RAGE™128
Software Development Guide

Technical Reference Manual

P/N: SDK-G04000 Rev 0.01

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.

Record of Revisions

Release	Date	Description of Changes
0.01	Aug 1999	First draft completed.

Related Manuals

RAGE 128 Register Reference Manuals

Table of Contents

Chapter 1: Overview

Chapter 2: Programming Basics

Chapter 6: CCE Packets

Chapter 7: Advanced Topics

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: CCE Command Packets

Chapter 1
Overview

1.1 Scope

This manual is a programming guide for the RAGE 128 graphics controller. The examples that are 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.* To request this manual, contact the ATI Developer Relations Department.

Background

The RAGE 128 is a fully integrated 128-bit graphics and multimedia accelerator. It combines astoundingly fast 3D and 2D acceleration with advanced multimedia capabilities. This accelerator incorporates new technologies such as Concurrent Command Execution (CCE). CCE was previously known as Programming Model 4 (PM4). CCE uses the RAGE 128's bus mastering capabilities to deliver excellent drawing performance, as well as simplifying the programming effort.

1.2 Major Features of the RAGE 128

- Highly optimized 128-bit engine.
- Triple 8-bit palette DAC with gamma correction for true WYSIWYG color. Pixel rates up to 250MHz (optional); 230MHz standard.
- Supports a variety of memory configurations for bandwidths of up to 2GB/s.
- Single Data Rate (SDR) SGRAM or SDRAM at up to 125MHz on a 128-bit interface (2GB/s).
- Double Data Rate (DDR) SGRAM at up to 125 MHz on a 64-bit interface (2GB/s).
- SDR SGRAM or SDRAM at up to 143MHz on a 64-bit interface (1.1GB/s)
- Flexible graphics memory configurations:
  - 2MB up to 32MB SDRAM or SDR/DDR SGRAM.
- DDC1 and DDC2B+ for plug and play monitors.
- Single-chip solution in 0.25µm, 2.5V CMOS technology.
  - Package options available for specific features.
- Hardware acceleration for the following:
  - BitBlt
  - Line Draw
  - Polygon/Rectangle Fill
  - Bit Masking
  - Monochrome Expansion
  - Panning/Scrolling
  - Scissoring
  - Full ROP support and hardware cursor (up to 64x64x2)
- Game acceleration including support for Microsoft's DirectDraw, Double Buffering, Virtual Sprites, Transparent Blit, and Masked Blit.
- Acceleration in 8-, 16-, 24-, 32-bpp modes.

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 RAGE 128.
2 Using the RAGE 128	Basic programming guide. A general understanding of the features and functions.
3 Getting Started	Using the RAGE 128 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 Execution Initialization and Usage	Overview of the CCE programming model: Setup and initialization of the CCE in various operational modes.
6 CCE Packets	Description of the CCE 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	CEE Command Packets

1.4 Notations and Conventions Used in this Manual

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.

Hexadecimal numbers will either be prefixed with “0x” (C-style) or appended with “h” (Intel assembly-style). Binary numbers will be appended with “b”. All other numbers are in decimal.

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
```

1.5 Nomenclature and Conventions

These conventions apply to the RAGE 128 Register Reference Manual.

1.5.1 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:

REGISTER_MNEMONIC

Example: **CONFIG_CHIP_ID** is the mnemonic for the Configuration Chip ID register.

REGISTER_MNEMONIC[Bit_Numbers]
- OR-
FIELD_NAME@REGISTER_MNEMONIC

For example, **CONFIG_CHIP_ID[15:0]** refers to the bit field that occupies bit positions [0] through [15] within this register.

CFG_CHIP_TYPE@CONFIG_CHIP_ID gives the field name **CFG_CHIP_TYPE** (Product Type Code) instead of the bits position.

1.5.2 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]**.

1.5.3 Register Description

All registers in this document are described with the format of the self-explained 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).

This page intentionally left blank.

Chapter 2
Programming Basics

2.1 Scope

This chapter details the basics about the RAGE 128's operation and drawing modes. The following topics are covered:

- Functional block diagram of the RAGE 128.
- Operation modes.
- Accelerator programming modes.
- Review of imaging terminology.
- Display modes and switching modes.

2.2 Overview

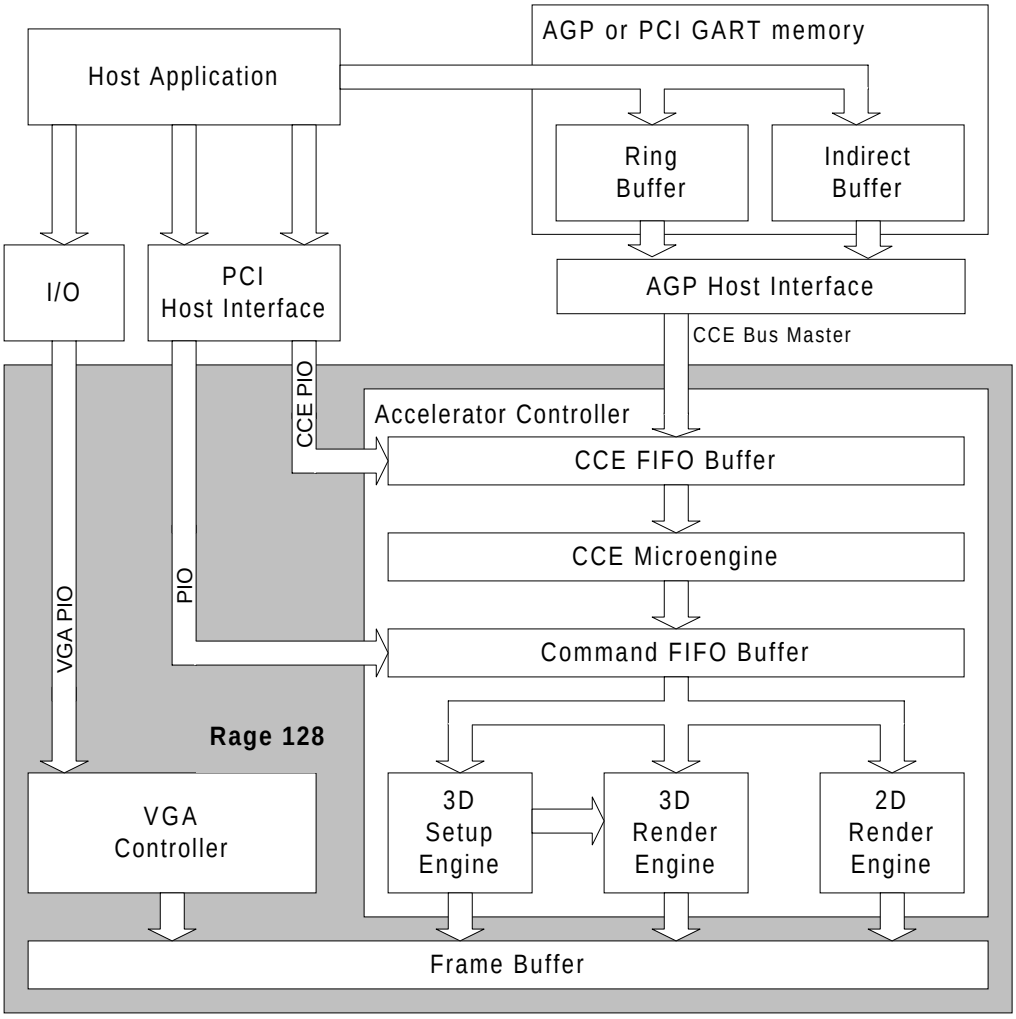

Figure 2-1. RAGE 128 Structure and Data Flow

This chapter presents a basic description of the functional blocks in this diagram. Detailed descriptions are presented in subsequent chapters. For a summary of the RAGE 128's functional blocks, *refer to Table 2-1*. For a summary of the RAGE 128's buffers, *refer to Table 2-2*.

Table 2-1 RAGE 128 Functional Blocks

Functional Unit	Purpose
Accelerated Graphics Port (AGP) Interface	Transfers data from the ring buffer (located in the system memory) to the RAGE 128's CCE FIFO buffer without direct involvement of the CPU.
VGA Controller	Manages pixel operations under VGA mode.
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.
3D Render Engine	Performs 3D primitive rasterization.
3D Setup Engine	Performs 3D primitive setup operations.

Table 2-2 RAGE 128 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	192 DWORDs	Contains register/data pairs for processing by the 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 8MB to 32MB.	Contains all on-screen and off-screen rendering buffers, such as: drawing, stencil and z buffers, bitmaps, and texture maps.

2.3 Operation Modes

The RAGE 128 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 (*refer to Figure 2-1.*). 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 Super VGA programming, the RAGE 128 supports the Video Electronics Standard Association (VESA) Video BIOS Extension (VBE) 2.0 programming interface. This interface was created by VESA to provide a standrard, 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 RAGE 128'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 three engines:

- 2D Rendering Engine that performs 2D rasterization.
- 3D Setup engine that performs 3D primitive setup operations.

- 3D Render Engine that performs 3D rasterization.

The three engines are collectively referred to as the Graphical User Interface (GUI) engine (*refer to Figure 2-1.*).

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 RAGE 128's memory-mapped registers. The registers are written through one of the RAGE 128's two register apertures over the bus interface. The register writes are queued in the RAGE 128's internal 192 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, *refer to Figure 2-1.*

For more details about the PIO-mode programming, *refer to Chapter 4*.

For more details about the RAGE 128's register apertures, *refer to 2.6*.

Concurrent Command Execution (CCE) Programming Mode

In this mode, the host sends commands to the RAGE 128 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 RAGE 128, 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 RAGE 128. 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 RAGE 128. In addition, there are several predefined single-purpose packets that greatly simplify the programming of common drawing operations.

The RAGE 128 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 or PCI GART memory. The PCI GART is a mechanism for simulating AGP functionality on the RAGE 128 over the PCI bus. For more details about the PCI GART, *refer to 2.6.6*. The host and RAGE 128 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 RAGE 128 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 RAGE 128 where to read the next packet by writing to a CCE write-pointer register. The RAGE 128 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 RAGE 128 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 RAGE 128 during CCE initialization.

To view a diagram of the ring buffer, *refer to Figure 2-2.*

For more details about programming the ring buffer *refer to Chapter 5*.

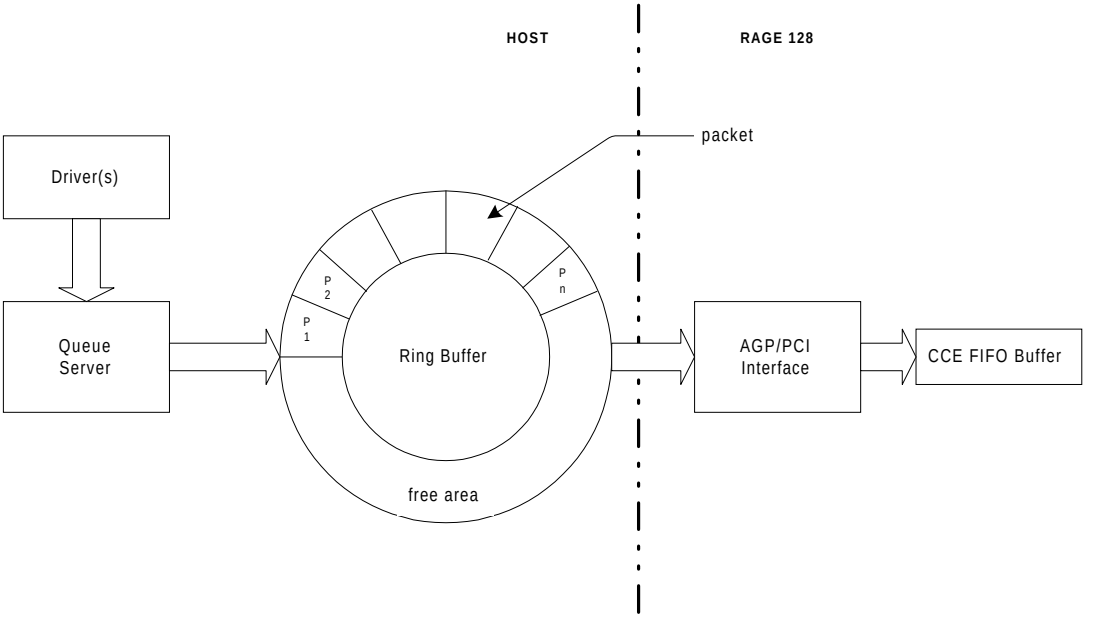

Figure 2-2. The Ring Buffer

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, *refer to Chapter 5*.

Indirect Buffer

The indirect buffer is a contiguous block of memory allocated by the host application in AGP or PCI GART memory. The host and RAGE 128 treat this as a linear buffer. They do not employ any buffer wrapping mechanisms for the indirect buffer.

The indirect buffer is similar to the ring buffer in that the host places packets in it and the RAGE 128 transfers them out using bus-mastering. But while the ring buffer is meant to be continuously updated, which results in content being constantly overwritten, the

indirect buffer may be filled with static packets that are merely updated but not completely overwritten. This is a more efficient way to handle common or frequently used packets such as blits, rectangle fills, etc. One additional difference is that packets may be placed and accessed from the indirect buffer in arbitrary order.

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 5.3*.

The most efficient combination is to use both the ring buffer and the indirect buffer. The indirect buffer may be used for storing frequently used packets, and the ring buffer may be used for general command streaming. Packets can be used to write to specific registers, so the register writes that trigger indirect buffer transfers can also be streamed as ring buffer packets. If the ring buffer is not used, these registers may be written through PIO.

To view a conceptual diagram of the indirect buffer, *refer to Figure 2-3.*

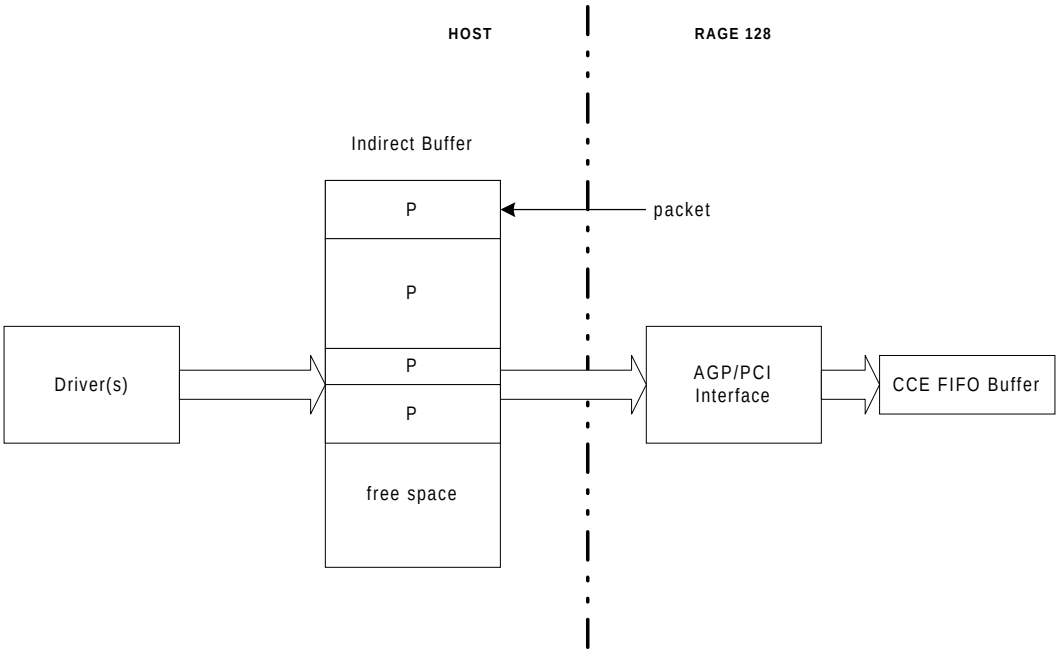

Figure 2-3. The Indirect Buffer

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 RAGE 128 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-, 24-, 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.

- Three bytes for the 24-bpp format.
- 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 RAGE 128			
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 LSB	9 A B C D E F 10	11 12 13 14 15 16 17 18	19 1A 1B 1C 1D 1E 1F 20 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 RAGE 128			
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 RAGE 128			
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 RAGE 128			
Pixel #2 aRRRRRGGGGGBBBBB		Pixel #1 aRRRRRGGGGGBBBBB	
Drawing Data Placed in Video Memory			
Pixel #1 low GGGBBBBB	Pixel #1 high aRRRRRGG	Pixel #2 low GGGBBBBB	Pixel #2 high aRRRRRGG
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 RAGE 128			
Pixel #2 RRRRRGGGGGBBBBB		Pixel #1 RRRRRGGGGGBBBBB	
Drawing Data Placed in Video Memory			
Pixel #1 low GGGBBBBB	Pixel #1 high RRRRRGGG	Pixel #2 low GGGBBBBB	Pixel #2 high RRRRRGGG

24-bpp Format

Each color component uses a byte to represent its intensity.

Table 2-8 24-bpp Format (display only)

24-bpp Format (display only)			
Structure of the Drawing Data as Used by the RAGE 128			
B2	R1	G1	B1
G3	B3	R2	G2
R4	G4	B4	R3
Drawing Data Placed in Video Memory			
B1	G1	R1	B2
G2	R2	B3	G3
R3	B4	G4	R4
Note: B2 means pixel 2, blue component; R1 is pixel 1, red component, etc.			

32-bpp, RGBa, or 8888 Format

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

Table 2-9 32-bpp, RGBa, or 8888 Format

32-bpp, RGBa, or 8888 Format			
Structure of the Drawing Data as Used by the RAGE 128			
a	R	G	B
Drawing Data Placed in Video Memory			
B	G	R	a

2.5.4 Pixels

The RAGE 128 supports pixel depths of 1, 8, 15, 16, and 32 bits per pixel. When operated in the 24-bpp format mode, some software assistance is required.

The pixels are consumed from the most significant bit (MSB) to the least significant bit (LSB) (or vise versa, depending on the rage 128'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 RAGE 128'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:

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

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

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.

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

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

Using Equation 2.3, the pitch can be written as:

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

The size of memory for the pitch is:

$$PitchMemSize = l \times Pitch \times 8$$ 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 x 100 x 8) provided each pixel is represented by 16-bit color. To enable the block-write capability of the RAGE 128, 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:

$$l \times (m+n) \,\mathrm{MOD}\, 128 = 0$$ 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 RAGE 128 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. Then, 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 RAGE 128 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 4MB. 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 0xFFFFF.

The RAGE 128 addresses the video memory from zero to the upper bound. The video memory address seen by RAGE 128 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.

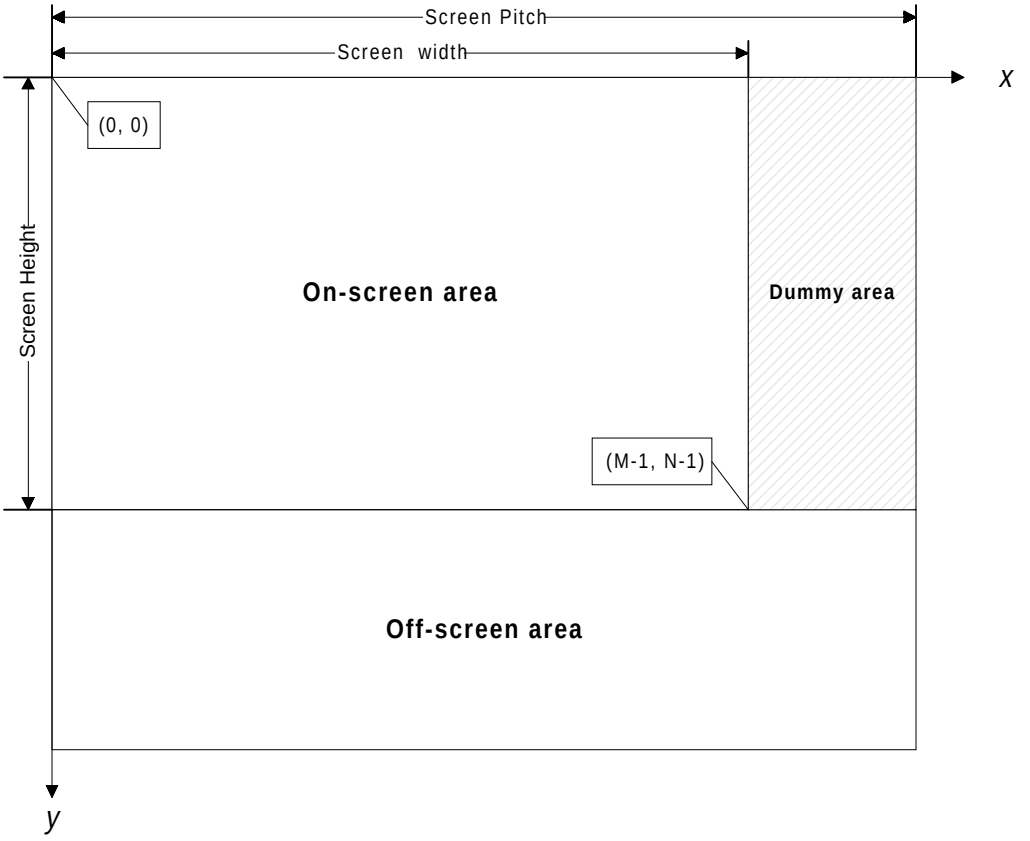

Figure 2-4. Video Memory

2.6 Memory Apertures

The RAGE 128 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 RAGE 128.
- The ***video aperture***.

Normally, the apertures are located somewhere within the 4GB address space where it does not conflict with the system (host) memory. Further, an aperture must be located on a 32MB boundary.

The following diagram shows the typical memory organization for the RAGE 128. Unless otherwise specified, all addresses in the register definition refer to a 64MB virtual address space.

The following groups are used:

- The first 32MB map to the frame buffer space.
- The next 32MB map to the AGP/PCI space, specifically:
  $Address_{AGP}$ = AGP_base + (offset - 32MB)
  $Address_{PCI}$ = Physical address in BM_GUI_TABLE.

The following registers are used to point to the apertures:

- Registers **`REG_BASE`** and **`CONFIG_REG_1_BASE`** point to the register apertures.
- Registers **`CONFIG_APER_0_BASE`** and **`CONFIG_APER_1_BASE`** point to the video aperture.

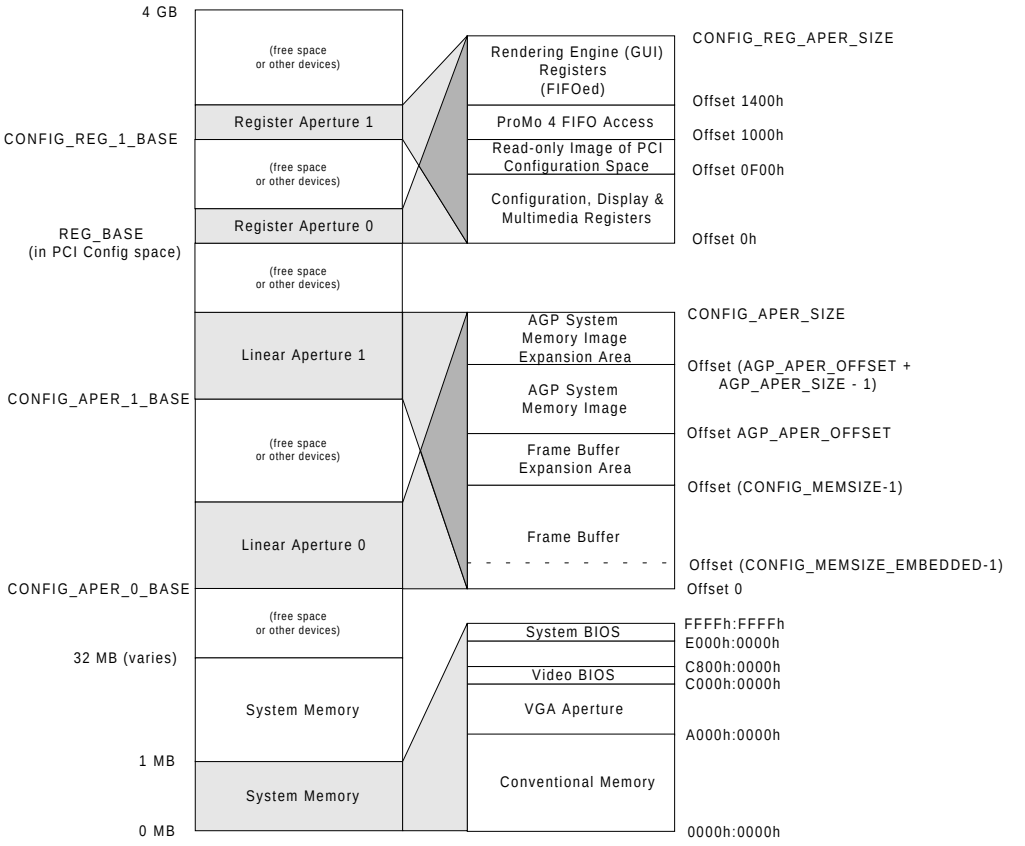

Figure 2-5. Memory Map

2.6.1 VGA Memory Aperture

When enabled for VGA, the RAGE 128 claims the standard VGA resources. The bits contained in **GRPH_ADRSEL** register determine the position and size of the VGA memory aperture. For most VGA graphics modes, the aperture is 128KB and starts at segment 0xA000.

2.6.2 Video BIOS

To relocate the RAGE 128'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.

2.6.3 Register Apertures

There are two memory-mapped register apertures in the RAGE 128. Each references the entire set of memory-mapped registers. Under the Intel architecture, one may be mapped for UnCached (UC) access, and the other for Write Combining (WC) access. WC is also uncached, but it is faster because it uses an out-of-order write buffer. The WC-mapped aperture may be used where speed is essential (e.g, when setting 3D states or setting up primitives). The UC-mapped aperture may be used when order is important (e.g, when initiating drawing operations).

Under the PowerMac architecture, the second aperture may be used for big-endian memory access.

Purpose

The register apertures contain all direct-accessed registers that are found in the RAGE 128, except for the VGA and PCI configuration registers. In addition, these registers also have the index/data pairs used for all indirectly accessed registers and memories.

Location

These registers are re-locatable. Base address of register aperture 0 is:

- Determined by the **REG_BASE** register (found in the PCI configuration space), or
- Readable in the I/O register aperture using **MM_INDEX** <= 0xF18 and reading **MM_DATA**.

Base address of register aperture 1 is determined by **CONFIG_REG_1_BASE**, which can be read in register aperture 0 once its base has been found as indicated above. Reading **CONFIG_REG_1_BASE** is the only method of determining register aperture 1's location that is forward compatible with future generations of the hardware.

Size

The size will grow for future generations of the RAGE 128. The contents of the **CONFIG_REG_APER_SIZE** register contains the sizes for each register aperture.

Memory Map

The following table shows the memory map.

Table 2-10 Memory Map

From	To	Description
0x0000	0x00FF	Non-GUI registers, also directly accessible in IOR space.
0x0100	0x0EFF	Non-GUI registers.
0x0F00	0x0FFF	Read-only copy of PCI configuration space.
0x1000	0x13FF	CCE FIFO direct access.
0x1400	0x1FFF	GUI registers.

2.6.4 Linear Memory Apertures

There are two copies of the linear memory aperture in the RAGE 128. Each copy is identical. The reason for two copies is to allow each to be independently marked as big-endian or little-endian in the PowerMac environment. For Wintel architectures, the second aperture may be used, but there is no valid reason to do so.

Purpose

The linear memory apertures allow access to the frame buffer memory, and for AGP systems to the AGP memory as seen by the RAGE 128.

Location

These apertures are re-locatable.

The **CONFIG_APER_0_BASE** register determines the base address of linear aperture 0. The **CONFIG_APER_1_BASE** register determines the base address of linear aperture 1. Both these registers can be read in any register aperture.

Size

The size of these registers will grow for future generations of the RAGE 128. The **CONFIG_APER_SIZE** register contains the size of each linear aperture.

Frame Buffer

The frame buffer image occupies the area in each aperture from offset 0 to CONFIG_MEMSIZE-1.

If **CONFIG_MEMSIZE_EMBEDDED** is greater than zero, the RAGE 128 uses an on-chip memory for the first piece of the frame buffer. This embedded memory is included in the **CONFIG_MEMSIZE** total. Currently, the RAGE 128 does not have any embedded memory. A future RAGE 128 model is planned to incorporate this embedded memory. The RAGE 128 supports up to 32MB of frame buffer memory.

2.6.5 AGP System Memory Image

Each linear aperture also contains an image of the AGP system memory as seen by the RAGE 128.

Typically, the host application would directly access the AGP system memory using the system processor. Since using this AGP image to access AGP memory generates an AGP slave and an AGP bus master cycle for each access (or group of accesses), it is highly inefficient; therefore, this method is not recommended.

Use this AGP image for debugging and allowing a method for flushing out pending AGP cycles still in the host chipset (before software directly accesses system memory).

The AGP image starts at offset **AGP_APER_OFFSET** in each linear aperture.

The **AGP_APER_SIZE** register contains the size of the AGP memory. This register is not a number, but an enumerated type that must be converted into a number (refer to the register definition).

The RAGE 128 supports up to 32MB of AGP memory.

2.6.6 RAGE 128 PCI GART

The RAGE 128 provides a mechanism for accessing system memory as AGP memory over the PCI bus. This mechanism allows up to 32MB of system memory on AGP cards, and up to 4MB on PCI cards, to be used as an AGP area using a scatter gather mechanism.

To use this feature, a 32KB table of page entries must be prepared and its physical base address must be written to the **PCI_GART_PAGE** register. The table must be 4KB aligned. Each table entry must contain the physical base address of a 4KB page allocated from system memory for the AGP area.

The PCI GART table is enabled by setting bit [0] of **PCI_GART_PAGE** to ‘0’. For AGP systems, this bit must be explicitly set to ‘1’ during initialization to disable the PCI GART.

To force use of the PCI GART on AGP-capable systems, these additional steps are necessary:

- The **BM_CHUNK_0_VAL:BM_PTR_FORCE_TO_PCI** field must be set to ‘1’.
- The **BM_CHUNK_0_VAL:BM_RD_FORCE_TO_PCI** field must be set to ‘1’.
- The **BM_CHUNK_0_VAL:BM_GLOBAL_FORCE_TO_PCI** field must be set to ‘1’.

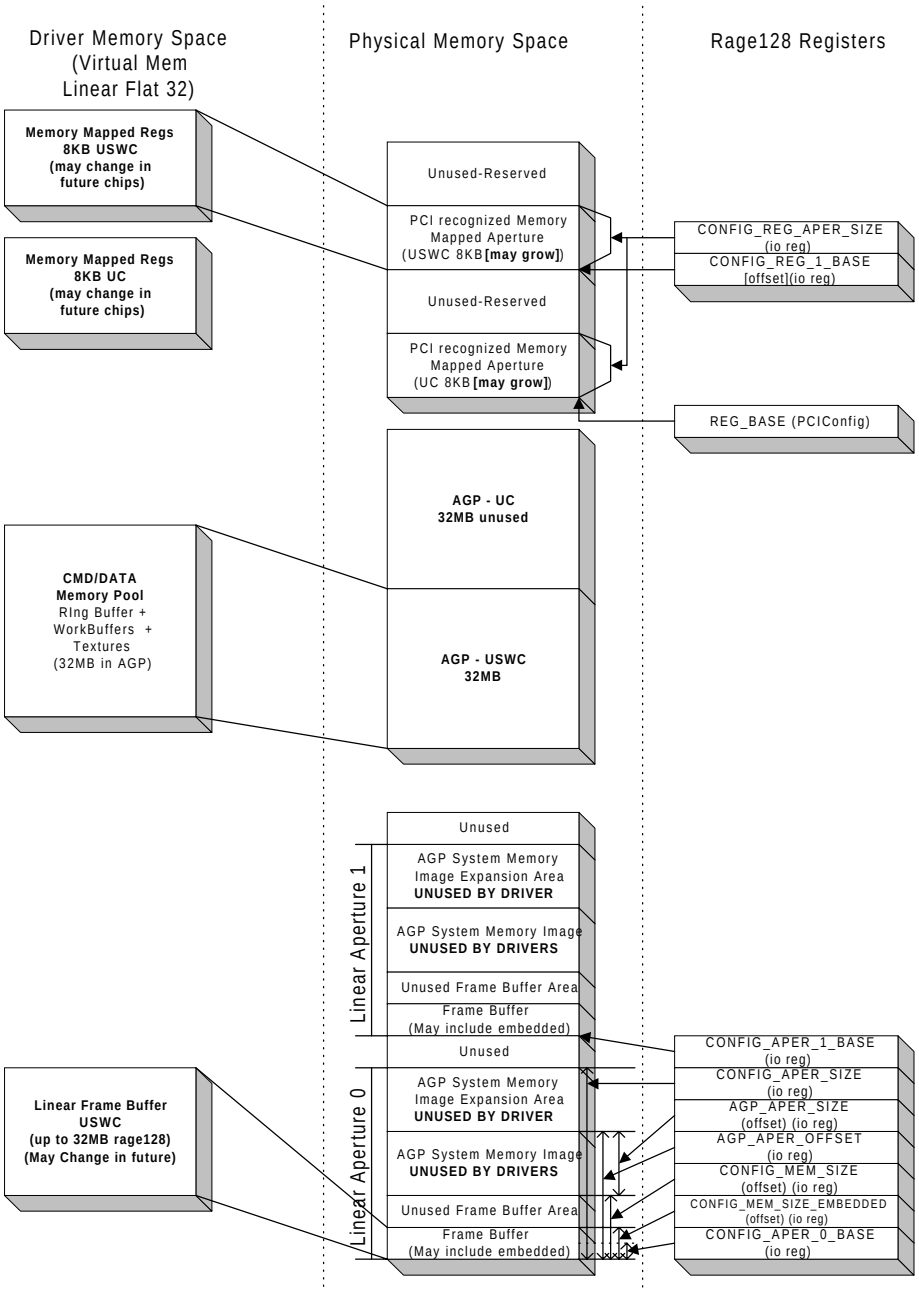

Figure 2-6. AGP Memory Architecture - Software Layout

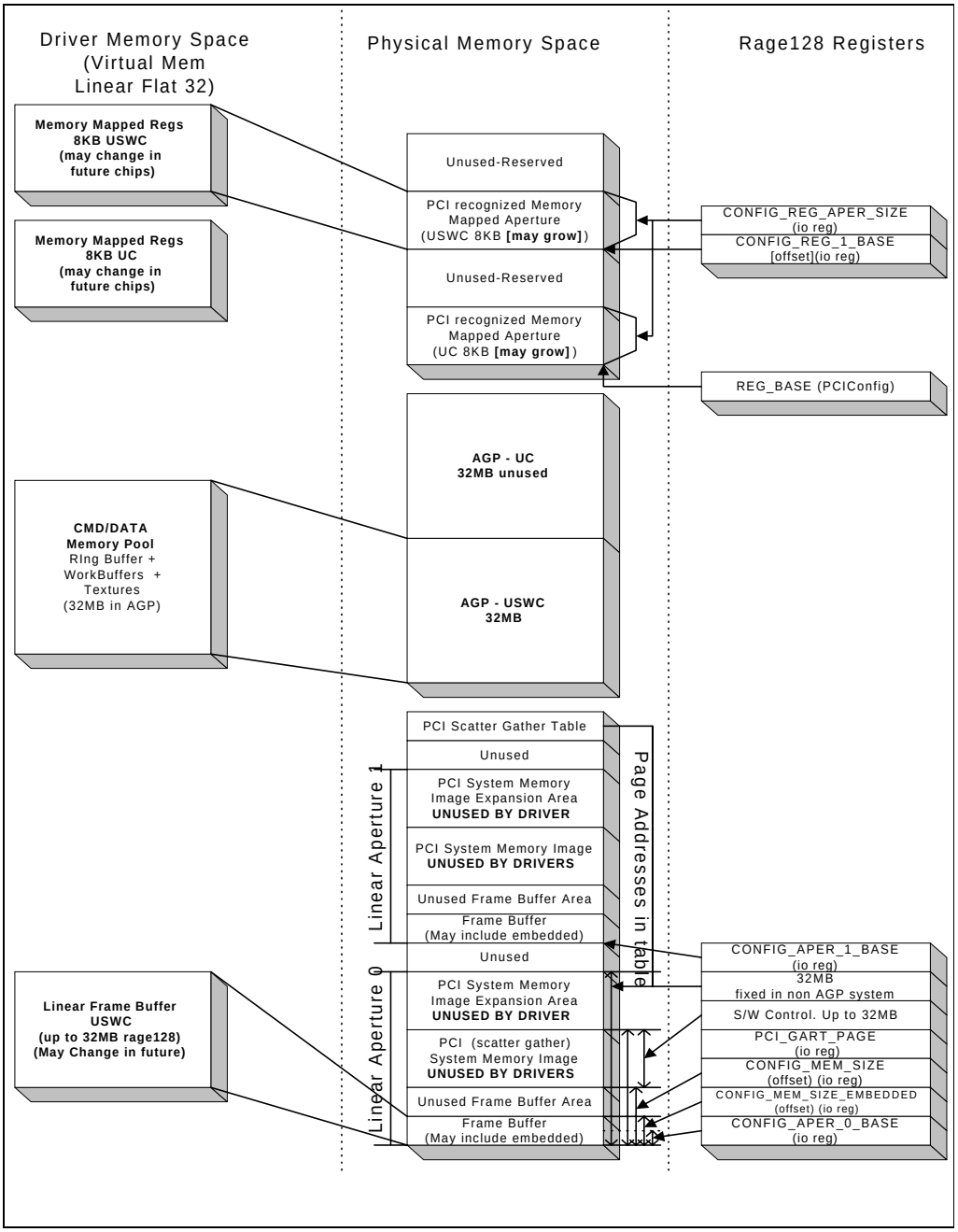

Figure 2-7. PCI Non-AGP Memory Architecture - Software Layouts

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 RAGE 128 according to the RAGE 128's capability and the available memory resource.

The RAGE 128 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 RAGE 128 supports the graphics mode with screen resolutions (from 320x200 to 1600x1200 pixels), and with depths of color (8, 16, 24 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 RAGE 128'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 RAGE 128 may hang (i.e. stop operating). To avoid hanging the RAGE 128, 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 **GUI_STAT** is set to zero.
- When the RAGE 128 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 RAGE 128 or configuring the display mode of the system.

For details about *BIOS Services*, see the *Appendix for the "BIOS Function Calls", on page A-1*.

This page intentionally left blank.

Chapter 3
Accelerator Operation Mode

3.1 Scope

This chapter contains information about setting up the RAGE 128 for accelerator-operation mode. The intended audience for this information is X-type OS driver developers.

This chapter shows how to detect the RAGE 128 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 RAGE 128 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 RAGE 128.
- 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 Step 1: Detect the RAGE 128

This step determines the following:

- The presence of a RAGE 128 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 RAGE 128 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.

3.2.1 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 RAGE 128 device IDs. The current list of RAGE 128 device IDs are as follows:

Table 3-1 RAGE 128 Device IDs

Package ID	Device ID	Description
RE	0x5245	312pin, PCI 33 only
RF	0x5246	312pin, AGP 1X & 2X
RK	0x524B	256pin, PCI 33 only
RL	0x524C	256pin, AGP 1X & 2X

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 RAGE 128 device was not found
     -OR-
   - Scan the BIOS segment to see if the 'R128' signature string is found (note: a way to detect a RAGE 128 revision that may not be in the list). This protects against a driver not detecting new or revised RAGE 128s. 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 RAGE 128, contact Developer Relations at ATI (www.atitech.com).

3.2.2 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'.
- RAGE 128 string
  The RAGE 128 string is 'R128'.

For a successful installation of the RAGE 128, all three of these items must be present. They should all be present within the first 512 bytes of the BIOS segment.

3.2.3 Scratch Register Test

To confirm the presence of a RAGE 128 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 RAGE 128 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**.

3.3 Step 2: Obtain the Configuration Information

After locating the PCI configuration space for a RAGE 128, 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 0xFFFFFF00.

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 RAGE 128-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).

3.4 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.

3.4.1 Using the BIOS Function

The RAGE 128 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 R128_SetMode (BYTE xres, BYTE yres, BYTE bpp)
{
    union REGS r;

    memset (&r, 0, sizeof (r));
    r.w.ax = 0xA000;                    // Function 00h: Set Mode.
    r.w.ch = 0x00;// Set initially to 0, will be filled in.

    // Determine requested resolution mode number.
    if ((xres == 320) || (xres == 640))
    {
        switch (yres)
        {
            case 200:   r.h.ch = 0xE2;
                        break;
            case 240:   r.h.ch = 0xE3;
                        break;
       case 350:r.h.ch = 0xE6;
                   break;
       case 400:r.h.ch = 0xE1;
                   break;
       case 480:r.h.ch = 0x12;
                   break;
            default:    break;
        }
    }
    else
    {
        switch (xres)
        {
            case 512:   r.h.ch = 0xE4;
                        break;
            case 800:   r.h.ch = 0x6A;
                        break;
            case 1024:  r.h.ch = 0x55;
                        break;
            case 1280:  r.h.ch = 0x83;
```

```
                        break;
                case 1600:  r.h.ch = 0x84;
                        break;
                default:    printf ("\nUnsupported resolution!\n");
                        return (0);
                        break;
            }

        } // if/else

        // if r.h.ch is still 0, an invalid xres or yres was passed.
        // we must return a failure
        if (r.h.ch == 0)
        {
            printf ("\nUnsupported Resolution!\n");
            return (0);
        }

        // 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 24:    r.h.cl = 0x05;
                        break;
            case 32:    r.h.cl = 0x06;
                        break;
            default:    printf ("\nUnsupported pixel depth!\n");
                        return (0);
                        break;
        } // switch

        // fill in the appropriate values for the global structure.
        R128_AdapterInfo.xres = xres;
        R128_AdapterInfo.yres = yres;
        R128_AdapterInfo.pitch = xres/8;  // we'll set pitch = xres/8 by
    default.
        R128_AdapterInfo.bpp = bpp;

        // Call the BIOS to set the mode.
        int386 (0x10, &r, &r);
        if (r.h.ah)
        {
            return (0);                     // Error setting mode.
        }
        else
```

```
    {
        return (1);
    } // if

} // R128_SetMode
```

3.4.2 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 R128_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 */
        R128_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 | R128_GetBPPValue (R128_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);
        }
    } // R128_SetDisplayModeFromTable ()...
```

3.4.3 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 RAGE 128 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.
- **MPP_TB_CONFIG** - disable MPP usage for TV out.
- **MPP_GP_CONFIG** -disable general purpose MPP.
- **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.
- **CAP1_TRIG_CNTL** - disable capture buffer 1.

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**
    - **CTRC_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 - t**he 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 RAGE 128 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.

3.4.4 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 RAGE 128-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 RAGE 128'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.

3.4.5 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 R128_PLLGetDividers (WORD Frequency)
//
// DESCRIPTION:
//  Generates post and feedback dividers for desired pixel clock frequency.
//
// PARAMETERS:
//  Frequency             Desired frequency in units of 10 kHz.
//
{
  DWORD FeedbackDivider;                    // Feedback divider value
  DWORD OutputFrequency;                    // Desired output frequency
  BYTE PostDivider = 0;                     // Post Divider for Pixel Clock

  //
// The internal clock generator uses a PLL feedback system to
// produce the desired frequency output according to the following
// equation:
  //
  // Output Frequency = (Reference Frequency * Feedback Divider) /
//                 (Reference Divider *  Post Divider)
  //
```

```
  // Where Reference Frequency is the reference crystal frequency,
// FeedbackDivider is the feedback divider (from 128 to 255 inclusive),
// and Reference Divider is the reference frequency divider.
  // Post Divider is the post-divider value (1, 2, 3, 4, 6, 8, or 12).
  //
  // The required feedback divider can be calculated as:
  //
  // Feedback Divider = (Post Divider * Reference Divider *
//                 Output Frequency) / Reference Frequency
  //


  // Make sure that the requested dot clock frequency does not exceed
  // the maximum possible output frequency.
  if (Frequency > PLL_INFO.max_freq)
  {
        Frequency = (WORD)PLL_INFO.max_freq;
  }

  // Make sure that the requested dot clock frequency is not less than
  // the lowest possible output frequency.
  if (Frequency * 12 < PLL_INFO.min_freq)
  {
        Frequency = (WORD)PLL_INFO.min_freq / 12;
  }

  OutputFrequency = 1 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.max_freq))
  {
        PostDivider = 1;
        goto _PLLGetDividers_OK;
  }

  OutputFrequency = 2 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.max_freq))
  {
        PostDivider = 2;
goto _PLLGetDividers_OK;
  }

  OutputFrequency = 3 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.max_freq))
  {
  PostDivider = 3;
```

```
        goto _PLLGetDividers_OK;
  }

  OutputFrequency = 4 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.max_freq))
  {
  PostDivider = 4;
        goto _PLLGetDividers_OK;
  }

  OutputFrequency = 6 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.max_freq))
  {
  PostDivider = 6;
        goto _PLLGetDividers_OK;
  }

  OutputFrequency = 8 * Frequency;

  if ((OutputFrequency >= PLL_INFO.min_freq) &&
(OutputFrequency <= PLL_INFO.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_INFO.ref_div * OutputFrequency,
    PLL_INFO.ref_freq);


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

```
    return;

}   // R128_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 RAGE 128. See the source code file "r128pll.c" for the steps required to program the actual PLL registers.

3.4.6 Programming the DDA

To determine how to access the display FIFO, the RAGE 128 uses a digital differential analyzer (DDA). In order for the display to work properly, program the two DDA registers with the proper values so that the display FIFO behaves properly and the resulting display is correct.

The affected registers are:

- **DDA_CONFIG**
- **DDA_ON_OFF**

Calculate the following values:

- Number of memory clock cycles (XCLKS) per transfer to the display FIFO: x
- Minimum number of bits required to hold the integer portion of x: b
- Useable precision: (b + 1) = p
- Display FIFO off point: r_{off}
- Display FIFO on point: r_{on}
- Loop latency factor for the hardware: r_{loop}

To calculate these values, use the following series of equations. First, determine the amount of memory clock cycles that are used in a transfer to the display FIFO.

x =
{Memory Clock (MHz) / Pixel Clock (MHz) } * {Display FIFO Width/Bits per Pixel}

Where x is equal to the number of memory clocks in a transfer. This is the display FIFO width (in bits) for the RAGE 128 (for all display modes).

Calculate the minimum number of bits required to hold the integer portion of x (i.e. calculate b). This value is easily found by shifting the x to the right until x = '0'. At this point, the number of shifts would represent the number of bits (b) required. The useable precision (p) is equal to the minimum number of bits previously calculated plus 1 (i.e. p = b + 1).

Use the following equation to calculate the display FIFO off point:

$$\mathbf{r_{off} = x * (d - 4)}$$

In the above equation, d is equal to the number of transfers (octwords) for a display FIFO entry. For extended accelerator modes, d = 32. For VGA modes, d = 64.

Calculating the display FIFO on point requires information about the type of memory installed on the adapter. The table below shows the currently used memories on the RAGE 128, with the necessary specification values required to calculate the display FIFO on setting. All the values in the table are expressed in memory clock cycles (XCLKS).

Table 3-3 Memory Specifications

	128 bit SDR 1:1	64 bit SDR 1:1	64 bit SDR 2:1	64 bit DDR
Memory Read Latency (ML)	4	4	4	4
Maximum Burst Length (MB)	4	8	4	4
RAS to CAS delay (t_{rcd})	3	3	1	3
RAS precharge (t_{rp})	3	3	2	3
write recovery (t_{wr})	1	1	1	2
CAS Latency (CL)	3	3	2	3
read to write delay (t_{r2w})	1	1	1	1
Loop Latency	16	17	16	16
DSP_ON	26	38	20	27
Note: All values expressed in XCLKS.				

Use the following equation to calculate the display FIFO on point:

$$\mathbf{r_{on} = 4 * MB + 3 * MAX(t_{rcd} - 2, 0) + 2 * t_{rp} + t_{wr} + CL + t_{r2w} + x}$$

For memory types that might be used in the future, the values in the table above are constant. In addition, they can be retrieved from the following registers (if you are using a platform that has a video BIOS):

- ML = MEM_LATENCY
- MB = 8 for 64 bit SDR 1:1, 4 otherwise
- t_{rcd} = 1 for SDR 2:1, MEM_TRCD otherwise
- t_{rp} = 2 for SDR 2:1, MEM_TRP otherwise
- t_{wr} = MEM_TWR
- CL = 2 for SDR 2:1, CAS_LATENCY otherwise
- t_{r2w} = MEM_TR2W

Be aware of a loop latency factor, which is incurred in the hardware. Ensure that the display FIFO on point plus the loop latency factor is less than the display FIFO off point (otherwise the display mode is not guaranteed to work).

Use the following equation to calculate the loop latency factor:

$$\mathbf{r_{loop} = 12 + ML}$$

However, for 64 bit SDR 1:1, use the following equation:

$$\mathbf{r_{loop} = 12 + ML + 1}$$

Therefore, to guarantee an operational mode, make sure that the following equation is true:

$$\mathbf{r_{on} + r_{loop} < r_{off}}$$

The values written to the DDA registers are as follows:

- **DDA_ON** = $r_{on} * 2^{11-p}$
- **DDA_OFF** = $r_{off} * 2^{11-p}$
- **DDA_PRECISION** = p
- **DDA_XCLKS_PER_XFER** = $x * 2^{11-p}$
- **DDA_LOOP_LATENCY** = r_{loop}

Program the registers as follows:

- **DDA_CONFIG** = **DDA_XCLKS_PER_XFER** | (**DDA_PRECISION** << 16) | (**DDA_LOOP_LATENCY** << 20)
- **DDA_ON_OFF** = **DDA_OFF** | (**DDA_ON** << 16)

3.5 Step 4: Initialize the GUI Engine

After setting a display mode, this step involves using the acceleration capabilities of the RAGE 128 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.
   - In 24-bpp format modes, the engine pitch values must be multiplied by 3.
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 RAGE 128 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 3-4 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).

Table 3-4 DP_GUI_MASTER_CNTL (Continued)

Field Name	Purpose
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.
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. 2 = 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_CONVERSION_TEMP	The default color conversion temperature is set here. This deals with color space conversions when using the front or back end scalars.
GMC_ROP3	The default raster operation is set here. The RAGE 128 supports all 256 ROPs as per the MS Win3.1 DDK. See appendix D regarding available ROP values.
DP_SRC_SOURCE	Determines the pixel source for source data. Possible sources are display memory and the host data registers.
GMC_3D_FCN_EN	Allows the clearing of the SCALE_3D_FCN register, which is required when initially setting up the drawing engine.

Table 3-4 DP_GUI_MASTER_CNTL (Continued)

Field Name	Purpose
GMC_CLR_CMP_CNTL_DIS	Enables or disables the color compare function.
GMC_AUX_CLIP_DIS	Enables or disables the auxiliary scissor registers.
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_BRES_ERR**
   - **DST_BRES_INC**
   - **DST_BRES_DEC**

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 R128_InitEngine (void)
{
  DWORD temp, bppvalue;

  // determine engine pitch
  temp = R128_AdapterInfo.pitch;
  if (R128_AdapterInfo.bpp == 24)
  {
        temp = temp * 3;
  }

  // setup engine offset registers
```

```
R128_WaitForFifo (4);
regw (DEFAULT_OFFSET, 0x00000000);

// setup engine pitch registers
regw (DEFAULT_PITCH, temp);

// set the default scissor registers to maximum dimensions
regw (DEFAULT_SC_TOP_LEFT, 0x00000000);
regw (DEFAULT_SC_BOTTOM_RIGHT, (0x1FFF << 16) | 0x1FFF);

// Set the drawing controls registers.
R128_WaitForFifo (1);
bppvalue = R128_GetBPPValue (R128_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         |
                (bppvalue << 8)              |
                GMC_SRC_DSTCOLOR             |
                GMC_BYTE_ORDER_MSB_TO_LSB    |
                GMC_CONVERSION_TEMP_6500     |
                ROP3_SRCCOPY                 |
                GMC_DP_SRC_RECT              |
                GMC_3D_FCN_EN_CLR            |
                GMC_DST_CLR_CMP_FCN_CLR      |
                GMC_AUX_CLIP_CLEAR           |
                GMC_WRITE_MASK_SET);

R128_WaitForFifo (7);

// Clear the line drawing registers
regw (DST_BRES_ERR, 0);
regw (DST_BRES_INC, 0);
regw (DST_BRES_DEC, 0);

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

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

// Wait for engine idle before returning
R128_WaitForIdle ();

return;
```

```
} // R128_InitEngine ()
```

Chapter 4
Programming

4.1 Scope

This chapter describes how to program the RAGE 128 to perform drawing operations. This chapter also discusses some aspects of programming the RAGE 128 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 RAGE 128, the command queue consists of 64 DWORD entries. The **GUI_FIFOCNT@GUI_STAT** 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 R128_WaitForFifo (DWORD entries)
{
  WORD starttick, endtick;

  starttick = *((WORD *) (DOS_TICK_ADDRESS));
  endtick = starttick;
  while ((regr (GUI_STAT) & 0x00000FFF) < entries)
  {
        endtick = *((WORD *) (DOS_TICK_ADDRESS));
        if (abs (endtick - starttick) > FIFO_TIMEOUT)
        {
              gui_stat = regr (GUI_STAT);
              R128_ResetEngine ();
        } // if
  } // while

  return;

} // R128_WaitForFifo ()
```

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) **GUI_FIFOCNT@GUI_STAT** until its contents equal 64.
2. Then, poll **GUI_FIFOCNT@GUI_STAT** 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 R128_WaitForIdle (void)
{
  WORD starttick, endtick;

  // Insure FIFO is empty before waiting for engine idle.
  R128_WaitForFifo (64);

  // Poll GUI_ACTIVE to wait for engine idle
  // Set the appropriate timeout values.
  starttick = *((WORD *) (DOS_TICK_ADDRESS));
  endtick = starttick;
  while ((regr (GUI_STAT) & GUI_ACTIVE) != ENGINE_IDLE)
  {
        endtick = *((WORD *) (DOS_TICK_ADDRESS));
        if (abs (endtick - starttick) > IDLE_TIMEOUT)
        {
              gui_stat = regr (GUI_STAT);
              R128_ResetEngine ();
        } // if
  } // while

  // flush the pixel cache to ensure that all pending writes
  // to the frame buffer are complete.
  R128_FlushPixelCache ();

  return;

} // R128_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
  - Scaled Block Transfer

- Methods for drawing lines:
  - Drawing Patterned Lines
  - Monochrome Expansion

4.3.1 Drawing Rectangles

To draw a simple, solid-colored rectangle, the RAGE 128 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 R128_DrawRectangle (DWORD x, DWORD y, DWORD width, DWORD height, DWORD
color)
{
  DWORD temp;
  DWORD save_dp_datatype;
```

```
    R128_WaitForFifo (6);
    // Save the previous DP_DATATYPE setting
    save_dp_datatype = regr (DP_DATATYPE);
    temp = R128_GetBPPValue ();
    regw (DP_DATATYPE,(temp|BRUSH_SOLIDCOLOR|ROP3_SRCCOPY));
    regw (DP_BRUSH_FRGD_CLR, R128_GetcolorCode (color));
    regw (DST_Y_X, (y << 16) | x);
    regw (DST_WIDTH_HEIGHT, (width << 16) | height);
    // Restore the DP_DATATYPE register
    regw (DP_DATATYPE, save_dp_datatype);
    return;
  } // R128_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 RAGE 128 uses hardware support.

Source - the location where the data is taken from.

Destination - the location where the data is transferred to.

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.

- ***Scaled BitBlt***
  The source is stretched or compressed in the process of data transfer and fitted into the destination 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 RAGE 128 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 RAGE 128 will convert the white bit (1) to the foreground color of the corresponding pixel and the black bit (0) to the background color.

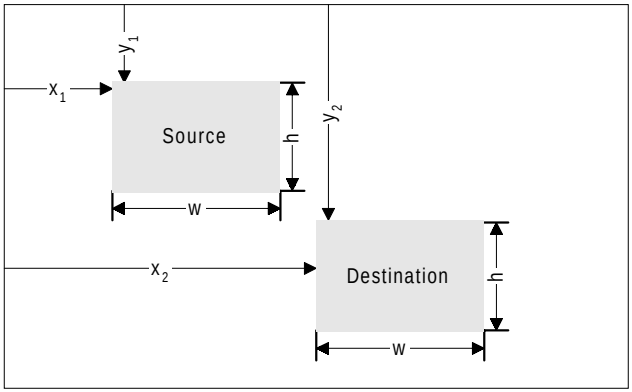

Figure 4-1. BitBlt - Bit Block Transfer Copying an Image from Source to Destination

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

```
void R128_Blt (WORD src_x, WORD src_y, WORD src_width, WORD src_height,
               WORD dst_x, WORD dst_y)
{
    DWORD temp;
    DWORD save_dp_datatype, save_dp_cntl;
    WORD bytepp;

    // save the registers we will be modifying
    R128_WaitForFifo (2);
    save_dp_datatype = regr (DP_DATATYPE);
    save_dp_cntl = regr (DP_CNTL);

    temp = R128_GetBPPValue ();

    // Set DP_DATATYPE for a SRCCOPY, current pixel depth, src=dst
    // Brush setting does not matter.
    R128_WaitForFifo (6);
    regw (DP_DATATYPE, temp | (BRUSH_SOLIDCOLOR << 16) | SRC_DSTCOLOR);

    // Set DP_MIX to SRCCOPY, rectangular source
    regw (DP_MIX, ROP3_SRCCOPY | DP_SRC_RECT);

    // Set DP_CNTL for left to right x direction, top to bottom y direction
    regw (DP_CNTL, DST_X_LEFT_TO_RIGHT | DST_Y_TOP_TO_BOTTOM);

    // Set the source and destination x and y values
    regw (SRC_Y_X, (src_y << 16) | src_x);
    regw (DST_Y_X, (dst_y << 16) | dst_x);

    // Perform the blt
    regw (DST_HEIGHT_WIDTH, (src_height << 16) | src_width);
```

```
        // restore the registers we modified
        R128_WaitForFifo (3);
        regw (SCALE_3D_CNTL, save_scale_3d_cntl);
        regw (DP_DATATYPE, save_dp_datatype);
        regw (DP_CNTL, save_dp_cntl);

        return;
    } // R128_Blt ()
    } // R128_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, destPixel = srcPixel XOR 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.
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 R128_TransparentBlt (_tbltdata TBLT)
{
  DWORD temp, save_dp_mix, save_dp_cntl, save_dp_datatype;
  WORD loop, space, num_images;

  R128_WaitForFifo (3);
  save_dp_mix = regr (DP_MIX);
  save_dp_cntl = regr (DP_CNTL);
  save_dp_datatype = regr (DP_DATATYPE);

   // Set up the drawing context
  R128_WaitForFifo (4);
  regw (DP_MIX, ROP3_SRCCOPY | DP_SRC_RECT);
  regw (SRC_SC_BOTTOM_RIGHT, 0x1FFF << 16 | 0x1FFF);
  regw (DP_CNTL, DST_X_LEFT_TO_RIGHT | DST_Y_TOP_TO_BOTTOM);
  temp = R128_GetBPPValue ();
  regw (DP_DATATYPE, temp | SRC_DSTCOLOR | BRUSH_SOLIDCOLOR);

  R128_WaitForFifo (4);
  // set up the transparency function in the color compare circuitry
  regw (CLR_CMP_CLR_DST, R128_GetcolorCode (TBLT.dst_clr));
  regw (CLR_CMP_CLR_SRC, R128_GetcolorCode (TBLT.src_clr));
  regw (CLR_CMP_MASK, 0xFFFFFFFF);
  regw (CLR_CMP_CNTL, (TBLT.cmp_src << 24) |
                        (TBLT.dst_cmp_fcn << 8) |
  TBLT.src_cmp_fcn);

  // Set up source and destination x and y values
  R128_WaitForFifo (3);
  regw (SRC_Y_X, (TBLT.src_y << 16) | TBLT.src_x);
  regw (DST_Y_X, (TBLT.dst_y << 16) | TBLT.dst_x);
  regw (DST_HEIGHT_WIDTH, (TBLT.src_height << 16) | TBLT.src_width);

      // Restore the modified registers
  R128_WaitForFifo (3);
  regw (DP_MIX, save_dp_mix);
  regw (DP_CNTL, save_dp_cntl);
  regw (DP_DATATYPE, save_dp_datatype);

  return;

} // R128_TransparentBlt ()
```

Scaled Block Transfer

This operation copies a block of pixels from the source to the destination while scaling the dimensions of the source to fit in the dimensions of the destination. In other words, the source rectangle is stretched or compressed in the process of copying according to the specified destination dimensions, and the resulting rectangle is placed at the location of the destination. *Refer to Figure 4-2. Scaled Image Transfer*.

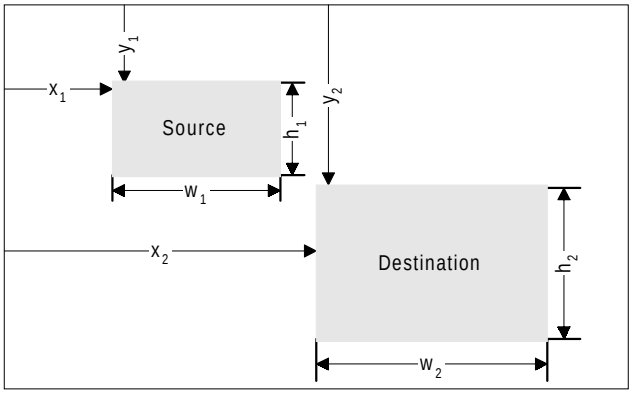

Figure 4-2. Scaled Image Transfer

In a scaled data transfer:

- Source is defined by:
  - Top-left corner coordinate is (X_1, Y_1)
  - Height and width is (h_1, W_1)

- Destination is defined by:
  - Top-left corner coordinate (X_2, Y_2)
  - Height and width (h_2, W_2)

The scaling factors between the source and destination may be defined as:

- Width (i.e. x-direction) scaling factor is $S_x = W_1/W_2$
- Height (i.e. y-direction) scaling factor is $S_y = H_1/H_2$

Since one of the three scaling parameters depends on the other two parameters, use only two parameters to specify the scaling of the source and destination.

Example Code: Scaled BitBlt operation

```
void R128_ScaleBlt (WORD src_x, WORD src_y, WORD src_width, WORD
src_height,
                    WORD dst_x, WORD dst_y, WORD dst_width, WORD dst_height)
{
    DWORD save_tex_cntl, save_scale_3d_cntl;
    DWORD temp;
    double factor = 65536.0;
    double scalex, scaley;

    // Save the registers that we intend to modify
    R128_WaitForFifo (2);
    save_tex_cntl = regr (TEX_CNTL);
    save_scale_3d_cntl = regr (SCALE_3D_CNTL);

    R128_WaitForFifo (13);

    // Set DP_MIX for SRCCOPY, using rectangular source.
    regw (DP_MIX, ROP3_SRCCOPY | DP_SRC_RECT);

    // Set SCALE_3D_DATATYPE to the current pixel depth
    temp = R128_GetBPPValue (R128_AdapterInfo.bpp);
    regw (SCALE_3D_DATATYPE, temp);

    // enable scaling with blending
    regw (SCALE_3D_CNTL, 0x00000040);

    // Clear TEX_CNTL, so all texturing functions are disabled.
    regw (TEX_CNTL, 0x00000000);

    // Disable any motion compensation functions
    regw (MC_SRC1_CNTL, 0x00000000);

    // Set up the height and width of the source data
    regw (SCALE_SRC_HEIGHT_WIDTH, (src_height << 16) | src_width);

    // set SCALE_PITCH equal to the screen pitch, as we are loading the
source
    // image in a rectangular trajectory in offscreen memory.
    regw (SCALE_PITCH, R128_AdapterInfo.pitch);

    // calculate the scaling factors for both x and y directions
    scalex = (double)src_width/(double)dst_width;
    scaley = (double)src_height/(double)dst_height;

    // Both the increment registers are 12 bit fractional, 4 bit integer
    // so we multiply the scaling factor by 65536 to convert the value
    // to this format.
    regw (SCALE_X_INC, (DWORD)(scalex * factor));
    regw (SCALE_Y_INC, (DWORD)(scaley * factor));
```

```
    // Clear out the accumulator registers
    regw (SCALE_HACC, 0x00000000);
    regw (SCALE_VACC, 0x00000000);

    // Set the dst location
    regw (SCALE_DST_Y_X, (dst_y << 16) | dst_x);

    // Perform the blt
    regw (SCALE_DST_HEIGHT_WIDTH, (dst_height << 16) | dst_width);

    // Now restore the registers we changed.
    R128_WaitForFifo (2);
    regw (TEX_CNTL, save_tex_cntl);
    regw (SCALE_3D_CNTL, save_scale_3d_cntl);

    return;
} // R128_ScaleBlt ()
```

4.3.2 Drawing Lines

Drawing lines can be accelerated by using the RAGE 128's hardware support for bresenham lines. The GUI must be programmed with the appropriate increment, decrement and error values to satisfy the bresenham algorithm as noted in the following code.

Example Code: Accelerated line drawing

```
void R128_DrawLine (WORD x1, WORD y1, WORD x2, WORD y2, DWORD color)
{
    int dx, dy;
    int small, large;
    int x_dir, y_dir, y_major;
    int err, inc, dec, temp;
    DWORD save_dp_cntl, save_dp_datatype, bppvalue;

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

    if (y1 < y2)
    {
        dy = y2 - y1;
        y_dir = 1 << 15;
```

```
    }
    else
    {
        dy = y1 - y2;
        y_dir = 0 << 15;
    } // if

    // Determine x & y min and max values; also determine y major bit.
    if (dx < dy)
    {
        small = dx;
        large = dy;
        y_major = 1 << 2;
    }
    else
    {
        small = dy;
        large = dx;
        y_major = 0 << 2;
    } // if

    // Calculate Bresenham parameters and draw line.
    err = (DWORD) (-large);
    inc = (DWORD) (2 * small);
    dec = (DWORD) (-2 * large);

    R128_WaitForFifo (11);

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

    // Set DP_DATATYPE
    bppvalue = R128_GetBPPValue (R128_AdapterInfo.bpp);
    regw (DP_DATATYPE, (bppvalue | BRUSH_SOLIDCOLOR | ROP3_SRCCOPY));

    // Draw Bresenham line.
    regw (DP_BRUSH_FRGD_CLR, R128_GetcolorCode(color));
    regw (DST_Y_X, (y1 << 16) | x1);

    // Allow setting of last pel bit and polygon outline bit for line draw-
ing.
    regw (DP_CNTL_XDIR_YDIR_YMAJOR, (y_major | y_dir | x_dir));
    regw (DST_BRES_ERR, err);
    regw (DST_BRES_INC, inc);
    regw (DST_BRES_DEC, dec);
    regw (DST_BRES_LNTH, (DWORD) (large + 1));
    regw (DP_CNTL, save_dp_cntl);
    regw (DP_DATATYPE, save_dp_datatype);

    return;

} // R128_DrawLine ()
```

Drawing Patterned Lines

The RAGE 128 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 R128_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, temp;
    DWORD save_dp_cntl, save_dp_datatype, bppvalue;

    R128_LoadPatternData (brushtype, data);

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

    if (y1 < y2)
    {
        dy = y2 - y1;
        y_dir = 1 << 15;
    }
    else
    {
```

```
            dy = y1 - y2;
            y_dir = 0 << 15;
        } // if

        // Determine x & y min and max values; also determine y major bit.
        if (dx < dy)
        {
            small = dx;
            large = dy;
            y_major = 1 << 2;
        }
        else
        {
            small = dy;
            large = dx;
            y_major = 0 << 2;
        } // if

        // Calculate Bresenham parameters and draw line.
        err = (DWORD) (-large);
        inc = (DWORD) (2 * small);
        dec = (DWORD) (-2 * large);

        R128_WaitForFifo (11);

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

        // Set DP_DATATYPE
        bppvalue = R128_GetBPPValue (R128_AdapterInfo.bpp);
        regw (DP_DATATYPE, (bppvalue | brushtype | ROP3_PATCOPY));

        // Draw Bresenham line.
        regw (DST_Y_X, (y1 << 16) | x1);

        // Allow setting of last pel bit and polygon outline bit for line draw-
    ing.
        regw (DP_CNTL_XDIR_YDIR_YMAJOR, (y_major | y_dir | x_dir));
        regw (DST_BRES_ERR, err);
        regw (DST_BRES_INC, inc);
        regw (DST_BRES_DEC, dec);
        regw (DST_BRES_LNTH, (DWORD) (large + 1));
        regw (DP_CNTL, save_dp_cntl);
        regw (DP_DATATYPE, save_dp_datatype);

        return;

    } // R128_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 RAGE 128 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;

    R128_WaitForFifo (7);

    temp = R128_GetBPPValue (R128_AdapterInfo.bpp);

    // First write GUI_MASTER_CNTL.
    regw (DP_GUI_MASTER_CNTL,
             (0 << 0) |      // Use DEFAULT_OFFSET and DEFAULT_PITCH for SRC
             (0 << 1) |      // Use DEFAULT_OFFSET and DEFAULT_PITCH for DST
              (0 << 2) |       // Use DEFAULT_SC_BOTTOM_RIGHT
             (1 << 3) |      // Use SC_TOP_LEFT and SC_BOTTOM_RIGHT for DST
              (0xC << 4) |    // Brush type ignored.
              (temp << 8) |   // Set DST_DATATYPE to the current bpp
              (0 << 12) |     // Expand to foreground and background.
             (1 << 14) |     // Consume monochrome data from LSbit to MSbit
              (0 << 15) |     // Set Conversion temp to 6500k
              (0xCC << 16) |  // Set ROP3 to SRC_COPY
              (3 << 24) |     // Source Data is from HOSTDATA registers.
              (0 << 27) |     // Clear 3D_SCALE_CNTL (Disable 3D engine)
              (1 << 28) |     // Clear CLR_CMP_CNTL (Disable Color Compare)
             (1 << 29) |     // Clear AUX_SC_CNTL (Disable Auxilary Scissors
              (1 << 30) |     // Set DP_WRITE_MASK to 0xFFFFFFFF
              (0 << 31)       // No BRUSH_X_Y required.
    );

    // Set the colors for the expanded data.
    temp = R128_GetcolorCode (pData->frgd);
    regw (DP_SRC_FRGD_CLR, temp);
    temp = R128_GetcolorCode (pData->bkgd);
    regw (DP_SRC_BKGD_CLR, temp);

    // Setup the destination trajectory.
    regw (DST_X_Y, ((pData->x << 16) | pData->y ) );
    regw (DST_WIDTH_HEIGHT, ((pData->w << 16) | pData->h) );

   // Write the data out to the HostData registers.  We write the number of
    // DWORDs less the last one, which we must write out to HOST_DATA_LAST
to
```

```
    // tell the GUI engine that the HOSTDATA operation is complete.

    for (loop = 0; loop < NumDWORDS-1; loop++ )
    {
        R128_WaitForFifo (1);
        regw (HOST_DATA0, *pSrc );
        pSrc += 1; // increment the data pointer
    }

    // Write out the final piece of data.
    R128_WaitForFifo (1);
    regw (HOST_DATA_LAST, *pSrc );
}
```

4.4 Hardware Cursor

The RAGE 128 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 RAGE 128's cursor is at the top-left corner of the display cursor.

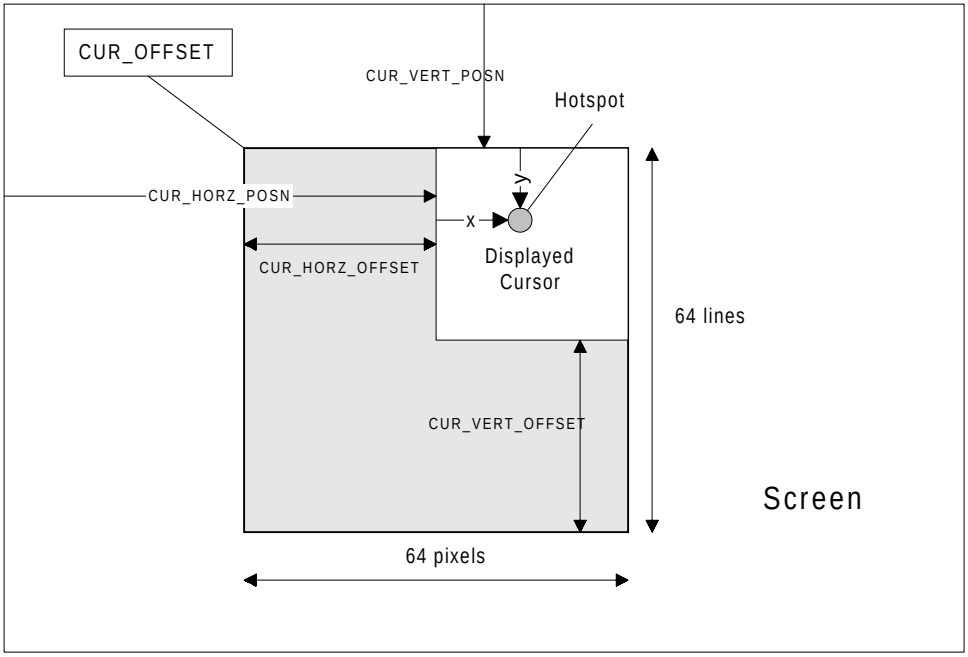

Figure 4-3. Cursor Related Parameters

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							
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, RAGE 128 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 R128_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 = R128_AdapterInfo.MEM_BASE +
```

```
CURSORDATA[cursor].cur_offset;

  // Set cursor size offsets.
  regw (CUR_HORZ_VERT_OFF, (horz_offset << 16) | vert_offset);

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

} // R128_SetHWCursor ()
```

Chapter 5
CCE Engine Initialization and Usage

5.1 Scope

The Concurrent Command Execution (CCE) Engine mode provides a simple method of programming 2D drawing operations. Instead of making register writes as you would in programmed I/O mode (PIO), simply submit a packet to the CCE ring buffer.

The CCE microengine automatically parses the packet and programs the necessary registers. This method of programming is very efficient because the CCE microengine uses the bus mastering capabilities of the RAGE 128 to send the packets from system memory to the graphics controller.

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 RAGE 128
- How the CCE microengine relates to the rest of the controller.

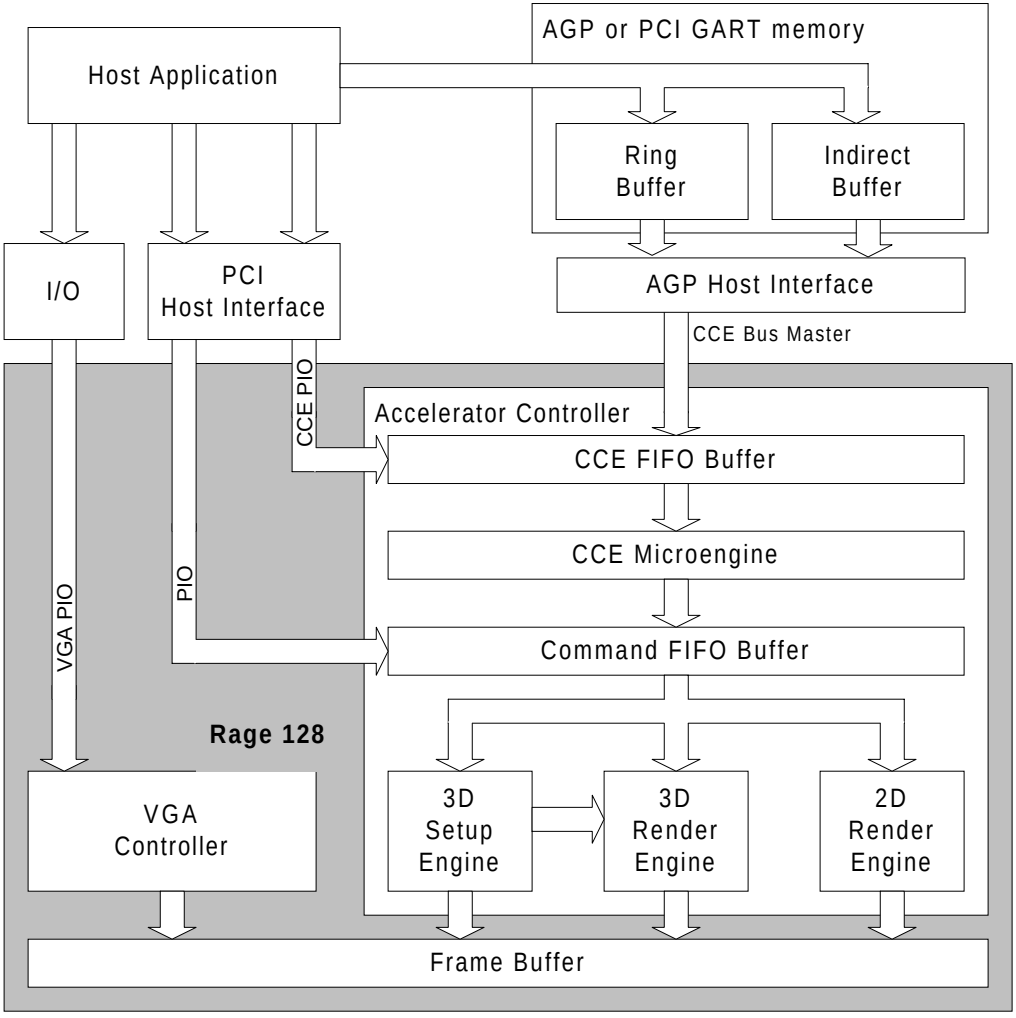

Figure 5-1. RAGE 128 Structure and Data Flow

5.2 Starting the CCE Microengine

For the purpose of this explanation, it is assumed that the RAGE 128 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:

- PM4_MICROCODE_DATAH, and
- PM4_MICROCODE_DATAL

The RAGE 128 needs to be informed of the microcode's starting address in PM4_MICROCODE_ADDR before loading begins.

Example Code: Loading the microcode into the microengine

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

void CCELoadMicrocode (void)
{
    int i;

    // Wait for engine idle before loading the microcode.

    R128_WaitForIdle ();

    // Set starting address for writing the microcode.

    regw (PM4_MICROCODE_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 (PM4_ MICROCODE_DATAH, CCE_Microcode[i][0]);
        regw (PM4_ MICROCODE_DATAL, CCE_Microcode[i][1]);
    } // for

    return;
} // CCELoadMicrocode
} // CCELoadMicrocode
```

5.2.3 Load the CCE Registers

Assuming that the size of the ring buffer is 1MB and it starts at address 0 in the virtual memory space, the free-area pointer must point to the beginning of the ring buffer and the RAGE 128 must be initiated to the CCE-bus mastering mode.

Also, the RAGE 128 must inform the host of the ring buffer's status after the transfer of the packets in the ring buffer is completed. That is, after a certain amount of data transfer, the head of packet queue in the ring buffer must be updated using the bus-mastering method.

In the following programming example, the RAGE 128 is set to update the queue head pPacketQueue after every transfer of 64 DWORDs. There are four parameters that decide the thresholds of initiating data transfer from the ring buffer, and when the pointer to the packet queue in the ring buffer is to be updated. The parameters are:

- The minimum amount of data to be transferred from the ring buffer before updating the packet queue pointer pPacketQueue at the host.
- The minimum number of DWORDs, L, in the Ring Buffer before initiating a data transfer.
- The minimum number of DWORDs, M, in the Command FIFO buffer before initiating a data transfer.
- The minimum number of DWORDs, N, in the CCE FIFO buffer before initiating a data transfer.

If we denote the actual numbers of DWORDs in the ring buffer, command FIFO buffer, and CCE FIFO buffer respectively by l, m and n, the condition for initiating a data transfer is as follows:

- l > L, or
- m < or = M, or
- n < or = N.

Now let L = 16, M = 8, and N = 8, the following example will set up the microengine according to the specification.

Example Code: Initializing the microengine

```
// Note that the ring buffer size must be power-of-2, min size 2 DWORDs

#define RING_SIZE       0x00040000  // 1MB ring buffer (256k DWORDs)
#define RING_SIZE_LOG2  17          // log2 (RING_SIZE) - 1

#define CCE_WATERMARK_L 16
#define CCE_WATERMARK_M 8
#define CCE_WATERMARK_N 8
#define CCE_WATERMARK_K 128

typedef struct tagRBINFO {
    volatile DWORD *ReadIndexPtr;       // Current Read pointer index
    DWORD ReadPtrPhysical;              // Physical address of read pointer
    DWORD WriteIndex;                   // Current write pointer index
    DWORD *LinearPtr;                   // Virtual address of ring buffer
    DWORD Size;                         // Size of ring buffer in DWORDs
} RBINFO;

#define RING_SIZE   0x00000800
DWORD dwRingBuf[RING_SIZE];
struct {
    DWORD *pPacketQueue,*pPacketQrec, *pFreeArea;
    DWORD dwRingSize, dwSpaceAvail, *pRingStart;
} Svr = {dwRingBuf, dwRingBuf, dwRingBuf, RING_SIZE, RING_SIZE, dwRingBuf};

void InitMicroEngine (void)
{
    // Set the start address of ring buffer.

    regw (PM4_BUFFER_OFFSET, dwRingBuf);

    // Set the pointer of the area for fill packets.

    regw (PM4_BUFFER_DL_WPTR, svr.pFreeArea);

    // Set up the threshholds of initiating a data transfer
    // from the ring buffer to the PM4 FIFO buffer.

    regw (PM4_BUFFER_WM_CNTL, 0x02020204);

    // Set Rage 128 to the CCE bus mastering mode with full use of the CCE
```

```
    // FIFO buffer (192 DWORDs), and set the size of ring buffer to 8 K.

    regw (PM4_BUFFER_CNTL, 0x2000000A);

    // Set the pointer to the head of the packet queue in the ring buffer

    regw (PM4_BUFFER_DL_RPTR_ADDR, &svr.pPacketQueue);
} // InitMicroEngine

int R128_CCEInit (int index)
{
    DWORD cce_buf_size;

    // Load CCE Microcode

    CCELoadMicrocode ();

    // Validate CCEmode and set up necessary parameters

    if ((index < CCE_MODE_192PIO) ||
        (index > CCE_MODE_64PIO_64VERPIO_64INDPIO))
    {
        return (CCE_FAIL_INVALID_CCE_MODE);
    } // if

    // Perform a soft reset of the engine

    R128_ResetEngine ();

    CCEFifoSize = CCEmode[index].fifosize;
    CCEBMFlag = CCEmode[index].busmaster;
    if (CCEBMFlag)
    {
        R128_CCESubmitPackets = CCESubmitPacketsBM;
        if (CCESetupBusMaster ())
        {
            return (CCE_FAIL_BUS_MASTER_INIT);
        } // if
        cce_buf_size = RING_SIZE_LOG2;
        bm_save_state = regr (BUS_CNTL);
        regw (BUS_CNTL, (bm_save_state & ~BUS_MASTER_DIS));
    }
    else
    {
        R128_CCESubmitPackets = CCESubmitPacketsPIO;
        cce_buf_size = 0;
    } // if

    // Set the Rage 128 to requested CCE mode.
    CCERequestedMode = CCEmode[index].pm4buffermode + cce_buf_size;
    regw (PM4_BUFFER_CNTL, CCERequestedMode);

    // Set the CCE to free running mode
```

```
    regw (PM4_MICRO_CNTL, PM4_MICRO_FREERUN);

    return (CCE_SUCCESS);
} // R128_CCEInit
int CCESetupBusMaster (void)
{
    _AGP_INFO *AGP_Info;
    DWORD ring_buf_offset, read_ptr_offset;

    // For the sake of simplicity, put the ring buffer at the start of AGP
    // space, factoring in alignment restrictions.

    ring_buf_offset = 0;

    // Align the offset to a 128-byte boundary. Strictly speaking, since an
    // offset of zero was chosen, the following step is unnecessary, but it
    // is good form to perform this step in case the ring buffer needs to be
    // elsewhere.

    ring_buf_offset = align (ring_buf_offset, 128);
    read_ptr_offset = align ((DWORD) readbuf, 32);
    RingBuf.ReadIndexPtr = (DWORD *) read_ptr_offset;
    R128_Delay (1);
    RingBuf.ReadPtrPhysical = GetPhysical (read_ptr_offset);
    RingBuf.Size = RING_SIZE;
    RingBuf.WriteIndex = 0;

    if (R128_InitAGP (APERTURE_SIZE_4MB))
    {
        CCEAGPFlag = TRUE;
        GetAGPINFO (&AGP_Info);
        RingBuf.LinearPtr = (DWORD *) (AGP_Info->LogicalAddress +
ring_buf_offset);
        regw (PCI_GART_PAGE, PCI_GART_DIS);
    }
    else
    {
        // No AGP available, use PCI GART mapping instead.

        CCEAGPFlag = FALSE;
        if (!(PCIGartInfo = SetupPCIGARTTable (APERTURE_SIZE_4MB)))
        {
            // If even a PCI GART table is not available, then bus mastering
             // is not possible.

             return (CCE_FAIL_BUS_MASTER_INIT);
        } // if

        RingBuf.LinearPtr = PCIGartInfo->pointer +
                            (ring_buf_offset/sizeof (DWORD));

        // Write the GART page address.  Since this address is 4KB
        // aligned, bit 0 is cleared. Hence, GART will be enabled.
```

```
            regw (PCI_GART_PAGE, PCIGartInfo->paddress);
        } // if

        // Set the start offset of the ring buffer.

        regw (PM4_BUFFER_OFFSET, (ring_buf_offset + 0x02000000));

        regw (PM4_BUFFER_DL_WPTR, RingBuf.WriteIndex);
        regw (PM4_BUFFER_DL_RPTR, RingBuf.WriteIndex);

        // Put the physical address of read pointer into PM4_BUFFER_DL_RPTR_ADDR

        regw (PM4_BUFFER_DL_RPTR_ADDR, RingBuf.ReadPtrPhysical);

        // Set watermarks for CCE

        regw (PM4_BUFFER_WM_CNTL, (CCE_WATERMARK_K/64) << 24 |
                                  (CCE_WATERMARK_N/4)  << 16 |
                                  (CCE_WATERMARK_M/4)  <<  8 |
                                  CCE_WATERMARK_L/4);

        return (CCE_SUCCESS);
    } // CCESetupBusMaster
```

5.2.4 Cautions When Programming RAGE 128 in CCE Mode

- All packets must be checked for proper formatting prior to submission to the server. Incorrectly-formatted packets will cause the RAGE 128 to hang.

5.3 Ring Buffer Management

5.3.1 The Ring Buffer Concept

When operating in CCE mode, the RAGE 128 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 RAGE 128 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, *refer to Section 2.6.6*.

The RAGE 128 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 RAGE 128 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 RAGE 128's on-chip CCE FIFO buffer, where they are processed by the microengine in the order they are received. The microengine places it's output into the command FIFO as register-datum pairs. When the host reaches the end of the block, it starts copying at the top again. The following figure shows a conceptual representation of the ring buffer.

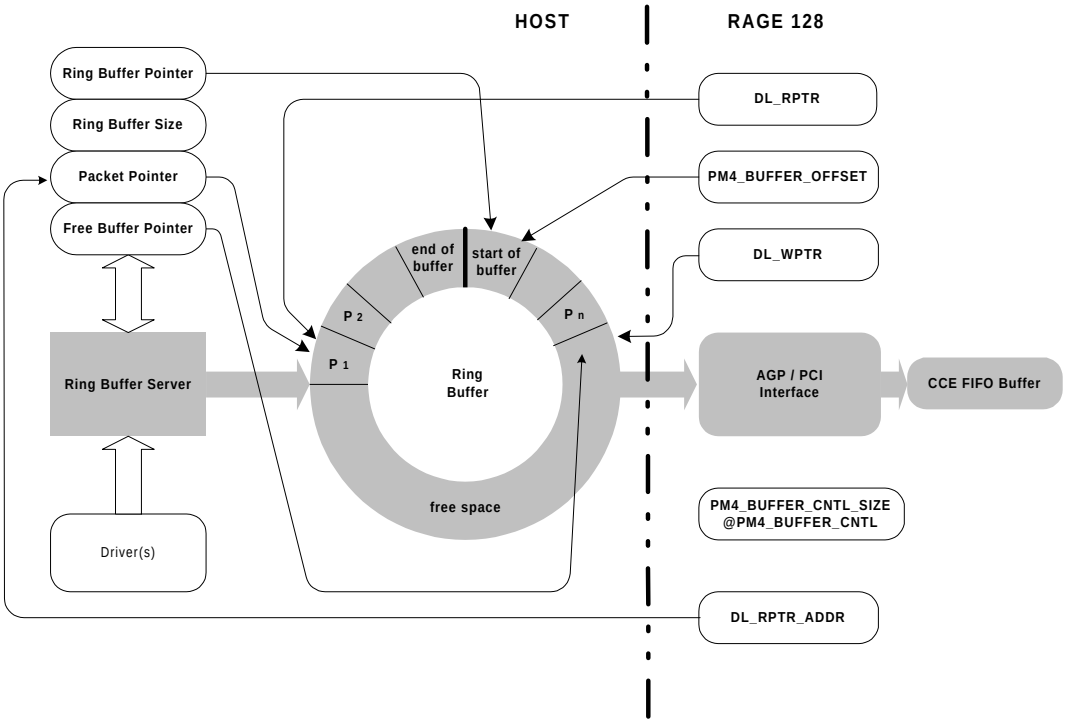

Figure 5-2. Ring Buffer and its Control Structure

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 Pn. The start of the queue, P1, is pointed to by both a packet pointer maintained by the application, and the RAGE 128's **PM4_BUFFER_DL_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 Rage 128's **PM4_BUFFER_DL_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 RAGE 128 is faster than the speed of data transfer from the ring buffer to the RAGE 128. 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 RAGE 128. 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 RAGE 128 free area pointer by writing the new free area address to the **PM4_BUFFER_DL_WPTR** register.

On the RAGE 128 side, packets are read one at a time from the head of the packet queue pointed to by the **PM4_BUFFER_DL_RPTR** register, and sent to the CCE FIFO buffer. The **PM4_BUFFER_DL_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 RAGE 128 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 RAGE 128's **PM4_BUFFER_DL_RPTR_ADDR** register during CCE initialization. Note that the AGP interface stops data transfer once pointer **PM4_BUFFER_DL_RPTR** meets **PM4_BUFFER_DL_WPTR**.

5.3.2 Ring Buffer 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 server. Under this scheme, all clients submit packets to the server, and the server mediates and schedules delivery of the packets to the ring buffer.

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 RAGE 128 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.

Example Code: Ring buffer management

```
#define FAIL    0
#define SUCCESS 1
DWORD dwRingBuf[RING_SIZE];
struct {
    DWORD *pPacketQueue,*pPacketQrec, *pFreeArea;
    DWORD dwRingSize, dwSpaceAvail, *pRingStart;
} Svr = {dwRingBuf, dwRingBuf, dwRingBuf, RING_SIZE, RING_SIZE, dwRingBuf};

WORD SubmitPackets (DWORD *ClientBuf, DWORD dwDataSize)
{
    long n1, n2;
    int i,j;
```

```
        // update available space to synchronize the record with Rage 128

        if (Srv.pPacketQueue > Srv.pPacketQrec)
        {
            Srv.dwSpaceAvail += Srv.pPacketQueue - Srv.pPacketQrec;
        } // if
        if (Srv.pPacketQueue < Srv.pPacketQrec)
        {
            Srv.dwSpaceAvail += Srv.pPacketQueue - Srv.pPacketQrec + RING_SIZE;
        } // if
        Srv.pPacketQrec = Srv.pPacketQueue;
        if (Srv.dwSpaceAvail >= dwDataSize)
        {
            if (Srv.pFreeArea + dwDataSize <= Srv.pRingStart + RING_SIZE)
            {
                for (i = 0; i < dwDataSize; i++)
                {
                    Srv.pFreeArea[i] = ClientBuf[i];
                } // for
                Srv.pFreeArea += dwDataSize;
            }
            else
            {
                n1 = Srv.pRingStart + RING_SIZE - Srv.pFreeArea;
                n2 = dwDataSize - n1;
                for (i = 0; i < n1; i++)
                {
                    Srv.pFreeArea[i] = ClientBuf[i];
                } // for
                Srv.pFreeArea = Srv.pRingStart;
                for (j = 0; i < n2; i++, j++)
                {
                    Srv.pFreeArea[j] = ClientBuf[i];
                } // for
                Srv.pFreeArea += n2;
            } // if
            WriteReg (PM4_BUFFER_DL_WPTR, svr.pFreeArea);
            Srv.dwSpaceAvail -= dwDataSize;
            return SUCCESS;
        }
        else
        {
            return FAIL;
        } // if
    } // SubmitPackets
```

Example Code: Submitting packets using programmed I/O (PIO) mode

```
    int CCESubmitPacketsPIO (DWORD *ClientBuf, DWORD DataSize)
    {
        // Consume entries in the buffer two DWORDs at a time, splitting up the
        // writes to the even and odd registers.
```

```
    while (DataSize > 1)
    {
        CCEWaitForFifo (2);
        regw (PM4_FIFO_DATA_EVEN, *ClientBuf++);
        regw (PM4_FIFO_DATA_ODD, *ClientBuf++);

        DataSize -= 2;
    } // while

    // At this point, DataSize should be either 0 or 1, handle odd packet
    // accordingly.

    if (DataSize & 1)
    {
        CCEWaitForFifo (2);
        regw (PM4_FIFO_DATA_EVEN, *ClientBuf);  // Write final packet
        regw (PM4_FIFO_DATA_ODD, CCE_PACKET2);  // Write dummy packet
    } // to align if

    // N.B.  A more sophisticated packet submission algorithm might try to
    // reduce the number of times that CCEWaitForFifo () is called and still
    // handle packets that are larger than the maximum CCE FIFO size.  A
    // somewhat inefficient approach (waiting for 2 free entries each time
    // through the loop) is used above since it simplifies the example and
    // can handle arbitrary sized buffers.

    return (CCE_SUCCESS);
} // CCESubmitPacketsPIO
```

Example Code: Submitting packets with Bus Mastering

```
int CCESubmitPacketsBM (DWORD *ClientBuf, DWORD DataSize)
{
    DWORD *tptr;

    // We shall arbitrarily fail if the incoming packet is bigger than our
    // ring buffer.  A better algorithm would break up the incoming packet
    // into small enough chunks to feed to the buffer.

    if (DataSize >= RingBuf.Size)
    {
        return (CCE_FAIL_BAD_PACKET);
    } // if

    tptr = RingBuf.LinearPtr + RingBuf.WriteIndex;
    while (DataSize > 0)
    {
        RingBuf.WriteIndex += 1;
        *tptr++ = *ClientBuf++;
        if (RingBuf.WriteIndex >= RingBuf.Size)
        {
            RingBuf.WriteIndex = 0;
            tptr = RingBuf.LinearPtr;
        } // if
```

```
            while (RingBuf.WriteIndex == *(RingBuf.ReadIndexPtr))
            {
                // Some form of timeout checking should be done here in
                // case the read pointer gets stuck due to an engine panic.
            } // while
            DataSize -= 1;
        } // while

        // Update pointer.

        regw (PM4_BUFFER_DL_WPTR, RingBuf.WriteIndex);
        return (CCE_SUCCESS);
    } // CCESubmitPacketsBM
```

Example Code: Shutting down the microengine

```
    void R128_CCEEnd (int waitmode)
    {
        if (CCEBMFlag)
        {
            // Signal CCE that we are done submitting bus-mastered packets

            regw (PM4_BUFFER_DL_WPTR, RingBuf.WriteIndex | PM4_BUFFER_DL_DONE);
        } // if

        if (waitmode == CCE_END_WAIT)
        {
            // Wait for engine idle before ending.  It does not matter if the
            // engine fails to idle as we will reset it shortly.

            CCEWaitForIdle ();
        } // if

        // Stop the CCE microengine by setting it to single-stepping mode

        regw (PM4_MICRO_CNTL, 0x00000000);

        // Perform a soft reset of the engine

        R128_ResetEngine ();

        // Set the Rage 128 to standard PIO mode.

        regw (PM4_BUFFER_CNTL, PM4_BUFFER_CNTL_NONPM4);

        if (CCEBMFlag)
        {
            regw (BUS_CNTL, bm_save_state);

            if (CCEAGPFlag)
            {
                // Shut down AGP

                R128_EndAGP ();
```

```
            }
            else
            {
                DPMI_freedosmem (PCIGartInfo->handle);
            } // if
        } // if

        return;
    } // R128_CCEEnd
```

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 5-16*.

The benefit of the indirect buffer is that unlike the ring buffer, it is not continuously being overwritten as a consequence of circular wrapping. This allows relatively static and frequently used packets to be written once and referenced multiple times. Only the operating parameters need to be changed for each instance of use (e.g., the top, left, width, and height parameters of a BITBLT type-3 packet). In contrast, the same packet would have to be continuously copied into the ring buffer because the buffer's contents are continuously overwritten.

The indirect buffer should be 4K page aligned.

The packet byte offset from the base of the indirect buffer is specified in the **PM4_IW_INDOFF** register. The size in even number of DWORDs is specified in the **PM4_IW_INDSIZE** register. If a packet's size is an odd count of DWORDs, it should be padded with a single type-2 NOP packet. Writing **PM4_IW_INDSIZE** initiates the packet transfer.

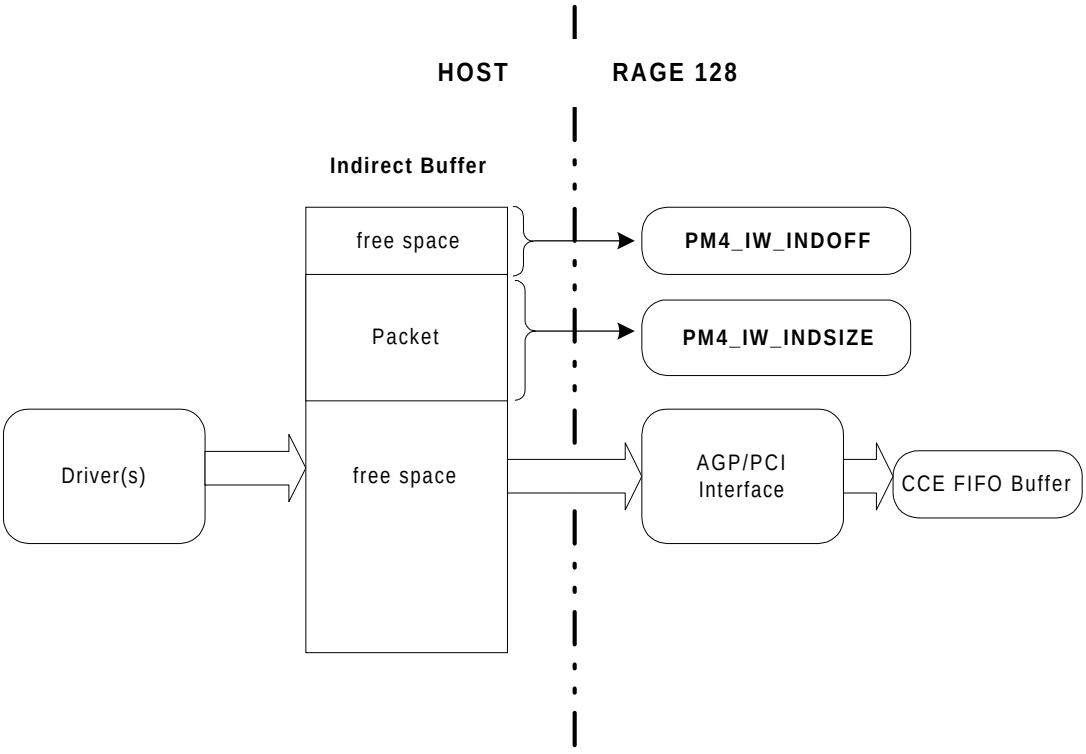

Figure 5-3. The Indirect Buffer

The most efficient programming model for the RAGE 128 is to use both the ring buffer and the indirect buffer. The ring buffer enables concurrency and command streaming, whereas the indirect buffer reduces copying overhead for commonly used packets. The packet transfers out of the indirect buffer may be streamed by writing **PM4_IW_INDOFF** and **PM4_IW_INDSIZE** 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.

Chapter 6
CCE Packets

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. CCE packets are used to draw three-dimensional (3D) images, such as:

- Shaded or textured mapped points.
- Shaded or textured mapped line lists and strips.
- Shaded or textured mapped triangle lists, strips, and fans.

CCE packets are used to control several features associated with 3D rendering such as:

- Texture map states
- Z buffering
- Stencil buffering
- Alpha blending
- Alpha testing
- Fog blending
- Culling
- Dithering

Programming examples will demonstrate how to use the CCE packets to draw 2D and 3D 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.

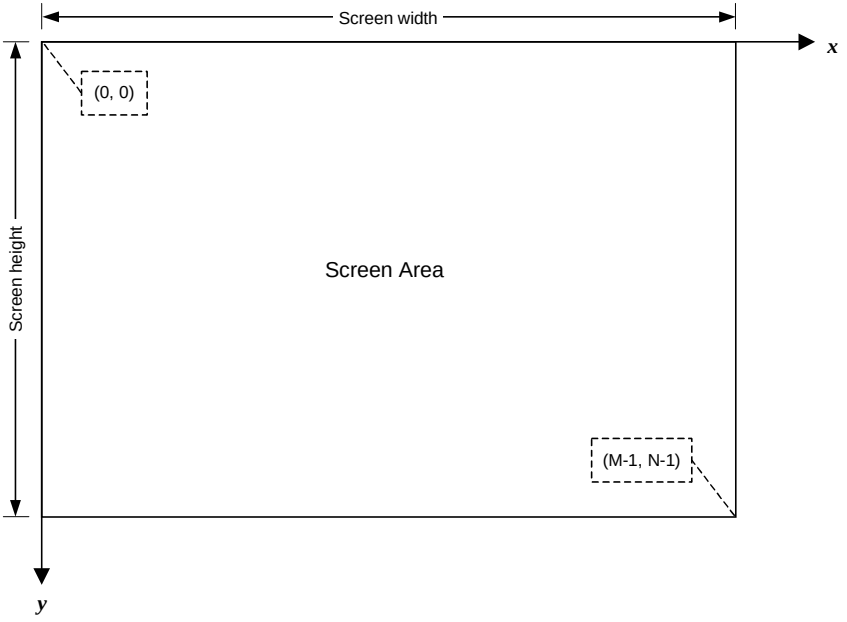

Figure 6-1. 2D Coordinate System

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 RAGE 128 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 RAGE 128 supports all 256 ROP3 raster operations.

As rendering operations occur in a specific display mode, the program must specify the following parameters to the RAGE 128 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, 24 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

RAGE 128 provides hardware assistance for drawing the following:

- Polylines.
- Polyscanlines.
- Rectangles.

The RAGE 128 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.

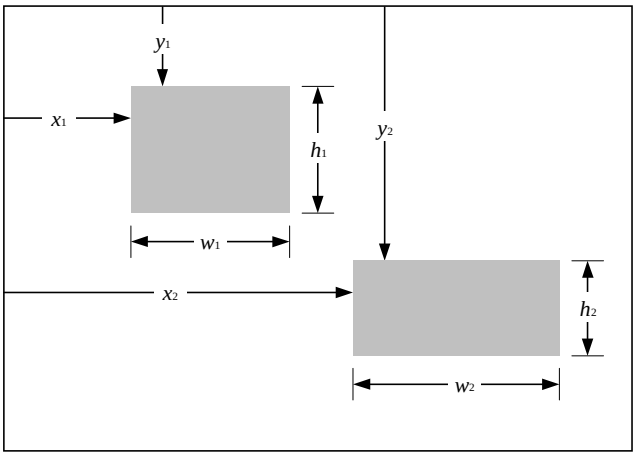

Figure 6-2. Rectangles

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

The destinational-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
#define DST_CLIPPING    0x00000008  // Clip the destination
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define SOLID_PEN       13          // Brush type selected
#define BLUE_COLOR      0x1F        // Colour in aRGB format
#define ROP_PAT_CPY     0xF0        // Copy the brush pattern to the dest
#define SRC_TYPE_3      3          // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
#define CLIP_TOP        10          // Clipping rectangle parameters
#define CLIP_LEFT       20
#define CLIP_BOTTOM     300
#define CLIP_RIGHT      200
DWORD dwBuf[20];
struct {
```

```
    WORD wLeft, wTop, wRight, wBottom;
} Rect[2]= {{20, 10, 80, 80}, {120, 10, 200, 160}};

int i = 0, j;

    // Compose the header

    dwBuf[i++] = CCE_PACKET3_CNTL_PAINT;

    // Compose GUI_CONTROL

    dwBuf[i++] = DST_CLIPPING | SOLID_PEN << 4 | PIXEL_TYPE_aRGB << 8 |
                 SRC_TYPE_3 << 12| DRAW_LEFT2RIGHT << 14 | ROP_PAT_CPY << 16;

    // Data of the clipping rectangle.

    dwBuf[i++] = CLIP_LEFT | CLIP_TOP << 16;
    dwBuf[i++] = CLIP_RIGHT | CLIP_BOTTOM << 16;

    // Colour used to draw the polyline.

    dwBuf[i++] = BLUE_COLOR;

    // Fill rectangles' data

    for (j = 0; j < 2; j++)
    {
        dwBuf[i++] = rect[j].wLeft + (rect[j].wTop << 16);
        dwBuf[i++] = rect[j].wRight + (rect[j].wBottom << 16);
    } // for
    dwBuf[0] |= (i - 2) << 16;  // Fill the header with packet size

    // Submit the packet to the ring buffer.

    SubmitPackets (dwBuf, i);
```

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 $p_1, p_2, \ldots, p_5$, 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 RAGE 128 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 draw 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 RAGE 128 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 Black.
- 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 specified by its top-left corner (10,10) and bottom-right corner (600,400).
- 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, …, p_5.

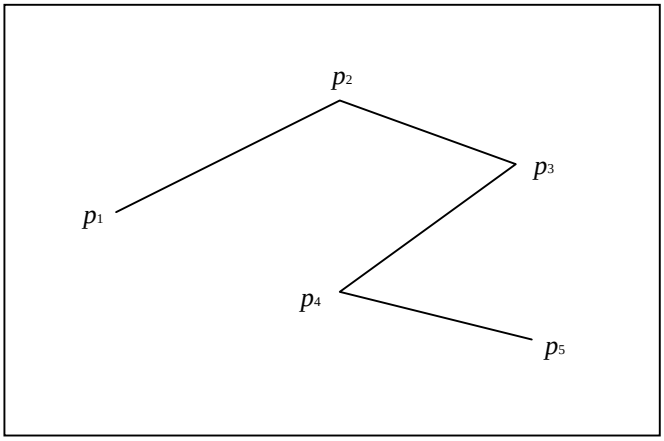

Figure 6-3. Polyline

Example Code: Drawing a polyline

```
#define CCE_PACKET3_CNTL_POLYLINE   0xC0009500
#define DST_CLIPPING    0x00000008  // Clip the destination
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define SOLID_PEN       13          // Brush type selected
#define BLACK_COLOR     0           // Colour in aRGB format
#define CLIP_TOP        10          // Clipping rectangle parameters
#define CLIP_LEFT       10
#define CLIP_BOTTOM     400
#define CLIP_RIGHT      600
#define ROP_PAT_CPY     0xF0        // Copy brush pattern to destination
#define SRC_TYPE_3      3          // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right

extern WORD SubmitPackets (DWORD *ClientBuf, DWORD dwDataSize);

DWORD dwBuf[20];
struct {
    int x, y;
} points[5] = {{10, 25}, {45, 57}, {156, 200}, {87, 260}, {160, 300}};

int i=0, j;

    // Compose the header
    dwBuf[i++]= CCE_PACKET3_CNTL_POLYLINE;
    // Compose GUI_CONTROL
    dwBuf[i++]=DST_CLIPPING |SOLID_PEN << 4 | PIXEL_TYPE_aRGB << 8 |
               SRC_TYPE_3 << 12| DRAW_LEFT2RIGHT <<14 | ROP_PAT_CPY <<16;
    // Data of the clipping rectangle.
```

```
dwBuf[i++]= CLIP_LEFT | CLIP_TOP << 16;
dwBuf[i++]= CLIP_RIGHT | CLIP_BOTTOM << 16;
// Colour used to draw the polyline.
dwBuf[i++]= BLACK_COLOR;
// Fill points' data
for (j = 0; j < 5; j++)
{
    dwBuf[i++]= points[j].x + (points[j].y << 16);
} // for
// Fill packet size into the packet header.
dwBuf[0] |= (i - 2) << 16;
// Submit the packet to the ring buffer.
SubmitPackets (dwBuf, i);
SubmitPackets (dwBuf, i);
```

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, …, 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).

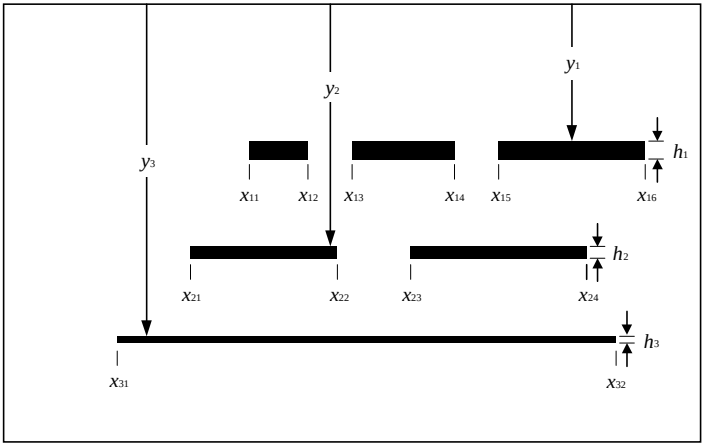

Figure 6-4. Polyscanlines

Assuming that the polyscanlines in *Figure 6-4.* are drawn completely without being clipped, in Blue 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 following parameters:
  - $y_1 = 80$, $h_1 = 3$, $x_{11} = 100$, $x_{12} = 150$, $x_{13} = 170$, $x_{14} = 230$, $x_{15} = 250$, $x_{16} = 300$
  - $y_2 = 100$, $h_2 = 2$, $x_{21} = 80$, $x_{22} = 160$, $x_{23} = 200$, $x_{24} = 290$

- $y_3 = 120$, $h_3 = 1$, $x_{31} = 60$, $x_{32} = 330$

Example Code: Drawing polyscanlines

```
#define CCE_PACKET3_CNTL_POLYSCANLINES  0xC0009800
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define SOLID_PEN       13          // Brush type selected
#define BLUE_COLOR      0x1F        // Colour in aRGB format
#define ROP_PAT_CPY     0xF0        // Copy brush pattern to destination
#define SRC_TYPE_3      3          // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
DWORD dwBuf[20];

WORD line1[] = {100, 150, 170, 230, 250, 300};
WORD line2[] = {80, 160, 200, 290};
WORD line3[] = {60, 330};
struct _polyscnline{
    DWORD numSegments;
    WORD wTop,wHeight;
    WORD *line;
} polyscnline[3] = {{3,80,3,line1}, {2,100,2,line2}, {1,120,1,line3}};


int i = 0, j, k;

    // Compose the header
    dwBuf[i++] = CCE_PACKET3_CNTL_POLYSCANLINES;
    // Compose GUI_CONTROL
    dwBuf[i++] = SOLID_PEN << 4 | PIXEL_TYPE_aRGB << 8 |
                 SRC_TYPE_3 << 12| DRAW_LEFT2RIGHT <<14 | ROP_PAT_CPY <<16;
    // Colour used to draw the polyscanlines.
    dwBuf[i++] = BLUE_COLOR;
    dwBuf[i++] = 3;             // Number of subpackets

    for (j = 0; j < 3; j++)
    {
        // Fill subpacket
        dwBuf[i++] = polyscnline[j].numSegments;
        dwBuf[i++] = polyscnline[j].wTop + (polyscnline[j].wHeight << 16);
        for (k = 0; k < 2*polyscnline[j].numSegments; k += 2)
        {
            dwBuf[i++] = polyscnline[j].line[k] +
                         (polyscnline[j].line[k+1] << 16);
        } // for
    } // for
    dwBuf[0] = CCE_PACKET3_CNTL_POLYSCANLINES | ((i-2) << 16);

    // Submit the packet to the ring buffer.
```

```
SubmitPackets (dwBuf, i);
SubmitPackets (dwBuf, i);
```

6.4 Block Transfers

The RAGE 128 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 three 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.
- ***Scaled BitBlt*** where the source is stretched or compressed in the process of data transfer and fitted into the destination dimensions.
- ***Transparent Scaled BitBlt*** is a transfer that is similar to Scaled BitBlt except that it makes the background image at the destination shown through the image copied from the source as if the source image is transparent.

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 RAGE 128 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. RAGE 128 will convert the white bit ('1') to the foreground color of the corresponding pixel and the black bit ('0') to the background color.

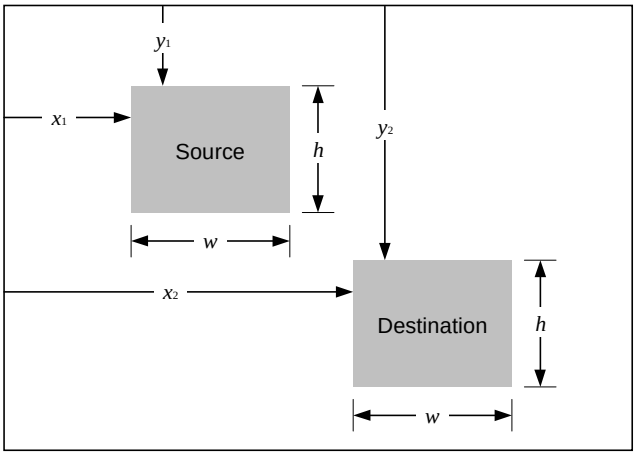

Figure 6-5. Copy an Image from Source to Destination

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 BLTBLT 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_MULTI   0xC0009B00
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define NO_BRUSH        15          // Brush type selected
#define LD_FRM_VRAM     2           // Source is loaded from the VRAM
#define ROP_SRCCOPY     0xCC        // Copy the source to the destination
#define SRC_TYPE_3      3          // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right

DWORD dwBuf[20];

int x1 = 20, y1 = 40, h = 50, w = 80, x2 = 120, y2 = 200;
int I = 0;

    // Compose the header
    dwBuf[i++] = CCE_PACKET3_CNTL_BITBLT_MULTI;
    // Compose GUI_CONTROL
    dwBuf[i++] = NO_BRUSH << 4 | PIXEL_TYPE_aRGB << 8 |
                 SRC_TYPE_3 << 12| DRAW_LEFT2RIGHT << 14 | ROP_SRCCOPY << 16
|
                   LD_FRM_VRAM << 24;
    // Fill rectangles' data
    dwBuf[i++] = y1 | (x1 << 16);   // Source location
    dwBuf[i++] = y2 | (x2 << 16);   // destination location
    dwBuf[i++] = h | (w << 16);     // dimensions of copied area
    // Submit the packet to the ring buffer.
    dwBuf[0] |= (i - 2) << 16;      // Add packet size to header
    SubmitPackets (dwBuf, i);
```

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).

Table 6-1 Source Comparator (Continued)

Decision Code	Description
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, destPixel = srcPixel XOR 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.

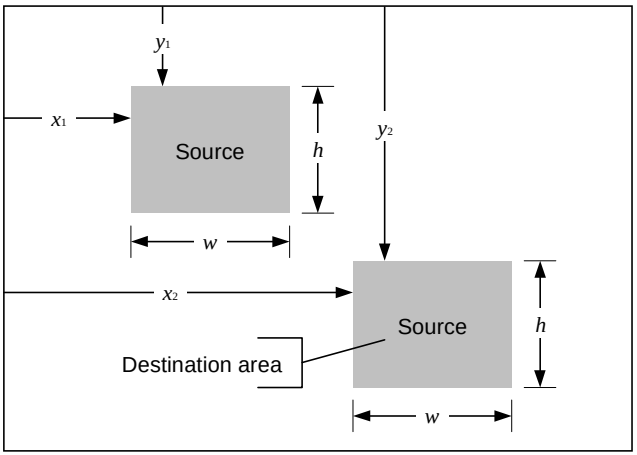

Figure 6-6. Transparent Bit-Block Transfer

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**). Supply data for fields **CLR_CMP_CNTL**, **SRC_REF_CLR**, and **DST_REF_CLR**. 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_SCALING  0xC0009700
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define NO_BRUSH        15          // Brush type selected
#define LD_FRM_VRAM     2           // Source is loaded from the VRAM
#define ROP_SRCCOPY     0xCC        // Copy the source to the destination
```

```
#define SRC_TYPE_3      3           // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
#define PACKET_SIZE     8           // Packet size including header
#define SRC_REF_COLOR   0x1F        // source reference
#define CLR_CMP_SRC     4           // Pixels equal to reference get to dest
#define CMP_ENABLE      1           // Enable source comparator
DWORD dwBuf[20];

int x1 = 20, y1 = 40, h2 = 100, w2 = 80, x2 = 120, y2 = 200;
int sx = 0x0C00;    // Representation of 3/4 (see Appendix B for details)
int sy = 0x0800;    // Representation of 1/2 (see Appendix B for details)
int i = 0;

    // Compose the header
    dwBuf[i++] = CCE_PACKET3_CNTL_TRANS_SCALING;
    // Compose GUI_CONTROL
    dwBuf[i++] = NO_BRUSH << 4 | PIXEL_TYPE_aRGB << 8 |
                 SRC_TYPE_3 << 12 | DRAW_LEFT2RIGHT << 14 |
                 ROP_SRCCOPY << 16 | LD_FRM_VRAM << 24 ;
    dwBuf[i++] = CLR_CMP_SRC | CMP_ENABLE << 24;
    dwBuf[i++] = SRC_REF_COLOR;
    dwBuf[i++] = 0;                 // dummy destination reference

    // Fill rectangles' data
    dwBuf[i++] = x1 | (y1 << 16);   // Source location
    dwBuf[i++] = x2 | (y2 << 16);   // destination location
    dwBuf[i++] = w2 | (h2 << 16);   // dimensions of destination area
    dwBuf[0] | = (i-2) << 16;       // Add packet size to header
    // Submit the packet to the ring buffer.
    SubmitPackets (dwBuf, i);
```

6.4.3 Scaled Block Transfer

The Scaled Block Transfer is a way to copy a block of pixels from the source to the destination while scaling the dimensions of the source to fit in the dimensions of the destination. In other words, the source rectangle is stretched or compressed in the process of copying according to the specified destination dimensions, and the resulting rectangle is placed at the location of the destination.

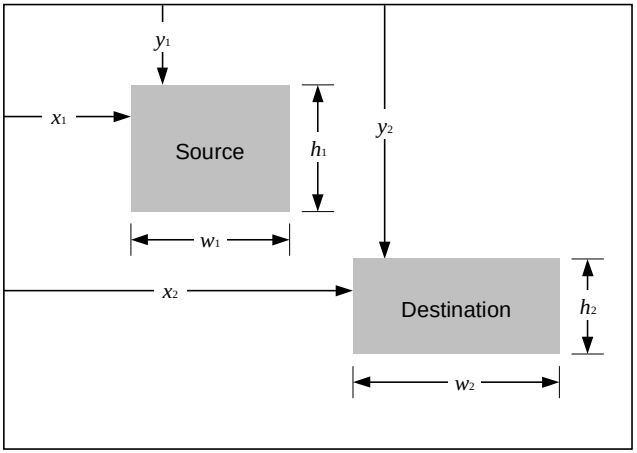

Figure 6-7. Scaled Image Transfer

In a scaled data transfer, the source is specified by its top-left corner coordinate (x_1, y_1) and height and width (h_1, w_1). The destination is specified by (x_2, y_2) and (h_2, w_2).

The scaling factors between the source and destination may be defined as:

- Sx = w1/w2
  - s_x is the factor in the *x*-direction

- Sy = h1/h2
  - s_y is the factor in the *y*-direction.

One of parameters w_1, w_2 and s_x is dependent on the other two. Use two of them to specify the horizontal dimensions of the source and destination. The same method can be used to specify the vertical dimensions of the source and destination.

The following example uses (x_1, y_1), (x_2, y_2), (s_x, w_2) and (s_y, h_2) to specify the locations and dimensions of the source and destination. Assuming $x_1 = 20$, $y_1 = 40$, $h_1 = 50$, $w_1 = 60$, $x_2 = 120$, $y_2 = 200$, $h_2 = 100$, $w_2 = 80$, then $s_x = ¾$ and $s_y = ½$.

It is obvious that packet SCALE is suitable for this operation. The setup parameters are the same as those given in the example of the previous section.

There are two types of scaled block transfer:

- Mapping a texture (bitmap) onto a screen area.
- Transfer a block of data from one screen area to anther as shown in *Figure 6-6.*

Both types of transfer require the user to specify for the packet the source location in terms of memory offset, and the vertical advance step of the source in terms of pitch. In the case of texture mapping, the memory offset points to the texture location in frame buffer and the pitch is set to the pitch of the texture. In the case of screen-to-screen transfer, convert the x- and y-coordinates of the source image to the memory offset to the screen origin and to indicate the pitch of the screen.

It is assumed that the display mode is set to 800x600 for screen resolution and 16 BPP for color. For this display mode:

- Each pixel is represented by 2 bytes.
- Each scanline of the screen is represented by 2x800 bytes.
- Screen pitch is 100.

Assuming the screen origin is at address 0 of the frame buffer, the memory offset of the source is determined by:

- offset = 1600y1 + 2x1

Example Code: Copying an image from the source to the destination with scaling

```
#define CCE_PACKET3_CNTL_SCALING    0xC0009600
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define NO_BRUSH        15          // Brush type selected
#define LD_FRM_VRAM     2           // Source is loaded from the VRAM
#define ROP_SRCCOPY     0x0CC       // Copy the source to the destination
#define SRC_TYPE_3      3          // No difference between source and dest
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
#define SCREEN_PITCH    100

DWORD dwBuf[20];

int x1 = 20, y1 = 40, h2 = 100, w2 = 80, x2 = 120, y2 = 200;
int sx = 0x0C00;    // Representation of 3/4 (see Appendix B for details)
int sy = 0x0800;    // Representation of 1/2 (see Appendix B for details)
int i = 0, j;

    // Compose the header
    dwBuf[i++] = CCE_PACKET3_CNTL_SCALING;
```

```
// Compose GUI_CONTROL
dwBuf[i++] = NO_BRUSH << 4 | PIXEL_TYPE_aRGB << 8 |
             SRC_TYPE_3 << 12 | DRAW_LEFT2RIGHT << 14 |
             ROP_SRCCOPY << 16 | LD_FRM_VRAM << 24;
dwBuf[i++] = 0;                 // disable 3D operations
dwBuf[i++] = 0;                 // disable lighting
dwBuf[i++] = 0;                 // disable texture mapping
dwBuf[i++] = PIXEL_TYPE_aRGB;   // set pixel type for source
// Fill rectangles' data
dwBuf[i++] = x1*2+y1*SCREEN_PITCH*2*8;  // Mem offset of source
dwBuf[i++] = SCREEN_PITCH;      // screen pitch
dwBuf[i++] = sx;                // Scaling factor in x-direction
dwBuf[i++] = sy;                // Scaling factor in y-direction
dwBuf[i++] = y2 | (x2 << 16);   // destination location
dwBuf[i++] = w2 | (h2 << 16);   // dimensions of destination area
dwBuf[0] | = (i - 2) << 16;
// Submit the packet to the ring buffer.
SubmitPackets (dwBuf, i);
```

6.4.4 Transparent Scaled Block Transfer

This packet combines the capabilities of both transparent BitBlts and scaled BitBlts.

6.5 Drawing 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. RAGE 128 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 6-8.*

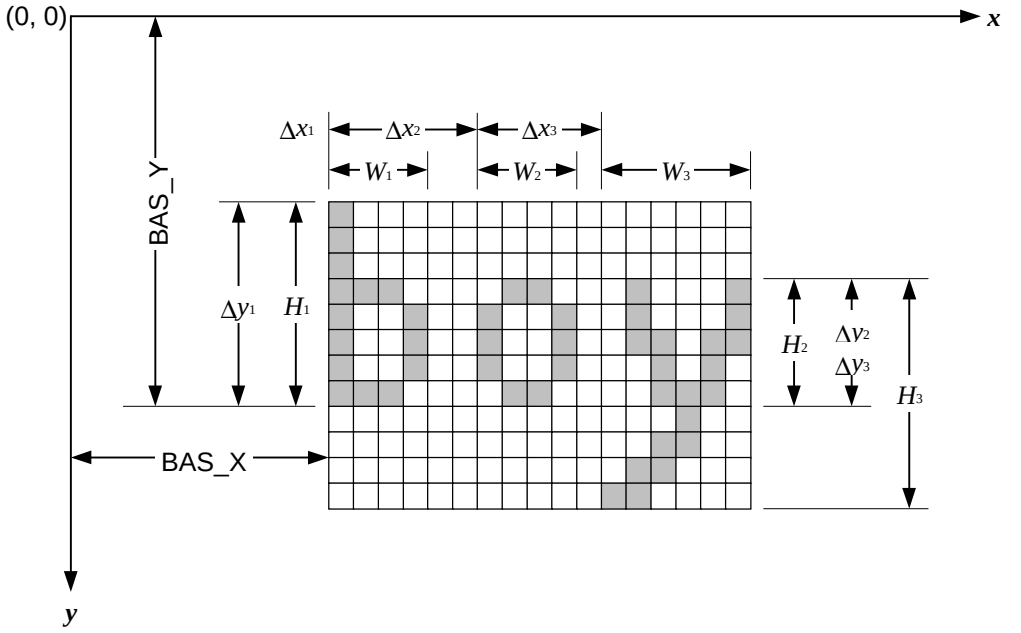

Figure 6-8. Parameters of Text

In this figure, the raster images of characters ‘b’, ‘o’ and ‘y’ are represented by 4×8, 4×5 and 6×9 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 6-8.* 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 6-8.* 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 the foreground color defined as Black (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 above.

Example Code: Drawing text in small font

```
#define CCE_PACKET3_CNTL_SMALLTEXT  0xC0009300
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define SRC_TYPE_1      1           // Mono bitmap with foreground colour
#define NO_BRUSH        15          // Brush type selected
#define LD_FRM_HOST     3           // Source is loaded from the host
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
#define DST_CLIPPING    0x00000008  // Clip the destination
#define ROP_SRCCOPY     0xCC        // Copy the source to the destination
#define PACKET_SIZE     4           // Actual size is 6 including header
#define CLIP_TOP        10          // Clipping rectangle parameters
#define CLIP_LEFT       20
#define CLIP_BOTTOM     300
#define CLIP_RIGHT      200
#define FRGD_COLOR      0           // Black
#define BAS_X           100         // starting location of the text (x)
#define BAS_Y           150         // starting location of the text (y)

// Define raster data for each character
DWORD Raster_B = 0x9E998E88;
DWORD Raster_O = 0x00609969;
DWORD Raster_Y[2] = {0x38CA1645, 0x00C01843};

DWORD dwBuf[20];

typedef struct {
    char delta_x, delta_y;
    BYTE width, height;
    WORD nPixels;
    DWORD *RastImg;
} SMALLBITGLYPH;
SMALLGLYPH text[3] = {
    {0, 8, 4, 8, 32, &Raster_B},
```

```
        {6, 5, 4, 5, 20, &Raster_O},
        {5, 5, 6, 9, 54, Raster_Y}};

    int i, j, m, n, PacketSize = 0;

        // Compose the header
        dwBuf[0] = CCE_PACKET3_CNTL_SMALLTEXT;
        // Compose GUI_CONTROL
        dwBuf[1] = DST_CLIPPING | NO_BRUSH << 4 | PIXEL_TYPE_aRGB << 8 |
                   SRC_TYPE_1 << 12 | DRAW_LEFT2RIGHT << 14 | ROP_SRCCOPY << 16|
                    LD_FRM_HOST1 << 24;
        // set up the destination clipping rectangle
        dwBuf[2] = CLIP_LEFT | CLIP_TOP << 16;
        dwBuf[3] = CLIP_RIGHT | CLIP_BOTTOM << 16;
        dwBuf[4] = FRGD_COLOR;              // Foreground clour
        dwBuf[5] = BAS_X | (BAS_Y <<16);    // starting location of the text
        m = 6;
        PacketSize = m;
        // Fill raster data for each character
        for (i = 0; i < 3; i++)
        {
            n = (text[i].nPixels + 31)/32;
            PacketSize += n + 1;
            dwBuf[m++] = text[i].delta_x | text[i].delta_y << 8 |
                         text[i].width << 16 | text[i].height << 24;
            for (j = 0; j < n; j++)
            {
                dwBuf[m++] = text[i].RastImg[j];
            } // for
        } // for
        dwBuf[0] |= PacketSize - 2 << 16;   // put the packet size into header
        // Submit the packet to the ring buffer.
        SubmitPackets (dwBuf, PacketSize);
```

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 6-8.* 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 *refer to "Drawing Text in Small Font" on page 6-25*, except for the raster operation code which is **SRCAND** (source

AND destination) instead of **SRCCOPY** (source copy), and 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
#define PIXEL_TYPE_aRGB 3           // Destination pixel type
#define SRC_TYPE_0      0           // Mono bitmap with foreground and
                                    // background colors defined
#define NO_BRUSH        15          // Brush type selected
#define LD_FRM_HOST     3           // Source is loaded from the host
#define DRAW_LEFT2RIGHT 0           // Pixels are drawn from left to right
#define DST_CLIPPING    0x00000008  // Clip the destination
#define ROP_SRCAND      0x88        // Source AND Destination (transparent)
#define PACKET_SIZE     4           // actual size is 6 including header
#define CLIP_TOP        10          // Clipping rectangle parameters
#define CLIP_LEFT       20
#define CLIP_BOTTOM     300
#define CLIP_RIGHT      200
#define FRGD_COLOR      0           // Black
#define BKGD_COLOR      0x7C00      // Red
#define BAS_X           100         // starting location of the text (x)
#define BAS_Y           150         // starting location of the text (y)

// Define raster data for each character
DWORD Raster_B = 0x9E998E88;
DWORD Raster_O = 0x00609969;
DWORD Raster_Y[2] = {0x38CA1645, 0x00C01843};

DWORD dwBuf[20];

typedef struct {
    WORD x, y;
    WORD width, height;
    DWORD nPixels;
    DWORD *RastImg;
} LARGEBITGLYPH;
LARGEGLYPH text[3] = {
    {100, 142, 4, 8, 32, &Raster_B},
    {106, 145, 4, 5, 20, &Raster_O},
    {111, 145, 6, 9, 54, Raster_Y}};

int i = 0, j, k, n;

    // Compose the header
    dwBuf[i++] = CCE_PACKET3_CNTL_HOSTDATA_BLT;
    // Compose GUI_CONTROL
    dwBuf[i++] = DST_CLIPPING | NO_BRUSH << 4 | PIXEL_TYPE_aRGB << 8 |
```

```
                SRC_TYPE_0 << 12 | DRAW_LEFT2RIGHT << 14 | ROP_SRCAND << 16
|
                 LD_FRM_HOST1 << 24;
    // set up the destination clipping rectangle
    dwBuf[i++] = CLIP_LEFT | CLIP_TOP << 16;
    dwBuf[i++] = CLIP_RIGHT | CLIP_BOTTOM << 16;
    dwBuf[i++] = FRGD_COLOR;    // Foreground clour
    dwBuf[i++] = BKGD_COLOR;    // Foreground clour
    // Fill raster data for each character
    for (j = 0; j < 3; j++)
    {
        // starting location of the text
        dwBuf[i++] = text[j].x | text[j].y << 16;
        dwBuf[i++] = text[j].width | text[j].height <<16;
        n = (text[j].nPixels + 31)/32;  // get size of raster image
        dwBuf[i++] = n;
        for (k = 0; k < n; k++)
        {
            dwBuf[i++] = text[j].RastImg[k];
        } // for
    } // for
    dwBuf[0] |= (i - 2) << 16;  // put the packet size into header
    // Submit the packet to the ring buffer.
    SubmitPackets (dwBuf, i);
```

6.6 3D Rendering

This section describes how to draw 3D primitives and set 3D rendering states on the RAGE 128 by using CCE command packets. It is assumed 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.

6.6.1 Setting Up the 3D Context

Prior to performing any 3D operations, configure the RAGE 128 into a predefined 3D context. This entails:

1. Enable the 3D operation on the RAGE 128. If this step is not taken, many 3D registers will not be writeable.
   - To enable 3D operation, set the **`MISC_3D_STATE_CNTL_REG:SCALE_3D_FN`** field to '2'.
2. Set a default set of 3D rendering states.

Once 3D operation has been enabled, individual 3D rendering states may be set as described in the section called: Setting 3D Render States.

An example of how to set the 3D context may be found in the file **`cntx3d.c`** in the **`CHAP6\3D\UTIL`** directory of the RAGE 128 DDK.

6.6.2 Drawing 3D Primitives

There are two Type-3 packets for drawing 3D primitives. Both render the following:

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

The first packet, **3D_RNDR_GEN_RPIM**, includes vertex data as part of the information body. The vertex data is copied into the ring buffer as part of the packet and bus mastered to the CCE FIFO buffer.

The second packet, **3D_RNDR_GEN_INDX_PRIM**, requires that the vertex data be placed in a dedicated buffer (called the vertex buffer) that is allocated either in the AGP space or the PCI GART space. For more details about PCI GART space, *refer to "RAGE 128 PCI GART" on page 2-23*. This method of rendering employs the RAGE 128's vertex walker mechanism. There are two ways to consume vertices from the vertex buffer when using the vertex walker:

- Specify the order of vertices through an index list provided in the information body of the **3D_RNDR_GEN_INDX_PRIM** packet. This allows vertices to be accessed in random order and to be used for more than one primitive without duplication.
- Consume vertices in sequential order from a specified location in the vertex buffer. This obviates the need for an index list in the information body of the packet.

Prior to using the vertex walker, the 192 entry CCE FIFO must be split into three distinct regions:

- 64 DWORDS for CCE command packet data.
- 64 DWORDS for vertex buffer data.
- 64 DWORDS for indirect bus-mastering data.

There are ten options for this field. Only the following three options are used with the vertex walker. The **PM4_BUFFER_CNTL** is usually written to when the CCE mode is initialized.

There are three options for setting the **PM4_BUFFER_CNTL:PM4_BUFFER_CNTL_FIFO_MODE** field:

Option 1: Write '7' to set the following configurations:

- 64 CCE packet PIO
- 64 Vertex Cache BM
- 64 Indirect BM.

Option 2: Write '8' to set the following configurations:

- 64 CCE packet BM
- 64 Vertex Cache BM
- 64 Indirect BM.

Option 3: Write ‘15’ to set the following configurations:

- 64 CCE packet PIO
- 64 Vertex Cache PIO
- 64 Indirect PIO.

For example, selecting 64 DWORD CCE packet PIO means only 64 DWORDs out of the 192 DWORDs of the CCE FIFO will be used for CCE packets. 64 out of the 192 DWORDS will be used for vertex data and the remaining 64 will be used for indirect bus mastering. Therefore, these terms represent how you want the FIFO configured.

- PIO refers to Programmed I/O.
- BM refers to Bus-Mastered.

Vertex Format

The RAGE 128 supports flexible vertex formats. This allows an application to tailor the vertex format according to its specific requirements. The vertex format is specified by setting the appropriate bits in the **VC_FORMAT** field of the primitive packet.

Drawing a Triangle List using the 3D_RNDR_GEN_PRIM Packet

The **3D_RNDR_GEN_PRIM** packet is described in depth in Appendix F. The packet consists of the following:

- HEADER field describing the kind of packet.
- VC_FORMAT field describing the vertex data block structure.
- VC_CNTL field describing the type of primitive to draw.
- A vertex array or vertex list of vertex data blocks.

Example Code: Setting up a packet to draw an independent triangle

```
#define CCE_PACKET3_3D_RNDR_GEN_PRIM  0xC0002500
#define CCE_VC_CNTL_PRIM_TYPE_TRI_LIST0x00000004
#define CCE_VC_CNTL_PRIM_WALK_RING    0x00000030
#define CCE_VC_FRMT_RHW               0x00000001
#define CCE_VC_FRMT_DIFFUSE_ARGB      0x00000008
#define CCE_VC_FRMT_SPEC_FRGB         0x00000040
#define CCE_VC_FRMT_S_T               0x00000080
#define CCE_VC_FRMT_S2_T2             0x00000100
#define VC_FORMAT_TLVERTEX2           CCE_VC_FRMT_RHW |\
                                      CCE_VC_FRMT_DIFFUSE_ARGB |\
                                      CCE_VC_FRMT_SPEC_FRGB |\
                                      CCE_VC_FRMT_S_T |\
                                      CCE_VC_FRMT_S2_T2;

// Vertex data block structure.

typedef struct {
      float x, y, z;
      float rhw;
      DWORD diffuse;
      DWORD specular;
      float s1, t1;
      float s2, t2;
} TLVERTEX2, *LPTLVERTEX2;

DWORD size=0;
TLVERTEX2* pv;
DWORD Buf[BUF_SIZE]
DWORD* pBuf = Buf;

// Set the packet HEADER, VC_FORMAT, and VC_CNTL fields.

*pBuf++ = CCE_PACKET3_3D_RNDR_GEN_PRIM;
*pBuf++ = VC_FORMAT_TLVERTEX2;
*pBuf++ = CCE_VC_CNTL_PRIM_TYPE_TRI_LIST | CCE_VC_CNTL_PRIM_WALK_RING |
                    (0x00000003L << 16);

pv = (TLVERTEX2*) pBuf;

// Copy triangle vertices into command packet buffer.

// Vertex 0:

pv->x = ((float)R128_AdapterInfo.xres/2.0f);
pv->y = ((float)R128_AdapterInfo.yres/4.0f);
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x000000ff;
```

```
pv->specular = 0x00000000;
pv->s1 = 0.5f;
pv->t1 = 1.0f;
pv->s2 = 0.5f;
pv->t2 = 0.0f;
pv++;

// Vertex 1:

pv->x = (float)R128_AdapterInfo.xres * 0.75f;
pv->y = (float)R128_AdapterInfo.yres * 0.75f;
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x0000ff00;
pv->specular = 0x00000000;
pv->s1 = 1.0f;
pv->t1 = 0.0f;
pv->s2 = 1.0f;
pv->t2 = 1.0f;
pv++;

// Vertex 2:

pv->x = (float)R128_AdapterInfo.xres * 0.25f;
pv->y = (float)R128_AdapterInfo.yres * 0.75f;
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x00ff0000;
pv->specular = 0x00000000;
pv->s1 = 0.0f;
pv->t1 = 0.0f;
pv->s2 = 0.0f;
pv->t2 = 1.0f;
pv++;
// Compute size of buffer.
size = ((DWORD)pv - (DWORD)(&Buf[0]))/sizeof (DWORD);
// Submit buffer.
Buf[0] |= ((size - 2) << 16);
R128_CCESubmitPackets (Buf, size);
```

Drawing a Triangle List using the 3D_RNDR_GEN_INDX_PRIM Packet

The **3D_RNDR_GEN_INDX_PRIM** packet is described in depth in Appendix F. The packet consists of the following:

- HEADER field describing the kind of packet.

PM4_VC_VLOFF field containing the offset of the vertex buffer from the base of AGP memory.

PM4_VC_SIZE field specifying the number of vertices in the vertex buffer.

- **VC_FORMAT** field describing the vertex data block structure.
- **VC_CNTL** field describing the type of primitive to draw.
- If the **VC_CNTL:PRIM_WALK** sub-field is **CCE_VC_CNTL_PRIM_WALK_IND**, an array of indices into the vertex buffer. Two indices are packed as WORDs per DWORD. For an odd number of indices, the high WORD of the last field is set to 0.

Example Code: Setting up a packet to draw an independent triangle using explicit vertex indices

```
#define CCE_PACKET3_3D_RNDR_GEN_INDX_PRIM     0xC0002300
#define CCE_VC_CNTL_PRIM_TYPE_TRI_LIST    0x00000004
#define CCE_VC_CNTL_PRIM_WALK_IND         0x00000010
#define CCE_VC_FRMT_RHW                   0x00000001
#define CCE_VC_FRMT_DIFFUSE_ARGB          0x00000008
#define CCE_VC_FRMT_SPEC_FRGB             0x00000040
#define CCE_VC_FRMT_S_T                   0x00000080
#define CCE_VC_FRMT_S2_T2                 0x00000100
#define VC_FORMAT_TLVERTEX2               CCE_VC_FRMT_RHW |\
                                          CCE_VC_FRMT_DIFFUSE_ARGB |\
                                          CCE_VC_FRMT_SPEC_FRGB |\
                                          CCE_VC_FRMT_S_T |\
                                          CCE_VC_FRMT_S2_T2;

// Vertex data block structure.

typedef struct {
      float x, y, z;
      float rhw;
      DWORD diffuse;
      DWORD specular;
      float s1, t1;
      float s2, t2;
} TLVERTEX2, *LPTLVERTEX2;
```

```
DWORD size=0;
TLVERTEX2* pv;
DWORD Buf[7]
// This example assumes that a vertex buffer has been allocated
// in a contiguous region of memory in AGP space. The global
// variable VertexBufferPtr contains the linear address where
// vertex data will be written into the vertex buffer. The
// global variable VertexBufferOffset contains the offset from the
// base of AGP memory to the address in VertexBufferPtr.

// Initialize packet command fields.

Buf[0] = CCE_PACKET3_3D_RNDR_GEN_INDX_PRIM;
Buf[1] = VertexBufferOffset;
Buf[2] = 3;
Buf[3] = VC_FORMAT_TLVERTEX2;
Buf[4] = CCE_VC_CNTL_PRIM_TYPE_TRI_LIST |
    CCE_VC_CNTL_PRIM_WALK_IND | 3L << 16;

// Set the vertex index write pointer. This is where vertex
// indices will be written into the packet information body.

pvertindex = (WORD*) &Buf[5];

// Write vertex data to vertex buffer.
pv = (TLVERTEX2*) VertexBufferPtr;

// Vertex 0:

pv->x = ((float)R128_AdapterInfo.xres/2.0f);
pv->y = ((float)R128_AdapterInfo.yres/4.0f);
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x000000ff;
pv->specular = 0x00000000;
pv->s1 = 0.5f;
pv->t1 = 1.0f;
pv->s2 = 0.5f;
pv->t2 = 0.0f;
pv++;

// Vertex 1:

pv->x = (float)R128_AdapterInfo.xres * 0.75f;
pv->y = (float)R128_AdapterInfo.yres * 0.75f;
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x0000ff00;
```

```
pv->specular = 0x00000000;
pv->s1 = 1.0f;
pv->t1 = 0.0f;
pv->s2 = 1.0f;
pv->t2 = 1.0f;
pv++;

// Vertex 2:

pv->x = (float)R128_AdapterInfo.xres * 0.25f;
pv->y = (float)R128_AdapterInfo.yres * 0.75f;
pv->z = 0.5f;
pv->rhw = 1.0f;
pv->diffuse = 0x00ff0000;
pv->specular = 0x00000000;
pv->s1 = 0.0f;
pv->t1 = 0.0f;
pv->s2 = 0.0f;
pv->t2 = 1.0f;
pv++;

// Write the WORD packed vertex indices into the packet
// information body.

*pvertindex++ = 0; // first vertex
*pvertindex++ = 1; // second vertex
*pvertindex++ = 2; // third vertex
*pvertindex++ = 0; // DWORD alignment padding

// Compute size of packet.

size = (DWORD)pvertindex - (DWORD)&pBuf[5]; // byte-size of
                                            // indices written.

// Convert size to DWORD count.

size = size/sizeof (DWORD);

// Adjust size for HEADER, PM4_VC _VLOFF, PM4_VC_VSIZE,
// PC_FORMAT and VC_CNTL in packet.

size += 5;

// Set packet size parameter in packet HEADER.

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

// Submit the packet to draw the batch.
```

```
R128_CCESubmitPackets (Buf, size);
```

Example Code: Setting up the packet to draw an independent triangle using the implicit vertex list in the vertex buffer

```
// This example is essentially the same as the last example.
// Only the differences are shown here.
#define CCE_VC_CNTL_PRIM_WALK_LIST 0x00000020

DWORD Buf[5];

// Initialize packet command fields.

Buf[0] = CCE_PACKET3_3D_RNDR_GEN_INDX_PRIM | (3L << 16);
Buf[1] = VertexBufferOffset;
Buf[2] = 3;
Buf[3] = VC_FORMAT_TLVERTEX2;
Buf[4] = CCE_VC_CNTL_PRIM_TYPE_TRI_LIST |
    CCE_VC_CNTL_PRIM_WALK_LIST | 3L << 16;

// Fill vertex buffer with data.
...

// Submit the packet to draw.

R128_CCESubmitPackets (Buf, 5);
```

6.6.3 Texture Mapping

The RAGE 128 contains powerful texture-combining units that can execute complex multi-texturing operations involving two textures in a single pass. Textures may reside in both local video and AGP memory. The RAGE 128 can texture map directly out of AGP memory.

Texture dimensions must be a power of two and cannot be greater than 1024. Textures may be rectangular (i.e., they need not have the same width and height). When multi-texturing, both textures may have different dimensions and data types.

The two textures in the multi-texturing stages of the texture-combining unit are referred to as the primary and secondary textures. Their features are configured by a largely similar set of registers. The primary texture registers are prefixed by PRIM_. The secondary texture registers are prefixed by **SEC_**.

For example, the general texture control registers for each are **PRIM_TEX_CNTL_C** and **SEC_TEX_CNTL_C**. For brevity, this text will describe common features in terms of the

primary texture registers only. Only secondary stage-specific features are described separately for the secondary texture.

Enabling Texture Mapping

Texture mapping may be enabled and disabled through the **TEX_CNTL_C:TEX_EN** field.

- '0' disables texture mapping.
- '1' enables texture mapping.

Texture Size and Pitch Parameters

For both the primary and secondary texture, the size and pitch parameters are entered in the **TEX_SIZE_PITCH_C** register. For the primary texture, the byte offset to the start of the texture data for the base texture is entered in register **TEX_0_OFFSET**. Byte offsets for mipmaps are entered in registers **TEX_1_OFFSET** to **TEX_10_OFFSET**. Byte offsets for the secondary texture and its mipmaps are loaded into registers **SEC_TEX_0_OFFSET** to **SEC_TEX_10_OFSET**.

Texture Format

The texture format is set through the **PRIM_TEX_CNTL_C:PRIMARY_DATATYPE**. The following formats are supported:

Table 6-3 PRIMARY_DATATYPE

State	Description
0	2-bpp VQ
1	4-bpp pseudo color
2	8-bpp pseudo color
3	16-bpp ARGB 1555
4	16-bpp RGB 565
5	24-bpp RGB
6	32-bpp ARGB 8888
7	8-bpp RGB 332
8	Y8 gray scale
9	RGB8 gray scale
10	16-bpp pseudo color

Table 6-3 PRIMARY_DATATYPE (Continued)

State	Description
11	YUV 422 packed
12	YUV 422 packed
14	AYUV 444
15	ARGB 4444

Texture Filtering

The texture magnification and minification filtering modes are set through the **PRIM_TEX_CNTL_C:PRIM_MIN_BLEND_FCN** and **PRIM_TEX_CNTL_C:PRIM_MAG_BLEND_FCN** fields. They may be set to the following values:

Table 6-4 PRIM_MIN_BLEND_FCN

State	Description
0	Pick nearest in largest map
1	Bilinear in largest map
2	Pick nearest in nearest map
3	Bilinear in nearest map
4	1x1 filtering
5	Trilinear

Table 6-5 PRIM_MAG_BLEND_FCN

State	Description
0	Pick nearest in largest map
1	Bilinear in largest map
2	Pick nearest in largest map
3	Bilinear in largest map
4	Pick nearest in largest map
5	Bilinear in largest map

States 2 to 5 for minification are only valid when mipmapping is enabled. Mipmapping may be enabled or disabled by writing ‘0’ or ‘1’ to **PRIM_TEX_CNTL_C:PRIM_MIP_MAP_DIS**.

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. Clamp mode replicates the texel at coordinate 1.0 to the edge of the primitive. Mirror mode ‘flips’ the texture about the 1.0 coordinate to produce a mirror image. Border color mode fills the remaining pixels after 1.0 with the texture's border color. Wrap addressing mode repeats, or tiles the texture along the texture coordinate axis.

Texture clamping for the texture S and T coordinates are set through the **PRIM_TEX_CNTL_C:PRIM_TEXTURE_CLAMP_MODE_S** and **PRIM_TEX_CNTL_C:PRIM_TEXTURE_CLAMP_MODE_T** fields. Texture clamping specifies how the texels should be drawn at coordinates beyond the range of 0.0 to 1.0f. The following states may be set:

Table 6-6 PRIM_TEXTURE_CLAMP_MODE_S

State	Description
0	Wrap the texture (tile)
1	Mirror the texture
2	Clamp the texture to the texel at 1.0
3	Use border color as texel color after 1.0

The border color is set through the **PRIM_TEXTURE_BORDER_COLOR** register in RGBA8888 format for RGB texture data types, or AYUV format for YUV datatypes.

Texture Wrapping

The **PRIM_TEX_CNTL_C:PRIM_TEX_WRAP_S** and **PRIM_TEX_CNTL_C:PRIM_TEX_WRAP_T** fields enable and disable cylindrical texture wrapping for the S and T coordinates, respectively.

Texture Combining

The RAGE 128 contains two texture combine units, one for the primary and one for the secondary texture. The units 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. The first and second arguments are called the color factor and input factor for the color combining function, and the alpha factor and input factor alpha for the alpha combining function. Both units have identical combining functions for the most part, but the secondary texture unit has additional functions and inputs to process the output from the primary unit.

The RAGE 128 can also combine the output of the texture combine units with the interpolated color of the primitive. This post-multitexturing combining operation is referred to as texture lighting in the RAGE 128 paradigm. Separate lighting functions are applied for the color and alpha channels. For this post-multitexturing lighting, the first argument is implicitly the output of the texture combine units, and the second is the interpolated color or alpha values of the primitive (i.e., the equivalents of the two arguments in the texture combine units).

The color combining function is set through the **PRIM_TEXTURE_COMBINE_CNTL_C: PRIMARY_COMB_FCN field**. It may be set to one of the following values:

Table 6-7 PRIMARY_COMB_FCN

State	Description
0	Disable. The output color is the texture color or interpolated color if shading.
1	Copy. Output color is the COLOR_FACTOR
2	Copy input. Output color is the INPUT_FACTOR
3	Modulate. Output color is COLOR_FACTOR * INPUT_FACTOR
4	Modulate * 2. Output color is COLOR_FACTOR * INPUT_FACTOR * 2
5	Modulate * 4. Output color is COLOR_FACTOR * INPUT_FACTOR * 4
6	Add. Output color is COLOR_FACTOR + INPUT_FACTOR
7	Add signed. Output color is COLOR_FACTOR + INPUT_FACTOR – 128
8	Blend vertex. Output color is (COLOR_FACTOR * interpolator alpha) + (INPUT_FACTOR * (1 – interpolator alpha))
9	Blend texture. Output color is (COLOR_FACTOR * primary texel alpha) + (INPUT_FACTOR * (1 – primary texel alpha))
10	Blend constant. Output color is (COLOR_FACTOR * CONSTANT_ALPHA) + (INPUT_FACTOR * (1 – CONSTANT_ALPHA))
11	Blend premultiply. Output color is COLOR_FACTOR + (INPUT_FACTOR * (1 – primary texel alpha))

Table 6-7 PRIMARY_COMB_FCN (Continued)

State	Description
12	Blend previous. Output color is (COLOR_FACTOR * primary texel alpha) + (INPUT_FACTOR * (1 – primary texel alpha))
13	Blend pre-multiply inverse. COLOR_FACTOR + (INPUT_FACTOR * primary texel alpha)
14	Add signed * 2. Output color is (COLOR_FACTOR + INPUT_FACTOR – 128) * 2
15	Blend constant color. Output color is (COLOR_FACTOR * CONSTANT_COLOR) + (INPUT_FACTOR * (1 – CONSTANT_COLOR))

The color factor is set through the **PRIM_TEXTURE_COMBINE_CNTL_C:COLOR_FACTOR** field:

Table 6-8 COLOR_FACTOR

State	Description
4	Texture color (or interpolator color if shading)
5	NOT Texture color (or NOT interpolator color if shading)
6	Texture alpha (or interpolator alpha if shading)
7	NOT Texture alpha (or NOT interpolator alpha if shading)

The input factor is set through the **PRIM_TEXTURE_COMBINE_CNTL_C:INPUT_FACTOR** field:

Table 6-9 INPUT_FACTOR

State	Description
2	CONSTANT_COLOR
3	CONSTANT_ALPHA
4	Interpolator color
5	Interpolator alpha

The alpha combine function is set through the **PRIM_TEXTURE_COMBINE_CNTL_C:COMB_FCN_ALPHA** field, which may be set to one of the following:

Table 6-10 COMB_FCN_ALPHA

State	Description
0	Disable. The output alpha is the texture alpha or interpolated alpha if shading.
1	Copy. Output alpha is the ALPHA_FACTOR.
2	Copy input. Output alpha is the INPUT_FACTOR_ALPHA
3	Modulate. Output alpha is ALPHA_FACTOR * INPUT_FACTOR_ALPHA.
4	Modulate * 2. Output alpha is ALPHA_FACTOR * INPUT_FACTOR_ALPHA * 2.
5	Modulate * 4. Output alpha is ALPHA_FACTOR * INPUT_FACTOR_ALPHA * 4.
6	Add. Output alpha is ALPHA_FACTOR + INPUT_FACTOR_ALPHA.
7	Add signed. Output alpha is ALPHA_FACTOR + INPUT_FACTOR_ALPHA – 128.
14	Add signed * 2. Output color is (COLOR_FACTOR + INPUT_FACTOR – 128) * 2.

The alpha factor is set through the **PRIM_TEXTURE_COMBINE_CNTL_C:ALPHA_FACTOR** field:

Table 6-11 ALPHA_FACTOR

State	Description
6	Texture alpha (or interpolator alpha if shading)
7	NOT Texture alpha (or NOT interpolator alpha if shading)

The alpha input factor is set through the **PRIM_TEXTURE_COMBINE_CNTL_C: INPUT_FACTOR_ALPHA** field:

Table 6-12 INPUT_FACTOR_ALPHA

State	Description
1	CONSTANT_ALPHA
2	Interpolator alpha

The **CONSTANT_COLOR** and **CONSTANT_ALPHA** are the RGB and A components, respectively, entered in RGBA8888 format in the **CONSTANT_COLOR** register.

The secondary texture allows two additional states for the input factor and one additional state for the alpha input factor.

The **SEC_TEX_COMBINE_CNTL_C:SECONDARY_INPUT_FACTOR** and the **SEC_TEX_COMBINE_CNTL_C:SECONDARY_INPUT_FACTOR_ALPHA** fields may be set to the following states:

Table 6-13 SECONDARY_INPUT_FACTOR

State	Description
2	CONSTANT_COLOR
3	CONSTANT_ALPHA
4	Interpolator color
5	Interpolator alpha
8	Previous color
9	Previous alpha

Table 6-14 SECONDARY_INPUT_FACTOR_ALPHA

State	Description
1	CONSTANT_ALPHA
2	Interpolator alpha
4	Previous alpha

The post-multitexturing lighting function is set through the **TEX_CNTL_C:TEX_LIGHT_FN**. It may be set to the following values:

Table 6-15 TEX_CNTL_C:TEX_LIGHT_FN

State	Description
0	Disable. The output color is the texture color or interpolated color if shading.
1	Copy. Output color is the COLOR_FACTOR
2	Copy input. Output color is the INPUT_FACTOR
3	Modulate. Output color is COLOR_FACTOR * INPUT_FACTOR
4	Modulate * 2. Output color is COLOR_FACTOR * INPUT_FACTOR * 2
5	Modulate * 4. Output color is COLOR_FACTOR * INPUT_FACTOR * 4
6	Add. Output color is COLOR_FACTOR + INPUT_FACTOR
7	Add signed. Output color is COLOR_FACTOR + INPUT_FACTOR - 128
8	Blend vertex. ZOutput color is (COLOR_FACTOR * interpolator alpha) + (INPUT_FACTOR * (1 - interpolator alpha))
9	Blend texture. Output color is (COLOR_FACTOR * primary texel alpha) + (INPUT_FACTOR * (1 - primary texel alpha))
10	Blend constant. Output color is (COLOR_FACTOR * CONSTANT_ALPHA) + (INPUT_FACTOR * (1 - CONSTANT_ALPHA))
12	Blend previous. Output color is (COLOR_FACTOR * primary texel alpha) + (INPUT_FACTOR * (1 - primary texel alpha))
14	Add signed * 2. Output color is (COLOR_FACTOR + INPUT_FACTOR - 128) * 2
15	Blend constant color. Output color is (COLOR_FACTOR * CONSTANT_COLOR) + (INPUT_FACTOR * (1 - CONSTANT_COLOR))

The post-multitexturing alpha lighting function is set through the **TEX_CNTL_C:ALPHA_LIGHT_FN** field. It may be set to the following:

Table 6-16 TEX_CNTL_C:ALPHA_LIGHT_FN

State	Description
0	Disable. The output color is the texture color or interpolated color if shading.
1	Copy. Output color is the COLOR_FACTOR
2	Copy input. Output color is the INPUT_FACTOR
3	Modulate. Output color is COLOR_FACTOR * INPUT_FACTOR
4	Modulate * 2. Output color is COLOR_FACTOR * INPUT_FACTOR * 2
5	Modulate * 4. Output color is COLOR_FACTOR * INPUT_FACTOR * 4
6	Add. Output color is COLOR_FACTOR + INPUT_FACTOR
7	Add signed. Output color is COLOR_FACTOR + INPUT_FACTOR - 128

Texture Coordinate Selection

In addition to the corresponding fields in the **PRIM_TEX_CNTL_C** register, the secondary texture contains the:

- **SEC_TEX_CNTL_C:SEC_SRC_SEL_ST** field for selecting the primary or secondary texture coordinate set.
- **SEC_TEX_CNTL_C:SEC_SRC_SEL_W** field for selecting the primary or secondary W coordinate.

For both fields:

- '0' selects the primary
- '1' selects the secondary.

Mipmapping

Mipmapping may be enabled separately for each texture stage. To enable mipmapping on the primary texture, set **PRIM_TEX_CNTL_C:PRIM_MIP_MAP_DIS** to '0'. A similar field exists in the **SEC_TEX_CNTL_C** register.

Loading Texture Data

Texture data can quite easily be copied into the frame buffer or AGP memory directly by the host application. However, this method has a shortcoming. It does not provide a way to synchronize the data load with the current drawing stream. There is no way to ensure the current texture data is not being fetched by the current drawing operation without taking extra, potentially performance-degrading precautions like waiting for engine idle. A better method (recommended by ATI) is to load the data using the **HOSTDATA_BLT** type-3 packet. The advantage of this method is that it streams the data load into the drawing stream.

For an example of how to use the **HOSTDATA_BLT** packet to load texture data, please see the **texture.c** file in the **Chap6\3D\Util** directory of the RAGE 128 DDK.

After loading texture data, the pixel cache should be flushed to ensure that all cached data is flushed to memory.

This may be done by setting the **PC_GUI_CTLSTATE:PC_FLUSH_GUI** field to '3'. Also, whenever switching textures, the texel cache should be flushed.

To flush the texel cache, set the **TEX_CNTL_C:TEX_CACHE_FLUSH** field to '1'. This will flush the texel cache at the start of the next primitive. This is a sticky bit. It will remain asserted until the next primitive is issued, then it will automatically be cleared.

6.6.4 Setting 3D Render States

This section describes how to set rendering states for the 3D functional blocks on the RAGE 128. The register 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);
R128_CCESubmitPackets (Buf, i);
```

Alpha Blending

Alpha blending allows the source primitive data to be combined with the destination data in various ways to achieve special effects like translucency.

The alpha blend equation is:

- CombFunc (SrcBlendFactor(SrcData), DestBlendingFactor (dstData))

Alpha blending is enabled by setting the **TEX_CNTL_C:ALPHA_EN** field to '1'. The source and destination alpha blending factors are set through the **MISC_3D_STATE_CNTL_REG:ALPHA_BLND_SRC** and **MISC_3D_STATE_CNTL_REG:ALPHA_BLND_DST** fields The following factors may be set:

Table 6-17 ALPHA_BLND_SRC

State	Description
BLEND_ZERO	Blend factor is (0, 0, 0, 0)
BLEND_ONE	Blend factor is (1, 1, 1, 1)
BLEND_SRCCOLOR	Blend factor is (Rs, Gs, Bs, As)
BLEND_INVSRCCOLOR	Blend factor is (1-Rs, 1-Gs, 1-Bs, 1-As)
BLEND_SRCALPHA	Blend factor is (As, As, As, As)
BLEND_INVSRCALPHA	Blend factor is (1-As, 1-As, 1-As, 1-As)
BLEND_DESTALPHA	Blend factor is (Ad, Ad, Ad, Ad)
BLEND_INVDESTALPHA	Blend factor is (1-Ad, 1-Ad, 1-Ad, 1-Ad)
BLEND_DESTCOLOR	Blend factor is (Rd, Gd, Bd, Ad)
BLEND_INVSDESTCOLOR	Blend factor is (1-Rd, 1-Gd, 1-Bd, 1-Ad)
BLEND_SRCALPHASAT	Blend factor is (f, f, f, 1), f = min (As, 1-Ad)
BLEND_BOTHSRCALPHA	SRC Blend factor is (As, As, As, As), force DST blend factor to (1-As, 1-As, 1-As, 1-As)
BLEND_BOTHINVSRCALPHA	SRC Blend factor is (1-As, 1-As, 1-As, 1-As), force DST blend factor to (As, As, As, As)

Table 6-18 ALPHA_BLND_DST

State	Description
BLEND_ZERO	Blend factor is (0, 0, 0, 0)
BLEND_ONE	Blend factor is (1, 1, 1, 1)

Table 6-18 ALPHA_BLND_DST (Continued)

State	Description
BLEND_SRCCOLOR	Blend factor is (Rs, Gs, Bs, As)
BLEND_INVSRCCOLOR	Blend factor is (1-Rs, 1-Gs, 1-Bs, 1-As)
BLEND_SRCALPHA	Blend factor is (As, As, As, As)
BLEND_INVSRCALPHA	Blend factor is (1-As, 1-As, 1-As, 1-As)
BLEND_DESTALPHA	Blend factor is (Ad, Ad, Ad, Ad)
BLEND_INVDESTALPHA	Blend factor is (1-Ad, 1-Ad, 1-Ad, 1-Ad)
BLEND_DESTCOLOR	Blend factor is (Rd, Gd, Bd, Ad)
BLEND_INVSDESTCOLOR	Blend factor is (1-Rd, 1-Gd, 1-Bd, 1-Ad)
BLEND_SRCALPHASAT	Blend factor is (f, f, f, 1), f = min (As, 1-Ad)

The Alpha-combing function is set through the **MISC_3D_STATE_CNTL_REG:ALPHA_COMB_FCN** field. The following alpha-combination functions may be set:

Table 6-19 ALPHA_COMB_FCN

State	Description
0	Add and clamp
1	Add but don't clamp
2	Subtract DST from SRC and clamp
3	Subtract DST from SRC but don't clamp

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 the **MISC_3D_STATE_CNTL_REG:REF_ALPHA** field. The alpha test function is set through the **MISC_3D_STATE_CNTL_REG:ALPHA_TEST_OP** field. The following states may be set:

Table 6-20 ALPHA_TEST_OP

State	Description
0	Never pass
1	Pass if Src < ref
2	Pass if Src <= Ref
3	Pass if Src == Ref
4	Pass if Src >= Ref
5	Pass if Src > Ref
6	Pass if Src != Ref
7	Always Pass

Fog Blending

Fog blending is performed according to the following equation:

- Final color = f x Cp + (1 – f) x Cf
  - f is the fog factor at the pixel.
  - Cp is the color of the source primitive pixel.
  - Cf is the fog color.

The RAGE 128 supports both table fog and vertex fog. Table fog determines the fog factor by using the interpolated z value at each pixel to index into a 256-element fog table. The fog factor in the table is an 8-bit value ranging from 0 to 255. Note that the RAGE 128 uses the vertex z value, and not the vertex w value, to index into the fog table. Vertex fog uses the interpolated alpha component of the vertex specular color as the fog factor at each pixel.

The fog method may be selected through the **MISC_3D_STATE_CNTL_REG:FOG_TABLE_EN** field.

- '0' selects vertex fog.
- '1' selects table fog.

The fog color is set through the FOG_COLOR register in RGB 888 format.

To set up the table fog, first write the table index at which new entries will be entered to the **FOG_TABLE_INDEX** register. Next, write the fog table entries to the **FOG_TABLE_DATA** register. The index is post-incremented after each write.

The following example shows how to setup the fog table using a Type-0 packet. Note that the **CCE_PACKET_0_ONE_REG_WR** flag has been added to the packet header for the table data packet, signifying that all data writes will go to the same register.

Example Code: Submitting a CCE packet

```
#define CCE_PACKET0                       0x00000000
#define CCE_PACKET_0_ONE_REG_WR(0x00000001 << 15)
#define FOG_TABLE_INDEX                   0x1a14
#define FOG_TABLE_DATA                    0x1a18

DWORD Buf[BUF_SIZE];

// Copy the fog table passed into this function.

Buf[0] = CCE_PACKET0 | (FOG_TABLE_INDEX >> 2);
Buf[1] = 0x00000000;
Buf[2] = CCE_PACKET0 | CCE_PACKET_0_ONE_REG_WR | (FOG_TABLE_DATA >> 2);

for (i=3; i < 259; i++)
Buf[i] = 258 - i;

Buf[2] |= (255L << 16);
R128_CCESubmitPackets (Buf, 259);
```

Table fog parameters such as the fog start, end, and density may be set through the **FOG_3D_TABLE_START**, **FOG_3D_TABLE_END**, and **FOG_3D_TABLE_DENSITY** registers, respectively.

Shading

The shading mode determines the color or colors used to render the primitive and how the colors are applied. The shading mode is set through the **PM4_VC_FPU_SETUP:PM4_COLOR_FCN** field. It may be set to the following states:

Table 6-21 PM4_COLOR_FCN

State	Description
0	Solid shade

Table 6-21 PM4_COLOR_FCN (Continued)

State	Description
1	Flat shade
2	Gouraud shade

Solid shading causes the primitive to be colored in the solid color set through the **CONSTANT_COLOR_C** register.

Flat shading causes the primitive to be colored according to the color of the first or third vertex. If **PM4_VC_FPU_SETUP:FLAT_SHADE_VERTEX** is set to '0', the D3D convention is used, which selects the color of the first vertex as the vertex color. '1' means use the OpenGL convention. The ***OpenGL*** convention selects the color of the third vertex as the primitive color.

Gouraud shading colors the primitive by interpolates the color at each vertex across the primitive.

Dithering

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.

The RAGE 128 implements two dithering algorithms: error diffusion, and table lookup. The algorithm may be selected through the **SCALE_3D_CNTL:SCALE_DITHER** field.

- '0' selects error diffusion.
- '1' selects table look up.

If error diffusion dither is selected, setting the **SCALE_3D_CNTL:DITHER_INIT** field to:

- '0' causes the current contents of the error register to be used at the start of the scanline.
- '1' causes the error value to be reset to '0' at the start of the line.

If table dither is selected, setting **SCALE_3D_CNTL:DITHER_INIT** disables dithering during alpha blending operations.

For best visual results, it is recommended that 32-bpp display modes be used instead of 16-bpp whenever possible. 32-bpp offers 16.7 million colors. This virtually eliminates all banding artifacts and obviates the need for dithering. 32-bpp rendering performance on the RAGE 128 is virtually identical to 16-bpp performance, resulting in negligible or no loss in rendering frame rates.

Culling

Culling allows specific operations to be performed on triangles based on their orientation. It is frequently used to eliminate back facing triangles. The culling capabilities of the RAGE 128 are configured through the **PM4_VC_FPU_SETUP** register.

PM4_VC_FPU_SETUP:FRONT_DIR selects the front facing orientation of the triangles.

- '0' selects clockwise.
- '1' selects counter clockwise.

Fields **PM4_VC_FPU_SETUP:BACKFACE_CULLING_FN** and **PM4_VC_FPU_SETUP:FRONTFACE_CULLING_FN** dictate what actions to take for back and front facing triangles, respectively. Both may be set to one of the following states:

Table 6-22 BACKFACE_CULLING_FN and FRONTFACE_CULLING_FN

State	Description
0	Cull the triangle
1	Draw the triangle as points
2	Draw the triangle as lines
3	Reverse area and draw the triangle as solid

Z Testing

Z testing is a method for performing hidden surface removal. The z values for source primitives are compared against the z values for destination pixels stored in a z buffer, and a decision is made to accept or reject, or occlude, the source pixel.

The z depth test is performed according to the following formula:

- Decision = ZDepthTestFunction (SourceZDepth, DestinationZDepth)

The z depth test function is set through the **Z_STEN_CNTL_C:Z_TEST** field. It may be set to the following states:

Table 6-23 Z_TEST

State	Description
0	Z test never passes
1	Pass if Source Z < destination Z
2	Pass if Source Z <= destination Z
3	Pass if Source Z == destination Z
4	Pass if Source Z >= destination Z
5	Pass if Source Z > destination Z
6	Pass if Source Z != destination Z
7	Z test always passes

The action to take following the z test with respect to updating the z buffer is controlled through the **TEX_CNTL_C:Z_MASK** field.

- '0' disables writes to the z buffer.
- '1' enables z writes.

Z testing is enabled and disabled through the **TEX_CNTL_C:Z_EN** field.

- '0' disables z testing
- '1' enables z testing.

The RAGE 128 supports 16bit, 24 bit, and 32 bit z buffers. The z buffer bit depth is set through the **Z_STEN_CNTL_C:Z_PIX_WIDTH** field. It may be set to the following values:

Table 6-24 Z_PIX_WIDTH

State	Description
0	16 bit z depth
1	24 bit z depth

Table 6-24 Z_PIX_WIDTH

State	Description
2	32 bit z depth

The depth of the z 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.

Stencil Buffer

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 RAGE 128 supports an eight-bit stencil buffer. The stencil buffer is interleaved with a 24-bit z buffer in a combined 32-bit buffer. The stencil buffer occupies the upper eight bits, and the z buffer occupies the lower 24 bits.

The stencil buffer operates according to the following equation:

- StencilCompareFunction ((StenciReference AND StenciMask), (StencilValue AND StencilMask))

The stencil reference is an eight-bit value ranging from ‘0’ to ‘255’. It is set by writing **STENCIL_REF_MASK_C:STEN_REF**. The stencil mask is set by writing the **STENCIL_REF_MASK_C:STEN_MSK** field.

The stencil compare function is selected by setting **Z_STEN_CNTL:STENCIL_TEST** field to one of the following values:

Table 6-25 STENCIL_TEST

State	Description
0	Never pass
1	Pass if <
2	Pass if <=
3	Pass if ==
4	Pass if >=
5	Pass if >
6	Pass if !=
7	Always pass

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 **Z_STEN_CNTL_C:STEN_SFAIL_OP**, **Z_STEN_CNTL_C:STEN_ZPASS_OP**, and **Z_STEN_CNT_CL:STEN_ZFAIL_OP** fields. Each may be set to one of the following states:

Table 6-26 States for Stencil Buffer

State	Description
0	Stencil buffer value = current
1	Stencil buffer value = 0
2	Stencil buffer value = Stencil Reference
3	Increment current stencil value by 1
4	Decrement current stencil value by 1
5	Stencil buffer value = NOT current

Data written back to the stencil buffer is masked by the stencil write mask. The stencil write mask is set through the **STEN_REF_MASK_C:STEN_WRITE_MSK** field.

This page intentionally left blank.

Chapter 7
Advanced Topics

7.1 Scope

This section covers advanced topics, such as:

7.2 Back-End Overlay and Scalar

The Back End Overlay is a hardware technique that allows the simultaneous display of two types of graphics on the CRT screen (i.e. composite video and computer graphics), while keeping the video and graphics sources in separate buffers of the graphics memory.

The overlaid video is seen through a window which is on top of the computer graphics. The the ***overlay surface*** buffer corresponds to the overlay window. The ***primary-surface buffer*** stores the graphics data.

As the two surfaces are independent of each other in display time, overlaying video on top of the primary surface will not alter the physical bits in the surface underneath it.

The overlay registers define a rectangle on the primary surface such that the rectangle will contain the overlay surface. When simultaneously displaying both computer graphics and video, the following events occurs:

1. The DAC (***Digital to Analog Converter***) reads the data in the primary surface along each scanline until it hits the left edge of the overlay rectangle.
2. Then, it switches to the overlay surface and reads from there until hitting the right edge of the rectangle.
3. The DAC switches its reading back to the original primary surface image. This switching from primary surface to the overlay and back may happen on every pass of the scanline until the video is completely overlaid.

The overlay can have a different pixel depth than the primary surface. For example, while 8bpp may look fine for the primary surface, a video clip may need to be 16bpp to be acceptable. The pixel depth switches seamlessly between the primary surface and the overlay.

The RAGE 128 uses the following scalars:

Video Input Scalar

This scalar can down-scale video horizontally (and sometimes vertically as well) before video is fed to the RAGE 128.

Horizontal Down Scalars

These scalars are located on the capture ports. Use them in case an external part such as an MPEG or HDTV decoder don not have their own downscalars.

Blits Scalar

The 2D/3D engine can also perform scale blits. Use these techniques to scale and color converted video.

Scan Conversion Scalars

These scalars are located in the “scan conversion” portion of a TV encoder product such as ATI’s ImpacTV.

Ratiometric Expander Scalars

These scalars are used in the RAGE 128 models that are dedicated for laptop computer applications.

Back-end Video Scalar

This was the first scalar put into a mainstream product. It has since been enhanced with every revision of ATI’s graphics accelerators.

Subpicture Scalar

This scalar supports DVD applications.

Back-end Video Scalar

This scalar has evolved into a very capable and feature rich scaling engine.Video can be displayed directly from the ***video frame buffer***(s) while other graphics are displayed from a ***graphics frame buffer***. Hardware will composite these two images on the way to the display. The Back End Video Scalar does the job of scaling and color converting the video.

- The main purpose for this scalar is ***upscaling***. It will read frames of video directly out of the frame buffer (generally in their native resolution) scale them up, color convert the images to RGB, and blend them with the primary display pixels.
- However, this scalar can also ***downscale***. It requires a formidable amount of signal processing to spatially resample (i.e. scale) an entire frame or field of video.
- This processing is performed during a short period of time during each display refresh. The more that the video is downscaled vertically, the less time there is to perform this processing, and the more powerful the filter engines have to be.

Recently Microsoft proposed new requirements related to video scalars. Before reading the remainder of this chapter, read the following documents:

- To review these requirements, look under NDA (they are not included here).
- For more details, refer to Microsoft's draft of PC99.

If you don't read these documents, you'll miss valuable background information, such as the definition of terms (e.g. ***tap*** and ***aliasing***) that are used in this manual. This information is continually evolving.

Some highlights about ***scaling quality*** are:

- For corporate and laptop PC applications, use the ***2x2 tap-filter kernel***. This means that the filter engine will interpolate from a 2x2 region of the source image (four pixels in a square) to create an output pixel.
- For PC used in an entertainment application, use the ***4x3 tap-filter kernel***.
- Generate a ***truncated-sinc curve*** (sinc = sin(x)/x) as a function of scaling ratio). Filter coefficients must be programmed with this curve in order to achieve an acceptable quality level.
- Minimize the ***spatial aliasing*** (artifacts created by imperfect resampling of some types of patterns in the source image).
- Enable the ability to ***zoom*** by a variable factor of up to 8:1 in 2-pixel increments.
- Enable the ability to ***shrink*** by a variable factor of up to 16:1 in 2-pixel increments.
- For ***Digital-TV,*** the scaling engine must accept and scale 1280 horizontal pixels.
- When shrinking by factors up to 2:1, image quality should not be perceptibly degraded (4:1 for PC's in an entertainment setting).

While these requirements are stringent, they can be realized with a combination of the RAGE 128's hardware and advanced drivers.

7.2.1 Feature Summary for the Back End Video Scalar

New Features

- A ***GUI stall feature*** allows MPEG decode to be synchronized with ***frame flipping***.
- ***Subpicture decoder*** and scalar interface and ***alpha-blending compositor***.

- When the video window is cropped by the desktop, it can be updated without any artifacts because there is sufficient double buffering of scalar control register fields. There is a register locking mechanism to allow autonomous updates of the overlay characteristics. Double buffering can be disabled.
- Weave 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.
- ***4-tap vertical filtering*** on all color components (Y, U, and V, RGB in RGB8888). Some restrictions apply.
- Two ***color-temperature*** settings. The recommended setting uses an improved color conversion equation.
- ***Vertical-filter engines*** are adaptively reconfigured to either filter more pixels with lower quality or fewer pixels with high quality as needed to keep the horizontal filters filled with as much data as possible data.

Other Features

- Performs ***deinterlacing*** and ***color adjustments***. Supports the following color formats:
  - ***RGB1555***, ***RGB565***
  - ***RGB8888***
  - ***Planer YUV9***, ***YUV12***
  - ***Packed YUYV***, ***UYVY***

- U's and V's, or R's and B's can be swapped in any format.
- Surfaces can be either linear or tiled surfaces. Tiled surfaces maximize the performance of hardware assisted video decompression.
- 4-tap horizontal filtering on all color components (Y, U, and V, RGB in RGB8888).
- 4-tap vertical filtering on all color components (Y, U, and V, RGB in RGB8888 - restrictions apply).
- ***4-tap filter coefficients*** are adaptively programmed to the optimal filter for the scaling ratio.
- For all four ***tap modes***, sharpness enhancing filters can be programmed.
- Vertical filters engines are adaptively reconfigured to either filter more pixels with lower quality or fewer pixels with high quality as needed to keep the horizontal filters filled with as much data as possible data.
- In four tap modes sharpness enhancing filters can be programmed.

- When downscaling, the scalar can read in up to four lines and blend them together to produce a single output line. This means up to 25% vertical reduction without line dropping is possible (50% only in RGB565 and 1555).
- The scalar can zoom in with sub-pixel windowing accuracy.
- When the video window is cropped by the desktop, it can be updated without any artifacts because there is sufficient double buffering of scalar control register fields. There is a register locking mechanism to allow autonomous updates of the overlay characteristics. Double buffering can be disabled.
- Supports either one of two ***capture ports***, or a software application as a video provider.
- Either ***bob*** the deinterlace fields (with vertical shift on either field) or ***weave*** two fields together.
- Can weave 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.
- There are two color temperature settings. The recommended setting uses an improved color conversion equation.
- Video-specific ***Gamma Correction***, ***Brightness Control***, and ***Saturation Control***.

7.2.2 Functional Overview

Table 7-1 Supported Modes

Mode	Scaling	Type of Filtering				
		Pick Nearest	2-Tap Horz	2-Tap Vert	4-Tap Horz	4-Tap Vert
RGB 1555	Up	Y	Y (**new**)	Y (**new**)	Y (**new**)	N
	Down	Y	Y (**new**)	Y (**new**)	Y (**new**)	N
RGB 565	Up	Y	Y (**new**)	Y (**new**)	Y (**new**)	N
	Down	Y	Y (**new**)	Y (**new**)	Y (**new**)	N
RGB 32	Up	Y	Y (**new**)	Y (**new**)	Y (**new**)	Y (**new**)
	Down	Y	Y (**new**)	Y (**new**)	Y (**new**)	N
YUV9	Up	Y	Y	Y	Y (**new** +uv)	Y (**new**)
	Down	Y	Y	Y	Y (**new** +uv)	N

Table 7-1 Supported Modes (Continued)

Mode	Scaling	Type of Filtering				
		Pick Nearest	2-Tap Horz	2-Tap Vert	4-Tap Horz	4-Tap Vert
YUV12	Up	Y	Y	Y	Y (**new** +uv)	Y (**new**)
	Down	Y	Y	Y	Y (**new** +uv)	N
YUYV	Up	Y	Y	Y	Y (**new** +uv)	Y (**new**)
	Down	Y	Y	Y	Y (**new** +uv)	N
UYVY	Up	Y	Y	Y	Y (**new** +uv)	Y (**new**)
	Down	Y	Y	Y	Y (**new** +uv)	N

Note: Y = YES, N = NO, new = new feature.

7.2.3 Additional Quality Enhancements

Filtering, especially 4-tap filtering, on a number of surface formats is now possible. These are shown in the previous table. In addition, when using a 4-tap filter to scale 1:1 or above, it is now possible to add a ***sharpening special effect***. The new 4-tap filters support extended coefficients that allow ***sharpening filters*** to be programmed as well as the traditional 4-tap filters and ***spatial-resampling filters***.

As an additional benefit of moving to a 128-bit wide memory system, the quality has been improved when dropping pixels to scale down horizontally. Both the RAGE PRO and RAGE 128 have sufficient filtering power to scale down an image by 50% without dropping pixels; however, below 50%, pixels are dropped. When pixels are dropped, ***aliasing*** can occur. This is especially apparent if the source image contains an image with a fine repeating pattern (e.g. striped shirt or text).

The following diagram (*Figure 7-1*.) shows the RAGE 128's improvement in the ***pixel-dropping technique***. This figure compares the RAGE 128 to its predecessor, the RAGE PRO.

- The vertical axis shows the quality of down scaling. As more pixels are dropped in a row, the lower the quality of the down-scaled image.
- The horizontal axis shows the down-scale ratio.

RAGE 128 can selectively drop either Y or UV pixels. Between 1/2 to 3/8 horizontal scale ratio, the RAGE 128 will drop the UV pixels, but not the Y pixels. Thus, aliasing of fine patterns will be avoided in this range.

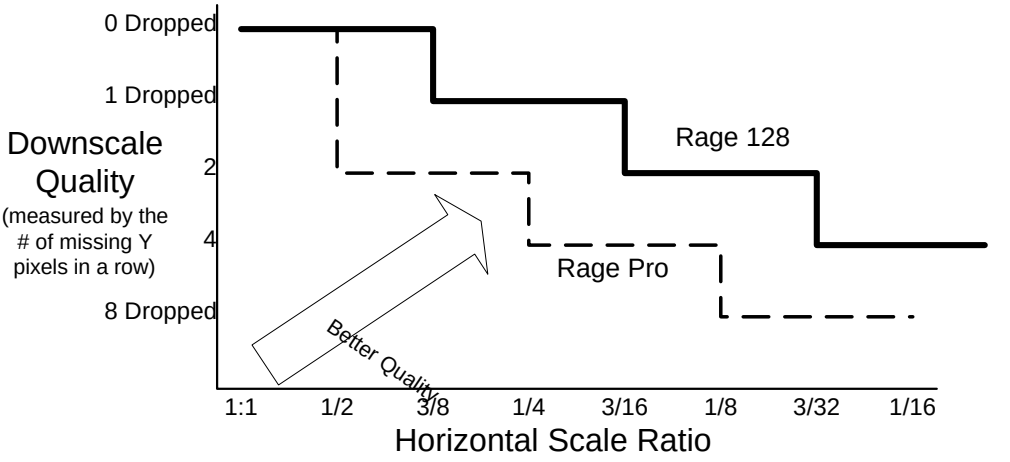

Figure 7-1. Scaling Quality Improvement

Below 3/8, when the RAGE 128 starts dropping Y pixels in order to down-scale, every second pixel is dropped rather than every second *pair* of pixels. This also improves quality, especially for text.

The following diagram (*Figure 7-2.*) shows the images that demonstrate the this concept.

- The top image shows the original text.
- The middle image is what the RAGE PRO starts with at 1/4 scaling.
- The lower image is what the RAGE 128 starts with at 1/4 scaling.

The vertical bars to the left show the relative coarseness with which the original bitmap is sampled.

- The upper “grill” samples two pixels, then drops two pixels.
- The lower “grill” uses every second pixel, and thus preserves finer detail of the original image).

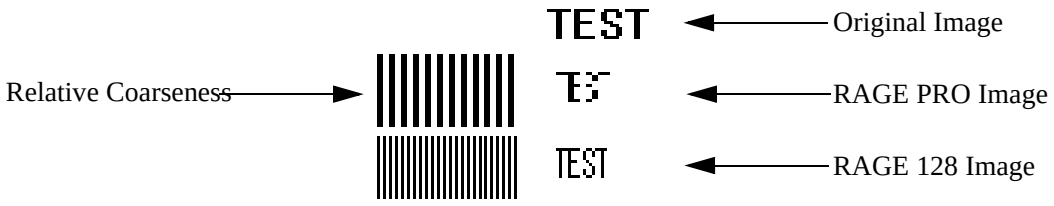

Figure 7-2. Quality Comparison between Filter Techniques

This feature allows an end user to display video in a ***small window*** (e.g. a video conference). The user can choose to simultaneously view a stock ticker or the progress of a meeting, and use most of the desktop for regular work activities. With RAGE 128, the window can be much smaller before problems occur, such as ***motion aliasing*** (i.e. ***flicker***) and readability.

If the memory system can provide enough bandwidth, vertical downscaling is improved. Previous ATI products were limited in that they could only fetch one line of source video for every line output to the display. In the RAGE 128, the number of lines fetched is programmable from one to four. Thus, in many display modes, it is possible to scale down vertically by a significant amount without dropping any lines. Remember that the vertical scalar is a 4-tap filter that drops down to a 2-tap filter when it is pushed to extremes.

If the scalar is simultaneously downscaling horizontally and vertically, then it may drop to 2-tap in order to avoid dropping pixels horizontally. Likewise, if the source is very wide, then it will drop to 2-tap to increase the length of line that can be stored in the line buffer. Even in 2-tap mode, the vertical filtering will be about twice as good as previous products when the scalar can read in two or more lines.

If the video arrives via the RAGE 128 capture port, there is a ***horizontal capture downscalar*** that will also downscale video with high quality. The capture engine's high-quality horizontal scalar and the back-end scalar can split the job of downscaling the video to achieve a very high level of overall quality.

If the source lines are pre-downscaled during capture, the back end scalar will be able to fetch up to four new lines per output line and apply the 4-tap vertical filtering to these lines. This means that the quality of the vertical downscaling will be excellent down to 1/4th the un-scaled size, and reasonably good below 1/4.

7.3 Auto-Flipping and Advanced Deinterlacing

The RAGE 128 supports ***buffer flipping*** between and up to six buffers in ***packed modes*** and between two buffers in ***planer modes***.

It also supports both bob and weave, but now the scalar will perform weaving as well as the capture hardware, and a full suite of bobbing and weaving modes is supported.

The RAGE 128 supports the following features:

Hop **- one foot on the ground, other not used**

- Where only the even or only odd fields are shown.
- Hop is best used for displaying ***freeze frame*** fields from a VCR. It produces the fewest artifacts when the two provided fields do not match.

Run **- one foot on the ground at a time, alternating feet**

- Where both fields are shown one at a time. During scaling, the even and odd fields are positioned independently so that the resulting image does not bob up and down.
- Run is a good default mode. It has the most tolerable set of artifacts over the entire range of video content. It is a good choice when reliable knowledge about the type of content is not available.

Jump **- both feet on the ground, both move together**

- Where two fields are weaved, but both fields are updated at the same time.
- Jump is best applied to video captured at a frame rate equal to half the video field rate, such as film converted to ***PAL/SECAM*** (except ***PAL M***) video. Some video is captured this way to improve slow motion replay.
- Jump may or may not weave compatible fields together. It will look good if is initiated so that the woven fields are compatible.

Walk **- both feet on the ground, move separately**

- Where two fields are weaved, but each field is updated as it arrives.
- Walk weaves fields together without regard for compatibility and updates fields as often as possible.

***Marriage Walk* - both feet on the ground, field pairs are compatible**

- Where only compatible fields are paired and weaved together. Display updates may occur any time provided that the resulting display image does not contain a mix of incompatible fields.
- Marriage Walk can be used if information about the compatibility of fields is available. If the content is suitable, and if fields can be correctly paired, the feathering artifact caused by weaving fields containing motion can be avoided. With Marriage Walk, the vertical resolution of video on a progressive display can be doubled over what is possible with Hop or Run.

7.4 Overlay Autonomous Updating

In order to solve past difficulties with making ***update-overlay commands*** execute autonomously, more registers have been double buffered and provided with a lock/unlock mechanism. This insures that a user will not notice any artifacts associated with scalar registers being partially updated when properties of the video window are changed.

Previously, only the scalar position could be updated autonomously. Now it will be possible to change the position, scale ratio, and source surfaces autonomously.

The address and pitch registers can be updated autonomously by alternating between subsets of the six address registers and two pitch registers. After programming a new set of values in unused address and pitch registers, software can point the hardware to these registers through a ***submit field*** mechanism which is lockable. Thus the submission of a field and thus the change of the address and position can be included in an autonomous update of the overlay.

7.5 Synchronizing Decoded Video Streams to the Display Refresh

To assist in a video-coded running in software so that synchronization is maintained with the display refresh, feedback will be provided to the driver to indicate whether the hardware began a new display refresh during the locked update.

If a software codec expects to be able to update the overlay properties within a single display refresh, this feedback mechanism will provide a sanity check to reassure the software that it did in fact complete it's update without dropping a frame.

7.5.1 GUI Stall Mechanism

When buffering frames of video, it may become necessary to stall the GUI engine in order to prevent the active front buffer from being overwritten to quickly with a future frame.

- If the codec determines that there is a possibility of this happening, it can enable an interlock mechanism which will temporarily stall the creation of this frame by the GUI engine.
- To enable this mechanism, a ***WaitUntilEvent command*** must precede the frame rendering commands in the GUI instruction queue.

The intent here is for the overlay to be able to tell the GUI that it is still using the surface that the GUI wants to render to even though from the software perspective that surface has been flipped and in no longer the front buffer.

What is proposed is that the overlay will send an **OV0_SURFACE_IS_FREE** signal to the GUI. It will make this signal go low when there is a danger of front buffer overwrite as determined by software.

There will be a new register bit called **OV0_STALL_GUI_UNTIL_FLIP**. If software wants to stall the GUI, then it will set this bit when if locks, updates, and unlocks overlay and subpicture registers. **OV0_SURFACE_IS_FREE** will go low at unlock and then high during VBlank (when the hardware double buffering flips the registers).

- Note that the behavior of **OV0_SURFACE_IS_FREE** is undefined if **OV0_STALL_GUI_UNTIL_FLIP** is written to when the lock bit is not set. Hardware sims will not send X's after reset however.

In the CCE/GUI environment, the event is **EVENT_OV0_FLIP**. The event happens when bit goes high. Upon a release (clearing of **OV0_LOCK**), bit is cleared. After the writes take (i.e. the pageflip is visually complete to the user), this bit is set.

OV0_SURFACE_IS_FREE is technically not an event trigger. If it is low, the WaitUntilEvent command must stall the GUI until it is high. It does not wait until the signal transitions from low to high (i.e. if it is already high, there is no stall).

7.6 Programming the Scalar

7.6.1 Overview

Setting up the parameters that define how a rectangle will be mapped onto the display is a problem that is in some ways similar to setting up the renderer to define how a triangle will be mapped into a rendering surface. Obviously, there are differences.

For example, the scalar doesn't have to do perspective correction, but it does have to perform 4x4-tap filtering and to minimize any spatial aliasing. The scalar does not have to support a Z buffer, but it does have to support many different surface formats.

7.6.2 Setup

Some setup of the scalar needs to be done in software before it can be used. Part of this setup is simply related to determining the increments that must be used for scaling. Another part concerns reconfiguring the filter engines (based on a table lookup) to best focus the filter engine's scaling power for the task.

7.6.3 Bandwidth

We have to consider the memory bandwidth that is required by the two video-capture ports, as well as the possibility of having wider (HDTV) formats that require more bandwidth per line. Handling the bandwidth problem isn't all that hard. There are just a few steps which will be explained in the following subsection. Another aspect of scalar setup is the generation of the filter coefficients. Another subsection that follows will discuss this topic.

7.6.4 Managing Bandwidth

Registers affected:

- **OV0_REG_LOAD_CNTL**
- **OV0_SCALE_CNTL**

Information must be gathered to feed into the bandwidth calculations. This should be done by reading the required registers and returning all of the parameters that are needed. It is very important that, for some of these parameters (such as the **CRTC_HTOTAL**), the driver insures that the parameter either will remain static or at least not change in a detrimental way.

For example, if the user wants to resize the CRTC:

- Either that application must prevent the **CRTC_HTOTAL** from shrinking to the point that the scalar is starved, or
- It must update the overlay when it shrinks the overlay to this point.

After the relevant information is gathered, the bandwidth routines are called.

The first routine, ***CalcFetchStartPoint***, determines the earliest point in time that line fetch requests can be made without overwriting a portion of the previously fetched that is still in use. The information that is returned must be programmed into the **H_LOAD_CMP** field of the **OV0_REG_LOAD_CNTL** register. The scalar's **OV0_SCALE_CNTL** and **OV0_PROGMBL_LOAD_START** bit must be set to one as well.

This routine contains a formula that calculates how much time (in pixel clocks) it will take to fill the line buffer in the best case. The fetch generation latency and the minimum memory latency are added to this time to generate the ***lead time***. The lead time is subtracted from the point in the horizontal timing that the scalar is finished with the last line. A little bit of margin is added for safety. The result is converted to character clocks, and programmed into the **H_LOAD_CMP** field.

The second routine, ***LineFetchSetup***, determines how many lines of data can be fetched, and what it takes to fetch those lines. In order to realize a display mode with the scalar on, it must be possible to fetch at least one line of video source data for each line of display data. Do this calculation to insure support for some of the higher resolution display modes. In lower-resolutions modes (where there is more bandwidth available), the calculation will tell you if it is possible to read in more than one video source line per display line. If it is possible, then the quality of the video when downscaling will be markedly increased. More importantly, the scalar will be compliant with Microsoft's downscaling quality requirements in these modes.

The LineFetchSetup routine is made up of a number of simple formulas that tally up the memory cycles consumed by other higher priority clients in the worst case and add extra cycles for page misses. This is subtracted from the total number of cycles to determine what is left for the scalar. This remaining bandwidth limits the number of lines (if any) that can be fetched. It calculates a value called **OV0_BURST_PER_PLANE** that controls the frequency at which Y, U, and V fetches alternate in planer modes. In the higher-resolution display modes with high refresh rates, this parameter becomes critical. It also becomes critical to reduce the size of the display FIFO buffer from 64 entries down to probably 48 entries. Either way, the remaining bandwidth drives a table look up which dictates the setting of a just a few fields that affect the fetch behavior and the amount of bandwidth consumed.

7.6.5 Physical Scaling Ratios

Once the bandwidth issue is resolved, the next step is to determine the vertical and horizontal scaling ratios that are required from the scalar. These ratios are basically functions of the source and destination window dimensions. However, there are a few mode-specific parameters (such as the type of deinterlacing, the pixel clock speed, and whether the CRT is in interlaced mode or not) that influence the calculation. As well the source window dimensions may be affected by the addition of black borders.

7.6.6 Setting up the Horizontal Accumulator

Registers affected:

- **OV0_H_INC**
- **OV0_STEP_BY**
- **OV0_P1_H_ACCUM_INIT**
- **OV0_P23_H_ACCUM_INIT**

Next, there is a table based lookup (mentioned earlier) that determines how to configure the scalar filter engines and pixel dropping hardware to achieve the required scaling. The table lookup is presently implemented in a function called **Calc_H_INC_STEP_BY** in the file **ov1calch.c**. This function's prototype is provided below:

Example Code: Setting up the horizontal accumulator

```
Calc_H_INC_STEP_BY(
    ov0field->val_OV0_SURFACE_FORMAT,
    H_scale_ratio,
    DisallowFourTapVertFiltering,
    DisallowFourTapUVVertFiltering,
    &ov0field->val_OV0_P1_H_INC,
    &ov0field->val_OV0_P1_H_STEP_BY,
    &ov0field->val_OV0_P23_H_INC,
    &ov0field->val_OV0_P23_H_STEP_BY,
    &P1GroupSize,
    &P1StepSize,
    &P23StepSize
);
P23GroupSize = 2;  // Current value for all modes
```

Calc_H_INC_STEP_BY will return values that should be programmed into the following fields:

- **OV0_P1_H_INC**
- **OV0_P23_H_INC**
- **OV0_P1_H_STEP_BY**
- **OV0_P23_H_STEP_BY**, as well as some additional information.

These fields control two horizontal accumulators, and address generation logic that fetches pixels from the line buffers to feed the vertical filter engine. These accumulators in turn generate pixel and line shift signals as well as blend ratios for the horizontal filters.

Calc_H_INC_STEP_BY will determine first if the vertical filters must be put in 2-tap mode or 4-tap mode.

- In 4-tap mode, the filters will blend using a more advanced 4-tap filtering kernel.
- In 2-tap mode, the filter engine blends using a less sophisticated liner filter (alpha blend), but it generates twice as many pixels per clock (three times as many in the case of RGB15/16).

There are a number of reasons why the filters should be put in the lower quality 2-tap mode. These are:

- Source is wider that 768.
- Source is RGB15/16 which doesn't have a four tap option.
- The vertical filters can't keep the horizontal filters supplied with pixels because the horizontal scaling ratio is too low.

When down scaling, it is better to drop from 4-tap filtering to 2-tap filtering, rather than drop pixels.

If the downscaling ratio is large, then the vertical filters can't keep up even in 2-tap mode. When this happens it becomes necessary to drop pixels.

The accumulators and shifters require initialization values that are also a function of the scaling ratio. These initialization values consist of two parts:

- **OV0_P1_H_ACCUM_INIT** and **OV0_P23_H_ACCUM_INIT** are used to initialize the accumulators.
- **OV0_PRESHIFT_P1_TO** and **OV0_PRESHIFT_P23_TO** are used to preshift the right amount of data into the horizontal filters at the beginning of each display line.

The following lines of code (taken from ov0setup.cpp) program these fields correctly.

Example Code: Setting up the horizontal accumulator

```
    tempAdditionalShift = ov0field->val_OV0_P1_X_START % P1GroupSize;
    if (ov0param->HORZ_PICK_NEAREST) {
            tempP1HStartPoint = tempAdditionalShift + 3.0 +
(float)ov0field->val_OV0_P1_H_INC / (1<<0xd);
    }
    else {
            tempP1HStartPoint = tempAdditionalShift + 2.5 +
(float)ov0field->val_OV0_P1_H_INC / (1<<0xd);
    }
    tempP1Init = (double)((int)(tempP1HStartPoint * (1<<0x5) + 0.5)) /
(1<<0x5);

    // P23 values are always fetched in pairs. If the start pixel is odd,
then we need to shift an additional pixel
    // Note that if the pitch is a multiple of two, and if we store fields
using the traditional planer format where
    // the V plane and the U plane share the same pitch, then
ov0field->val_OV0_P2_X_START % P23GroupSize should equal
    // ov0field->val_OV0_P3_X_START % P23GroupSize. Either way it is a
requirement that the U and V start on the same
    // polarity byte (even or odd).
    tempAdditionalShift = ov0field->val_OV0_P2_X_START % P23GroupSize;
    if (ov0param->HORZ_PICK_NEAREST) {
            tempP23HStartPoint = tempAdditionalShift + 3.0 +
(float)ov0field->val_OV0_P23_H_INC / (1<<0xd);
    }
    else {
            tempP23HStartPoint = tempAdditionalShift + 2.5 +
(float)ov0field->val_OV0_P23_H_INC / (1<<0xd);
    }
    tempP23Init = (double)((int)(tempP23HStartPoint * (1<<0x5) + 0.5)) /
(1<<0x5);
    ov0field->val_OV0_P1_H_ACCUM_INIT = (int)((tempP1Init -
(int)tempP1Init) * (1<<0x5));
    ov0field->val_OV0_PRESHIFT_P1_TO = (int)tempP1Init;
    ov0field->val_OV0_P23_H_ACCUM_INIT = (int)((tempP23Init -
(int)tempP23Init) * (1<<0x5));
            ov0field->val_OV0_PRESHIFT_P23_TO = (int)tempP23Init;
```

7.6.7 Setting up the Destination Window

Registers affected:

- **OV0_Y_X_START**
- **OV0_Y_X_END**
- **OV0_EXCLUSIVE_HORZ (indirectly)**
- **OV0_EXCLUSIVE_VERT (indirectly)**

The ***destination window coordinates*** are specified using 'inclusive' display pixel coordinates. There are horizontal restrictions when the pixel clock is running faster that the scalar's maximum clock of 125MHz. In this case, the scalar's destination window will always start on the correct pixel, but the total number of pixels across will be rounded up to the nearest multiple of two. For most applications this is not a problem because the overlay is only displayed where the key color is, and thus the extra pixels are hidden.

7.6.8 Setting up the Source Window

Registers affected:

- **OV0_P1_BLANK_LINES_AT_TOP**
- **OV0_P23_BLANK_LINES_AT_TOP**
- **OV0_VID_BUF*_BASE_ADRS**
- **OV0_P*_X_START_END**

The scalar can zoom in on a source window with pixel accuracy. The zoom region is referred to as the ***view window***. Any source data outside the view window will be replaced with black. The filter engine will blend black pixels with source pixels for the best possible quality at the edges.

The top of left corner of the view window (e.g. the viewing window into the source image) is approximately defined by the **OV0_VID_BUF*_BASE_ADRS** field of the **OV0_VID_BUF*_BASE_ADRS** register. This field points to the octword that contains the top left pixel in the viewing window.

The memory system and the scalar handle data in octwords. Thus, both the start pixel and end pixel in a line are specified relative to the beginning of the first octword in the line.

The exact starting pixel in the first octword is defined by the **OV0_P*_X_START** field in the **OV0_P*_X_START_END** register.

The last pixel is specified by the **OV0_P*_X_END** field in the **OV0_P*_X_START_END** register. The **P in OV0_P*_X_END** stands for 'Plane'. P1 represents the Y plane, P2 represents the U plane (and sometimes also the V plane), and P3 represents the V plane (and sometimes it's not used).

Vertically, it is possible to have the scalar add black boarders (additional black lines) to the video image without having to write them into the frame buffer. These boarders are needed for the MPEG letterbox mode. This is done by indicating the number of black lines required at the top and the height of the 'active' portion of the video.

Note that black lines require no read bandwidth, and thus they are never dropped during downscaling. Thus, if the image is predecimated by doubling the pitch, also adjust the number of black lines to be added.

7.6.9 Calculating the Filter Coefficients

Registers affected:

- **OV0_FILTER_CNTL**
- **OV0_FOUR_TAP_COEF_***

To understand how to create suitable coefficients four a ***spatial resampling filter***, you'll need to understand some background in ***spatial resampling***. Spatial resampling means scaling. Scaling is done by interpolating new pixels from nearby pixels in an original source image. This interpolation is generally performed by multiplying the nearby pixels by ***filter coefficients***, summing the results, and then dividing by the sum of the coefficients.

Video images tend to be ***band limited*** (i.e. ***separable filters*** can be used).

- Band-limited video means that there are no frequencies in the image above the maximum frequency that the raster can accurately represent.
- Separable filter means that first the image is filtered in the vertical direction and then in the horizontal direction. Separating the filtering into two steps reduces the amount of math, and thus the amount of signal processing, that the scalar must perform.

The RAGE 128 Back-End scalar uses separable 4-tap filters to interpolate new pixels from a 4x4 region of nearby source pixels. First it interpolates vertically, and then it interpolates horizontally.

The scalar allows you to select whether to use either hard-coded set or a programmable set of coefficients for each of the main ***scaling operations***. The scaling operations are:

- Vertical Y Scaling
- Vertical UV Scaling
- Horizontal Y Scaling
- Horizontal UV Scaling

The **OV0_FILTER_CNTL** register allows you to control the selection.

A given set of filter coefficients are useful for interpolating a pixel at a given position between source pixels. For example, the coefficients {-0.1, 0.6, 0.6, -0.1} would interpolate a pixel that was 50% of the way between the second and third pixel in a set of four.

For each position, a different set of four coefficients is needed. The RAGE 128 supports 8 possible positions, or “phases”. They are 0%, 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, and 87.5% of the way between the second and third pixel in a set of four. The coefficients for the three last positions are mirror images of the 2nd, 3rd, and 4th positions, thus extra registers for these values are not needed.

The hard-coded coefficients are suitable for upscaling, or downscaling slightly (9/10ths of the original). To downscale below 90%, different filter coefficients are needed to achieve the best quality. Thus, if one scaling operation is performing a downscale, it should be assigned the programmable coefficients. The coefficients should computed as a function of the scaling ratio and the sharpness control setting.

If more than one scaling operation is downscaling, either they must share the set of programmable coefficients, or one of them must use hard-coded coefficients. When you examine all the possibilities, you’ll discover that programmable coefficient contention isn’t a large problem.

If no scaling operations are downscaling, all of the scaling operations can use the programmable coefficients, and these coefficients can be programmed for upscaling with a sharpness control.

All the coefficients values for a given position (or phase) must add up to 32.

7.6.10 Setting up the Vertical Accumulator

Registers affected:

- **OV0_V_INC**
- **OV0_P1_V_ACCUM_INIT**
- **OV0_P23_V_ACCUM_INIT**
- **OV0_VID_BUF_PITCH0_VALUE** (indirectly)
- **OV0_VID_BUF_PITCH1_VALUE** (indirectly)

The vertical-scaling ratio determines the value of a vertical increment value that is used by the vertical accumulator. If you indicate to the hardware that there is bandwidth available to fetch in more than one line each Hblank (by programming the **OV0_P1_MAX_LN_IN_PER_LN_OUT** and **OV0_P23_MAX_LN_IN_PER_LN_OUT** fields), the scalar will fetch more than one line if necessary; otherwise, it will automatically drop lines when scaling down.

For large vertical downscaling ratios there may be cases when the scalar can fetch more than one line, but not all the lines it needs to downscale without dropping lines. In this case the scalar may end up ***dropping lines*** in clumps. If this situation happens, then the driver should predecimate the image vertically by doubling, tripling, quadrupling, etc. the pitch that it uses to program the scalar. This will spread out the groups of lines that are fetched and reduce the size of the gaps in the source image. Note that if the scalar can only fetch one line per HBlank, then there is no “group of lines” to spread out.

The exact vertical position must be tweaked by setting the **OV0_P1_V_ACCUM_INIT** and **OV0_P23_V_ACCUM_INIT** fields. These registers initialize the vertical accumulators so that the image will be properly aligned.

7.6.11 Autonomous Update

Registers affected:

- **OV0_REG_LOAD_CNTL.*LOCK***
- **OV0_SCALE_CNTL.OV0_DOUBLE_BUFFER_REGS**

If a new display refresh were to start (while changing some scalar registers), some of your changes would be in effect and others would not. The current refresh would be partially updated and may appear distorted. To prevent this from happening, many important scalar register fields have been double buffered.

A double buffered register field has a register field that the driver can modify and a second internal register field that the hardware uses. The driver's register field is copied to the hardware's register field at the start of the display VBlank if the **OV0_REG_LOAD_CNTL.OV0_LOCK** bit is not set.

To update autonomously, you must:

1. Set the **OV0_REG_LOAD_CNTL.OV0_LOCK** bit
2. Check that the hardware saw that this bit was set by polling **OV0_REG_LOAD_CNTL.OV0_LOCK_READBACK** until it goes high.
3. Update any double buffered registers that require updating.
4. Update the **OV0_VID_BUF_PITCH0_VALUE** register that is not in use if needed.
5. Update one (or three for planer modes) **OV0_VID_BUF*_BASE_ADRS** register(s) if needed.
6. Submit a field or frame if **OV0_VID_BUF_PITCH0_VALUE** or **OV0_VID_BUF*_BASE_ADRS** was modified. The submission is double buffered, even though the **OV0_VID_BUF*** registers are not.
7. Unlock the registers by resetting the **OV0_REG_LOAD_CNTL.OV0_LOCK** bit.

To find out if a VBlank occurred while you had the registers locked, read the **OV0_VBLANK_DURING_LOCK** field.

7.6.12 Autoflipping and Advanced Deinterlacing

Registers affected:

- **OV0_AUTO_FLIP_CNTL**
- **OV0_DEINTERLACE_PATTERN**

To use autoflipping and advanced deinterlacing, a video provider (either a capture port, or software application) must be selected by the overlay using the **OV0_VID_PORT_SELECT** register field. This video provider must submit fields to the back end video scalar.

To submit a field, a small structure is filled out that indicates:

- Which **OV0_VID_BUF?_BASE_ADDRESS**(es) point(s) to the new field.
- Whether the field is even or odd (for non-planer surfaces), and if the current field is a ***repeated field*** (if the video follows a 3:2 pulldown pattern).

A software application provides this information by writing to fields in the **OV0_AUTOFLIP_CNTL** register, and then toggling a bit (change it to ‘0’ if it is ‘1’, or change it to ‘1’ if it is ‘0’). The video capture hardware talks directly to the video scalar hardware through a similar mechanism.

The scalar keeps a record of the last three submissions from each video provider. The scalar can be directed to apply any of the above ***deinterlacing techniques*** (described below) to the fields that are submitted.

A record of each video provider’s submissions is kept by shifting the submission information into registers. Theses are referred to as the ***Next register***, the ***Curr register***, and the ***Prev register***, where:

- Next is the most recent
- Curr is the second most recent
- Prev in the third most recent.

The **OV0_VID_PORT_SELECT** the register selects, which active video provider’s submissions are dedicated to the deinterlace control block.

The repeat-field information is not recorded, but instead it is used immediately. It can reset the ***Deinterlace Pattern Pointer*** (which, BTW, is readable via **OV0_DEINT_PAT_PNTR**). Normally, the Deinterlace Pattern Pointer will increment every time that a field is submitted by the selected video provider. This pointer selects one of up to ten ***Deinterlace Pattern Directives*** that are stored in the **OV0_DEINT_PAT** register field.

The deinterlace control block establishes what surfaces the scalar will pick up and how they will be combined to create a display frame. The deinterlace control block is directed by the 2-bit ***Directive Value*** to do one of four actions. The decoding is as follows:

1. Weave the ***Next field*** with the ***Curr field***.
2. Weave the Curr field with the ***Prev field***.
3. Bob the Next field.
4. Bob the Curr field.

Different patterns can be used to achieve different types of deinterlacing. For example:

- **OV0_DEINT_PAT** = 0xAAAAA (i.e. ten ‘2’s)
  - Indicates bob with the most recent field.

- **OV0_DEINT_PAT** = 0xFFFFF (i.e. ten ‘3’s)
  - Indicates bob with the second most recent field.

- **OV0_DEINT_PAT** = 0xEEEEE (i.e. 3232323232)
  - Indicates bob using the second most recent field alternating with the most recent field. Causes that single-field mode to occur.

- **OV0_DEINT_PAT** = 0x00000 (i.e. ten ‘0’s)
  - Indicates weave using the most recent two fields. Causes 50/60 frames to be displayed per second.

- **OV0_DEINT_PAT** = 0x11111 (i.e. 0101010101)
  - Indicates weave by using the most recent two fields. Then, when the next field arrives, it uses the 2nd and 3rd most recent fields (the same two). Then repeat.

This weaves pairs of even and odd fields and displays the pairs at 30 frames per second.

- **OV0_DEINT_PAT** = 0x04411 (i.e. 0?1010?101, where '?' = don't care = 0).
  - This pattern will weave fields in a way that will undo a 3:2 pull down. The pattern will need to be rotated to match the phase of the incoming video's pulldown pattern. If a repeat field indication is available, then this can be used to sync the pattern with the incoming video.

Currently, the length of the pattern is programmable, but there doesn't appear to be any good reason to program it to less than the maximum value of '9'. The current pointer value can be read and this information can be helpful if the driver needs to dynamically change the pattern or style of deinterlacing on the fly. To switch between bob and weave, change the vertical scaling ratio as well in a single autonomous operation.

Because fields are labeled even or odd as they are submitted, the hardware can be told how to position even fields with respect to odd fields. When bobbing, the fields **OV0_SHIFT_EVEN_DOWN** and **OV0_SHIFT_ODD_DOWN** are used.

- When set, appropriately labeled fields will be shifted by one half source line.
- When weaving, the **OV0_FIRST_LINE_EVEN** will control an even field's positioning relative to an odd field's.

Avoid allowing a software application from weaving together an even field with an even field, or an odd field with an odd field. Never use weave on captured data if there is a risk that the capture hardware will submit two fields of the same type in a row.

The video capture ports are not designed to provide planer data. However, a software application, such as a hardware-assisted MPEG decoder, will provide planer data. It will not be necessary to weave planer data with planer data. Even with field based motion compensation, the data will be manipulated in the frame buffer in a pre-weaved fashion.

- To achieve weave deinterlacing, read frames directly out of the frame buffer.
- To achieve bob deinterlacing, read every second line of the Y frame and every line of the UV frames.

7.7 Color Controls

The Color controls in the RAGE 128 are the same as RAGE PRO.

7.8 Keying Controls

The following registers are used to enable color keying with respect to the overlay:

- **OV0_KEY_CNTL**
- **OV0_VIDEO_KEY_CLR**
- **OV0_VIDEO_KEY_MSK**
- **OV0_GRAPHICS_KEY_CLR**
- **OV0_GRAPHICS_KEY_MSK**

As noted above, there are color keys for both graphics and video. The graphics color key applies to data that is retrieved from the engine or the frame buffer. The video color key is applied to data that originates from the capture buffer(s). Overlay key-color registers is 24 bits wide, while the display key-color registers and key mask are 32 bits wide. The value of the color key should be entered as it applies to the current graphics mode. The mask registers should be set up to mask out the bits that you will not use in your color key. For instance, in 16 bpp-mode (565), bits [16] to [31] should be masked out, as we will not be using those bits when comparing the source data against the destination.

OV0_VIDEO_KEY_FN @ OV0_KEY_CNTL and OV0_GRAPHICS_KEY_FN @ OV0_KEY_CNTL determine how the color keys are applied. It is also possible to compare the graphics and video outputs by using OV0_CMP_MIX @ OV0_KEY_CNTL.

7.9 Tabulating Cycles in the HBlank

The scalar is limited by the number of cycles available in its VBlank. If too many other real-time clients are active and making requests in the worst possible way, there will be a minimal amount of time for the scalar to fetch the data it needs to get started. This sub-section explains how to determine how many cycles are left for the scalar if all the other clients are engaged in worst case behavior.

7.9.1 Part 1

First, we must determine the earliest point that data transfers from the frame buffer into the scalar-line buffer can occur. This point actually depends on the best-case behavior of the memory system. If the first request is serviced immediately by the memory controller and the data is returned without any page faults, the data will arrive at the earliest time possible. The data must not overwrite data from the previous line that is still in use.

The scalar actually starts and finishes reading lines from its line buffer a little ahead of the actual display timing due to the hardware's pipeline delay. The line buffer fill for the next line can begin before the current line has finished being read, provided that it doesn't overwrite the end of the current line. *Figure 7-3.* shows the fetch request beginning as early as possible.

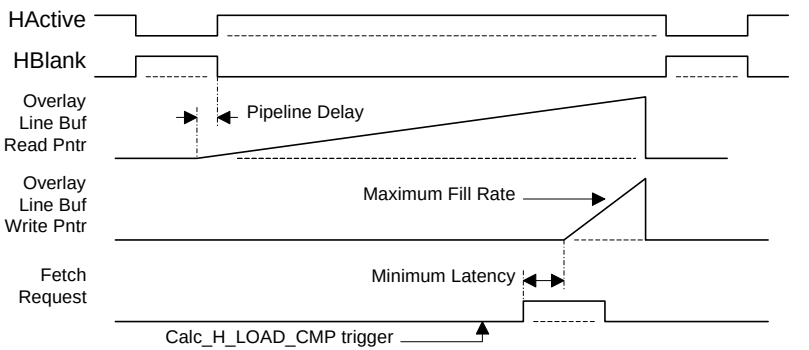

Figure 7-3. The fetch request beginning as early as possible

For planer fetches, the fill rate can be faster in the best case (when only lines of U and V are fetched). The fetch may alternate between the U and V planes. To accurately determine how soon the fetch can begin, take into account how often switching between the planes will occur.

There are two switching mechanisms.

- One method causes a switch to occur to the plane that has the least data after a fixed burst size in octwords. The **OV0_BURST_PER_PLANE** field in the **OV0_SCALE_CNTL** register defines this burst size.
- Another method is by enabling **OV0_SMART_SWITCH**. This mechanism causes the switch to occur at page boundaries, and makes the switch to a plane that is in an opposite memory bank.

In tiled mode, page boundaries occur frequently. Page faults are hidden when switches between Y, U, and V planes occur.

If the switching behavior can be accurately described in a formula, it will be possible to begin fetching at the earliest possible point in time. Currently, the start point is conservatively determined (by the ***CalcScalarHBlank function***) using the assumption that a whole line of U data is fetched, followed by a whole line of V data.

7.9.2 Part 2

The ***CalcScalarHBlank routine*** returns the following values:

- ***EarliestDataTransfer***
- ***LatestDataTransfer***
- ***VCLK_Offset***

EarliestDataTransfer (the left most dotted arrow in *Figure 7-3.*) shows where the start of the write into the line buffer can occur. There are a several cycles from when **Calc_H_LOAD_CMP** triggers a line fetch to when the first-data transfer occurs.

Subtract the minimum memory latency (about 10 cycles) from the scalar's computation delay (dependent on the minimum number of lines dropped) to obtain the earliest fetch request from the EarliestDataTransfer point. The earliest fetch request is converted to a character clock by dividing by 8 and programmed into **H_LOAD_CMP**.

The scalar's computation delay is derived from a formula that uses ***MinDroppedP1Lines*** and ***MinDroppedP23Lines*** among other variables. The address generator spends extra cycles adding the pitch to the line address when there are dropped lines. Thus if the minimum number of dropped lines is known, the trigger point can be moved a little earlier. However, the minimum number of dropped lines is not known until after the bandwidth calculation is complete and a decision is made about how many lines to fetch and how many to drop.

7.9.3 Part 3

Find when the scalar will read the first bytes of data. This is the beginning of the ***Hactive scalar*** minus a variable pipeline delay minus 16-ECP cycle-lead time.

- For non-planer modes, this defines the latest point in time that the data must start arriving by.
- For planer modes, it defines the latest point in time at which some Y, U, and V data must have arrived by.

As long as the scalar has some Y, U, and V data in its line buffer, it can get started.

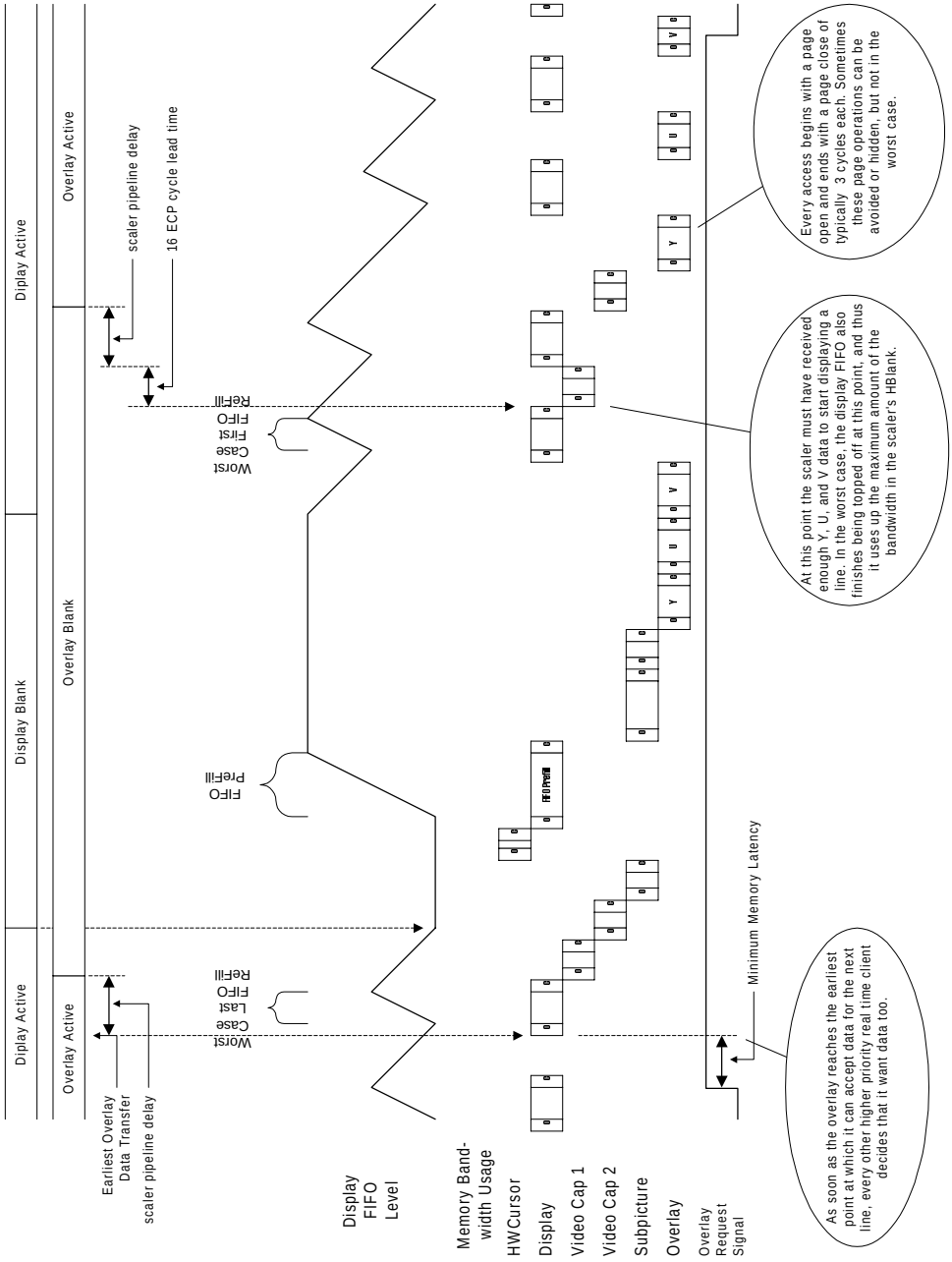

Figure 7-4. Modeling Worst Case Behavior

As show in *Figure 7-4.*, it is important to model the worst case behavior of all the other clients. This figure assumes the following:

- All the clients are turned on.
- All of their accesses will cause page faults.
- Each client will make the maximum number of small accesses to maximize the total number of page faults.
- The timing of all accesses will be as inconvenient as possible for the scalar.

In *Figure 7-4.*, the scalar is not full-screen width. If it was full width, there would be even less time for scalar data transfers. However, then it would not be possible to illustrate how the display's last and first bursts exhibit worst case behavior.

7.10 Tips for Getting More Bandwidth

- Turn off the video capture units, or correctly specify the capture rate for these units. Currently, you can report whether the capture port is turned on or off, but not what the data rate is. The data rate is assumed to be 35MB/s (which is a worst case for PAL).
- If the virtual desktop mode is off and all the display lines are aligned on 64 byte boundaries, the display accesses will improve and the display bandwidth will decrease.
- If the hardware cursor is not in use, there will be more cycles left for the scalar.
- The ***display FIFO size*** is currently set to 32 Octwords by the BIOS. In some cases, a smaller value could be set. While this may reduce the overall memory efficiency slightly, it would also free up more cycles in the HBlank for the scalar.
- Run both the scalar and the display in tiled mode. In tiled mode, transactions are more likely to dovetail together, and thus memory efficiency will improve. In fact, the scalar will automatically switch between Y, U, and V fetches in a way that will hide page turns. It is difficult to be 100% deterministic about how much of an efficiency gain will be achieved. Thus, any assumption about the benefit of tiling should be carefully tested.
- If the vertical filter coefficients are programmed so that the last (lowest on the screen) of four lines is always multiplied by a coefficient of zero, you don't have to worry about it not being fetched in time for the current display line.
    - This only works so long as the vertical filters are in 4-tap mode. They are generally in this mode when the scalar is short of bandwidth because the scalar is scaling horizontally to the full-screen width. The exception to this is sources that are wider that 768 pixels. These sources require the scalar to double up the lines. This in turn forces the filters into 2-tap vertical filtering mode.

7.11 Front-end Scalar

The front-end scalar of the RAGE 128 can be used for a few different purposes. As discussed in Chapter 4, it can be used to performed scaled-interpolated bit-block transfers (i.e. blts).

One of the other important features of the front-end scalar is the color-space conversion capability. While the overlay can be used for this purpose, the major difference between the two is that the front-end scalar actually writes the converted data back into the frame buffer, while the overlay does the conversion at the DAC level. Some applications may require access to the converted data, thus the front-end scalar would be best suited for this task.

The front-end scalar accepts the following input pixel formats:

- 8-, 15-, 16-, 24- and 32-bpp RGB
- 8-bpp RGB332
- Y8 greyscale
- RGB8 greyscale (8 bits of intensity, duplicated for all 4 channels, the RED channel is used for writes)
- 16-bpp: a psuedocolor greyscale (8:8)
- YUV 422 packed (VYUY)
- YUV 422 packed (YVYU)
- aYUV 444 (8:8:8:8)
- aRGB 4444

7.12 Bus Mastering

7.12.1 Bus Master Operation

The RAGE 128 can act as a bus master. The bus mastering capabilities allows the transfer of data from system memory to the frame buffer and vice versa with minimal CPU usage.

The RAGE 128 performs two types of transfers:

- System transfers
  - A system transfer involves moving memory between the system memory and the frame buffer memory (an visa versa).
  - Use a system transfer to move a bitmap that is loaded into system memory into the frame buffer.
  - Use the bus master to move data that was captured into the frame buffer over to system memory for modification by the CPU or other devices.

- GUI transfers
  - A GUI transfer involves moving data from system memory to the frame buffer through the GUI (or engine).
  - Use a GUI transfer (also known as a ***virtual fifo***) to queue up a series of engine register writes in system memory; then, bus master the list to the GUI using the bus master. If an application constantly performs the same type of blt or screen setup, use the bus master.

7.12.2 Creating a Descriptor Table

The bus master is instructed where to retrieve data through the use of descriptor tables. A descriptor entry consists of four DWORDs, with the following values:

Table 7-2 Descriptor Table

	Name	Bit	Function
DWORD 0	BM_FRAME_BUF_OFFSET	23:0	Frame buffer offset for data transfer
DWORD 1	BM_SYSTEM_MEM_ADDR	31:0	Physical system memory address for data transfer

Table 7-2 Descriptor Table (Continued)

	Name	Bit	Function
DWORD 2	BM_COMMAND	11:0	Count of bytes to transfer (4KB max.)
		30	Disable incrementing frame buffer offset.
		31	End of descriptor list
DWORD 3	(reserved)	31:0	

Transfers use the same byte offsets for both the frame-buffer and system-memory addresses.

- For transfers from system memory, the bus master hardware will use system memory address bits [1:0] for the frame buffer offset bits [1:0].
- For transfers from the frame buffer, frame buffer offset bits [1:0] will be used in place of the system memory bits [1:0].

Thus, the source address of the transfer will always dictate the byte alignment bit [1:0] and override the destination setting.

A maximum of 4096 bytes of data can be transferred per descriptor. As a result, when transferring an image that is larger than 4KB, create a table-of-descriptor entries. The last entry must have bit [31] of the **BM_COMMAND** DWORD set to '1' to indicate to the bus master hardware that this is the last descriptor entry. The entire descriptor table must be in contiguous memory, and the physical memory address of the head of the table must be known.

Pseudo Code to set up a Descriptor

1. **loop**:
2. Write the frame buffer destination offset address to **BM_FRAME_BUFF_OFFSET**.
3. Write the physical address of the memory to be transferred to **SYSTEM_MEM_ADDR**.
4. Write the amount of bytes to be transferred to **BM_COMMAND** (4096 bytes maximum).
5. If this is the last descriptor entry, set bit [31] to '1'.
6. If you are writing to one memory address (e.g. for a GUI transfer), set bit [30] to '1'.
7. Write a '0' for the reserved DWORD.
8. If there is still more data to be transferred, increment the **BM_FRAME_BUFF_OFFSET** and **SYSTEM_MEM_ADDR** appropriately, and go to **loop** to create another descriptor.

7.12.3 Setting up a System Bus Master Transfer

When a program requires a transfer of data from system memory to the frame buffer, the bus mastering capabilities of the RAGE 128 can be used to allow the CPU to perform other tasks while the RAGE 128 moves the data into the frame buffer.

The RAGE 128 now allows the use of several different bus mastering buffers, including:

- Three video capture buffers.
- Four VIP buffers.

These buffers do not have to be used for these (capture and VIP transfer) purposes, but are customized somewhat to these tasks.

Use the following steps to set up the RAGE 128 to perform a bus-master operation from system memory to the frame buffer. However, before performing these steps, make sure that the descriptor table is set up and the physical memory address (of the descriptor table) is paragraph aligned.

1. To enable bus mastering, clear **BUS_MASTER_DIS@BUS_CNTL**.
2. To enable the interrupt, set **BUSMASTER_EOL_INT_EN@GEN_INT_CNTL**.
3. To clear the bus master end of a transfer-interrupt set, **BUSMASTER_EOL_INT_AK@GEN_INT_STATUS** to ‘1’.
4. Set **SYSTEM_TRIGGER@BM_SYSTEM_TABLE** to the desired transfer method (‘0’ in this case).
5. Then OR this value with **SYSTEM_TABLE_ADDR@BM_SYSTEM_TABLE** (the physical memory address of the head of the descriptor table - the first descriptor entry).
6. Then, write result to **BM_SYSTEM_TABLE**. Writing to **BM_SYSTEM_TABLE** initiates the bus master operation.

At this point, allow the CPU to perform other tasks. To find out if the bus master transfer is complete, read **BUSMASTER_EOL_INT@GEN_INT_STATUS** to see if it is set to ‘1’. A ‘1’ indicates that the transfer is complete. Once **BUSMASTER_EOL_INT** has been acknowledged (i.e. set to ‘1’), write a ‘1’ to this bit to clear the interrupt.

This page intentionally left blank.

Appendix A
BIOS Function Calls

A.1 Scope

This section describes the various aspects of the VGA Controller.

A.2 AH = 0; Set Video Mode (AL = Video mode)

Table A-1 For IBM Compatible Modes

AL	MODE/TYPE	RESOLUTION	DIM/COLOR	START ADDRESS
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	B800h:0
05h	color/graphics	320x200	40x25/BW	B800h:0
06h	color/graphics	320x200	80x25/BW	B800h: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	color/graphics	640x480	80x30/BW	A000h:0
12h	color/graphics	640x480	80x30/16	A000h:0
13h	color/graphics	320x200	80x25/256	A000h:0

Table A-2 For ATI Enhanced Modes

AL	MODE/TYPE	RESOLUTION	DIM/COLOR	START ADDRESS
21h	color/alpha	800x400	100x25	B800h:0
22h	color/alpha	800x480	100x30	B800h:0

Table A-2 For ATI Enhanced Modes (Continued)

AL	MODE/TYPE	RESOLUTION	DIM/COLOR	START ADDRESS
23h	color/alpha	1056x200	132x25/16	B800h:0
33h	color/alpha	1056x352	132x44/16	B800h:0
55h	color/graphics	1024x768	128x48/16	A000h:0
61h	color/graphics	640x400	80x25/256	A000h:0
62h	color/graphics	640x480	80x30/256	A000h:0
63h	color/graphics	800x600	100x42/256	A000h:0
64h	color/graphics	1024x768	128x48/256	A000h:0
6Ah	color/graphics	800x600	100x42/16	A000h: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 = 4; Read Current Light Pen Position

VGA does not support light pen.

A.7 AH = 5; Select Active Display Page

AL = page number to be active

A.8 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.9 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.10 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.11 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.12 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.13 AH = 0Bh; Set Color Palette

This function is 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.14 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.15 AH = 0Dh; Read Dot (graphics mode)

BH = page number
DX, CX = row, column of dot position
On Exit:
AL = color value of dot

A.16 AH = 0Eh; Write Teletype to Active Page

AL = character to write
BL = foreground color in graphics mode

A.17 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.18 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.19 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


AL =  23h       ; update alternative character generator pointer (INT 43)
                    with the 8x8 character
                ;generator in ROM


AL =  24h       ; update alternative character generator pointer (INT 43)
                    with the 8x16 character
                ; generator in ROM
```

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.20 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

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 = 36h ; video screen on/off

AL = 0; video screen on

= 1; video screen off

On Exit:

AL = 12h; function supported BX=5506h

; VGAWONDER BIOS extension

AL = video mode

BP = 0FFFFh

DI = 0

SI = 0

On Exit:

if BP is not equal to 0FFFFh then ES:BP = pointer to parameter table

if SI is not equal to 0 then ES:SI = pointer to parameter table supplement

A.21 AH = 13h; Write String to Specified Page

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.22 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)

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.23 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)		
[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

Functionality and State Information (AH = 1Bh) (Continued)		
[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
[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
static functionality table format: 0 - function not supported 1 - supported function		
[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

Functionality and State Information (AH = 1Bh) (Continued)		
[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
[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

Functionality and State Information (AH = 1Bh) (Continued)		
[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.24 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
Extended BIOS Function Calls

B.1 Scope

This section provides details about the extended BIOS function calls.

For details about the *"BIOS Extensions" refer to page B-2*.

For details about the *"Mode Table Structure" refer to page B-16*.

For details about the *"RAGE 128 Internal Parameter Table Format" refer to page B-17*.

B.2 BIOS Extensions

B.2.1 Video BIOS Base Address

Extended video BIOS call can be invoked by a FAR CALL instruction in x86's 16-bit real or V86 mode. It can be accomplished in protected mode as well with proper handling of physical addresses. The physical address of video BIOS is stored in register BIOS_1_SCRATCH (base address + 014h) as a 16-bit real mode segment address value. The following assembler code will save the segment address in register DS. The starting address of the video BIOS will be DS:0

```
mov     DX, BIOS_1_SCRATCH
in      AX, DX
mov     DS, AX
```

Based on this mechanism, the video BIOS can be in RAM or in ROM, or can be anywhere in memory. For the current video BIOS, the initialization has to executed below 1M in real mode. Applications using extended video BIOS functions should work without any assumptions regarding video BIOS locations.

B.2.2 Calling Extended Functions

The video BIOS address is stored in register BIOS_1_SCRATCH and the extended video BIOS services are accessible by far call to offset 64h with the following instructions.

CALL BIOS_ADDR:64h

Another way to invoke the extended BIOS service is by calling a INT 10h with ah=0A0h. The support of INT 10h is also available with VGA disabled mode.(Multiple Display Support Document). Registers AX, BX, CX, DX, SI and DI may be destroyed during the extended function call.

VGA/VESA BIOS functions can be invoked through a far call to the offset location 68h in the BIOS.

CALL BIOS_ADDR:68h

Extended and VGA/VESA services support both x86's 16-bit real and protected mode. However, when invoked in protected mode, the applications need to call a protected mode initialization function in the BIOS and setup some segment addresses. The details of this protected mode support are described in the proposed VBE 3.0 documentation. In the current implementation, the VESA BIOS is VBE 2.0 withx86's real and protected modes support.

B.2.3 Compatibility

The purposes of these extended ROM services are to provide a set of the most commonly used hardware dependent functions in a standard interface such that application programmers need not to worry about the details of hardware programming. It is recommended that drivers developed for Rage128 should use extended function AL =0 to set display mode and the drivers have to work in VGA share mode.

B.2.4 Extended BIOS Services

BIOS_ADDR:64h
All functions return with 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] [1] [2]	= = =	0;CRT 1;TV 2;DFP
CRT STANDARD	=	0;NO MONITOR 1;MONOCHROME MONITOR 2;COLOR MONITOR
DFP STANDARD[0] [2] other bits a values	= = =	1;TFT 1;Scalable DFP Reserved
TV STANDARD	=	1;NTSC 2;PAL 3;PALM 4;PAL60 5;NTSC-J 6;SCART RGB
TVSTANDARDMASK[0] **[1]** **[2]** **[3]** **[4]** **[5]**	= = = = = =	1;NTSC 1;PAL 1;PALM 1;PAL60 1;NTSC-J 1;SCART RGB

B.2.5 Function 00h - Set Display Mode

To Call: AL = 00h Set Display Mode

CL[3-0] = Color depth
- = 1 ; 4bpp
- = 2 ; 8bpp
- = 3 ; 15bpp (555)
- = 4 ; 16bpp (565)
- = 5 ; 24bpp in RGB format
- = 6 ; 32 bpp in xRGB format

CH = Resolution
- = E1h ; 640x400
- = E2h ; 320x200
- = E3h ; 320x240
- = E4h ; 512x384
- = E5h ; 400x300
- = E6h ; 640x350
- = 12h ; 640x480
- = 6Ah ; 800x600
- = 55h ; 1024x768
- = 81h ; load CRTC table from buffer in DX:BX (see)
- = 82h ; load CRTC table from frame buffer, pointer in DX:BX (supported in VGA disable products)
- = 83h ; 1280x1024

DX:BX = pointer to parameter table if CH = 81h
= 32-bit linear address offset (in dword boundary) into frame buffer if CH = 82h

B.2.6 Function 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 (AL=00h) is called through the BIOS.

To Call: AL = 01h Set Display Controller State

CL = 0 ; VGA
= 1 ; Extended

B.2.7 Function 02h - Set DAC State

To Call:	AL	= 02h	Set DAC State
	CL	= 0	; set DAC to active mode (This function will not alter the number of bit for the DAC)
		= 1	; set DAC to sleep mode
		= 2	; set DAC to 6 bit
		= 3	; set DAC to 8 bit

B.2.8 Function 03h - Program Specified Clock Entry

To Call:	AL	= 03h	Program Specified Clock Entry
	CL[2-0]	= 0	; MCLK, engine clock
		= 1	; XCLK, memory clock
		= 2	; PCLK, dot clock
	CH	= entry in the frequency table for programming PCLK	
	BX	= value in KHz/10	
Returns:	AL	= clock chip type	
	BX, CL	= programming word depending on type	

B.2.9 Function 04h - Short Query Function 0

To Call: AL = 04h Short Query Function 0

Returns:

CH [3-0] = DAC type

CH [7,6,5,4] = Sync on green, gamma correction, 8bit, sleep

CL = Color depth support

```
Bit 7 = 1   ; 4bpp
Bit 4 = 1   ; 32bpp (unpack 24bpp in xRGB, x is byte MSB)
Bit 3 = 1   ; 24bpp in RGB
Bit 2 = 1   ; 16bpp (555)
Bit 1 = 1   ; 16bpp (565)
Bit 0 = 1   ; 8bpp
```

DL [2-0] = 000b ; generic BIOS

= 001b ; fix frequency monitor BIOS, should only use default CRTC in BIOS

= 010b ; fix frequency monitor BIOS, and can use external CRTC values

BL [3-0] = bus type

BH][3-0] = memory type

DI = subsystem vendor ID

SI = subsystem ID

B.2.10 Function 05h - Short Query Function 1

To Call: AL = 05h Short Query Function 1

Returns:

CL [3-0] = Card ID

DX = I/O base address

DI = BIOS segment address

SI = Bus/device information

B.2.11 Function 06h - Short Query Function 2

To Call: AL = 06h Short Query Function 2

Returns:

AL = Revision ID

BX = Aperture address (frame buffer address in Mbytes)

CL = Memory size in number of 512K blocks

CH	=	Reserved memory by hardware in number of 2K blocks
DX	=	PCI device ID
DI:SI	=	Alternative aperture address (memory mapped registers in linear 32bit)

B.2.12 Function 07h - Query Graphics Hardware Capability and Capture Width Info

To Call: AL = 07h Query Graphics Hardware Capability and Capture Width Info

Returns:

DX:DI	=	Pointer to table specifying max dot clock information, the table is terminated by a zero in the first column
DX:[DI-1]	=	number of bytes per row
DX:[DI-2]	=	DX:[DI-2]
CL	=	support mask to be used

H_DISP	SUPPORTMASK (use bit 7-4 only)	MEMREQ	MAX DOTCLOCK	PIXEL WIDTH
0 (end of table)				

where,

H_DISP	=	Horizontal resolution in number of characters
SUPPORT MASK	=	A bit value to indicate the valid condition of the entry
MEMREQ	=	Minimum memory required to support the specified resolution and color depth (in numbers of 512K blocks)
MAX DOTCLOCK	=	Max dot clock with the specified resolution and color depth in MHz
PIXEL WIDTH	=	Color depth

To determine if a video mode is supported, the following algorithm can be used:

if ((H_DISP <= horizontal disp(in char)&&
(SUPPORTMASK & CL)&&
(MEMREQ <= current 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:DI = Pointer to table specifying max capture width.
The table is terminated by a zero in the first column.
If SI = 0, no table is provided and driver needs to use its 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)					

where

H_DISP = Horizontal resolution in number of characters

SCALER SOURCE[7] = 1; scaler source format is in 32bpp aRGB888

[6] = 1; scaler source format is in 15bpp aRGB, 16bpp RGB565, YUV12, VYUY422 and YVYU422

MEMREQ = Minimum memory required to support the specified resolution and color depth (in numbers of 512K blocks)

MAX DOTCLOCK = Max dot clock with the specified resolution and color depth in MHz

PIXEL WIDTH = Color depth

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;
```

B.2.13 Function 08h - Query Installed Modes

To Call:	AL	=	08h Query Install Modes
	DI	=	DISPLAY DEVICE ID
	DX:BX	=	Pointer to buffer (64 bytes)
Returns:	DX:BX	=	Pointer to a list of supported modes terminated by a zero

B.2.14 Function 09h - Query Supported Mode

To Call:	AL	=	09h Query Supported Mode
	DI	=	DISPLAY DEVICE ID
	CL	=	color depth (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)
Returns:	DX:BX	=	Pointer to CRTC parameter table (if the mode is supported)

B.2.15 Function 0Ah - Display Power Management Service (DPMS)

To Call:	AL	=	0Ah	Display Power Management Service (DPMS)
	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
Returns:	CL	=	Current DPMS state	

B.2.16 Function 0Bh - Display Data Channel (DDC) Service

To Call:	AL	=	0Bh	Display Data Channel (DDC) Service
	BH	=	DISPLAY DEVICE ID	
	BL	=	0	; return DDC format supported by Graphics controller and monitor
Returns:	BL	=	0	; DDC not supported
	BL [0]	=	1	; DDC1 supported by monitor
	BL [1]	=	1	; DDC2B supported by monitor
	AL	=	0	; DDC not supported by BIOS
	AL [0]	=	1	; DDC1 supported by monitor
	AL [1]	=	1	; DDC2B supported by monitor
	AL [2]	=	1	; DDC2AB supported by BIOS
	AL [3]	=	1	; DDC2Bi supported by BIOS
	AL [6]	=	1	; BIOS supports detailed EDID timing at power-up
	AL [7]	=	1	; BIOS can us/uses EDID setup the board at power-up

To Call:	AL	= 0Bh	Display Data Channel (DDC) Service
	BL	= 1	; read EDID data (support DDC1/DDC2B only, first EDID block for DDC2B)
	BH	= DISPLAY DEVICE ID	
	CX	= Buffer size	
	DX:DI	= Pointer to buffer (not less than 128 bytes)	
Returns:	DX:DI	= Pointer to EDID data	

To Call:	AL	= 0Bh	Display Data Channel (DDC) Service
	BL	= 2	; read from device to buffer (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)	
Returns:	DX:DI	= Pointer to buffer with data read	

To Call:	AL	= 0Bh	Display Data Channel (DDC) Service
	BL	= 3	; write to device from buffer/[read from device to buffer] (only supported by DDC2B/2AB/2Bi), master write/[slave read (supported if DDC2AB is supported)]
	CX	= Buffer size	
	DX:DI	= Pointer to buffer	
	DX:[DI]...DX:[DI + CX - 1]	= Data to write	
	DX:[DI+CX]	= Max bytes to read after write (<= buffer size)	
	DX:[DI+CX+1]	= Waiting limit for slave read in msec	
Returns:	DX:DI	= Pointer to buffer with data read (if required)	

To Call:	AL	= 0Bh	Display Data Channel (DDC) Service
	BL	= 4	; return DDC format supported by BIOS
Returns:	AL[0]	= 1	; DDC1 supported by BIOS
	AL[1]	= 1	; DDC2B supported by BIOS
	AL[2]	= 1	; DDC2AB supported by BIOS
	AL[3]	= 1	; DDC2Bi supported by BIOS
	AL[6]	= 1	; BIOS support detailed EDID timing at power-up
	AL[7]	= 1	; BIOS can use EDID information to setup the board at power-up

B.2.17 Function 0Ch - Save and Restore Graphics Controller Data

To Call:	AL	= 0Ch	Save and Restore Graphics Controller Data
	CL	= 0	; return buffer size required to fit saved data in number of bytes
	CH [0]	= 0	; do not include GUI registers
		= 1	; include GUI registers (not supported)
Returns:	CX	= buffer size	

To Call:	AL	= 0Ch	Save and Restore Graphics Controller Data
	CL	= 1	; save controller data
	DX:DI	= pointer to buffer	
Returns:	DX:DI	= pointer to buffer with saved data	

To Call:	AL	= 0Ch	Save and Restore Graphics Controller Data
	CL	= 2	; restore controller data
	DX:DI	= pointer to buffer	

B.2.18 Function 0Dh - Get/Set Refresh Rate (CRT only)

To Call:	AL	= 0Dh	Get/Set Refresh Rate (CRT only)
	CL	= 0	; get current refresh rate information
		= 1	; change current refresh rate information
		= 2	; save refresh rate information
	DX:DI	= pointer to buffer (min 20 bytes required and is terminated by 0FFFFh)	

Table B-1 Refresh Rate Information Table

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)

Table B-1 Refresh Rate Information Table (Continued)

Offset (word)	Content
3	6Ah(800x600) refresh mask bit 4 = 85Hz bit 3 = 56Hz bit 2 = 60Hz bit 1 = 72Hz bit 0 = 75Hz
4	55h(1024x768)
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

To Call:

AL	= 0Dh	Get/Set Refresh Rate (CRT only)
CL	= 3	; restore factory default refresh rate information

The following functions are available if TV or Flat Panel are supported

B.2.19 Function 14h - Detect CRT/TV/DFP

To Call:

AL	= 14h	Detect CRT/TV/DFP
CH [0]	= 0	; return monitor information based on previous detection
	= 1	; return current monitor 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 [7-3]	=	00000b	; reserved
Returns:	CH	=	000b	; no TV attached
		=	001b	; TV attached to composite connector
		=	100b	; TV attached to S-Video connector
		=	101b	; TV attached to both composite and S-Video connectors
	CL	=	CRT STANDARD	
	BL	=	1	; DFP detected
	BH	=	TV STANDARD	

B.2.20 Function 15h - Get/Set Active Display(s)

To Call:	AL	=	15h	Get/Set Active Display(s)
	CH	=	0	return displays that will be set active at next mode call
Returns:	CL	=	requested display	
	CL [0]		1	; CRT
	CL [0]		1	; TV
	CL [0]		1	; DFP
	CL [0]		1	; auto-switch
	CL [7-4]		0000b	; reserved
To Call:	AL	=	15h	Get / Set Active Display(s)
	CH	=	1	; set active display, will take effect at next setmode
	CL	=	requested display	
	CL [0]		1	; CRT
	CL [1]		1	; TV
	CL [2]		1	; DFP
	CL [3]		1	; auto-switch
	CL [7-4]		0000b	; reserved

B.2.21 Function 16h - Get/Set TV Standard

This function returns an error if dynamic switching of TV standard is **NOT** supported.

To Call:	AL	= 16h	Get / Set TV Standard
	CH	= 0	; return current TV standard
Returns:	CH	= current active standard value	
	CL	= TV standard mask that can be supported on the fly	
To Call:	AL	= 16h	Get / Set TV Standard
	CH	= 1	; set TV standard, only active at next setmode
	CL	= TV standard value	
Returns:	None		

B.2.22 Function 17h - Get TVOut Info

This function will return an **error** code if TVOut is **NOT** supported.

To Call:	AL	= 16h	Get TVOut info
	DI	= 0	; get TVOut Information
Returns:	CH	= TVOut chip revision code	
	CL	= Reference Frequency	
		= 0	; 29.49892 MHz
		= 1	; 28.63636 MHz
		= 2	; 14.31888 MHz
		= 3	; 27.00000 MHz
Returns: (cont'd)	DX	= 0	; no TVOut chip is detected
		= 1	; TVOut chip is detected but not supported in BIOS
		= 3	; TVOut chip is detected and is supported in BIOS
To Call:	AL	= 16h	Get TVOut info
	DI	= 1	; reset Graphics Controller DSP value based on current setting
Returns:	None		

B.3 Mode Table Structure

B.3.1 CRTC Parameter Table

Table B-2 CRTC Parameter

Offset (byte)	Description
Installed Mode Table 1	
0 - 1	Horizontal resolution, in pixels
2 - 3	Vertical resolution, in lines
4	Mode number of this mode table
5	Maximum pixel depth
6 - 7	Reserved
8 - 9	Bits [15-0]Reserved Bit 8 Disable pipeline delay adjustment in BIOS Bits [7-6]Reserved Bit 4Enable Composite Sync Bits [3-2]Reserved Bit 1Enable interlace Bit 0 Enable double scan
0Ah	CRTC_H_TOTAL
0Bh	CRTC_H_DISP
0Ch	CRTC_H_SYNC_STRT
0Dh	CRTC_H_SYNC_WID
0Eh - 0Fh	CRTC_V_TOTAL
10h - 11h	CRTC_V_DISP
12h - 13h	CRTC_V_SYNC_STRT
14h - 15h	Bits [15-8]- Reserved for CRTC_H_DISP Bits [7-0]- CRTC_V_SYNC_WID
16h - 17h	Dot Clock for coprocessor mode (for programmable clock chip)
18h - 19h	Bits [15-8]- CRTC_H_SYNC_DLY Bits [7-4]- OVR_WID_LEFT Bits [3-0]- OVR_WID_RIGHT
1Ah - 1Bh	Bits [15-8]- OVR_WID_TOP Bits [7-0]- OVR_WID_BOTTOM
1Ch - 1Dh	Bits [15-8]- OVR_CLR_G Bits [7-0]- OVR_CLR_B
1Eh - 1Fh	Bits [15-8]= 0 Bits [7-0]- OVR_CLR_R

B.4 RAGE 128 Internal Parameter Table Format

B.4.1 CRTC Parameter Table

Table B-3 RAGE 128 internal CRTC parameter

Offset (words)	Bits	Description
0	1 5 - 8 7 - 0	Not used Video Mode Number
1	15 - 8 7 - 0	Reserved CRT Refresh rate bit mask
2	15 - 9 8 7- 5 4 2 - 3 1 0	Reserved Disable Hsync delay adjust in BIOS Reserved Enable Composite Sync Reserved Enable interlace Enable double scan
3	15 - 8 7 - 0	CRTC_H_DISP CRTC_H_TOTAL
4	15 - 8 7 - 0	CRTC_H_SYNC_WID CRTC_H_SYNC_STRT
5	15 - 0	CRTC_V_TOTAL
6	15 - 0	CRTC_V_DISP
7	15 - 0	CRTC_V_SYNC_STRT
8	15 - 8 7 - 0	Reserved for CRTC_H_DISP CRTC_V_SYNC_WIDTH
9	15 - 0	Dot Clock in KHz /10
A	15 - 8 7 - 4 3 - 0	CRTC_H_SYNC_DLY OVR_WID_LEFT OVR_WID_RIGHT
B	15 - 8 7 - 0	OVR_WID_TOP, OVR_WID_BOTTOM
C	15 - 8 7 - 0	OVR_CLR_G OVR_CLR_B
D	15 - 8 7 - 0	0 OVR_CLR_R

This page intentionally left blank.

Appendix C
BIOS Header, Scratch Registers and Information Tables

C.1 Scope

This section provides details about the BIOS Header, Scratch Registers and Information Tables.

For details about the *"Video BIOS Header" refer to page C-2*.

For details about the *"Scratch Registers" refer to page C-6*.

For details about the *"Information Tables" refer to page C-8*.

C.2 Video BIOS Header

There is information stored in the BIOS header. This information is not intended for application program development.

Table C-1 Video BIOS Header

Byte offset	Content
0	=2, Type definition
1	extended function code, 0a0h,0a1h...etc.
2	OEM_ID1
3	OEM_ID2
4	BIOS_MAJOR_REV
5	BIOS_MINOR_REV
6-7	Size of structure in number of bytes
8-9	Pointer to SMI (BIOS entry + 1)
0Ah-0Bh	Pointer to PMID
0Ch-0Dh	Pointer to initialization table
0Eh-0Fh	Pointer to CRC checksum block
10h-11h	Pointer to config file name
12h-13h	Pointer to logon message
14h-15h	Pointer to misc. information
16h-17h	PCI bus/device/function code
18h-19h	BIOS runtime segment address
1Ah-1Bh	I/O base address
1Ch-1Dh	Subsystem vendor ID
1Eh-1Fh	Subsystem ID
20h-21h	Post vendor ID
22h-23h	Int 10h offset, Coprocessor Only BIOS
24h-25h	Int 10h segment, Coprocessor Only BIOS
26h-27h	Monitor information, OEM specific
28h-29h	Pointer to configuration block (if non-zero)
2Ah-2Bh	Pointer to DAC pipeline delay information
2Ch-2Dh	Pointer to capability data structure
2Eh-2Fh	Pointer to internal CRT tables
30h-31h	Pointer to PLL information block
32h-33h	Pointer to TV information table (if non-zero)
34h-35h	Pointer to DFP information table (if non-zero)

Table C-1 Video BIOS Header (Continued)

Byte offset	Content
36h-37h	Pointer to hardware configuration table
38h-39h	Pointer to multimedia table (if non-zero)
3Ah-3Dh	TV standard BIOS table support signature "$TVS" (if dynamic bootup TV standard is supported, otherwise this field is zero)
3Eh-3Fh	Pointer to TV standard BIOS table (if non-zero and if offset 3Ah-3Dh is equal $TVS)

The following code will locate the BIOS header and extract the PCI bus/device/function information from the ROM header.

```
unsigned     far *ip;

char         far *cp;

FP_SEG(ip) = BIOSLocation();/* assume BIOSLocation()
                            /* will return the BIOS segment
                            /* address */

FP_OFF(ip) = 0x48;          /* pointer to offset to the BIOS
                            /* header */

FP_OFF(ip) = ip[0];         /* update word pointer to
                            /* point to the BIOS header */

FP_SEG(cp) = FP_SEG(ip);    /* update byte pointer to
                            /* point to the BIOS header as
                            /* well */

PciBusDev = ip[0x0b];       /* get the PCI bus/dev/func
                            /* word */
```

BIOS revision print out format

The BIOS revision number should be in the following format (AAA.BBB.CCC.DDD.CONFIG.FILE)

printf("%03d.%03d.%03d%03d.%s", OEM_ID1, OEM_ID2, BIOS_MAJOR_REV, BIOS_MINOR_REV,config.file);

Configuration block

dw2
dwbuffersize
dwbuffersize dup (0)

Code Layout

StringMeaning

‘1COD’primary runtime code ends
‘1INI’primary initialization code ends
‘DDCP’paged DDC code ends
‘AH1C’paged function ah=1ch and CXSTATE ends
‘BOOT’dual boot image ends

Misc Block

StringContent

‘R128’Rage 128 indicator
‘PCI’ PCI bus
‘AGP’ AGP
‘SGS1’SDR SGRAM 1:1
‘SGS2’SDR SGRAM 2:1
‘SGD1’DDR SGRAM

Table C-1 Initialization Block

Register (word)	‘OR’ Mask (dword)	‘AND’ Mask (dword)
BIOS_0_SCRATCH		
BUS_CNTL		
HW_DEBUG		
GEN_RESET_CNTL		
MEM_CNTL		
EXT_MEM_CNTL		
MEM_INTF_CNTL		
MEM_STR_CNTL		
MEM_INIT_LAT_TIMER		
MEM_SDRAM_MODE_REG		
MEM_ADDR_CONFIG		
GUI_DEBUG0		
GPIO_MONID_REG		
SURFACE_DELAY		
PC_GUI_MODE		
00000h		

Table C-2 PLL Block

Byte offset	Content
0	6, Clock chip type
1	Size of the structure in byte
2	Dot lock entry used for accelerated modes
3	Dot lock entry used for extended VGA modes
4 - 5	Offset into internal clock table used by VGA parameter table
6 - 7	Offset into actual programmed frequency table at POST
8 - 9	XCLK setting, (memory clock in KHz / 10)
10 - 11	MCLK setting, (engine clock in KHz / 10)
12	Number of PLL information blocks to follow, currently value is 3
13	Size of each PLL information block
14 - 15	Reference frequency of the dot clock
16 - 17	Reference Divider of the dot clock
18 - 21	Min frequency can be supported before post divider for the dot clock
22 - 25	Max frequency can be supported for the dot clock
26 - 27	Reference frequency of the MCLK, engine clock
28 - 29	Reference divider of the MCLK, engine clock
30 - 33	Min frequency can be supported before post divider for the MCLK, engine clock
34 - 37	Max frequency can be supported for the MCLK, engine clock
38 - 39	Reference frequency of the XCLK, memory clock
40 - 41	Reference divider of the XCLK, memory clock
42 - 45	Min frequency can be supported before post divider for the XCLK, memory clock
46 - 49	Max frequency can be supported for the XCLK, memory clock

C.3 Scratch Registers

Table C-3 Scratch Registers

Scratch Register	Content
BIOS_0_SCRATCH(base address + 010h)	bit 7 -Windows DOS emulation bit 6 -Mode checking by pass bit 5 -External CRTC table indicator bit 4 -Reserved (DOS) bit 3 -VBE linear frame buffer bit 2 -VBE single R/W page bit 1 bit 0
BIOS_0_SCRATCH + 1(base address + + 011h)	bit 3 -FP VGA auto scaling bit 2 -FP autoswitch, internal use bit 1- FP autoswitch pending bit 0 - TV enable state
BIOS_0_SCRATCH + 2(base address + + 012h)	bit 7-TV S-Video connected bit 6 -FP connected bit 5 -TV composite connected bit 4 -CRT connected bit 3 -Autoswitch enabled bit 2 -FP active bit 1-TV active request bit 0 -CRT active
BIOS_0_SCRATCH + 3(base address + + 013h)	Bits [7- 5]- DFP features Bit 7 - panel scalable Bit 6 - use the scalability of the chip itself Bit 5 - use the scalability of the panel Bits [4- 2] - TV STANDARD Bits [1- 0] - CRT STANDARD
BIOS_1_SCRATCH + 0(base address + + 014h)	BIOS segment address (7 - 0)
BIOS_1_SCRATCH + 1(base address + + 015h)	BIOS segment address (15 - 8)
BIOS_1_SCRATCH + 2(base address + + 016h)	Bus/device/function information
BIOS_1_SCRATCH + 3(base address + + 017h)	Bus/device/function information
BIOS_2_SCRATCH + 0(base address + + 018h)	640 refresh mask
BIOS_2_SCRATCH + 1(base address + + 019h)	800 refresh mask

Table C-3 Scratch Registers (Continued)

Scratch Register	Content
BIOS_2_SCRATCH + 2(base address + + 01Ah)	1024 refresh mask
BIOS_2_SCRATCH + 3(base address + + 01Bh)	1280 refresh mask
BIOS_3_SCRATCH + 0(base address + + 01Ch)	1600 refresh mask
BIOS_3_SCRATCH + 1(base address + + 01Dh)	PDF resolution supported by panel Bits [7-4]- Reserved Bit 3- 1280x1024 **Bit 2- 1024x768** Bit 1- 800x600 Bit 0- 640x480
BIOS_3_SCRATCH + 2(base address + + 01Eh)	Reserved
BIOS_3_SCRATCH + 3(base address + + 01Fh)	Reserved

C.4 Information Tables

C.4.1 TV Information

Table C-4 TV Information Table

Byte offset	Content
0 -2	Signature “$TV”
3	Table version = 1
4 - 5	TV information table size
6	TVOut support information: ‘N’ - default TVOut chip not found ‘T’ - TVOut chip on board
7	BIOS built-in initialization TV standard Bits [3-0] = 0001b for NTSC = 0010b for PAL = 0011b for PAL-M = 0100b for PAL-60 = 0101b for NTSC-J = 0110b for SCART-PAL
8	Checksum
9	TVOut information Bits [1-0] = 00b invalid = 01b TV off, CRT on = 10b TV on, CRT off = 11b TV on, CRT on Bits [3-2] = 00b 29.498928713 MHz = 01b 28.636360000 MHz = 10b 14.318180000 MHz = 11b 27.000000000 MHz
10	Run time supported TV standard Bit 0 NTSC Bit 1 PAL Bit 2 PAL-M Bit 3 PAL-60 Bit 4 NTSC-J Bit 5 SCART-PAL

Table C-4 TV Information Table (Continued)

Byte offset	Content
11	Initialization time supported TV standard Bit 0 NTSC Bit 1 PAL Bit 2 PAL-M Bit 3 PAL-60 Bit 4 NTSC-J Bit 5 SCART-PAL

C.4.2 DFP Information

Table C-5 DFP Information Table (Revision 0)

Byte offset	Content
0	Table revision = 0
1	Table size in bytes = 5
2	Bits [3-0] - Type of hardware support 0 None 1 1042x768 2 1280x1024 3 1600x1200 4 800x600 5 - Fh Reserved Bits [7-4] - Reserved
3	DFP standard(s) supported Bit 0 = 1 TFT Bit 1 = 1 DSTN Bit s [7-2] Reserved
4 - 5	DFP ID

Table C-6 DFP Information Table (Revision 1)

Byte offset	Content
0	Table revision = 1
1	Table size in bytes = 5
2	Bits [3-0] - Type of hardware support 0 640x480 1 800x600 2 1024x768 3 1280x1024 4 1600x1200 5 - Fh Reserved Bits [7-4] - Reserved
3	DFP standard(s) supported Bit 0 = 1 TFT Bit 1 = 1 DSTN Bit s [7-2] Reserved
4	Vendor Specific Support Flag Bit 0 = 1 Toshiba System BIOS Support for EDID
5	Pointer to Vendor Specific Table = 0h for NO TABLE DEFINED, i.e. byte 4 should be 0

Appendix D
VESA BIOS Extension

D.1 Scope

This section provides details about the VESA BIOS Extension. 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).

For details about the *"Status Information" refer to page D2*.

For details about the *"Function 00h - Return Super VGA Information" refer to page D3*.

For details about the *"Function 01h - Return Super VGA Mode Information" refer to page D6*.

For details about the *"Function 02h - Set Super VGA Video Mode" refer to page D12*.

For details about the *"Function 03h - Return Current Video Mode" refer to page D13*.

For details about the *"Function 04h - Save/Restore State" refer to page D14*.

For details about the *"Function 05h - Display Window Control" refer to page D15*.

For details about the *"Function 06h - Set/Get Logical Scan Line Length" refer to page D17*.

For details about the *"Function 07h - Set/Get Display Start" refer to page D18*.

For details about the *"Function 08h - Set/Get AC Palette Format" refer to page D19*.

For details about the *"Function 09h - Set/Get AC Palette Data" refer to page D20*.

For details about the *"Power Management Services" refer to page D21*.

For details about the *"Display Identification Extensions" refer to page D23*.

D.2 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.

D.3 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:

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
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:

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
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 D-2 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.

D.4 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:

```
ModeInfoBlock struc

;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
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:

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
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:

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 tp 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 his 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:

00h	= Text mode
01h	= CGA graphics
02h	= Hercules graphics
03h	= 4-plane planar
04h	= Packed pixel

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.

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.

D.5 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.

D.6 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.

D.7 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.

D.8 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.

D.9 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.

D.10 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.

D.11 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.

D.11.1 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	

D.11.2 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	

D.12 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 id 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
			80h	Set Palette Data during Vertical Retrace with Blank Bit on
	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	

D.13 Power Management Services

D.13.1 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	

D.13.2 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	

D.13.3 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

D.14 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.

D.14.1 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.

D.14.2 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 E
BIOS Hardware Configuration and Multimedia Tables

E.1 Scope

This section describes Multimedia Table and Hardware Configuration Table for multimedia devices in the graphics controller BIOS. The Multimedia table is used to describe the on board multimedia hardware configuration. It only exists in AIW type configuration products. The Hardware Configuration table is used to describe the graphics controller multimedia configuration.

For details about the *"BIOS Multimedia Table" refer to page E-2*.

For details about the *"BIOS Hardware Configuration Table" refer to page E-8*.

For details about the *"BIOS Tables for RAGE 128 / RAGE THEATER Board" refer to page E-10*.

E.2 BIOS Multimedia Table

The BIOS Multimedia table is used for any AIW type board and OEM solution equipped with multimedia hardware. It specifies the configuration of the multimedia devices on board. The table includes a header and a body. The header contains a unique 6 characters signature, a revision byte and a table size byte. A pointer to the table byte 0 location can be derived from the ROM header table.

Currently, the BIOS table is defined as a 8 bytes header and 7 bytes body with revision 1. Revision 0 of this table must not be used to build BIOS of any board that uses either RAGE 128 or RAGE THEATER.

The Multimedia table field definitions are shown below in . Please note that the signature is removed from the header to save ROM space. The ROM header table pointer still points to the byte 0 location. The table size and revision number can be calculated by subtracting 1 and 2 respectively from the pointer. Besides the table header, a physical connector ID field and 5 video inputs are introduced.

Table E-1 Multimedia Table, Revision 1

Offset (byte)	Field	Definition	Code Description
- 2			Hardware info table revision
- 1			Hardware info table size

Table E-1 Multimedia Table, Revision 1 (Continued)

Offset (byte)	Field	Definition	Code Description
0	Bit [4:0]	Tuner Type	= 0, No Tuner Installed = 1, Philips FI1236 MK1 NTSC M/N North America = 2, Philips FI1236 MK2 NTSC M/N Japan = 3, Philips FI1216 MK2 PAL B/G = 4, Philips FI1246 MK2 PAL I = 5, Philips FI1216 MF MK2 PAL B/G, SECAM L/L' = 6, Philips FI1236 MK2 NTSC M/N North America = 7, Philips FI1256 MK2 SECAM D/K = 8, Philips FM1236 MK2 NTSC M/N North America = 9, Philips FI1216 MK2 PAL B/G - External Tuner POD = 10, Philips FI1246 MK2 PAL I - External Tuner POD = 11, Philips FI1216 MF MK2 PAL B/G, SECAM L/L' - External Tuner POD = 12, Philips FI1236 MK2 NTSC M/N North America - External Tuner POD = 13, Temic FN5AL.RF3X7595 PAL I/B/G/DK & SECAM DK = 14, Reserved = 15, Reserved = 16, Alps TSBH5 NTSC M/N North America = 17, Alps TSC?? NTSC M/N North America = 18, Alps TSCH5 NTSC M/N North America with FM = 19-30, Reserved = 31, Unknown Tuner Type
	Bit [7:5]	Video Input for Tuner	= 0, Video input0 = 1, Video input1 = 2, Video input2 = 3, Video input3 = 4, Video input4 = 5 - 15, Reserved

Table E-1 Multimedia Table, Revision 1 (Continued)

Offset (byte)	Field	Definition	Code Description
1	Bit [3:0]	Audio Chip Type	= 0, Philips TEA5582 NTSC Stereo, no dbx, no volume control = 1, Mono with audio mux = 2, Philips TDA9850 NTSC NA. Stereo, dbx, EEPROM, mux, no volume = 3, Sony CXA2020S Japan NTSC Stereo, mux, no volume = 4, ITT MSP3410D Europe Stereo, volume, internal mux = 5, Crystal CS4236B = 6, Philips TDA9851 NTSC stereo, volume control, no dbx, no mux = 7, ITT MSP3415 (Europe) = 8, ITT MSP3430 (NA) = 9 - 14, Reserved = 15, No Audio Chip Installed
	Bit [4]	Product Type	= 0, OEM Product = 1, ATI Product
	Bit [7:5]	OEM Revision	
2	Bit [7:0]	Product ID (defined as OEM ID or ATI board ID that is dependent on Product Type Setting)	= 0, ATI Prototype Board = 1, ATI All in Wonder = 2, ATI All in Wonder Pro, no MPEG/DVD decoder = 3, ATI All in Wonder Pro, CD11 or similar MPEG/DVD decoder on MPP = 4, ATI All in Wonder Plus = 5, ATI Kitchener Board = 6, ATI Toronto Board (analog audio) = 7, ATI TV-Wonder = 8, ATI Victoria Board (RAGE XL plus RAGE THEATER) = 9-255, Reserved
3	Bit [1:0]	Tuner Voltage Regulator Control Control	= 0, No Tuner Power down feature = 1, Tuner Power down feature = 2-3, Reserved
	Bit [3:2]	Hardware Teletext Support	= 0, No Hardware Teletext = 1, Philips SAA5281 = 2-3, Reserved
	Bit [5:4]	FM Audio Decoder	= 0, No FM Audio decoder = 1, FM Audio decoder (Rohm BA1332F) installed = 2-3, Reserved
	Bit [6]	Reserved	
	Bit [7]	Audio Scrambling	= 0, Not Supported = 1, Supported

Table E-1 Multimedia Table, Revision 1 (Continued)

Offset (byte)	Field	Definition	Code Description
4	Bit [0]	I^2S Input Configuration	= 0, Not Supported = 1, Supported
	Bit [1]	I^2S Output Configuration	= 0, Not Supported = 1, Supported
	Bit [[3:2]	I^2S Audio Chip	= 0, TDA1309_32Strap. = 1, TDA1309_64Strap = 2, ITT MSP3430 = 3, ITT MSP3415 = 4-7 Reserved
	Bit [5]	S/PDIF Output Configuration	= 0, Not Supported = 1, Supported
	Bit [7:6]	Reserved	
5	Bit [[3:0]	Video Decoder Type	= 0, No Video Decoder = 1, Bt819 = 2, Bt829 = 3, Bt829A = 4, Philips SA7111 = 5, Philip SA7112, or SA7112A = 6, RAGE THEATER = 7-15, Reserved
	Bit [7:4]	Video-In Standard / Crystal	= 0, NTSC and PAL Crystals Installed (for Bt8xx) = 1, NTSC Crystal Only (for Bt8xx) = 2, PAL Crystal Only (for Bt8xx) = 3, NTSC, PAL, SECAM single crystal for Bt829 & BT879 = 4, 28.63636 MHz Crystal = 5, 29.49892713 MHz Crystal = 6, 27.0 MHz Crystal = 7, 14.31818 MHz Crystal = 8-15, Reserved
6	Bit [2:0]	Video Decoder Host Config	= 0, I^2C Device = 1, MPP Device = 2, 2 bits VIP Device = 3, 4 bits VIP Device = 4, 8 bits VIP Device = 5-6, Reserved = 7, PCI Device
	Bit [7:3]	Reserved	

Table E-1 Multimedia Table, Revision 1 (Continued)

Offset (byte)	Field	Definition	Code Description
7	Bit [1:0]	Video Input0 Type	= 0, Unused / Invalid = 1, Tuner Input = 2, Composite Input = 3, S-Video Input
	Bit [2]	Video Input0 F/B setting	= 0, Front Connector = 1, Rear Connector
	Bit [5:3]	Physical Connector ID	
	Bit [7:6]	Reserved	
8	Bit [1:0]	Video Input1 Type	= 0, Unused / Invalid = 1, Tuner Input = 2, Composite Input = 3, S-Video Input
	Bit [2]	Video Input1 F/B setting	= 0, Front Connector = 1, Rear Connector
	Bit [5:3]	Physical Connector ID	
	Bit [7:6]	Reserved	
9	Bit [1:0]	Video Input2 Type	= 0, Unused / Invalid = 1, Tuner Input = 2, Composite Input = 3, S-Video Input
	Bit [2]	Video Input2 F/B setting	= 0, Front Connector = 1, Rear Connector
	Bit [5:3]	Physical Connector ID	
	Bit [7:6]	Reserved	
10	Bit [1:0]	Video Input3 Type	= 0, Unused / Invalid = 1, Tuner Input = 2, Composite Input = 3, S-Video Input
	Bit [2]	Video Input3 F/B setting	= 0, Front Connector = 1, Rear Connector
	Bit [5:3]	Physical Connector ID	
	Bit [7:6]	Reserved	

Table E-1 Multimedia Table, Revision 1 (Continued)

Offset (byte)	Field	Definition	Code Description
11	Bit [1:0]	Video Input4 Type	= 0, Unused / Invalid = 1, Tuner Input = 2, Composite Input = 3, S-Video Input
	Bit [2]	Video Input4 F/B setting	= 0, Front Connector = 1, Rear Connector
	Bit [5:3]	Physical Connector ID	
	Bit [7:6]	Reserved	

Video decoder device offers a certain input selection that defines the video input and the combination of composite and S-video source. The following table maps out the possible video selection for each decoder type used by ATI products.

Table E-2 ATI-used decoder types and its video selections

	BT819/829		BT829A/B		RAGE THEATER	
Video Input0	Mux0	C/SV	Mux0	C/SV	Comp0	C
Video Input1	Mux1	C/SV	Mux1	C/SV	Comp1	C
Video Input2	Mux2	C/SV	Mux2	C/SV	Comp2	C
Video Input3	X	-	Mux3	C/SV	Comp3	C/SV
Video Input4	X	-	X	-	Comp4	C/SV

C/SV -Composite/S-Video. C can be used for tuner source or composite source.
For BT8x9, only one S-Video can be selected

E.3 BIOS Hardware Configuration Table

Table E-3 Hardware Configuration Table

Offset (byte)	Field	Definition	Code Description
0 - 3			Hardware info table signature string "$ATI"
4			Hardware info table revision
5			Hardware info table size
6	Bits [3:0]	I2C_Type	= 0, Normal GP_IO (I2C data=GP_IO2, clock=GP_IO1 = 1, ImpactTV GP_IO = 2, Dedicated I^2C Pin = 3, GPIO (I^2C data=GP_IO12, clock=GP_IO13 = 4, GPIO (I^2C data=GPIO12, clock=GPIO10 = 5, RAGE THEATER I^2C Master = 6, Using Rage128 MPP2 Pin (MPP2 is not used in this configuration) = 7-14, Reserved = 15, No I^2C Configuration
	Bits [7:4]	Reserved	
7	Bits [3:0]	TVOut Support	= 0, No TVOut supported = 1, ImpactTV1 supported = 2, ImpactTV2 supported = 3, Improve Impact TV2 supported = 4, RAGE THEATER supported = 5-15, Reserved
	Bits [6:4]	Video Out Crystal Frequency	= 0, TVOut not Installed = 1, 28.63636 MHz Crystal = 2, 29.49892713 MHz Crystal = 3, 27.0 MHz Crystal = 4, 1431818 MHz Crystal = 5-7, Reserved
	Bit 7	Impact TV Data Port	= 0, MPP1 = 1, MPP2

Table E-3 Hardware Configuration Table (Continued)

Offset (byte)	Field	Definition	Code Description
8	Bit 0	Video Port Capability	= 0, AMC/DVS0 Video Port (VP) un-supported = 1, AMC/DVS0 VP supported
	Bit 1		= 0, ZV VP un-supported. = 1, Zoom Video (ZV) VP supported
	Bit 2		= 0, AMC/DVS1 not supported = 1, AMC/DVS1 supported.
	Bit 3		= 0, VIP 16 bit not supported = 1, VIP 16 bit supported.
	Bits [7:4]	Reserved	
9	Bits [3:0]	Host Port Configuration	= 0, No Host Port = 1, MPP Host Port = 2, 2 bit VIP Host Port = 3, 4 bit VIP Host Port = 4, 8 bit VIP Host Port = 5-15, Reserved
	Bits [7:4]	Reserved	

E.4 BIOS Tables for RAGE 128 / RAGE THEATER Board

E.4.1 Multimedia Table

Please note the OEM default setting. The OEM default profile is defined when OEM ID (Product ID) is equal 10 and OEM revision is equal 0. These conditions provide for Multimedia table field values to be within the predefined seeing range and avoid any OEM-specific configurations. If there is an OEM-specific setting, an OEM ID as well as an OEM revision must be defined.

Table E-4 RAGE 128 / RAGE THEATER board Multimedia table

Offset (bytes)	Field	Definition	OEM Default	Kitchener NA	Toronto NA	Victoria NA	AIWPro NA
- 2			1	1	1	1	1
- 1			12	12	12	12	12
0	Bit [4:0]	Tuner Type	X	6	6	6	6
	Bit [7:5]	Video Input for Tuner	X	0	0	0	0
1	Bit [3:0]	Audio Chip Type	X	2	8		2
	Bit [4]	Product Type	0	1	1	1	1
	Bit [7:5]	OEM Revision	0	0	0	0	0
2	Bit [7:0]	Product ID	10	5	6	8	2
3	Bit [1:0]	Tuner Voltage Regulator Control	X	0	0	0	1
	Bit [3:2]	Hardware Teletext Support	X	0	0	0	0
	Bit [5:4]	FM Audio Decoder	X	0	0	0	0
	Bit [6]	Reserved	X	0	0	0	0
	Bit [7]	Audio Scrambling	X	0	0	0	0
4	Bit [0]	I^2S Input Configuration	0	0	1	0	0
	Bit [1]	I^2S Output Configuration	0	0	0	0	0
	Bit [4:2]	I^2S Audio Chip	0	0	2	0	0
	Bit [5]	S/PDIF Configuration	0	0	1	0	0
	Bit [7:5]	Reserved	0	0	0	0	0
5	Bit [3:0]	Video Decoder Type	X	3	6	6	3
	Bit [7:4]	VideoIn Standard/Crystal	X	1	5	6	1
6	Bit [2:0]	Video Decoder Host Config	0	0	2	1	0
	Bit [7:3]	Reserved	0	0	0	0	0

Table E-4 RAGE 128 / RAGE THEATER board Multimedia table (Continued)

Offset (bytes)	Field	Definition	OEM Default	Kitchener NA	Toronto NA	Victoria NA	AIWPro NA
7	Bit [1:0]	Video Input0 Type	X	1	1	1	1
	Bit [2]	Video Input0 F/B setting	0	0	0	0	0
	Bit [5:3]	Physical Connector ID	0	0	0	0	0
	Bit [7:6]	Reserved	0	0	0	0	0
8	Bit [1:0]	Video Input1 Type	X	3	0	0	3
	Bit [2]	Video Input0 F/B setting	0	0	0	0	0
	Bit [5:3]	Physical Connector ID	0	0	0	0	0
	Bit [7:6]	Reserved	0	0	0	0	0
9	Bit [1:0]	Video Input2 Type	X	2	2	2	2
	Bit [2]	Video Input0 F/B setting	0	0	0	0	0
	Bit [5:3]	Physical Connector ID	0	0	0	0	0
	Bit [7:6]	Reserved	0	0	0	0	0
10	Bit [1:0]	Video Input3 Type	0	0	0	0	0
	Bit [2]	Video Input0 F/B setting	0	0	0	0	0
	Bit [5:3]	Physical Connector ID	0	0	0	0	0
	Bit [7:6]	Reserved	0	0	0	0	0
11	Bit [1:0]	Video Input4 Type	0	0	3	3	0
	Bit [2]	Video Input0 F/B setting	0	0	1	1	0
	Bit [5:3]	Physical Connector ID	0	0	0	0	0
	Bit [7:6]	Reserved	0	0	0	0	0

E.4.2 Hardware Configuration Table

Table E-5 RAGE 128 / RAGE THEATER board Hardware Configuration table

Offset (bytes)	Field	Description	Default (without MM)			Default (with MM)				
			RAGE PRO / XL/XC	LT PRO	RAGE Mobility	RAGE 128	RAGE PRO	LT PRO / XL	RAGE Mobility	RAGE 128
0 -3		Table Signature	$ATI	$ATI	$ATI	$ATI	$ATI	$ATI	$ATI	$ATI
4		Revision	2	2	2	2	2	2	2	2
5		Table size	10	10	10	10	10	10	10	01
6	Bit [3:0]	I2C_Type	2	0	15	2	2	3	3	2
	Bit [7:4]	Reserved	0	0	0	0	0	0	0	0
7	Bit [3:0]	TVOut Support	0	3	3	2	0	3	3	2
	Bit [6:4]	Crystal Frequency	0	0	0	1	0	0	0	1
	Bit [7]	Impact TV data Port	0	0	0	0	0	0	0	0
8	Bit [0]	AMC/DVS0 Port	1	0	0	1	1	0	0	1
	Bit [1]	Zoom Video Port	0	1	1	0	0	1	1	0
	Bit [2]	AMC/DVS1 Port	0	0	0	0	0	0	0	0
	Bit [3]	VIP 16 bit Port	0	0	0	0	0	0	0	0
	Bit [7:4]	Reserved	0	0	0	0	0	0	0	0
9	Bit [3:0]	Host Port Configuration	0	0	0	0	0	0	0	0
	Bit [7:4]	Reserved	0	0	0	0	0	0	0	0

Appendix F
CCE Command Packets

F.1 Scope

This section provides a summary of the CCE command packets. In CCE mode, programming the RAGE 128 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.

F.2 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.

4.1 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.

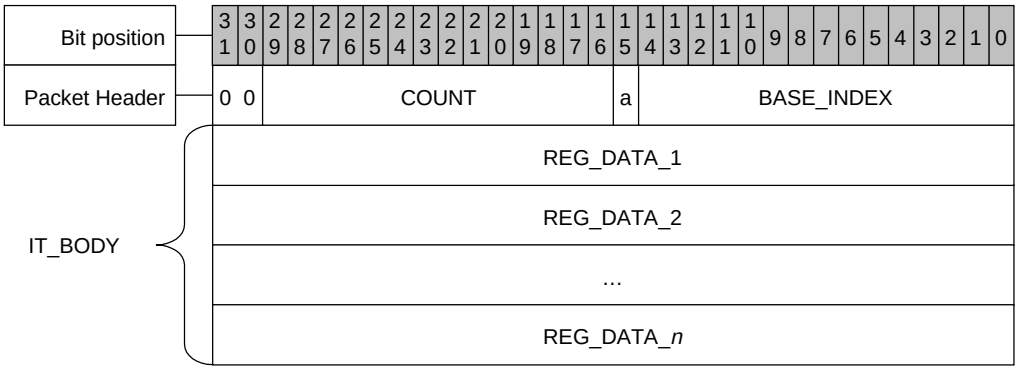

Figure 4-1. Type 0 CCE Packet

Table 4-3 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 4-4 Header Fields for a Type-0 CCE Packet

Bit(s)	Field Name	Description
10:0	BASE_INDEX	Memory-mapped address (in units of DWORDs) of the first register to be written.
14:11	Reserved	N/A
15	ONE_REG_WR	0 - Write the data to N consecutive registers. 1 - Write all the data to the same register.

Table 4-4 Header Fields for a Type-0 CCE Packet

Bit(s)	Field Name	Description
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 0.

Table 4-5 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 the RAGE 128 Register Reference for details.

F.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.

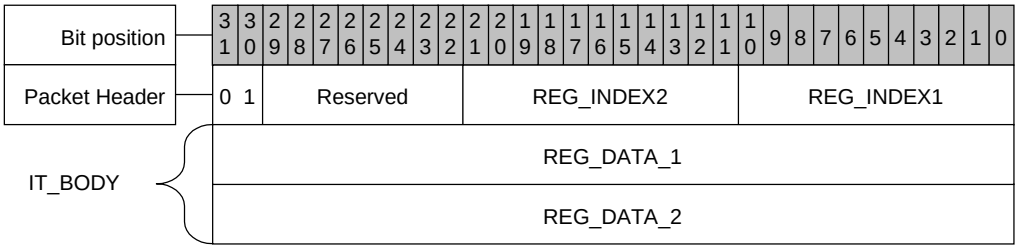

Figure 4-2. Type 1 CCE Packet

Table 4-6 Format for a Type 1 CCE Packet

Ordinal	Field Name
1	[HEADER]
2	[REG_DATA_1]
3	[REG_DATA_2]

Table 4-7 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	N/A
31:30.	TYPE	Packet identifier. It should be 1.

Table 4-8 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 the RAGE 128 Register Reference for details.

F.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.

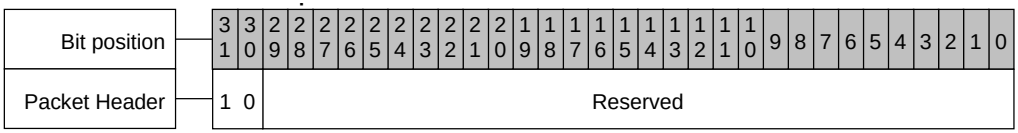

Figure 4-3. Type 2 CCE Packet

Table 4-9 Format of a Type 2 CCE Packet

Ordinal	Field Name
1	[HEADER]

Table 4-10 Header Fields of a Type 2 CCE Packet

Bit(s)	Field Name	Description
29:0	reserved	N/A
31:30	TYPE	Packet identifier. It should be 2.

F.5 Type 3 CCE Packet

Purpose: For carrying out the operation indicated by field IT_OPCODE.

Type-3 packets have 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 the 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.

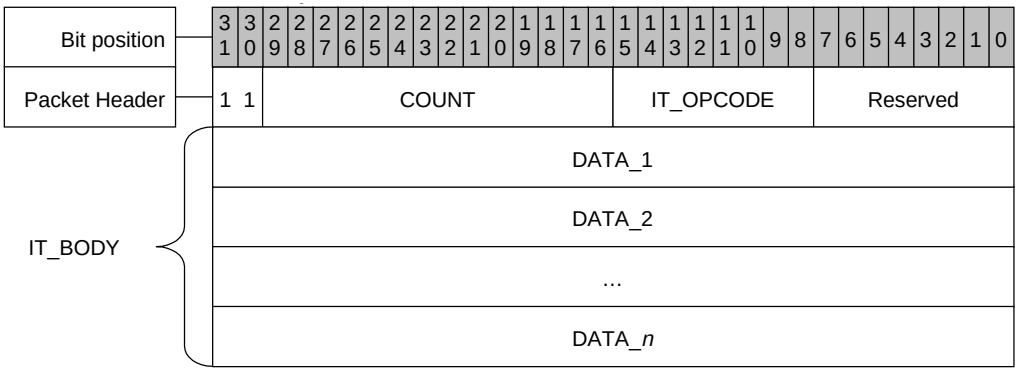

Figure 4-4. Type 3 CCE Packet

Table 4-11 Formal for a Type 3 CCE Packet

Ordinal	Field Name
1	[HEADER]
2	{IT_BODY} Information Body

Table 4-12 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.

Table 4-12 Header Fields for a Type 3 CCE Packet

Bit(s)	Field Name	Description
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.

F.6 Summary of the CEE Packets

Table 4-13 Summary of the CEE Packets

Packet Name	IT_ OPCODE	Description
NOP	0x10	Skip N DWORDs to get to the next packet.
PAINT	0x91	Paint a number of rectangles with a color brush.
SMALL_TEXT	0x93	Draw a string of small characters on the screen.
HOSTDATA_BLT	0x94	Draw a string of large characters on the screen, or copy a number of bitmaps to the video memory.
POLYLINE	0x95	Draw a polyline (lines connected with their ends).
SCALE	0x96	Scale the given rectangular screen area by a factor. This packet is used by both 2D and 3D operations.
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 3D graphics.
POLYSCANLINES	0x98	Draw polyscanlines or scanlines.
NEXTCHAR	0x19	Print a character at a given screen location using the default foreground and background colors.
PLY_NEXTSCAN	0x1D	Draw polyscanlines using current settings.
SET_SCISSORS	0x1E	Set up scissors.
SET_MODE_24BPP	0x1F	Set the 24bpp mode flag.
PAINT_MULTI	0x9A	Paint a number of rectangles on the screen with one color. The difference between this function and PAINT is the representation of parameters.
BITBLT_MULTI	0x9B	Copy a number of source rectangles to destination rectangles of the screen respectively.
TRANS_BITBLT	0x9C	2D transparent bitblt operation.
3D_RNDR_GEN_INDX_PRIM	0x23	Draw 3D objects using the vertex walker.
3D_RNDR_GEN_PRIM	0x25	Draw 3D points, lines, triangles, strips, fans using the ring buffer.
LOAD_PALETTE	0x2C	Load a palette onto RAGE 128 for 2D scaling.

Table 4-13 Summary of the CEE Packets (Continued)

Packet Name	IT_OPCODE	Description
PURGE	0x2D	Purge the pixel cache.
NEXT_VERTEX_BUNDLE	0x2E	Add more vertices to the end of a 3D_RNDR_GEN_INDX_PRIM packet.

F.7 2D Packets

Table 4-14 Information Body (IT_BODY) of 2-D packets

Ordinal	Field Name
1	{SETTINGS}
2	{DATA_BLOCK}

Table 4-15 SETTINGS FIELD for the IT_BODY

Ordinal	Field Name
1	[GUI_CONTROL]
2	{SETUP_BODY}

GUI_CONTROL

This subfield will be used to setup the RAGE 128 (register **DP_GUI_MASTER_CNTL**), and it also decides the content of **SETTINGS.SETUP_BODY**.

Table 4-16 GUI_CONTROL Subfield for the SETTINGS Field

Bit(s)	Field Name	Description
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.
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 point to the off screen area of 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.

Table 4-16 GUI_CONTROL Subfield for the SETTINGS Field (Continued)

Bit(s)	Field Name	Description
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.
11:8	DST_TYPE	The pixel type of the destination. 0, 1 - (reserved) 2 - 8 bpp pseudocolor 3 - 16 bpp aRGB 1555 4 - 16 bpp RGB 565 5 - 24 bpp RGB 6 - 32 bpp aRGB 8888 7 - 8 bpp RGB 332 8 - Y8 greyscale 9 - RGB8 greyscale (8 bit intensity, duplicated for all 3 channels. Green channel is used on writes) 10 - (reserved) 11 - YUV 422 packed (VYUY) 12 - YUV 422 packed (YVYU) 13 - (reserved) 14 - aYUV 444 (8:8:8:8) 15 - aRGB4444 (intermediate format only. Not understood by the Display Controller) Note: choices 7-15 are only valid in 3D mode
13:12	SRC_TYPE	The field indicates the pixel type of blitting source. 0 - The source data type is mono opaque, and the fore- and back-ground colors need to be redefined. 1 - The source data type is mono transparent, and only the foreground color needs to be redefined. 2 - Reserved. 3 - The source pixel type is the same as that given in field DST_TYPE.
14	PIX_ORDER	The bit decides the order of bits (or pixels) in DWORD to be consumed. Only applicable to monochrome mode. 0 - Bits to be consumed from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). 1 - Bits to be consumed from LSB to MSB.

Table 4-16 GUI_CONTROL Subfield for the SETTINGS Field (Continued)

Bit(s)	Field Name	Description
15	COLOR_CONVT	YUV to RGB conversion temperature 0 - Red at 6500K, GB at 9300K 1 - RGB at 9300K
23:16	WIN31_ROP	This field tells the GUI engine how the raster operation is to be carried out. The code of this field follows the ROP3 code defined by Microsoft. See Windows 3.1 DDK for reference.
26:24	SRC_LOAD	The field indicates where the source data come from. 0, 1 - Reserved 2 - loaded from video memory (rectangular trajectory) 3 - loaded through the HOSTDATA registers (linear trajectory) 4 - loaded through the HOSTDATA registers (linear trajectory and 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	GMC_3D_FCN_EN(Reserved)	0 - clear SCALE_3D_FCN,Z_EN and STENCIL_EN fields 1 - leave SCALE_3D_FCN,Z_EN, and STENCIL_EN fields alone
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
29	GMC_AUX_CLIP_DIS	0 - No change to AUXn_SC_ENB 1 - clear all AUXn_SC_ENB bits to 0
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 SETTINGS.SETUP_BODY. 0 - No such a field in SETTINGS.SETUP_BODY. 1 - There is a field in SETTINGS.SETUP_BODY.

SETUP_BODY

This field may contain the following subfields. Their presence depends on the bits 0-7 of **`SETTINGS.GUI_CONTROL`**.

Table F-1 SETUP_BODY Subfield for the SETTINGS Field

Ordinal	Field Name	Description
1	[SRC_PITCH_ OFFSET]	[20:0] - offset address in units of 32 bytes. This address points to the memory reference location of the source rectangle. [30:21] - pitch size (in units of 8 pixels) of the source. Note that in monochrome mode the source pitch must be a multiple of 128 pixels. In 8bpp mode, source pitch must be a multiple of 16 pixels. [31] - this is a flag bit that indicates whether the source memory is in “tiled” format. 1: Tiled format; 0: not a tiled format.
2	[DST_PITCH_ OFFSET]	[20:0] - offset address in the unit of 32 bytes. This address points to the memory reference location of the destination rectangle. [30:21] - pitch size (in unit of 8 pixels) of the destination. Note that in monochrome mode the destination pitch must be a multiple of 128 pixels. In 8bpp mode, source pitch must be a multiple of 16 pixels. [31] - this is a flag bit that indicates whether the destination memory is in “tiled” format. 1: Tiled format; 0: not a tiled format.
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.

Table F-1 SETUP_BODY Subfield for the SETTINGS Field (Continued)

Ordinal	Field Name	Description
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.

Table F-2 SETTINGS for SETUP_BODY.BRUSH_PACKET

BRUSH _TYPE	Description of the brush	Packet size	Packet content
0	A 8 x 8 mono pattern with the foreground and background colors specified in the packet. Here the matrix is represented in the format *column-by-row.*	4 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
1	A 8 x 8 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted.	3 DWORDs	[FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
2	A 8 x 1 (8 columns by 1 row) mono pattern with the foreground and background colors specified in the packet.	3 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1]
3	A 8 x 1 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted.	2 DWORDs	[FRGRD_COLOR] [MONO_BMP_1]
4	A 1 x 8 mono pattern with the foreground and background colors specified in the packet.	3 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1]
5	A 1 x 8 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted.	2 DWORDs	[FRGRD_COLOR] [MONO_BMP_1]
6	A 32 x 1 mono pattern with the foreground and background colors specified in the packet.	3 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1]
7	A 32x1 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted.	2 DWORDs	[FRGRD_COLOR] [MONO_BMP_1]

Table F-2 SETTINGS for SETUP_BODY.BRUSH_PACKET (Continued)

BRUSH_TYPE	Description of the brush	Packet size	Packet content
8	A 32x32 mono pattern with the foreground and background colors specified in the packet.	34 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1] … [MONO_BMP_32]
9	A 32x32 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted.	33 DWORDs	[FRGRD_COLOR] [MONO_BMP_1] … [MONO_BMP_32]
10	A 8x8 color pattern. The pixel type is given by the 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	A 8x1 color pattern. The pixel type is given by field `SETTINGS.GUI_CONTROL.DST_TYPE`	2* N DWORDs	[COLOR_BMP_1] [COLOR_BMP_2] … [COLOR_BMP_2*N]
12	A 1x8 color pattern. The pixel type is given by field `SETTINGS.GUI_CONTROL.DST_TYPE`	2* N DWORDs	[COLOR_BMP_1] [COLOR_BMP_2] … [COLOR_BMP_2*N]
13	Use the color specified in the packet as the solid (plain) color for the brush, i.e. a color brush without a pattern.	1 DWORD	[FRGRD_COLOR]
14	reserved	not applicable	
15	No brush used.	0	

Table F-3 Pixel size in bytes

SETTINGS for GUI_CONTROL.DST_TYPE	N
0-1	not applicable
2	1
3	2

Table F-3 Pixel size in bytes (Continued)

SETTINGS for GUI_CONTROL.DST_TYPE	N
4	2
5	3
6	4
7	1
8-15	not applicable

Table F-4 Contents of Brush Packet

Field Name	Description
[FRGRD_COLOR]	The foreground color of the text in RGBQUAD format. [7:0] - intensity of Blue; [15:8] - intensity of Green [23:16] - intensity of Red. [31:25] - reserved.
[BKGRD_COLOR]	The background color of the text in RGBQUAD format. [7:0] - intensity of Blue; [15:8] - intensity of Green [23:16] - intensity of Red. [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 color 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.

F.8 NOP

Packet Type: 2D

Purpose: For skipping a number of DWORDs to get to the next packet.

Table F-5 format for NOP

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

DATA BLOCK for NOP

This field may consists of a number of DWORDs, and the content may be anything.

F.9 PAINT

Packet Type: 2D

Purpose: For painting a number of rectangles with a color brush.

Table F-6 Format for PAINT

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-7 DATA BLOCK for PAINT

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.

F.10 SMALL_TEXT

Packet Type: 2D

Purpose: For printing a string of characters on the screen in the format of the bit-packed Small Glyph.

Table F-8 Format for SMALL_TEXT

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-9 DATA_BLOCK for SMALL_TEXT

Ordinal	Field Name	Description
1	[FRGD_COLOUR]	The foreground color 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.

Table F-10 DATA BLOCK for 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: [31:25] - height of the character bitmap.
2	[RASTER_1]	The 1st DWORD of the mono bitmap data.

Table F-10 DATA BLOCK for SMALLCHAR_x (Continued)

Ordinal	Field Name	Description
…		
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, the starting position of text is at (*bas_x, bas_y*). The actual sizes of characters 'b', 'o' and 'y' respectively are 4×8, 4×5 and 6×9. Therefore, the related 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 $\Delta y_3 = 5$

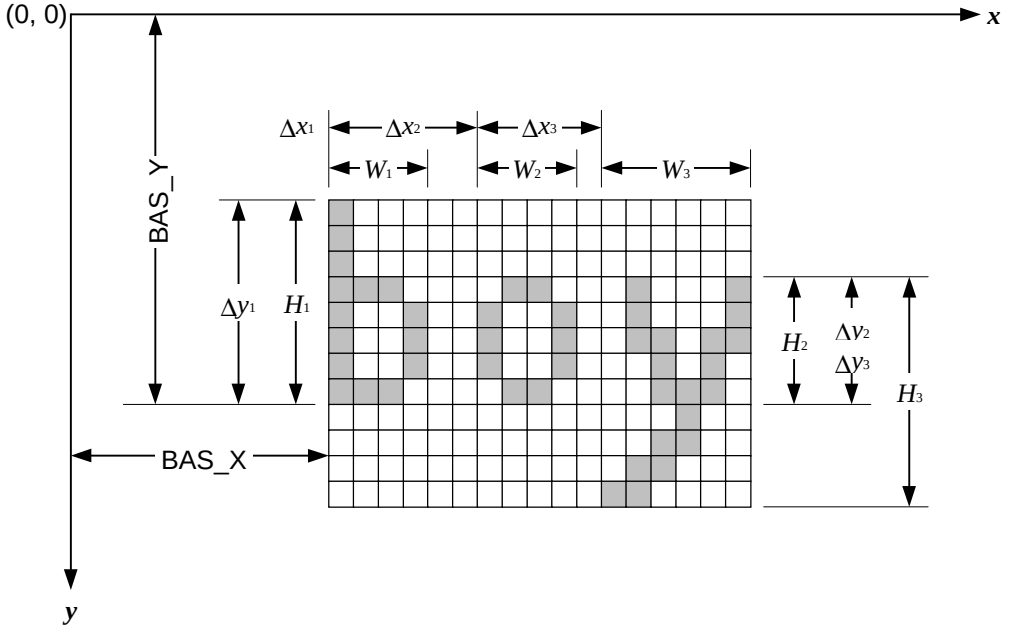

Figure 4-5. Drawing Small Text

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.

F.11 HOSTDATA_BLT

Packet Type: 2D

Purpose: For copying 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.

Table F-11 Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-12 DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOUR]	Foreground color in RGBQUAD format. For mono-to-color 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 color in RGBQUAD format. For mono-to color 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.

Table F-13 DATA BLOCK for 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.
3	[NUMBER]	The number of DWORDs in the bitmap. It should be *m* in this case.

Table F-13 DATA BLOCK for BIGCHAR_x (Continued)

Ordinal	Field Name	Description
4	[RASTER_1]	The 1st DWORD of the mono bitmap data.
…		
m+3	[RASTER_m]	The m-th DWORD of the mono bitmap data.

F.12 POLYLINE

Packet Type: 2D

Purpose: For drawing a polyline specified by a set of coordinates (x_0, y_0), (x_1, y_1), …, (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.

Table F-14 Format for POLYLINE

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-15 DATA_BLOCK for POLYLINE

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.
…		
n+1	[Yn | Xn]	The ending coordinate of the polyline.

F.13 SCALE

Packet Type: 2D

Purpose: For stretch or compressing the texture pattern stored in a bitmap, and put the scaled pattern to the destination area in the video memory.

Table F-16 Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-17 DATA_BLOCK for SCALE

Ordinal	Field Name	Description
1	[MISC_3D_STATE]	This field specifies the operation to be carried out. See following for details.
2	[TEX_CNTL]	The bits of this field enable or disable the operations (alpha and fog) specified in field MISC_3D_STATE. See below for details.
3	[TEX_COMB_CTL]	This field corresponds to register PRIMARY_TEXTURE_COMBINE_CNTL. See below for details.
4	[SCALE_DATATYPE]	See below.
5	[SCALE_OFFSET]	[25:0] - Offset of texture in video memory. (Alias to TEX_0_OFFSET) [31:30] - Texture mapping mode 0 - Texture surface is not tiled. 1 - Texture surface is tiled by the host application. 2, 3 - Texture surface is stored in a tiled surface.
6	[SCALE_PITCH]	See below.
7	(Reserved)	This field should be set to 0.
8	[SCALE_X_INC]	Scaling factor in x-direction. Its value is SRC_W/DST_W, where SRC_W and DST_W denote the widths of the source and destination images respectively. [19:16] - Integer part of the factor. [15:4] - Fractional part of the factor. [Other bits] - reserved.

Table F-17 DATA_BLOCK for SCALE (Continued)

Ordinal	Field Name	Description
9	[SCALE_Y_INC]	Scaling factor in y-direction. Its value is SRC_H/DST_H, where SRC_H and DST_H denote the heights of the source and destination images respectively. [19:16] - Integer part of the factor. [15:4] - Fractional part of the factor. [Other bits] - reserved.
10	[DST_X | DST_Y]	The coordinate of the top-left corner of the destination bitmap. DST_X: [29:16] - x-coordinate expressed in a signed integer. DST_Y: [13:0] - y-coordinate expressed in a signed integer.
11	[DST_H | DST_W]	The width and height of the destination bitmap expressed in unsigned integers. DST_W: [13:0] - width. DST_H: [29:16] - height

Table F-18 DATA BLOCK for MISC_3D_STATE

Bit(s)	Field Name	Description
7:0	REF_ALPHA	Reference Alpha value for alpha testing when the test is enabled.
9:8	SCALE_3D_FCN	Set to 1 to enable the scaling operation.
11:10	(Reserved)	
13:12	ALPHA_COMB_FCN	This field defines how the resultant Alpha is computed in Alpha blending. Let Alpha, Src and Dst denote the resultant, the source and destination alphas respectively, where Src=Ws*Sa and Dst=Wd*Sa. The combining operation can be written as Alpha=Src OP Dst, where operator OP is defined as: 0 - Add operation, and the resultant Alpha is limited to range [0, 1] 1 - Add operation 2 - Subtract operation and Alpha = Alpha MOD 1.0 3 - Subtract operation. Note: Generally, Alpha is defined in range [0, 1]. However, it may be represented in an integer ranging from 0 to 255. In this case, the right-operand of MOD should be 256.
14	FOG_TABLE_EN	0 - FOG_VERTEX 1 - FOG_TABLE
15	(Reserved)	

Table F-18 DATA BLOCK for MISC_3D_STATE (Continued)

Bit(s)	Field Name	Description
19:16	ALPHA_BLND_SRC	This field select the weighting factors (Wa, Wr, Wg, Wb) for the source pixel in the blending operation. Assume that source and destination pixels are denoted respectively as (Sa, Sr, Sg, Sb) and (Da, Dr, Dg, Db). 0 - BLEND_ZERO (Wa=Wr=Wg=Wb=0) 1 - BLEND_ONE (Wa=Wr=Wg=Wb=1) 2 - BLEND_SRCCOLOUR (not applicable) 3 - BLEND_INVSRCCOLOUR (not applicable) 4 - BLEND_SRCALPHA (Wa=Wr=Wg=Wb=Sa) 5 - BLEND_INVSRCALPHA (Wa=Wr=Wg=Wb=1-Sa) 6 - BLEND_DESTALPHA (Wa=Wr=Wg=Wb=Da) 7 - BLEND_INVDESTALPHA (Wa=Wr=Wg=Wb=1-Da) 8 - BLEND_DESTCOLOUR (Wa=Da, Wr=Dr, Wg=Dg, Wb=Db) 9 - BLEND_INVDESTCOLOUR (Wa=1-Da, Wr =1-Dr, Wg=1-Dg, Wb=1-Db) 10 - BLEND_SRCALPHASAT (Wa=1, Wr=Wg=Wb=min(Sa, 1-Da)) 11 - BLRND_BOTHSRCALPHA (Wa=Wr=Wg=Wb=Sa, and assign factor 1-Sa to each weighting factor of the destination pixel) 12 - BLEND_BOTHINVSRCALPHA (Wa=Wr=Wg=Wb=1-Sa, and assign factor Sa to each weighting factor of the destination pixel) 13-15 - Reserved

Table F-18 DATA BLOCK for MISC_3D_STATE (Continued)

Bit(s)	Field Name	Description
23:20	ALPHA_BLND_DST	This field select the weighting factors (Wa, Wr, Wg, Wb) for the destination pixel in the blending operation. Assume that source and destination pixels are denoted respectively as (Sa, Sr, Sg, Sb) and (Da, Dr, Dg, Db). 0 - BLEND_ZERO (Wa=Wr=Wg=Wb=0) 1 - BLEND_ONE (Wa=Wr=Wg=Wb=1) 2 - BLEND_SRCCOLOUR (Wa=Sa, Wr=Sr, Wg=Sg, Wb=Sb) 3 - BLEND_INVSRCCOLOUR (Wa=1-Sa, Wr=1-Sr, Wg=1-Sg, Wb=1-Sb) 4 - BLEND_SRCALPHA (Wa=Wr=Wg=Wb=Sa) 5 - BLEND_INVSRCALPHA (Wa=Wr=Wg=Wb=1-Sa) 6 - BLEND_DESTALPHA (Wa=Wr=Wg=Wb=Da) 7 - BLEND_INVDESTALPHA (Wa=Wr=Wg=Wb=1-Da) 8 - BLEND_DESTCOLOUR (not applicable) 9 - BLEND_INVDESTCOLOUR (not applicable) 10 - BLEND_SRCALPHASAT (not applicable) 11-15 - Reserved
26:24	ALPHA_TST_OP	Specifies the acceptance criterion in comparing the alpha component of the new pixel against the reference alpha stored at field REF_ALPHA. The test form is: if (NEWa OPCODE REFa) then {Accept New Pixel}. The OPCODE is defined as: 0 - The test always fails, i.e. the new pixel is always rejected. 1 - CMP_LESS (Less than) 2 - CMP_EQUAL (Equal to) 3 - CMP_LESSEQUAL (Less than or equal to) 4 - CMP_GREATER (Greater than) 5 - CMP_NOTEQUAL (Not equal to) 6 - CMP_ALWAYS (The new pixel is always accepted.)
29:27	(Reserved)	

Table F-18 DATA BLOCK for MISC_3D_STATE (Continued)

Bit(s)	Field Name	Description
31:30	CLR_CMP_FCN_3D	NOTE: This type of color keying is available. when using the old texture interface (execute buffer, DrawPrimitve etc.). When the new multi-texture API is used, then the APP must use the texel alpha. This is what MS is advocating ALIASED to CLR_CMP_CNTL_3D) bits 1:0 0 - False (always write the source to the destination) 1 - True (never write the source to the destination) 2 - Texel != CLR_CMP_CLR_3D (Write to the destination if texel is equal to the color stored in register CLR_CMP_CLR_3D). 3 - Texel = CLR_CMP_CLR_3D (Write to the destination if texel is NOT equal to the color stored in register CLR_CMP_CLR_3D).

Table F-19 DATA BLOCK for TEX_CNTL

Bit(s)	Field Name	Description
6:0	NIL1	Set to constant 0.
7	FOG_EN	0 - Disable fogging. 1 - Enable fogging.
8	DITHER_EN	0 - Disable dithering. 1 - Enable dithering.
9	ALPHA_EN	0 - Disable Alpha blending. 1 - Enable Alpha blending.
10	ALPHA_TST_EN	0 - Disable Alpha testing. 1 - Enable Alpha testing.
31:11	NIL2	Set to constant 0.

Table F-20 DATA_BLOCK for SCALE_DATATYPE

Bit(s)	Field Name	Description
3:0	SRC_DATATYPE	This field specifies the pixel type of the source bitmap. 0 - 2 bpp VQ (Not supported in the initial part) 1 - 4 bpp pseudocolor. Upper 4 bits of the byte are unused. 2 - 8 bpp pseudocolor 3 - 16 bpp aRGB 1555 4 - 16 bpp RGB 565 5 - (Reserved) 6 - 32 bpp aRGB 8888 7 - 8 bpp RGB 332 8 - Y8 greyscale 9 - RGB8 greyscale (8 bit intensity, duplicated for all 4 channels. Green channel is used on writes) 10 - 16 bpp a:pseudocolor (8:8) 11 - YUV 422 packed (VYUY) 12 - YUV 422 packed (YVYU) 13 - 16 bpp a:RGB8 greyscale (8:8) 14 - aYUV 444 (8:8:8:8) 15 - aRGB4444
7:4	PALETTE	This field select a palette for pseudo color textures. The interpretation of the code depends on field SRC_DATATYPE. If SRC_DATATYPE = 1 (4 bpp color), this field selects 1 of 16 possible palettes stored in the system. If SRC_DATATYPE = 2 (8 bpp pseudo color/VQ textures), this field selects one of the following: 0 - either of 2 palettes 1 - Palette 1 2 - Palette 2 3-15 - (Reserved)
31:8	Reserved	

Table F-21 DATA BLOCK for SCALE_PITCH

Bit(s)	Field Name	Description
8:0	SCALE_PITCH	Pitch in units of 8 pixels of the source data for RGB and packed modes. The pitch is required to be programmed so that all source lines are an integer number of QWORDs
16:9	Reserved	

Table F-21 DATA BLOCK for SCALE_PITCH (Continued)

Bit(s)	Field Name	Description
20:17	SCALE_OFFSET_PTR	This field points to one of registers TEX_x_OFFSET, where x can be one of values 0, 1, …, 10. A TEX_x_OFFSET points to the address of a texture stored in the video memory. This field should be set to zero (0) for scaling. 0 - Use the texture pointed to by TEX_0_OFFSET as the source. 1 - Use the texture pointed to by TEX_1_OFFSET as the source. … 10 - Use the texture pointed to by TEX_10_OFFSET as the source. 11 - Use the texture pointed to by SEC_TEX_0_OFFSET as the source. 12 - Use the texture pointed to by SEC_TEX_1_OFFSET as the source. … 15 - Use the texture pointed to by SEC_TEX_14_OFFSET as the source.
29:21	Reserved	
31:30	SCALE_PITCH_ADJ	Indicate whether SCALE_PITCH should be adjusted prior to use. 0 - no adjustment on SCALE_PITCH 1 - multiply SCALE_PITCH by 2 prior to use 2 - multiply SCALE_PITCH by 4 prior to use 3 - (Reserved)

Table F-22 DATA_BLOCK.TEX_COMB_CTL

Bit(s)	Field Name	Description
3:0	COMB_FCN	Specifies the function used to modify the color component of primary texels during the texture combine stage. 0 - Disable. Output color is: texture color (or Interpolator Color if shading) 1 - Copy. Output color is COLOR_FACTOR 2 - Copy Input. Output color is INPUT_FACTOR 3 - Modulate. Output color is COLOR_FACTOR*INPUT_FACTOR 4 - Modulate*2. Output color is COLOR_FACTOR*INPUT_FACTOR*2 5 - Modulate*4. Output color is COLOR_FACTOR*INPUT_FACTOR*4 6 - Add. Output color is COLOR_FACTOR + INPUT FACTOR 7 - Add Signed. Output color is COLOR_FACTOR + INPUT FACTOR - 128 8 - Blend Vertex. Output color is (COLOR_FACTOR*interpolator alpha) + (INPUT_FACTOR*(1 - interpolator alpha)). 9 - Blend_Texture. Output color is (COLOR_FACTOR*primary texel alpha) + (INPUT_FACTOR*(1 - primary texel alpha)). 10 - Blend Constant. Output color is (COLOR_FACTOR*CONSTANT_ALPHA) + (INPUT_FACTOR*(1 -CONSTANT_ALPHA)). 11 - Blend Pre Multiply. Output color is COLOR FACTOR + (INPUT_FACTOR*(1 - primary texel alpha)). 12 - Blend_Previous. Output color is (COLOR_FACTOR*primary texel alpha) + (INPUT_FACTOR*(1 - primary texel alpha)). 13 - Blend Pre Multiply Inverse. Output color is COLOR FACTOR + (INPUT_FACTOR*(primary texel alpha)). 14 - Add Signed2X. Output color is (COLOR_FACTOR + INPUT_FACTOR - 128)*2 15 - Blend Constant Color. Output color is (COLOR_FACTOR*CONSTANT_COLOR) + (INPUT_FACTOR*(1 -CONSTANT_COLOR)).
7:4	COLOR_FACTOR	0-3 - (Reserved) 4 - Texture Color (or Interpolator Color if shading) 5 - ~Texture Color (or ~Interpolator Color if shading) 6 - Texture Alpha (or Interpolator Alpha if shading) 7 - ~Texture Alpha (or ~Interpolator Alpha if shading) 8-15 - (Reserved)
9:8	Reserved	
13:10	INPUT_FACTOR	0-1 - (Reserved) 2 - CONSTANT_COLOR 3 - CONSTANT_ALPHA 4 - Interpolator Color 5 - Interpolator Alpha 6-15 - (Reserved)

Table F-22 DATA_BLOCK.TEX_COMB_CTL

Bit(s)	Field Name	Description
17:14	COMB_FCN_ALPHA	Specifies the function used to modify the alpha component of primary texels during the texture combine stage. 0 - Disable. Output color is primary texture alpha (or Interpolator Alpha if shading) 1 - Copy. Output color is ALPHA_FACTOR 2 - Copy Input. Output color is INPUT_FACTOR_ALPHA 3 - Modulate. Output color is ALPHA_FACTOR*INPUT_FACTOR_ALPHA 4 - Modulate*2. Output color is ALPHA_FACTOR*INPUT_FACTOR_ALPHA*2 5 - Modulate*4. Output color is ALPHA_FACTOR*INPUT_FACTOR_ALPHA*4 6 - Add. Output color is ALPHA_FACTOR + INPUT_FACTOR_ALPHA 7 - Add Signed. Output color is ALPHA_FACTOR + INPUT_FACTOR_ALPHA - 128 8-13 - (Reserved) 14 - Add Signed2x. Output color is (ALPHA_FACTOR + INPUT_FACTOR_ALPHA - 128)*2 15 - (Reserved)
21:18	ALPHA_FACTOR	0-5 - (Reserved) 6 - Texture Alpha (or Interpolator Alpha if shading) 7 - ~Texture Alpha (or ~Interpolator Alpha if shading) 8-15 - (Reserved)
24:22	Reserved	
27:25	INPUT_FACTOR_ALPHA	0 - (Reserved) 1 - CONSTANT_ALPHA 2 - Interpolator Alpha 3 - (Reserved) 4 - (Reserved) 5-8 - (Reserved)
31:28	Reserved	

F.14 TRANS_SCALE

Packet Type: 2D

Purpose: For stretching or compressing the texture pattern (or bitmap), and put the scaled pattern to the destination area in the video memory. The scaled pattern may be transparent to the background of the destination according to some conditions set out by the user.

Table F-23 Format for TRANS_SCALE

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-24 DATA_BLOCK for TRANS_SCALE

Ordinal	Field Name	Description
1	[CLR_CMP_CNTL]	This field determines how the transparent scaled blitting is done. See below for details.
2	[SRC_REF_CLR]	Source reference color in the RGBQUAD format. This is the color to be stripped off from the source.
3	[DST_REF_CLR]	Destination reference color in the RGBQUAD format. This is the color to be preserved at the destination.
4	[MISC_3D_STATE]	This field specifies the operation to be carried out. See section B.2.2.6 SCALE for details.
5	[TEX_CNTL]	The bits of this field enable or disable the operations (alpha and fog) specified in field MISC_3D_STATE. See section B.2.2.6 SCALE for details.
6	[TEX_COMB_CTL]	This field corresponds to register PRIMARY_TEXTURE_COMBINE_CNTL. See section B.2.2.6 SCALE for details.
7	[SCALE_DATATYPE]	See section B.2.2.6 SCALE for details.
8	[SCALE_OFFSET]	[25:0] - Byte pointer to the smallest texture map. [31:30] - Texture mapping mode 0 - Texture surface is not tiled. 1 - Texture surface is tiled by the host application. 2,3 - Texture surface is stored in a tiled surface.

Table F-24 DATA_BLOCK for TRANS_SCALE (Continued)

Ordinal	Field Name	Description
9	[SCALE_PITCH]	See section B.2.2.6 SCALE for details.
10	(Reserved)	This field should be set to zero (0).
11	[SCALE_X_INC]	Advancing step size (in units of pixels) in x-direction for the source bitmap. [19:4] - X accumulator increment, 12 bits fractional, 4 bits unsigned integer. For packed or planar YUV pixels, this applies only to the Y values. [Other bits] - reserved.
12	[SCALE_Y_INC]	Advancing step size (in units of pixels) in y-direction for the source bitmap. [19:4] - Y accumulator increment, 12 bits fractional, 4 bits unsigned integer. [Other bits] - reserved.
13	[DST_X | DST_Y]	The coordinate of the top-left corner of the destination bitmap. DST_X: [29:16] - x-coordinate expressed in a signed integer. DST_Y: [13:0] - y-coordinate expressed in a signed integer.
14	[DST_H | DST_W]	The width and height of the destination bitmap, expressed in unsigned integers. DST_W: [13:0] - width. DST_H: [29:16] - height

CLR_CMP_CNTL

This field controls how the source pixels are written to the destination, depending on the source and destination reference colors and comparison settings. The source pixels may be filtered against the source reference color, and the destination pixels with a specific color may be preserved according to field CLR_CMP_DST.

Table F-25 DATA_BLOCK for CLR_CMP_CNTL

Bit(s)	Field Name	Description
2:0	CLR_CMP_SRC	Strip off the source reference color from the source pixels. 0 - Do not strip off source pixels. All source pixels are written to the destination. 1 - Block the blitting source. No source pixel is written to the destination. 2, 3 - Reserved. 4 - The source pixels whose color is equal to the reference color are written to the destination. 5 - The source pixels whose color is NOT equal to the reference color are written to the destination. 6 - Reserved. 7 - 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, destPixel = srcPixel XOR foregrndColor if srcPixel is equal to the foreground color of a mono bitmap, specifically text. This is referred to as flipping sometimes.
7:3	Reserved	
10:8	CLR_CMP_DST	Preserve pixels at the destination. 0 - Do not preserve the destination pixels. All pixels from the source are written to the destination. 1 - Preserve all the destination pixels. No source pixel is written to the destination. 2, 3 - Reserved. 4 - The destination pixels whose color is equal to the reference color are preserved. No source pixel is written on top of the pixels. 5 - The destination pixels whose color is NOT equal to the reference color are preserved. 6, 7 - Reserved.
23:11	Reserved	
25:24	CMP_ENABLE	The bits controls what type of operation to be carried out. 0 - Enable function CLR_CMP_DST. 1 - Enable function CLR_CMP_SRC. 2 - Enable both CLR_CMP_SRC and CLR_CMP_DST. The final decision is based on the agreement between decisions made separately. 3 - Reserved.
31:26	Reserved	

F.15 POLYSCANLINES

Packet Type: 2D

Purpose: For drawing 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.

Table F-26 Format POLYSCANLINES

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-27 DATA_BLOCK for POLYSCANLINES

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.

Table F-28 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.
…		
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.

F.16 NEXTCHAR

Packet Type: 2D

Purpose: For printing a character at a given screen location using the default foreground and background colors.

Table F-29 Format for NEXTCHAR

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

Table F-30 DATA BLOCK for NEXTCHAR

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.

F.17 PAINT_MULTI

Packet Type: 2D

Purpose: For painting a number of rectangles on the screen with one color. The color used is specified in field SETTINGS while the location and geometry of the rectangles are specified in field DATA_BLOCK.

Table F-31 Format for PAINT_MULTI

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-32 DATA_BLOCK for PAINT_MULTI

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.

F.18 BITBLT_MULTI

Packet Type: 2D

Purpose: For copying 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.

Table F-33 Format for BITBLT_MULTI

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-34 DATA_BLOCK for BITBLT_MULTI

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 bottom-right 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 bottom-right 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.

F.19 TRANS_BITBLT

Packet Type: 2D

Purpose: For copying pixels from the source rectangle to the destination with transparency.

Table F-35 Format for TRANS_BITBLT

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

Table F-36 DATA BLOCK for TRANS_BITBLT

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 color in the RGBQUAD format. This is the color to be stripped off from the source.
3	[DST_REF_CLR]	Destination reference color in the RGBQUAD format. This is the color to be preserved at the destination.
4	[SRC_Y | SRC_X]	The coordinates of the top-left corner of the source bitmap. SRC_X: [15:0] - x-coordinate represented by a signed integer. SRC_Y: [31:16] - y-coordinate represented by a signed integer.
5	[DST_Y | DST_X]	The coordinates of the top-left corner of the destination bitmap. DST_X: [15:0] - x-coordinate expressed in a signed integer. DST_Y: [31:16] - y-coordinate expressed in a signed integer.
6	[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.

F.19.1 CLR_CMP_CNTL

This field controls how the source pixels are written to the destination, depending on the source and destination reference colors and comparison settings. The source pixels may be filtered against the source reference color, and the destination pixels with a specific color may be preserved according to field CLR_CMP_DST.

Table F-37 DATA BLOCK for CLR_CMP_CNTL

Bit(s)	Bit-Field Name	Description
2:0	CLR_CMP_SRC	Strip off the source reference color from the source pixels. 0 - Do not strip off source pixels. All source pixels are written to the destination. 1 - Block the blitting source. No source pixel is written to the destination. 2, 3 - reserved. 4 - The source pixels whose color is equal to the reference color are written to the destination. 5 - The source pixels whose color is NOT equal to the reference color are written to the destination. 6 - Reserved. 7 - 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, destPixel = srcPixel XOR foregrndColor if srcPixel is equal to the foreground color of a mono bitmap, specifically text. This is referred to as flipping sometimes.
7:3	Reserved	
10:8	CLR_CMP_DST	Preserve pixels at the destination. 0 - Do not preserve the destination pixels. All pixels from the source are written to the destination. 1 - Preserve all the destination pixels. No source pixel is written to the destination. 2, 3 - Reserved. 4 - The destination pixels whose color is equal to the reference color are preserved. No source pixel is written on top of the pixels. 5 - The destination pixels whose color is NOT equal to the reference color are preserved. 6, 7 - Reserved.
23:11	Reserved	
25:24	CMP_ENABLE	The bits controls what type of operation to be carried out. 0 - Enable function CLR_CMP_DST. 1 - Enable function CLR_CMP_SRC. 2 - Enable both CLR_CMP_SRC and CLR_CMP_DST. The final decision is based on the agreement between decisions made separately. 3 - Reserved.
31:26	Reserved	

F.20 PLY_NEXTSCAN

Packet Type: 2D

Purpose: For drawing a number of scanlines or polyscanlines using the current settings.

Table F-38 Format for PLY_NEXTSCAN

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 nth dash. END_n: [31:16] - the ending x-coordinate of the nth dash.

F.21 LOAD_PALETTE

Packet Type: 2D

Purpose: For setting up the 3D engine scaler and load a palette onto RAGE 128 for a consequent 2D scaling operation.

Table F-39 Format LOAD_PALETTE

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 1st entry of the palette. [7:0] - Blue component. [15:8] - Green component. [23:16] - Red component. [31:24] - Alpha component if applicable.
4	[COLOUR_2]	The 2nd entry of the palette. Bits are defined as above.
…		
n+2	[COLOUR_n]	The nth entry of the palette. n = 16 (4bpp) or 256 (8bpp)

F.22 SET_SCISSORS

Packet Type: 2D

Purpose: For setting the scissors to the given parameters.

Table F-40 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).

F.23 SET_MODE_24BPP

Packet Type: 2D

Purpose: For setting the 24bpp flag in the microcode engine.

Table F-41 Format for SET_MODE_24BPP

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[FLAG]	1 - Set the 24bpp flag in the microcode engine. 0 - Clear the 24bpp flag.

F.24 3D_RNDR_GEN_PRIM

Packet Type: 3D

Purpose: For rendering 3D primitives points, lines and triangles 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 VC_FORMAT that indicates how the vertex data blocks should be interpreted.
- A control field VC_CNTL that defines the type of primitive being drawn and the drawing method to be used.
- 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 VC_FORMAT. 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.

Table F-42 Format for 3D_RNDR_GEN_PRIM

Ordinal	Field Name
1	[HEADER]
2	[VC_FORMAT]
3	[VC_CNTL]
4	{FTLVERTEX_1}
…	
n+3	{FTLVERTEX_n}

F.24.1 VC_FORMAT

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.

Table F-43 VC_FORMAT

Bit(s)	Field Name	Description
0	RHW	1 - The FTLVERTEX block (defined below) contains a RHW field. 0 - The FTLVERTEX block does not contain such a field.
1	DIFFUSE_BGR	1 - The FTLVERTEX block contains a diffuse component of 3 colors (B,G,R) expressed in the float type. 0 - The FTLVERTEX block does not contain such a component (B,G,R).
2	DIFFUSE_A	1 - The FTLVERTEX block contains field DIFFUSE_ALPHA expressed in the float type. 0 - The FTLVERTEX block does not contain such a field.
3	DIFFUSE_ARGB	1 - The FTLVERTEX block contains field DIFFUSE_ARGB. 0 - The FTLVERTEX block does not contain such a field.
4	SPEC_BGR	1 - The FTLVERTEX block contains a specular component of 3 colors represented in the OpenGL format, i.e., each of colors B,G,R is represented by a number between 0.0 and 1.0. 0 - The FTLVERTEX block does not contain such a component.
5	SPEC_F	1 - The FTLVERTEX block contains field SPEC_FOG represented by a number between 0.0 and 1.0. 0 - The FTLVERTEX block does not contain such a field.
6	SPEC_FRGB	1 - The FTLVERTEX structure contains a combined fog/specular color component in the form of DWORD FRGB. 0 - The FTLVERTEX structure does not contain a combined fog/specular color component
7	S_T	1 - The FTLVERTEX block contains a set of texture coordinates (S, T). They are stored in two FLOATs. 0 - The FTLVERTEX block does not contain any texture coordinates (S, T).
8	S2_T2	1 - The FTLVERTEX block contains the second set of texture coordinates (S,T). They are stored in two FLOATs. Note that they are mainly used by D3D's multi-texture API. 0 - The FTLVERTEX block does not contain the second set of texture coordinates (S,T).
9	RHW2	1 - The FTLVERTEX block contains field floatRHW2 for the second set of texture coordinates. 0 - The FTLVERTEX block does not contain such a field.
31:10	Reserved	

F.24.2 VC_CNTL

This field corresponds to RAGE 128 register **PM4_VC_CNTL**. It selects the type of rendering primitive and the method of using the hardware.

Table F-44 VC_CNTL

Bit(s)	Field Name	Description
3:0	VC_PRIM_TYPE	The field defines the types of rendering primitive. 0 - Draw nothing. 1 - Draw a number of points. 2 - Draw a number of independent lines. 3 - Draw a number of polylines (line strips). 4 - Draw a number of independent triangles. 5 - Draw a triangle fan. 6 - Draw a triangle strip. 7 - Draw type-2 triangles. (for the vertex walker only) 8 -15 - Reserved
5:4	PRIM_WALK	This field defines the method of rendering. The object being drawn relates to the setting of VC_PRIM_TYPE. 0 - Reserved 1 - Draw the primitives pointed to by the given vertex indices using the vertex walker method. 2 - Draw all the primitives given in the vertex list using the vertex walker method. 3 - Use the ring buffer method to draw all the primitives. The following data consists of a number of FTLVERTEX structures.
15:6	Reserved	
31:16	NUM_VERTEX	The number of vertices in the packet. It should be n in this case.

F.24.3 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_VC_FORMAT**. 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.

Table F-45 FTLVERTEX

Field Name	Data Type	Description
[X_COORDINATE]	FLOAT	The x-coordinate of the vertex. Always present in FTLVERTEX.
[Y_COORDINATE]	FLOAT	The y-coordinate of the vertex. Always present in FTLVERTEX.
[Z_COORDINATE]	FLOAT	The z-coordinate of the vertex. Always present in FTLVERTEX.

Table F-45 FTLVERTEX (Continued)

Field Name	Data Type	Description
[RHW]	FLOAT	This value is equal to 1/ Z_COORDINATE. Conditional presence if VC_FORMAT.RHW =1
[DIFFUSE_BLUE]	FLOAT	The Blue component of diffuse color. Color intensity represented in the OpenGL format. Its value is between 0.0 and 1.0, where 1.0 represents the highest intensity while 0.0 represents the lowest. Conditional presence if VC_FORMAT.DIFFUSE_BGR=1.
[DIFFUSE_GREEN]	FLOAT	The Green component of diffuse color. Expressed in the OpenGL format. Conditional presence if VC_FORMAT.DIFFUSE_BGR=1.
[DIFFUSE_RED]	FLOAT	The Red component of diffuse color. Expressed in the OpenGL format. Conditional presence if VC_FORMAT.DIFFUSE_BGR =1.
[DIFFUSE_ALPHA]	FLOAT	Diffuse component. Its value is between 0.0 and 1.0. Conditional presence if VC_FORMAT.DIFFUSE_A=1.
[DIFFUSE_ARGB]	DWORD	Diffuse component expressed in integer. [31:24] - The Alpha component in unsigned integer. [23:16] - The Red component in unsigned integer. [15:8] - The Green component in unsigned integer. [7:0] - The Blue component in unsigned integer. Conditional presence if VC_FORMAT.DIFFUSE_ARGB = 1.
[SPEC_BLUE]	FLOAT	The blue component of specular color in the OpenGL format. Conditional presence if VC_FORMAT.SPEC_BGR=1.
[SPEC_GREEN]	FLOAT	The green component of specular color in the OpenGL format. Conditional presence if VC_FORMAT.SPEC_BGR=1.
[SPEC_RED]	FLOAT	The red component of specular color in the OpenGL format. Conditional presence if VC_FORMAT.SPEC_BGR=1.
[SPEC_FOG]	FLOAT	The fog component of specular color. Its value is between 0 and 1. Conditional presence if VC_FORMAT.SPEC_F=1.
[SPEC_FRGB]	DWORD	The integer format of specular color. [31:24] - The Fog component in unsigned integer. [23:16] - The Red component in unsigned integer. [15:8] - The Green component in unsigned integer. [7:0] - The Blue component in unsigned integer. Conditional presence if VC_FORMAT.SPEC_FRGB = 1.
[TEXTURE1_U]	FLOAT	The u-coordinate of the 1st texture. Conditional presence if VC_FORMAT.S_T =1
[TEXTURE1_V]	FLOAT	The v-coordinate of the 1st texture. Conditional presence if VC_FORMAT.S_T =1
[TEXTURE2_U]	FLOAT	The u-coordinate of the 2nd texture. Conditional presence if VC_FORMAT.S2_T2=1

Table F-45 FTLVERTEX (Continued)

Field Name	Data Type	Description
[TEXTURE2_V]	FLOAT	The v-coordinate of the 2nd texture. Conditional presence if VC_FORMAT.S2_T2=1
[RHW2]	FLOAT	Not used for DirectX. Conditional presence if VC_FORMAT.RHW2=1

F.25 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).

F.25.1 Points (1)

A point is specified by one vertex.

Table F-46 Points (1)

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.

F.25.2 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.

Table F-47 Lines (2)

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.

F.25.3 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.

Table F-48 Polylines (3)

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.
	…	
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.

F.25.4 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.

Table F-49 Triangles (4)

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.

F.25.5 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.

Table F-50 Triangle Fan (5)

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.
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.

F.25.6 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.

Table F-51 Triangle Strip (6)

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.

F.26 3D_RNDR_GEN_INDX_PRIM

Packet Type: 3D

Purpose: Render 3-D primitives points, lines and triangles 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 VC_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.

Table F-52 Format for 3D_RNDR_GEN_INDX_PRIM

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	[VC_FORMAT]	Same as field VC_FORMAT of packet 3D_RNDR_GEN_PRIM.
5	[VC_CNTL]	Same as field VC_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 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.

F.26.1 Vertex Array Format

Table F-53 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

F.27 NEXT_VERTEX_BUNDLE

Packet Type: 3D

Purpose: This is a continuation of packet 3D_RNDR_GEN_INDX_PRIM. Using this packet implies that the primitives in this packet will be rendered in the same manner as those of the previous 3D_RNDR_GEN_INDX_PRIM packet.

Table F-54 Format for NEXT_VERTEX_BUNDLE

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
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 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.

This page intentionally left blank.

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Appendix G
List of Tables

G.1 List of Tables Sorted by Name

Table G-1 List of Tables Sorted by Name

Table G-1 List of Tables Sorted by Name (Continued)

Table G-1 List of Tables Sorted by Name (Continued)

This page intentionally left blank.

Appendix H
List of Figures

H.1 List of Figures Sorted by Name

Table H-1 List of Figures Sorted by Name

Table H-1 List of Figures Sorted by Name (Continued)

Appendix I
List of Example Code

I.1 List of Example Code Sorted by Name

Table I-1 List of Example Code Sorted by Name

Table I-1 List of Example Code Sorted by Name (Continued)

Appendix J
Revision History

J.1 SDK-G04000 Rev 0.01 (SD40001.pdf)

First draft completed in Aug 1999.

This page intentionally left blank.