive Type 0 = None (will not trigger Setup Engine to run) 1 = Point List 2 = Line List 3 = Line Strip 4 = Triangle List 5 = Triangle Fan 6 = Triangle Strip 7 = Triangle with wFlags (aka, RADEON "Type-2" triangles) * 8 = Rectangle List 9 = 3-Vertex Point List 10 = 3-Vertex Line List * Encoding 7 indicates whether a 16-bit word of wFlags is present in the stream of indices arriving when the VTX_AMODE is programmed as a '0'. The Setup Engine just steps over the wFlags word; ignoring it. 0 = Stream contains just indices, as: [Index1, Index0] [Index3, Index2] [Index5, Index4] ...etc... 1 = Stream contains indices and wFlags: [Index1, Index0] [wFlags, Index2] [Index4, Index3] [wFlags, Index5] ...etc...

FTLVERTEX

A vertex data block is denoted by FTLVERTEX_x where x is the ordinal number of the block. FTLVERTEX supplies the coordinates and associated attributes of a point in a 3-D space. The presence of some fields in a FTLVERTEX_x block depends on the fields of PM4_SE_VTX_FMT. Therefore, the size of a FTLVERTEX block may vary. The definition of FTLVERTEX is given as follows and the ordering of the fields in a FTLVERTEX block follows their ordering in the following table.

Type	Parameter	Description	Format
Position0 XY	X0	The x coordinate of the vertex	IEEE floating point
	Y0	The y coordinate of the vertex	IEEE floating point

Type	Parameter	Description	Format
Position0 Z	Z0	The z coordinate of the vertex	IEEE floating point
Position0 W	W0	The w coordinate of the vertex	IEEE floating point
Blend Weight(s)	BW0 BW1 BW2 BW3	0-4 Skinning Blend Weights (for vertex blending)	IEEE floating point
Vertex Normal 0	Nx0	The x coordinate of the vertex normal	IEEE floating point
	Ny0	The y coordinate of the vertex normal	IEEE floating point
	Nz0	The z coordinate of the vertex normal	IEEE floating point
Diffuse	ARGB	Diffuse color and alpha weight	Usually 8888, but can be three or four separate IEEE floating point values
Specular	ARGB	Specular color and fog weight	Usually 8888, but can be three or four separate IEEE floating point values
Texture Coordinate Set n n= 0,1,2,3	Sn	The 1st coordinate for texture number n (usually the single dimension horizontal component S)	IEEE floating point
	Tn	The 2nd coordinate for texture number n (usually the two dimension vertical component T)	IEEE floating point
	Qn	The 3rd coordinate for texture number n (usually the homogeneous W value)	IEEE floating point
Position1 XY	X1	The x coordinate of the vertex for blending	IEEE floating point
	Y1	The y coordinate of the vertex for blending	IEEE floating point
Position1 Z	Z1	The z coordinate of the vertex for blending	IEEE floating point
Position1 W	W1	The w coordinate of the vertex for blending	IEEE floating point
Vertex Normal 1	Nx1	The x coordinate of the vertex normal	IEEE floating point
	Ny1	The y coordinate of the vertex normal	IEEE floating point
	Nz1	The z coordinate of the vertex normal	IEEE floating point

Interpretation of vertices

The vertices in the packet are represented by an array of fields FTLVERTEX_1 through FTLVERTEX_n. The interpretation of the vertex array depends on the field VC_PRIM_TYPE. The following list the interpretations with respect to a given VC_PRIM_TYPE code (in parentheses).

Points (1)

A point is specified by one vertex.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The 1st point to be drawn.
2	FTLVERTEX_2	The 2nd point to be drawn.
	...	
n	FTLVERTEX_n	The n-th point to be drawn.

Lines (2)

A line is specified by 2 vertices, one representing the start point and the other representing the end point. To specify m lines, we need 2m vertices.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The start of the 1st line.
2	FTLVERTEX_2	The end of the 1st line.
3	FTLVERTEX_3	The start of the 2nd line.
4	FTLVERTEX_4	The end of the 2nd line.
	...	
n-1	FTLVERTEX_2m-1	The start of the m-th line.
n	FTLVERTEX_2m	The end of the m-th line.

Polylines (3)

A polyline is composed of a number of line segments with their ends connected to each other. Therefore, we need m+1 vertices to specify an m-segment polyline.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The start of the 1st line segment.
2	FTLVERTEX_2	The end of the 1st line segment, and the start of the 2nd line segment.
3	FTLVERTEX_3	The end of the 2nd line segment, and the start of the 3rd line segment.
	...	

Ordinal	Field Name	Description
n-1	FTLVERTEX_m	The end of the (m-1)-th line segment, and the start of the m-th line segment
n	FTLVERTEX_m+1	The end of the m-th line segment.

Triangles (4)

Three vertices are required to specify an independent triangle. Therefore, the total number of vertices required for specifying m independent triangles is 3m.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The 1st vertex of the 1st triangle.
2	FTLVERTEX_2	The 2nd vertex of the 1st triangle.
3	FTLVERTEX_3	The 3rd vertex of the 1st triangle.
4	FTLVERTEX_4	The 1st vertex of the 2nd triangle.
5	FTLVERTEX_5	The 2nd vertex of the 2nd triangle.
6	FTLVERTEX_6	The 3rd vertex of the 2nd triangle.
	...	
n-2	FTLVERTEX_3m-2	The 1st vertex of the m-th triangle.
n-1	FTLVERTEX_3m-1	The 2nd vertex of the m-th triangle.
n	FTLVERTEX_3m	The 3rd vertex of the m-th triangle.

Triangle Fan (5)

In drawing a triangle fan, vertex 1 is shared by all the triangles, and two neighboring triangles share two vertices (vertex 1 is one of them). That is, vertices 1, 2 and 3 are used to draw the first triangle; vertices 1, 3 and 4 to draw the second triangle; vertices 1, 4 and 5 to draw the third; and so on. If the triangle fan is composed of m triangle, the number of vertices required for specifying the fan is $n=m+2$.

Ordinal	Field Name	Description
1	FTLVERTEX_1	This vertex is shared by all the triangles, and is referred to as the 1st vertex by all the triangles.
2	FTLVERTEX_2	The 2nd vertex of the 1st triangle.
3	FTLVERTEX_3	The 3rd vertex of the 1st triangle and the 2nd vertex of the 2nd triangle.

Ordinal	Field Name	Description
4	FTLVERTEX_4	The 3rd vertex of the 2nd triangle and the 2nd vertex of the 3rd triangle.
	...	
n-1	FTLVERTEX_n-1	The 3rd vertex of the (m-1)-th triangle and the 2nd vertex of the m-th triangle.
n	FTLVERTEX_n	The 3rd vertex of the m-th triangle.

Triangle Strip (6)

A triangle strip is composed of a number of triangles where an adjacent pair share two vertices. With a triangle strip, only the first triangle uses three vertices, the subsequent triangles only need one new vertex for the rendering (two vertices from the previous triangle are re-used). That is, the drawing of the first triangle makes use of vertices 1, 2 and 3. The drawing of the second makes use of vertices 2, 3 and 4; and so on. If a triangle strip is composed of m triangle, the number of vertices required for specifying the strip is $n=m+2$.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The 1st vertex of the 1st triangle.
2	FTLVERTEX_2	The 2nd vertex of the 1st triangle and the 1st vertex of the 2nd triangle
3	FTLVERTEX_3	The 3rd vertex of the 1st triangle and the 2nd vertex of the 2nd triangle.
4	FTLVERTEX_4	The 3rd vertex of the 2nd triangle and the 2nd vertex of the 3rd triangle.
	...	
n-1	FTLVERTEX_n-1	The 3rd vertex of the (m-1)-th triangle and the 2nd vertex of the m-th triangle.
n	FTLVERTEX_n	The 3rd vertex of the m-th triangle.

Rectangles (8)

Three vertices are required to specify an independent rectangle. Any three of the four rectangle vertices can be used to specify the rectangle. Therefore, the total number of vertices required for specifying m independent rectangles is 3m.

Ordinal	Field Name	Description
1	FTLVERTEX_1	The 1st vertex of the 1st rectangle.
2	FTLVERTEX_2	The 2nd vertex of the 1st rectangle.
3	FTLVERTEX_3	The 3rd vertex of the 1st rectangle.
4	FTLVERTEX_4	The 1st vertex of the 2nd rectangle.
5	FTLVERTEX_5	The 2nd vertex of the 2nd rectangle.
6	FTLVERTEX_6	The 3rd vertex of the 2nd rectangle.
	...	
n-2	FTLVERTEX_3m-2	The 1st vertex of the m-th rectangle.
n-1	FTLVERTEX_3m-1	The 2nd vertex of the m-th rectangle.
n	FTLVERTEX_3m	The 3rd vertex of the m-th rectangle.

E.4.7 3D_RNDR_GEN_INDX_PRIM

Functionality

Render 3-D primitives points, lines, triangles, and rectangles using the Vertex Walker method. The data buffer pointed to by field PM4_VC_VLOFF is filled by the application. The vertex walker draws primitives according to the settings of field SE_VF_CNTL of the packet. The indices in the packet serve as pointers to the vertex data blocks in the associated vertex array which are selected for rendering. The selected vertex data are used by the vertex walker to carry out the rendering operation. If the packet does not have the index portion, i.e. the packet only consists of 5 fields (HEADER and 4 fields that follow it), it implies that the entire vertex array is used for rendering.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[PM4_VC_VLOFF]	The offset of the vertex array with respect to the physical address of the AGP space, As special service is required to convert the array address from the virtual space to this offset.
3	[PM4_VC_VSIZE]	The total number of vertices in the vertex array
4	[SE_VTX_FMT]	Same as field SE_VTX_FMT of packet 3D_RNDR_GEN_PRIM.

Ordinal	Field Name	Description
5	[SE_VF_CNTL]	Same as field SE_VF_CNTL of packet 3D_RNDR_GEN_PRIM. Its subfields should be set to the values relevant to the vertex walker operation. Also, registers PM4_VC_VLOFF, PM4_VC_VSIZE and PM4_VC_VFORMAT should be set up accordingly.
6	[INDX_2 INDX_1]	INDX_1: [15:0] :- the index of the 1st selected element in the vertex list. INDX_2 : [31:16] :- the index of the 2nd selected element in the vertex list.
7	[INDX_4 INDX_3]	The 3rd and 4th selected elements in the vertex list.
...		
n+5	[INDX_2n INDX_2n-1]	The last two selected elements in the vertex list. Note: the chosen elements can be any vertices, and their indices don't have to be contiguous. For example, one may select 5 vertices from 10 for rendering primitives. The indices of the selected vertices can be 0, 4, 5, 8 and 9. If the number of selected vertices is not even, the high word of the last DWORD of the packet may be filled with 0.

Vertex Array Format

Ordinal	Field Name	Description
1	{FTLVERTEX_1}	The 1st vertex data block
2	{FTLVERTEX_2}	The 2nd vertex data block
...		
2n	{FTLVERTEX_2n}	The n-th vertex data block

E.5 Miscellaneous Packets

E.5.1 WAIT_FOR_IDLE

Functionality

Wait for the Idle state of the engine. The packet is for debug use only.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[0x11111111]	Dummy field. It must be constant 0x11111111.

This page intentionally left blank.

Appendix F

Revision History

F.1 SDK-215R6-00-01 (SDK-215R6-00-01.pdf)

First draft completed in September 2001.

This page intentionally left blank.