mach64 Programmer's Guide

Technical Reference Manual

P/N: PRG-G01000 Rev 2.01

Record of Revisions

Release	Date	Description of Changes
2.00	Mar. 98	Updates; New format.
2.01	July 98	Updated Chapter 7 (table 7-1).

Technical Reference Manuals

- *mach64* BIOS Kit (BIO-G01000)
- *mach64* Graphics Controller Specifications ATI-264CT/ET (GCS-C022001-00)
- *mach64* Graphics Controller Specifications ATI-88800GX (GCS-C012001-00)
- *mach64* Graphics Controller Specifications ATI-264VT (GCS-C02500)
- *mach64* Programmer's Guide (PRG-G01000)
- *mach64* Register Reference Guide (RRG-S022001-00)
- *mach64* Register Reference Guide ATI-264VT (RRG-C02500)
- *mach64* VGA Register Guide (VGA-S022001-00)

Table of Contents

Chapter 1: Overview

Chapter 2: Using the mach64

Chapter 3: Getting Started

Chapter 4: Linear Aperture

Chapter 5: Engine Initialization

Chapter 6: Engine Operations

Chapter 7: Advanced Topics

Chapter 8: mach64VT/GT Specific Features

Appendix A: BIOS Services

Appendix B: Extended BIOS (LT Specific)

Appendix C: CRTC Parameters

Appendix D: Clock Chip Reference

Appendix E: Register Summary

Appendix F: Programming PLL Registers in mach64 CT Family

Bibliography

Chapter 1
Overview

1.1 Introduction

This manual is a guide to understanding and programming the *mach64* accelerator. The *mach64* accelerator is a fixed-function, 2D graphics accelerator. It is function-compatible, but not register-compatible, with its predecessor – the *mach32* accelerator. It is not register compatible, yet it is function compatible, with *mach32.*

Those seeking a general understanding of the features and functions of the *mach64* only need to read *Chapter 2: Using the mach64.* Very specific examples and techniques are described in following chapters - *Chapter 3: Getting Started; Chapter 4: Linear Aperture; Chapter 5: Engine Initialization; Chapter 6: Engine Operations; Chapter 7: Advanced Topics* and *Chapter 8: mach64TV/GT Specific Features.*

The scope of this programmer's guide includes the *mach64* VT and GT (3D RAGE) accelerator chips. Those wishing to obtain programming information on earlier *mach64* variants (GX and CT) should obtain the older version of the *mach64* Programmer's Guide (contact ATI Developer Relations).

1.2 Brief History of ATI Graphics Products

The *mach64* is the latest member of ATI's line of graphics chips. To understand how it relates to earlier ATI chips for compatibility, a short discussion of these earlier chips is necessary.

Although ATI did manufacture graphics boards prior to the introduction of the Video Graphics Array (VGA) by IBM in 1987, they will not be covered in the following discussion.

1.2.1 VGAWONDER

The VGAWONDER family (ATI18800 and ATI28800) were non-accelerated chips that fully implemented the IBM VGA standard. In addition, they also supported SuperVGA graphics modes of up to 1024x768 at 8bpp or 640x480 at 24bpp, depending on chip revision and amount of memory. These additional modes were supported with ATI-specific extended VGA registers.

VGAWONDER-based boards only came in ISA bus versions as it predates most of the extended bus architectures.

1.2.2 mach8

The *mach8* (ATI38800) was ATI's first true Graphics Accelerator, providing hardware assisted drawing capabilities for 2D primitives like lines, rectangles and polygons. It was register compatible with the IBM 8514/A Display Adapter. Thus any applications or drivers that supported the 8514/A would run on a *mach8* without any modification. The *mach8* also extended on the 8514/A specification.

The *mach8* did not have any VGA compatibility so a separate VGA controller was required for standard text and VGA modes. Some *mach8* boards, like the GRAPHICS VANTAGE and GRAPHICS ULTRA included a VGAWONDER controller on the same board as the *mach8* to provide this VGA support. The VGA controller had its own memory, completely separate from the *mach8* accelerator's memory.

mach8-based boards were produced in both ISA and Microchannel versions.

1.2.3 mach32

The *mach32* chip (ATI68800) is the immediate predecessor to the current *mach64* family. The *mach32* was register compatible with both the IBM 8514/A and the *mach8*. The *mach32* also contained a VGA controller on the chip that was compatible with the VGAWONDER so a separate VGA controller was not needed. The memory on the *mach32* board was shared between the VGA controller and the *mach32* accelerator.

The *mach32* improved upon the *mach8* by providing a linear aperture to allow fast image data transfer by mapping the video memory to the system memory address space. Later revisions of the *mach32* also were able to memory map the *mach32* registers to overcome the performance penalty incurred in going through I/O port-mapped registers. Finally, the *mach32* contained a hardware cursor.

mach32-based boards were produced in five bus types: ISA, EISA, VESA Local Bus, Microchannel, and PCI.

1.2.4 mach64

The *mach64* represented a departure from the *mach32* in that it was no longer register compatible with previous ATI graphics accelerators or the 8514/A. (VGA register compatibility was retained, however.) This departure was necessary to resolve some design limitations that were a legacy of the older generation chips. Fortunately, almost all the functionality that was in the *mach32* was preserved in the *mach64* design, and some useful additions and enhancements were incorporated.

As indicated on the table below, the *mach64* can be divided into two major types, the GX family and the CT family. While applications that use the *mach64* should run on both types with little or no modification, there are some important differences between the two

families that are highlighted in the following sections.

Boards based on *mach64* are produced in ISA, VESA Local Bus and PCI bus versions.

Table 1-1 *mach64* Product Families

***mach64* Feature Set Variations**							
	***mach64*GX Family**				***mach64*CT Family**		
Feature	**GX-C/D**	**GX-E***	**GX-F**	**CX**	**CT**	**VT**	**GT (3D RAGE)**
Relocatable I/O (PCI only)†			✓†		✓†	✓	✓
Maximum Memory	8MB	8MB	8MB	4MB	4MB	4MB	8/16 MBΔ
Minimum Memory	512KB	1MB	1MB	512KB	1MB	1MB	1MB
Standard Linear Aperture (little endian)	✓	✓	✓	✓	✓	✓	✓
Extended Linear Aperture (big endian)		✓	✓		✓	✓	✓
Linear Aperture Boundary	8MB‡	16MB	16MB	8MB	16MB	16MB	16MB
ATI SVGA Extended Register Set	✓	✓	✓	✓			
Supported bus types	ISA, VLB, PCI	PCI	ISA, VLB, PCI	ISA, VLB, PCI	PCI	PCI	PCI, AGP

* Revision E was a short-lived version that was only used in Apple Power Macintosh-based boards.
†Relocatable I/O requires a hardware strap to be enabled. If the feature is enabled, the standard I/O base addresses do not apply.
‡ A 4MB boundary is possible if the linear aperture size is set to 4MB.
Δ 16 MB maximum on 3D RAGE PRO chips only.
C/DRevisions C and D.

1.2.4.1 *mach64*GX Family

The *mach64*GX Family encompasses the *mach64*GX (ATI888GX00) and *mach64*CX (ATI888CX00) variants. The major distinguishing characteristics of this family are:

- Uses an external DAC
- Uses an external clock synthesizer
- Support for VRAM
- VGA controller is ATI VGAWONDER compatible
- VGA controller is independently programmable from the accelerator controller

From a very rough architectural perspective, the *mach64*GX family more resembles the *mach32* than it does the *mach64*CT family. However, from a functionality and register level perspective, the *mach64*GX is almost identical to the *mach64*CT.

1.3 *mach64*CT Family

The *mach64*CT Family encompasses the *mach64*CT (ATI264CT), *mach64*VT (ATI264VT) and *mach64*GT (3D RAGE) variants. The major distinguishing characteristics of this family are:

- Integrated DAC
- Integrated clock synthesizer
- No VRAM support
- VGA controller is “pure” VGA, not VGAWONDER compatible
- VGA controller is not independently programmable from the accelerator controller

1.3.1 *mach64*VT

The *mach64*VT family of chips is built upon the previously mentioned CT. They have the same feature set as the CT, plus some additional video features such as:

- back end hardware overlay
- back end hardware scaler

1.3.2 *mach64*GT (3D RAGE)

The *mach64*GT (commonly known as the 3D RAGE) introduces hardware support for 3D operations. While low level 3D operations are not discussed in this guide, we do demonstrate the usage of front and end scaler, which is part of the 3D pipeline. The 3D RAGE includes all *mach64*VT features with the addition of:

- hardware 3D acceleration
- improved video filtering

1.4 Features

1.4.1 *mach64* Major Features

- Full draw capability at 1, 4, 8, 15, 16, and 32 bits per pixel color resolutions. Hardware-assisted draw functions are available for packed 24 bits per pixel draw modes.
- Standard spatial resolution of 640x480, 800x600, 1024x768, and 1280x1024. Other resolutions with pixel clocks of up to 220 MHz can be supported, limited only by the DAC, memory size, and memory bandwidth.
- Full read/writable memory-mapped registers.
- Up to 8MB of memory (16 for 3D RAGE PRO).
- 32x32 command FIFO.
- Four-color (two fixed colors, complement, and transparent) hardware cursor of size up to 64x64.
- Overscan.
- Linear frame buffer is locatable on 16MB boundaries anywhere in a 4GB system memory address space.
- Paged frame buffer with two 32KB pages (independent read and write pages), pagable on 32KB boundaries anywhere in the 8MB video memory address space.
- Draw functions include rectangle fill, line draw, bitblt, polygon boundary lines, and polygon fill.
- Generalized 2D patterns with rotation.
- A linear memory mode for efficient memory management.
- Efficient monochrome expansion.
- Bit masking and scissoring capabilities.
- Seventeen-function ALU for full suite of logical ROPs.
- Source compare logic suitable for transparent blits.
- Destination compare logic suitable for alpha channel mixing.
- Scrolling and panning on a virtual desktop.
- Big endian support (*mach64*GX-E/F, *mach64*CT Family).
- EEPROM hardware support for non-volatile storage. (Certain controllers are EEPROM-less.)
- Four-level hardware Display Power Management System (DPMS) mode support.

- DAC power-down support.
- Diagnostic test modes.

1.4.2 Functional Enhancements Relative To *mach32*

- Full draw capability in 1 bpp and 32 bpp modes, and hardware assist in packed 24 bpp mode has been added.
- Full 32-bit registers. Some register pairs may be written in a single 32-bit write.
- Device coordinates have been expanded from –4096 to +4095 in the X direction, and from –16384 to +16383 in the Y direction.
- Bresenham parameters have been expanded from 12 bits to 18 bits.
- Packed monochrome expansion.
- The paged frame buffer is now pagable on 32KB boundaries instead of 64KB.
- The source trajectory types, strictly-linear, general-pattern, and general-pattern-with-rotation, have been added.
- Source compare.
- Four-level hardware Display Power Management System (DPMS) mode support.
- DAC power-down support.
- Diagnostic test modes.

1.4.3 Deletions Relative To *mach32*

- Point-to-point line draw.
- Line clip exception handling.
- VNIB and VPIX type rectangles.
- Short-stroke vectors.
- Scan line draw.
- Four compare functions.
- Bounds accumulators.
- CRTC shadow sets.
- Host reads; screen-to-host transfers can still be accomplished by aperture reads.
- Degree mode lines; Bresenham lines are still supported.

All the deleted functions listed above are redundant and may still be accomplished by other means.

1.4.4 Functional Differences From *mach32*

- Monochrome blits are now packed instead of sparse.
- Host writes are packed to 32 bits. The 1 bpp and 4 bpp modes may be optionally aligned to a byte.
- Pixel consumption order from the host data register is only programmable in 1 bpp and 4 bpp modes.
- Polygon fills are always inclusive on both edges and optionally right edge exclusive on the mach64CT.
- Polygons derive their boundary data from an implicit polygon source instead of an explicit monochrome source.
- Rectangular trajectories are specified in width and height instead of start and end.
- The ALU carry chain mask is set explicitly instead of implicitly from the pixel depth.
- Line drawing options do not affect rectangular trajectories and rectangle options do not affect line drawing trajectories.
- Destination side effects (tiling) are now programmable.
- Source pointer always returns to the original SRC_X, SRC_Y position after draw completion.
- Pixel depths, pitches and offsets are independently specified for CRTC, source, destination, and host.
- Bresenham parameters have been expanded from 12 bits to 18 bits.

1.5 Overview of the Manual

1.5.1 Chapter-By-Chapter Summary

Chapters 1 to *7* cover the general functionality that is available in all variants of the *mach64*. In *Chapter 8*, the specific details of each particular variant will be covered in depth.

Chapter 2 provides details of the features and basic programming model of the *mach64*.

Chapter 3 demonstrates the fundamental steps that are necessary to use the *mach64* in accelerator mode. Issues such as card detection and setting a display mode are covered here. Programming considerations are also discussed.

Chapter 4 covers the usage of the linear aperture, which provides immediate benefit to programs as they no longer have to deal with bank switching and the 64KB page limit.

Chapter 5 goes into issues covering the accelerator engine itself, such as the command FIFO queue and engine initialization.

Chapter 6 discusses general engine operation, and provides numerous examples of standard engine operations.

Chapter 7 contains some advanced topics that highlight some of the special features and capabilities of the *mach64*.

Chapter 8 covers some other advanced topics specific to the VT and 3D RAGE, including use of the hardware overlay/scaler, the front end scaler of the 3D RAGE, and the bus mastering capabilities of the 3D RAGE PRO

1.5.2 Notations And Conventions Used In This Manual

Mnemonics are used throughout this manual in place of hardware register names. 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]

CONFIG_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.

Sample Code Example

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

This page intentionally left blank.

Chapter 2
Using the mach64

2.1 Introduction

This chapter discusses the functionality of the *mach64*. The capabilities and features of the *mach64* are also summarized.

2.2 Intel Based Architecture

This section focuses on the features and services that are available on systems that have Intel and Intel-compatible CPUs as well as those systems that can emulate Intel CPUs.

2.2.1 Memory Map

The *mach64* requires a memory aperture so that an application can access the frame buffer and the memory mapped registers. Normally, this aperture is located somewhere within the 4GB address space where it does not conflict with system (host) memory. Further, this aperture must be located on a 4MB, an 8MB, or a 16MB boundary, depending upon the particular *mach64* chip and configuration. The following diagram illustrates a typical memory organization for a *mach64* board with 4MB of display memory installed on system with 16MB of main memory:

Typical Organization Of *mach64* Aperture Within Host Address Space (PC-compatible)

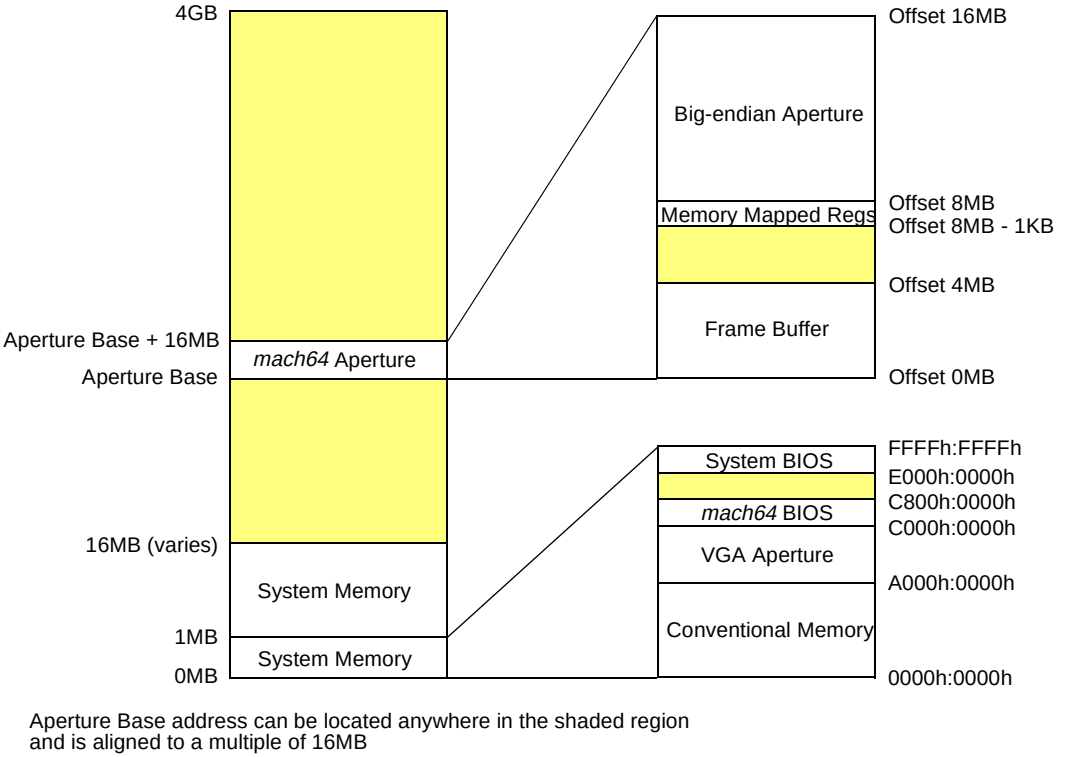


Figure 2-1. Aperture Within Host Address Space (PC-compatible)

2.2.2 BIOS Services

The BIOS Services provide a straightforward way of setting up and using the *mach64*. The BIOS Services also provide a way of querying the *mach64* hardware in order to determine its capabilities.

VGA modes are initialized with the standard INT 10h interface as described in the *mach64 Register Reference Guide*. For further information on using standard VGA BIOS Services, see *Programmer's Guide to the EGA, VGA, and Super VGA Cards*, by Richard Ferraro.

For accelerator BIOS services, either INT 10h (AH=A0h) or a far call to the ROM can be used. The key services that are provided include loading and setting a display mode, and the BIOS query functions. See *Appendix A, BIOS Services* for a complete definition of all accelerator BIOS services.

2.2.3 Registers

All of the *mach64* accelerator engine functions are performed through the use of the registers. There are 6 classes of registers that are available:

- **VGA Registers** are completely segregated from the accelerator registers. Their functions are mutually exclusive. They are addressed at I/O ports 3B0h-3BFh, 3C0h-3CFh, and 3D0h-3DFh. These are the registers that are provided for compatibility with the IBM VGA Display Adapter. Note that the ATI VGA extended registers at 1CEh-1CFh are only available on the *mach64*GX family in standard (non-relocatable) I/O mode. (*See mach64 Register Reference Guide for more details*.)
- **Setup and Control Registers** are usually initialized only once during boot time and are used for basic configuration of the *mach64* hardware and to report back hardware capabilities. The *mach64* diagnostic registers are also included in this category.
- **Accelerator CRTC and DAC registers** are used to program the resolution, refresh rate, and pixel depth of the display mode, and to provide hardware cursor services.
- **Draw Engine Control Registers** are used for manipulating *mach64* draw engine in terms of general data path setup and control.
- **Draw Engine Trajectory Control Registers** are used to set up and control specific engine drawing operations.
- **Host Bus Dependent Registers** are used for bus-specific information.

Registers must be accessed in order to be useful to the programmer. Most registers are memory mapped. Others are I/O mapped. Some are both. In general, the VGA registers are I/O mapped only, the *mach64* Draw Engine registers are memory mapped only, and the rest of the registers are both I/O and memory mapped. See *mach64 Register Reference Guide* for specifics and exceptions. The following sections demonstrate how to access these registers.

2.2.3.2 Memory Mapping

All registers not associated with the draw engine are I/O mapped, and all have memory mapped register aliases (except for CONFIG_CNTL on *mach64*GX-C/D). All registers are 32 bits wide, except for DAC_REGS, which are 4x8 bit registers. All draw engine registers are memory mapped with DWORD offsets greater than or equal to 40h.

- If the small apertures are enabled, the memory mapped registers may be accessed through a 1KB area at a segment:offset of B000h:FC00h.
- If the big aperture is enabled, the memory mapped registers occupy the address space located at the base address of the aperture, plus an offset of 3FFC00h for a 4MB aperture, or 7FFC00h for an 8MB aperture configuration. A method of accessing extended memory is required to access the registers at this location.

On the *mach64*GX family, memory mapped registers may be read from and written to in 8-bit, 16-bit and 32-bit quantities.

On the *mach64*CT family, writes to the memory mapped registers must be performed in one 32-bit write. The memory mapped registers on the *mach64*CT family may be read in the same manner as on the *mach64*GX family.

Referring to the *mach64 Register Reference Guide*, the **DWORD Offset** or **Memory Map (MM) select** is given to describe the register's address. If access through the small apertures is desired, the physical address can be determined by the following equation:

```
physical memory address = (MM select << 2) + B000h:FC00h
```

For example, if the **MM select** = 21h (SCRATCH_REG1), the physical address would be B000h:FC84h.

If the big aperture is enabled, the equation becomes:

```
physical memory address = (MM select << 2) + aperture base +
                        + memmap offset
```

where **memmap offset** is either 3FFC00h or 7FFC00h. Using the example above, if the aperture base address is A0000000h, the aperture size is 8MB (offset 7FFC00h) and the **MM select** = 21h (SCRATCH_REG1), the physical memory address would be A07FFC84h.

For some registers, it is necessary to access individual bytes within the 32-bit register (such as DAC_REGS). The **MM select** must be converted to a byte offset before adding the individual byte offset (0, 1, 2, or 3). For example, to access the DAC_MASK byte of DAC_REGS through the small aperture the equation is:

```
          byte offset  = MM select << 2 = 30h << 2 = 00C0h (DAC_REGS)

individual byte offset  = 2 (DAC_MASK byte)


physical memory address  = byte offset + individual byte offset +
                           B000h:FC00h

                         = 00C0h + 2 + B000h:FC00h = B000h:FCC2h
```

For the big aperture, the equation is:

```
            byte offset  = MMselect << 2 = 30h << 2 = C0h (DAC_REGS)

individual byte offset  = 2 (DAC_MASK byte)
```

```
            aperture base  = A0000000h

         memmap offset (8MB)  = 7FFC00h

    physical memory address  = byte offset + individual byte offset +
                               aperture base + memmap offset

                             = C0h + 2 + A0000000h + 7FFC00h = A07FFCC2h
```

2.2.3.3 I/O Mapping

Since the I/O base address may be different depending on the card configuration, it cannot be assumed to be a specific value. The easiest way to obtain the I/O base address is to call *mach64* BIOS function 12h (*see Appendix A, BIOS Services for more information*). The BIOS services can be called in two ways: FAR CALL or INT 10h (it is recommended that the INT 10h method be used).

This function also returns the I/O base address type -- standard or relocatable. If it is standard, the I/O base address will typically be 2ECh. If it is relocatable (only on PCI), the I/O base address can be any value within a 64KB I/O space. The value is decided by the system to insure that no conflicts exist and is in accord with the “plug and play” specification of a PCI system.

In order to use the FAR CALL method, the *mach64* ROM segment is required. The ROM segment for a *mach64* card with the VGA enabled is always at C000h and is normally 32KB in size. To access the BIOS services in the ROM, the offset must be set to 64h. If the VGA is disabled, the ROM segment is usually at C000h or C800h but can be located at other segments. A VGA disabled ROM size is 2KB to 8KB.

Referring to the *mach64 Register Reference Guide*, the **I/O select** is given to describe the register’s address. The physical address can be determined by the following equation:

```
physical I/O address = (I/O select << 10) + I/O base address
```

For example, if the I/O base address = 2ECh and the I/O select = 11h (SCRATCH_REG1), the physical I/O address would be 46ECh.

If the relocatable feature is enabled (PCI only), the DWORD Offset or Memory Map (MM) select is used to describe the register's address. For this case, the equation becomes:

```
physical I/O address = (MM select << 2) + I/O base address
```

Using the example above, if the I/O base address = E000h and the **MM select** = 21h (SCRATCH_REG1), the physical I/O address would be E084h.

For some I/O registers, it is necessary to access individual bytes within the 32-bit register (such as DAC_REGS). The **I/O select** or **MM select** must be converted to a byte offset before adding the individual byte offset (0, 1, 2, or 3).

For example, to access the DAC_MASK byte of DAC_REGS, the equation is:

```
            byte offset  = I/O select << 10 = 17h << 10 = 5C00h (DAC_REGS)
individual byte offset   = 2 (DAC_MASK byte)
       I/O base address  = 2ECh

  physical I/O address   = byte offset + individual byte offset + I/O base
                           address
                         = 5C00h + 2 + 2ECh = 5EEEh
```

For relocatable I/O, the equation is:

```
            byte offset  = MMselect << 2 = 30h << 2 = C0h (DAC_REGS)
individual byte offset   = 2 (DAC_MASK byte)
       I/O base address  = E000h

  physical I/O address   = byte offset + individual byte offset +
                           I/O base address
                         = C0h + 2 + E000h = E0C2h
```

2.3 Non-Intel Based Architecture

This section will focus on the features and services that are available on systems that cannot fully emulate Intel CPUs (such as the Apple Power Macintosh).

Note that the *mach64*GX-C/D cannot be used in non-Intel environments. Also, non-Intel platforms must conform to the PCI specification. Thus this section is restricted to PCI versions of the *mach64*GX-E/F and the *mach64*CT family.

2.3.1 Memory Map

The *mach64* requires a memory aperture so that an application can access the frame buffer and the memory mapped registers. Normally, this aperture is located somewhere within the 4GB address space where it does not conflict with system (host) memory. Further, this aperture must be located on a 16MB boundary. The little endian aperture is located at offset 0MB of this aperture space, while the big endian aperture is located at offset 8MB The following diagram illustrates a typical memory organization for a *mach64* board with 4MB of display memory installed:

Typical Organization Of *mach64* Aperture Within Host Address Space (non-Intel)

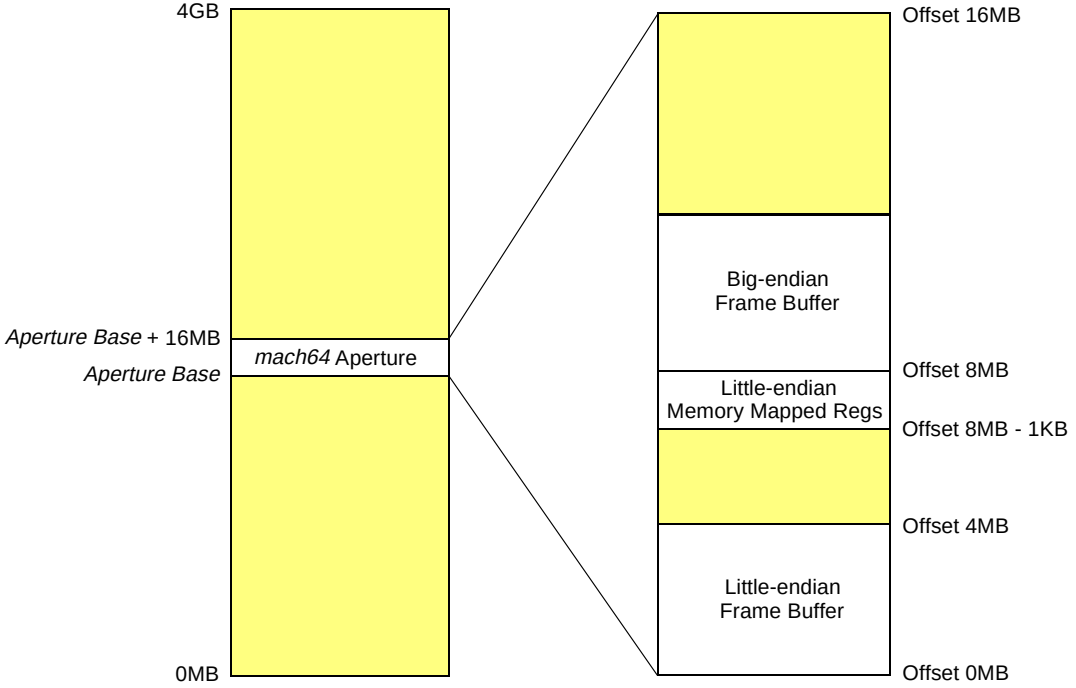


Figure 2-4. Aperture Within Host Address Space (non-Intel)

2.3.2 BIOS Services

BIOS Services are unavailable on non-Intel platforms. The BIOS is typically replaced with a ROM that conforms to the IEEE OpenBoot specification. Upon system powerup, the ROM will initialize the *mach64* board to a known state. The ROM image will disappear at the end of the boot process. All further access to the *mach64* must be done via the memory mapped registers. Setting modes, for example, must be done manually.

2.3.3 Registers

The VGA registers are normally not available, unless the non-Intel platform in question contains hardware support for an I/O address space that is distinct from Memory address space.

All registers are memory mapped and are 32 bits wide, except for DAC_REGS, which are 4x8 bit registers.

- If the small apertures are enabled, the memory mapped registers may be accessed

through a 1KB area at a linear address 0x000BFC00.

- If the big aperture is enabled, the memory mapped registers occupy the address space located at the base address of the aperture, plus an offset of 0x003FFC00 for a 4MB aperture, or 0x007FFC00 for an 8MB aperture configuration. A method of accessing extended memory is required to access the registers at this location.

On the *mach64*GX family, memory mapped registers may be read from and written to in 8-bit, 16-bit and 32-bit quantities.

On the *mach64*CT family, writes to the memory mapped registers must be performed in one 32-bit write. The memory mapped registers on the *mach64*CT family may be read in the same manner as on the *mach64*GX family.

Referring to the *mach64 Register Reference Guide*, the **DWORD Offset** or **Memory Map (MM) select** is given to describe the register's address. If access through the small apertures is desired, the physical address can be determined by the following equation:

```
physical memory address = (MM select << 2) + 0x000BFC00
```

For example, if the MM **select** = 0x21 (SCRATCH_REG1), the physical address would be 0x000BFC84.

If the big aperture is enabled, the equation becomes:

```
physical memory address = (MM select << 2) + aperture base +
                        + memmap offset
```

where **memmap offset** is either 0x003FFC00 or 0x007FFC00. Using the example above, if the aperture base address is 0xA0000000, the aperture size is 8MB (offset 0x007FFC00) and the **MM select** = 0x21 (SCRATCH_REG1), the physical memory address would be 0xA07FFC84.

For some registers, it is necessary to access individual bytes within the 32-bit register (such as DAC_REGS). The **MM select** must be converted to a byte offset before adding the individual byte offset (0, 1, 2, or 3). For example, to access the DAC_MASK byte of DAC_REGS through the small aperture the equation is:

```
          byte offset  = MM select << 2 = 0x30 << 2 = 0xC0 (DAC_REGS)

individual byte offset  = 2 (DAC_MASK byte)


physical memory address  = byte offset + individual byte offset +
                           0x000BFC00

                         = 0xC0 + 2 + 0x000BFC00 = 0x000BFCC2
```

For the big aperture, the equation is:

```
          byte offset  = MMselect << 2 = 0x30 << 2 = 0xC0 (DAC_REGS)
individual byte offset = 2 (DAC_MASK byte)
         aperture base = 0xA0000000
   memmap offset (8MB) = 0x007FFC00


physical memory address = byte offset + individual byte offset +
                          aperture base + memmap offset
                        = 0xC0 + 2 + 0xA0000000 + 0x007FFC00
                        = 0xA07FFCC2
```

I/O mapped registers may not be available on non-Intel platforms as this method of register access implicitly assumes Intel-style I/O port addressing capability. If I/O mapped registers are available, see section *2.2.3.3* for information on how to access these ports.

This page intentionally left blank.

Chapter 3
Getting Started

3.1 Introduction

This chapter discusses the basics of using the *mach64* and covers detection of the *mach64* and setting up a display mode. Refer to the sample code for this chapter in the *mach64* Programmer's Guide Companion Diskette.

3.2 Before you start

Before programming the *mach64* there are several issues that should be discussed as they will determine how the *mach64* will be used on the desired platform. These issues are discussed below.

3.2.1 Accelerator vs. VGA

The *mach64* has two distinct operating modes:

- **VGA mode**
- **Accelerator mode**

For more information on standard VGA programming, please refer to any of the texts that are mentioned in the Bibliography such as *Programmer's Guide to the EGA, VGA, and Super VGA Cards*, by Richard F. Ferraro.

Note that the *mach64* also supports the VESA VBE 1.2 programming interface. This interface was created by the Video Electronic Standards Association (VESA) to provide a standardized method for using SuperVGA display modes on non-accelerated hardware. Effectively, VBE 1.2 folded in VGA support for common high resolution modes such as 1024x768 with 256 colors. Contact VESA for further information on VBE.

The accelerator provides the ability to draw into screen memory concurrently with the operation of the host CPU. In accelerator mode, there are two ways of accessing the graphics memory:

- **Memory aperture**
- **Draw engine**

The host application may read or write screen memory directly through a memory aperture (an **aperture** is an address space that maps directly to on-board memory).

Accesses through the aperture provide no acceleration, and the speed of these accesses is generally bound by the speed of the host expansion bus.

The second way of accessing the memory is to use the draw engine to write to it. The draw engine can do two things:

- **Rectangle fills**
- **Lines**

These are known as **destination trajectories** (a **trajectory** defines a path through graphics memory which the draw engine reads or writes data). These trajectories may be filled with pixel data from various sources. If the source data comes from graphics memory, this is called a **bitblt** (or **blit**) and follows one of four different source trajectories.

A more detailed description of trajectories can be found in section 6.1.3: *Trajectories*.

3.2.2 Linear Aperture vs. VGA Aperture

Memory on the *mach64* may be directly accessed in one of three ways:

- **Standard paged 64KB VGA aperture**
- **Small dual paged apertures**
- **Big aperture**

Note that it is completely legitimate to have all three apertures simultaneously accessing framebuffer memory, but if the Big linear aperture is active there is typically no need to access memory through either the VGA or the small apertures.

3.2.2.1 Standard Paged 64KB VGA Aperture

If the VGA is enabled and the *mach64* is in VGA mode, memory may be accessed through the **standard paged 64KB VGA aperture**. The segment base address of this aperture is either A000h or B000h depending on the video mode.

The *mach64*GX family also use this aperture to access the lower 1MB of memory in planar (16 color) SVGA modes. The ATI VGA extended registers are used to select the 64KB read or write page mapped to the aperture space. As the ***mach64*CT family do not contain the VGA extended register set**, the small dual paged apertures described in the next section are used to access video memory. Any memory writes via the VGA aperture are inhibited when the memory boundary is enabled.

For more information on how to page the 64KB aperture, see the *mach64 Register Reference Guide*.

3.2.2.2 Small Dual Paged Apertures

If the *mach64* is in an accelerator mode or a SVGA packed pixel mode, **two small 32KB apertures** may be enabled at segment base addresses A000h and A800h. The read and write pages are set independently on 32KB boundaries for each of the two apertures with the MEM_VGA_WP_SEL and MEM_VGA_RP_SEL registers. This aperture mode is a type of VGA aperture configuration that is not available in standard VGA modes. If the memory boundary is enabled, writes to these apertures are inhibited.

- These small apertures can access the full 8MB.
- These small apertures may be enabled only if the VGA is enabled on the chip; otherwise, a memory address conflict would exist between the accelerator and the existing VGA.

Some special initialization is required to enable the small apertures and the memory mapped registers in the VGA address space:

- The VGA must be put into packed pixel mode.
- The VGA must have a 128KB aperture enabled if both the small apertures and the memory mapped registers are addressed.
- The bit CFG_MEM_VGA_AP_EN@CONFIG_CNTL must be set.

If access to the VGA memory mapped registers is not required, the setting of CFG_MEM_VGA_AP_EN@CONFIG_CNTL is not necessary.

Because the small aperture page size is 32KB, programs which assume the page size to be 64KB need to double the page number within their page setting routine. When selecting the write page, for instance, the doubled page number must be written to MEM_VGA_WPS0@MEM_VGA_WP_SEL to set the page number for the first 32KB aperture. This value plus one must then be written to MEM_VGA_WPS1@MEM_VGA_WP_SEL to set the page number for the second 32KB aperture. A similar process may be used to set the read page in the MEM_VGA_RP_SEL register. In this way, programs which assume a page size of 64KB can use the small apertures transparently.

The ATI VGA extended registers are used to change the display page in the *mach64*GX family. Because the *mach64*CT family does not include the VGA extended register set, they must use the small aperture to change the read or write page.

3.2.2.3 Big Aperture

If the *mach64* is in accelerator mode, a big linear aperture may be enabled to access the entire frame buffer. The size and location of the aperture depends on the *mach64* variant. For example, on the *mach64*GX-C/D, the aperture size may be set to 4MB or 8MB and require an 8MB boundary for the aperture location. The *mach64*GX-E/F and the

*mach64*CT family always require a 16MB boundary since enabling the big linear aperture also enables the 8MB big endian aperture (the big endian aperture starts at the standard linear aperture address plus 8MB). The *mach64*GX-E/F do allow 4MB or 8MB aperture sizing whereas the *mach64*CT family allows only an 8MB sized aperture. To produce code that works across all *mach64* variants, it is recommended that the aperture size be set to 8MB and located on a 16MB boundary.

The availability of this aperture is assured on all board configurations except ISA bus configurations. On an ISA system, the following two conditions must be met in order to use the big aperture:

- The aperture must fit within a 16MB address space.
- The aperture must not overlap host CPU memory.

An ISA system with greater than 12MB of host CPU memory cannot use a big aperture.

Given the restrictions imposed on using the linear aperture on ISA systems, it is recommended that the VGA dual paged aperture be used for ISA systems.

3.2.3 Protected Mode vs. Real Mode

Writing to and reading from the linear aperture can be done in several ways. Since the linear aperture is located in the “extended memory” space, a real mode application must use the extended memory services to access memory through the linear aperture. There are several services available:

- System BIOS INT 15h, function 87h.
- DOS Protected Mode Interface (DPMI).
- Virtual Control Program Interface (VCPI).

The last two points mentioned require protected mode memory managers that support these services. The first point mentioned will typically be available since it is supported by the system BIOS. This method is also the slowest and is not practical for performance applications. It should be noted that EMS (Expanded Memory Services) and XMS (eXtended Memory Services) are not practical for accessing graphics frame buffers.

If the application is in protected mode, the linear aperture can be accessed easily with virtually no overhead.

Before attempting to access the big aperture, the host application must enable it with BIOS services function 5 or by writing to the CONFIG_CNTL register. (*See Appendix A, BIOS Services for more details*.)

3.3 mach64 Detection

There are several steps that are required in order to properly detect the *mach64*. Some of these steps may need to be performed differently on a non-Intel platform or if the BIOS is unavailable. The key steps to detection are:

1. Detect *mach64* signatures.

2. Determining I/O base address.

3. Read/Write tests.

4. Determining specific *mach64* variant.

3.3.1 Card Detection

Find an ATI *mach64* ROM and its ROM segment by scanning through ROM segments C000h – FE00h, in 2KB steps. To match, the ROM ID, ATI product signature, and *mach64* product string must be found:

ROM ID = AA55h (PC compatibles)
ATI product signature = “761295520”
mach64 string2 = “MACH64”
mach64 string1 = “GXCX” (older ROMs)

The ROM ID bytes will occur as the first two bytes in the segment. The ATI product signature will occur somewhere within the first 256 bytes of the segment and will identify the ROM as belonging to an ATI display adapter. One of the *mach64* strings will occur somewhere within the first 1024 bytes of the segment and will identify the ROM as belonging to a *mach64*-based product.

Only one the *mach64* strings will be present in a *mach64* ROM. Therefore, the ROM should be searched for string1 first. If it is not present, string2 should be searched for. If it is also not present, a *mach64* ROM is not present.

For non-Intel platforms, access to the PCI configuration space is required. Scan for a PCI card with a VendorID of 0x1002. This number is ATI’s VendorID as registered with the PCI Special Interest Group, and all PCI-based products manufactured by ATI will have this VendorID. Once found, scan for the following DeviceID codes to identify a *mach64*:

Table 3-1 PCI DeviceID Codes

***mach64* PCI DeviceID Codes**	
Variant	**DeviceID**
*mach64*GX	0x4758
*mach64*CX	0x4358
*mach64*CT	0x4354
*mach64*VT	0x5654
*mach64*VTB	0x5655
*mach64*VT4	0x5656

***3D RAGE* PCI DeviceID Codes**	
Variant	**DeviceID**
3D RAGE (GT)	0x4754
3D RAGE II+ (GTB)	0x4755
3D RAGE IIC	0x4756
3D RAGE PRO (BGA, AGP)	0x4742
3D RAGE PRO (BGA, AGP, 1X ONLY)	0x4744
3D RAGE PRO (BGA, PCI)	0x4749
3D RAGE PRO (PQFP, PCI)	0x4750
3D RAGE PRO (PQFP, limited 3D)	0x4751

On all systems that support multiple *mach64* cards installed the above procedure should be repeated until all *mach64* images have been located.

3.3.2 I/O Base

Call the ROM (BIOS service 12h) to find the I/O base address and type (standard/relocatable). The CX register should be preloaded with zero before calling this BIOS function. This insures that CX is zero on return for older ROMs.

Standard (also known as Fixed or Sparse) I/O is the only I/O type available on ISA and VLB *mach64* boards. The lower 10 bits of the I/O port address are fixed and set to 0x2EC and the upper 6 bits are used to index the various *mach64* registers. Relocatable (also known as Block) I/O is available on PCI *mach64* boards with the exception of the *mach64*GX-C/D. The I/O base can be anywhere within the 64KB I/O address space and will occupy 256 consecutive registers.

For non-Intel platforms, the Base Addresses section of the PCI configuration space will indicate what the I/O base address is.

3.3.3 Read/Write Test

Perform a write/read test on SCRATCH_REG1 (its contents must be saved and restored since they are used by the BIOS services). This is done by writing the 32-bit value 55555555h to SCRATCH_REG1 and then reading it back. If the value is different, a *mach64* is not present. If the value is the same, repeat this test with AAAAAAAAh. Ensure that the register's contents are restored.

3.3.4 CONFIG_CHIP_ID

Read the CONFIG_CHIP_ID register for additional information such as the chip type, class, and revision.

Additional configuration information can be obtained with a BIOS query call (functions 6, 7, 8, 9, and Ah). (See *Appendix A, BIOS Services* for more information.).

3.4 Mode Switching

It is highly recommended that all mode switching be done by a BIOS service function call rather than by manually setting the CRT controller (CRTC). The main reasons for doing this are:

- Simplicity.
- The characteristics of the non-volatile storage device that stores mode and monitor information may not be known. Without monitor information, the only mode guaranteed to work on all analog monitors is 640x480 at 60 Hz non-interlaced.
- CRTC compatibility with future devices is not guaranteed.

On *mach64*GX family, there are separate CRTCs for the VGA and accelerator. The CRTC parameters may be set independently. On *mach64*CT family, both the VGA and accelerator share the same CRTC and therefore they cannot be set independently. For all *mach64* chips, the BIOS can be used to switch between VGA and accelerator modes.

The following table lists the VESA VBE modes that may be available on the *mach64*. These modes are set with the standard VESA VBE set mode call (INT 10h, AX=4F02h, BX=*mode*). Consult the VBE specification for further details.

Table 3-2 VESA Compatible Mode Support

VESA VBE 1.2 Compatible Mode Support for *mach64*			
Mode number	**Resolution**	**Pixel Depth**	**Memory Used**
100h	640x400	8	250KB
101h	640x480	8	300KB
102h	800x600	4	235KB
6Ah	800x600	4	235KB
103h	800x600	8	469KB
104h	1024x768	4	384KB
105h	1024x768	8	768KB
107h	1280x1024	8	1280KB
110h	640x480	15	600KB
111h	640x480	16	600KB
112h	640x480	24	900KB
113h	800x600	15	938KB
114h	800x600	16	938KB
115h	800x600	24	1407KB
116h	1024x768	15	1536KB
117h	1024x768	16	1536KB
118h	1024x768	24	2304KB
119h	1280x1024	15	2560KB
11Ah	1280x1024	16	2560KB
11Bh	1280x1024	24	3840KB

The following table lists the modes that are available on the *mach64* when in accelerator mode. These modes are set through the *mach64* BIOS. See *Appendix A: BIOS Services* for details.

Table 3-3 Accelerator Modes

***mach64* Accelerator Modes**		
Resolution	**Pixel Depth**	**Memory Used**
640x480	4	150KB
640x480	8	300KB
640x480	15/16	600KB
640x480	24	900KB
640x480	32	1200KB
800x600	4	235KB
800x600	8	469KB
800x600	15/16	938KB
800x600	24	1407KB
800x600	32	1875KB
1024x768	4	384KB
1024x768	8	768KB
1024x768	15/16	1536KB
1024x768	24	2304KB
1024x768	32	3072KB
1280x1024	4	640KB
1280x1024	8	1280KB
1280x1024	15/16	2560KB
1280x1024	24	3840KB
1280x1024	32	5120KB
1600x1200	4	938KB
1600x1200	8	1875KB
1600x1200	15/16	3750KB
1600x1200	24	5625KB
1600x1200	32	7500KB

Note that the availability of each mode depends upon the amount of memory and the type of DAC that is on board, and that not all modes may be available on all *mach64* boards.

3.4.1 BIOS Interface

VGA modes are initialized with the standard INT 10h interface as described in the *mach64 VGA Register Guide.*

For accelerator BIOS services, either INT 10h (AH=A0h) or a far call to the ROM can be used. Since the ROM segment is determined during card detection, it will be available for ROM calls. If the I/O base address is known, it can also be retrieved from SCRATCH_REG1 using the following calculation:

```
segment = (SCRATCH_REG1 & 0x7F) * 0x80 + 0xC000
```

The offset used for the call is always 64h. See *Appendix A, BIOS Services* for a complete definition of all accelerator BIOS services. In particular, BIOS Service 02h (Load and Set Mode) is the main function that is called when setting a mode. The following example shows how to set an accelerator mode using both methods:

FAR CALL Method

```
; Data for FAR CALL
romAddr         dw    64h
                dw    0c000h

; Set a 1024x768 8 bpp accelerator mode using the FAR CALL method.
mov      ax, 2              ; BIOS service 2 (load and set
                                  mode)
mov      ch, 55h            ; resolution = 1024x768
mov      cl, 82h            ; pitch = X resolution, 8 bpp
mov      romAddr, 64h       ; offset of ROM call address
mov      romAddr+2, romseg  ; segment of ROM call address
call     DWORD PTR romAddr  ; call BIOS service
```

INT 10h Method

```
; Set a 1024x768 8 bpp accelerator mode using the INT 10h method.
mov      ax, 0a002h         ; BIOS service 2 (load and set
                                  mode)
                            ; -- note value of AH
mov      ch, 55h            ; resolution = 1024x768
mov      cl, 82h            ; pitch = X resolution, 8 bpp
int      10h                ; call BIOS service
```

3.4.2 Manual Mode Switching and Custom CRT Modes

Mode switching by manual means is not recommended. If for some reason this cannot be avoided, refer to *Chapter 7, Advanced Topics.*

This page intentionally left blank.

Chapter 4
Linear Aperture

4.1 Introduction

This chapter discusses the use of the linear apertures on the *mach64*. The apertures provide a means to access all of framebuffer memory without resorting to bank switching.

The first and most obvious advantage of having a *mach64* Display Adapter is the ability to access the entire video memory as a linear frame buffer, regardless of the spatial resolution and pixel depth. The Linear Frame Buffer is accessed via the Linear Aperture or Big Aperture. Similar to the standard IBM VGA mode 13h, the Linear Frame Buffer allows the programmer access to all points on the display without the impediment of having to bank switch the standard 64KB VGA aperture. The *mach64* allows for either a 4MB or an 8MB aperture, which is ample for all supported modes.

The following sections will outline all of the required steps to implement usage of the Linear Frame Buffer.

4.2 Aperture Base Address

To use the linear aperture, the Base Linear Address and the size of the aperture must be determined. The BIOS Query Services 06h and 09h provide a method for obtaining aperture location and size. The Short Query Function call (06h) returns the aperture base address, in megabytes, in the BX register, and the size of the aperture in the lower nybble of AL. The Query Device call (09h) returns the entire Query Structure. The aperture base address is located at offset 10h and the size is located at offset 12h of the Query Structure.

For non-Intel environments, the aperture base address can be found through the PCI configuration space. Since the *mach64*GX-E/F and all members of the *mach64*CT family do not support 4MB apertures, it can be safely assumed that the aperture size is 8MB for these chips.

See next page for the example code.

Example code for Querying the BIOS for the Base Address and Limit

```
int  apertureBaseMB, apertureSizeMB;

regs.h.ah = 0xA0;                  // mach64 BIOS call
regs.h.al = 0x06;                  // mach64 Service 06h: Short
                                   // Query
int386 (0x10, &regs, &regs);       // Call the Video BIOS

apertureBaseMB = regs.w.bx;              // Save base address
apertureSizeMB = (regs.h.al & 0x1F) * 4;// Save aperture size
```

4.3 Convert Physical Address

After the base address has been obtained, it may be necessary to convert this address into a format that the operating system can use. For example, under a DPMI DOS Extender environment, the above physical address information must be converted into a linear address (or logical address). DPMI service 0800h (Physical Address Mapping) will perform this mapping. Continuing the above example we will have the following:

Example code for converting the address from physical to linear

```
regs.w.ax = 0x0800;                // DPMI Service 0800h:
                                   //   Physical Addr Map
regs.w.bx = apertureBaseMB << 4;   // Convert base address in
                                   //   megabytes
regs.w.cx = 0x0000;                // into 32-bit address in
                                   //   BX:CX
regs.w.si = apertureSizeMB << 4;   // Convert size in megabytes
regs.w.di = 0x0000;                // into 32-bit value in SI:DI
int386 (0x31, &regs, &regs);       // Call DPMI
```

Other operating environments may have to manually create a protected mode selector with a selector base equal to the aperture base address and the selector limit equal to the aperture size.

4.4 Enable the Aperture

Once the aperture's location and size have been determined, the aperture should be enabled in order to use it. BIOS service 05h (Memory Aperture Service) provides a method for doing so.

Example code for enabling the aperture

```
regs.h.ah = 0xA0;                  // mach64 BIOS call
regs.h.al = 0x05;                  // mach64 Service 05h:
                                   //   Aperture Service
regs.h.cl = 0x01;                  // CL[0] = 1: Enable Linear
                                   //   Aperture
int386 (0x10, &regs, &regs);       // Call the Video BIOS
```

4.5 Using the Linear Aperture

The organization of the linear aperture is similar to VGA mode 13h except that higher resolutions and greater pixel depths are possible. Bank switching is not necessary.

4.5.1 Memory Organization Of Pixels

The draw engine directly supports pixel depths of 1, 4, 8, 15, 16, and 32 bits per pixel. The only draw function supported in packed 24 bpp mode is rectangle fill. This mode is actually an 8 bpp mode with special rotations done to the DP_FRGD_CLR, DP_BKGD_CLR, DP_WRT_MASK, and 8x8x1 monochrome pattern registers.

The CRTC supports display pixel depths of 4 and 8 bpp pseudocolor, and 15, 16, 24, and 32 bpp direct color modes.

Note that the draw engine and CRTC must be configured with the same value of BYTE_PIX_ORDER if 4 bpp mode is selected (see DP_PIX_WIDTH and CRTC_GEN_CNTL in the *mach64 Register Reference Guide*).

Bit definitions of all pixel configurations are shown below in DWORD and BYTE representations (this is the “little endian” representation). The ordinal values represent the pixel ordering in memory for a left to right pixel trajectory beginning on a DWORD boundary, i.e. the ordinal value ‘1’ represents the position in memory of the leftmost pixel in the DWORD.

1 BPP, BYTE_PIX_ORDER = 0, Draw Engine Only				
DWORD	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
BYTE	1 2 3 4 5 6 7 8	9 A B C D E F 10	11 12 13 14 15 16 17 18	19 1A 1B 1C 1D 1E 1F 20

1 BPP, BYTE_PIX_ORDER = 1, Draw Engine Only				
DWORD	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
BYTE	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

4 BPP Pseudocolor, BYTE_PIX_ORDER = 0								
DWORD	7	8	5	6	3	4	1	2
BYTE	1	2	3	4	5	6	7	8

4 BPP Pseudocolor, BYTE_PIX_ORDER = 1								
DWORD	8	7	6	5	4	3	2	1
BYTE	2	1	4	3	6	5	8	7

8 BPP Pseudocolor				
DWORD	4	3	2	1
BYTE	1	2	3	4

15 BPP (aRGB 1555)				
DWORD	Pixel 2, aRRRRRGGGGGBBBBB		Pixel 1, aRRRRRGGGGGBBBBB	
BYTE	P1 low, GGGBBBBB	P1 high, aRRRRRGG	P2 low, GGGBBBBB	P2 high, aRRRRRGG

16 BPP (RGB 565)				
DWORD	Pixel 2, RRRRRGGGGGBBBBB		Pixel 1, RRRRRGGGGGBBBBB	
BYTE	P1 low, GGGBBBBB	P1 high, RRRRRGGG	P2 low, GGGBBBBB	P2 high, RRRRRGGG

24 BPP, Display only				
DWORD	B2	R1	G1	B1
	G3	B3	R2	G2
	R4	G4	B4	R3
BYTE	B1	G1	R1	B2
	G2	R2	B3	G3
	R3	B4	G4	R4

32 BPP (RGBa 8888)				
DWORD	R	G	B	a
BYTE	a	B	G	R

Note that 4 bpp is generally not supported any more. It was useful in planar pixel mode and it allowed for 1280x1024 with only 1MB of RAM.

4.6 Complete Example of Using the Aperture

The following example is a full demonstration of aperture access.

Example On Using The Aperture

```
int apertureBaseMB, apertureSizeMB;
char *apertureLinearAddress;

// Assume that an accelerator mode (e.g. 640x480x8bpp) has
// been set up

// Call the BIOS to find the aperture
regs.h.ah = 0xA0;                  // mach64 BIOS call
regs.h.al = 0x06;                  // mach64 Service 06h: Short
                                   //   Query
int386 (0x10, &regs, &regs);       // Call the Video BIOS

apertureBaseMB = regs.w.bx;              // Save base address
apertureSizeMB = (regs.h.al & 0x1F) * 4;// Save aperture size

// Convert address returned by BIOS to linear address
regs.w.ax = 0x0800;                // DPMI Service 0800h:
                                   //   Physical Addr Map
regs.w.bx = apertureBaseMB << 4;   // Convert base address in
                                   //   megabytes
regs.w.cx = 0x0000;                // into 32-bit address in
                                   //   BX:CX
regs.w.si = apertureSizeMB << 4;   // Convert size in megabytes
regs.w.di = 0x0000;                // into 32-bit value in SI:DI
int386 (0x31, &regs, &regs);       // Call DPMI
```

```
// Linear Address is returned in BX:CX
apertureLinearAddress = (char *) ((regs.w.bx << 16L) +
                                  regs.w.cx);

// Enable the aperture
regs.h.ah = 0xA0;                 // mach64 BIOS call
regs.h.al = 0x05;                 // mach64 Service 05h:
                                  //   Aperture Service
regs.h.cl = 0x01;                 // CL[0] = 1: Enable Linear
                                  //   Aperture
int386 (0x10, &regs, &regs);      // Call the Video BIOS

// We can now directly access the aperture through
// apertureLinearAddress;
apertureLinearAddress[0] = 0xFF;  // Change pixel in upper left
                                  //   corner
```

4.7 VGA Interaction

Remember that physical memory is shared between the on-chip VGA and the accelerator. On the *mach64*GX family a logical boundary may be enabled with the MEM_CNTL register to inhibit the two logical devices from accessing the other's memory.

- **When the memory boundary is disabled**, each device (draw engine, VGA aperture or small apertures) has full access to on-board memory.
- **When the memory boundary is enabled**, any memory accesses through the VGA aperture or small apertures are inhibited. All draw engine functions that access the memory below the boundary are inhibited. The boundary may be set to zero. Remember to set all draw engine offsets above the memory boundary.
- Memory accesses through the big linear aperture are not affected by the memory boundary register.
- If the application destroys VGA memory, the application must re-initialize the VGA mode before exiting.

The memory boundary feature is not supported on the *mach64*CT.

Chapter 5
Engine Initialization

5.1 Introduction

This section covers the steps necessary to setup and use the *mach64* accelerator engine.

5.2 Background Information on the mach64 Engine

The *mach64* engine provides more efficiency in processing common drawing functions by letting the (much faster) hardware do the work. However, it is impossible to hardwire every facet of computer graphics into the accelerator, so some drawing functions may still needs to be done by software.

5.2.1 Command FIFO Queue

All writes to draw engine registers are automatically routed through a 32-bit-wide, 16-entry-deep command FIFO. All entries are consumed in the same order as they are written.

- Note that host data registers do not generate extra wait states as on the *mach32*, and complete FIFO discipline is therefore required for these registers.
- Register reads are not FIFOed in any fashion.
- Register writes to registers with DWORD offsets less than 40h are not FIFOed.

5.2.1.1 Waiting For Sufficient FIFO Entries

Prior to any writes to any draw engine register, it is essential to check the state of the command FIFO to ensure that enough FIFO entries are available. Failure to do so may cause the draw engine to lock. C source code that waits for *n* free entries is shown below:

Example Code for Waiting for Sufficient Empty Entries in the Command FIFO

```
VOID WaitForFifo(short entries)
{
    while ((regr(FIFO_STAT) & 0xffff) >
       ((UNSIGNED INT)(0x8000 >> entries)));
}
```

5.2.1.2 Resetting The FIFO

If the FIFO has locked because of improper FIFO discipline, the FIFO and the draw engine must be reset before continuing.

Example Code for Resetting the Command FIFO

```
VOID ResetEngine(VOID)
{
    // reset engine
   iow32(GEN_TEST_CNTL, (ior32 (GEN_TEST_CNTL) & 0xFFFFFEFF));
    // enable engine
   iow32(GEN_TEST_CNTL, (ior32 (GEN_TEST_CNTL) | 0x00000100));
    // ensure engine is not locked up by clearing any FIFO or
    // HOST errors
    iow32(BUS_CNTL, (ior32 (BUS_CNTL) | 0x00A00000));
}
```

5.2.1.3 Waiting For Draw Engine Idle

There are two cases where the application must wait for the draw engine to become idle:

- The first case occurs when the application is depending on the draw engine to update a register or bit field (such as DST_X, or the scissor status bits in the GUI_STAT register). The application must ensure idleness so that those registers will not be read back while in an intermediate state.
- The second one occurs when the same memory region is being accessed by the draw engine and reads/writes through an aperture at the same time. If an engine write and an aperture write are occurring in the same region, the pixel that lands on top will not be deterministic. If an engine write and an aperture read are occurring in the same region, the pixel that is read back may or may not be the pixel just drawn.

Example Code for Waiting for the Draw Engine to be Idle

```
VOID WaitForIdle(VOID)
{
    WaitForFifo(16);
    while ((regr(GUI_STAT) & 1)!= 0);
}
```

A state of idleness implies 16 free FIFO entries, but 16 free FIFO entries **do not** imply a state of idleness

5.2.2 Other Essentials

Before beginning, a firm grasp of the *mach64* accelerator registers is essential. Specifically, accessing them through I/O ports or direct memory access, where applicable. A very important register is the CONFIG_CNTL register as it indicates the current state of the apertures. BIOS Services 06h and 09h (the query functions) also report the aperture state and may be used instead.

5.3 Preliminary Essentials

Any programming of the engine must perform certain tasks for initialization. This section outlines these tasks. Once the following tasks are complete, the *mach64* is set up to make full use of the engine.

5.3.1 *mach64* Detection

Before attempting to initialize the engine, ensure that a *mach64* card is present and functioning properly. See *Section 3.3*: *mach64 Detection* for the detailed steps.

5.3.2 Hardware Query

It is very helpful to perform a BIOS Service 09h (Query Device), and store the information for future reference. This data contains essential information for tasks such as setting the aperture, and determining how to access the registers. A complete description of the query structure can be found in *Section A.2.: Services*

5.3.3 Save/Restore Old Video Mode Information

Normally, it should be possible to return the system to its original state after using the engine. Thus, old video mode information should be saved so that this information is not lost.

5.3.4 Open Mode

For the engine, there are a few extra steps that are needed in opening a new video mode over simply performing a BIOS call 02h. Section *5.4* outlines all the essentials for opening and closing an engine mode.

5.3.5 Initializing The Engine

Finally, the engine must actually be initialized to a known state. The FIFO queue must be cleared and many of the registers must be reset. Section *5.4* details all the necessary requirements.

5.4 Opening and Closing a Mode

5.4.1 Opening

Opening an accelerator mode involves the following steps:

1. Determine if the *mach64* board will support the requested mode.

2. Initialize the Linear Aperture and/or the Small Apertures. Check the Query Structure to determine which apertures are available.

3. Load and Set the Mode with BIOS Service 02h.

4. Reset the *mach64* Engine. See *Section 5.2.1.2 Resetting the FIFO*.

5. Initialize the palette.

For the *mach64*GX family, palette initialization is only necessary for 4 bpp and 8 bpp. Since the *mach64*CT family contains an internal DAC, special handling is required as indicated below.

5.4.1.4 Programming The Internal DAC On The *mach64*CT Family

The internal DAC on the *mach64*CT family is upward compatible with a stock VGA DAC. For 4 and 8 bits per pixel (bpp) modes, the data is masked with DAC_MASK and then used to index a 256 entry look-up table (LUT) or palette. The LUT is 18 bits wide (RGB 6:6:6). The color in the palette corresponding to the index is then displayed. DAC_8BIT_EN@DAC_CNTL should be set to zero (6 bits operation) for these modes. The lower two bits are ignored, so when data is written to the palette, the value should be shifted up by 2 bits (for example, 0x3F becomes 0xFC).

For 15, 16, 24 and 32 bpp modes, the data is represented directly. For example, a value of 0x001F in video memory for a 16 bpp mode will be displayed as LIGHT BLUE. In order for the correct colors to display for these modes, the palette addresses must be initialized. Also, the DAC_8BIT_EN@DAC_CNTL should be set to one (8 bits operation). An initialization example follows:

Example for Initializing the Internal DAC on the *mach64*CT Family

```
#define    DAC_W_INDEX 0
#define    DAC_DATA    1
#define    DAC_MASK 2
#define    DAC_R_INDEX 3

void InitHiColorPaletteForCT(void)
{
   int index;
   // Set to 8 bit DAC operation (bit 8 in DAC_CNTL).
   iow8 (ioDAC_CNTL + 1, ior8 (ioDAC_CNTL + 1)  |  0x01);

   // Set the DAC MASK to FFh.
   iow8 (ioDAC_REGS + DAC_MASK, 0xFF);

   // Fill the palette starting at 0 to insure direct color
   // mapping is employed.
   iow8 (ioDAC_REGS + DAC_W_INDEX, 0);
   for  (index = 0;  index < 256;  index++)
   {
      iow8 (ioDAC_REGS + DAC_DATA,   index);
      iow8 (ioDAC_REGS + DAC_DATA,   index);
      iow8 (ioDAC_REGS + DAC_DATA,   index);
   }
}
```

If the mode is invoked by the BIOS, the palette initialization will be done there.

The internal 8-bit DAC registers may be programmed through the VGA DAC I/O addresses (3C6 through 3C9), the accelerator I/O space (I/O register select 17, byte offsets 0 through 3), or the accelerator memory mapped space (DWORD memory offset 30, byte offsets 0 through 3). Each of these address mappings correspond to the same DAC registers.

Table 5-1 DAC Register Mappings

DAC Register Mappings			
DAC Register	**VGA I/O Address**	**Accelerator I/O Select**	**Accelerator Memory Mapped Byte Offset**
DAC_MASK	3C6	17, offset 2 (5EEE, 5DCA or 5DDE)	C2
DAC_R_INDEX	3C7	17, offset 3 (5EEF, 5DCB or 5DDF)	C3
DAC_W_INDEX	3C8	17, offset 0 (5EEC, 5DC8 or 5DDC)	C0
DAC_DATA	3C9	17, offset 1 (5EED, 5DC9 or 5DDD)	C1

5.4.2 Reading from the Palette

1. Write the desired palette entry whose color will be read to the DAC_R_INDEX register.

2. Read from the DAC_DATA register three times in succession. The first read is the red component of the color data; the next is green; and the last is blue.

3. The DAC_R_INDEX register will auto-increment after the last read from DAC_DATA so that the host may read from the next palette entry without re-writing the DAC_R_INDEX register. Repeat step 2 to read successive palette entries.

5.4.3 Writing to the Palette

1. Write the palette entry that the host desires to be programmed to the DAC_W_INDEX register.

2. Write the red component data to the DAC_DATA register, followed in succession by the green and blue components to the same register. Remember that in 6 bit mode, the high two bits of the pixel component data are ignored.

3. The DAC_W_INDEX register will auto-increment after the last write to DAC_DATA so that the host may program the next palette entry without re-writing the DAC_W_INDEX register. Repeat step 2 to write successive palette entries.

5.5 Initializing the Engine

The following sections summarize the key items involved in initializing the *mach64* engine to a known state.

- Reset and enable the *mach64* engine.
- Set the VGA page pointers, if needed.
- Setup a standard engine context.

The engine must be reset, enabled, and the FIFO must be cleared. Resetting and enabling the engine is normally done by first clearing, then setting GEN_GUI_EN@GEN_TEST_CNTL. Clearing a locked FIFO involves setting BUS_FIFO_ERR_ACK@BUS_CNTL and BUS_HOST_ERR_ACK@BUS_CNTL. The function **ResetEngine ( )**, as defined in *Section 5.2.1.2*, gives an example of the above steps.

The dual 32KB small aperture page pointers MEM_VGA_RP_SEL and MEM_VGA_WP_SEL, are set to the start of display memory. Normally, the lower page pointers are set to page 0 and the upper page pointers are set to page 1 to provide compatibility with a standard 64KB VGA aperture.

Setting up a standard engine context is discussed in detail below.

5.5.1 Setup Standard Engine Context

The following table indicates a suggested set of initialized values for the *mach64* engine.

Table 5-2 Recommended Initialization Values

Recommended Initialization Values For *mach64* Engine		
Register Group	**Register Name**	**Initialized Value**
Context Control	CONTEXT_MASK	0xFFFFFFFF
Destination Draw	DST_OFF_PITCH	pitch: (mode pitch)/8, offset: 0
	DST_Y_X	0
	DST_HEIGHT	0
	DST_BRES_ERR	0
	DST_BRES_INC	0
	DST_BRES_DEC	0
	DST_CNTL	x: left to right, y: top to bottom, last pel: enable

Table 5-2 Recommended Initialization Values (Continued)

Recommended Initialization Values For *mach64* Engine		
Register Group	**Register Name**	**Initialized Value**
Source Draw	SRC_OFF_PITCH	pitch: (mode pitch)/8, offset: 0
	SRC_Y_X	0
	SRC_HEIGHT1_WIDTH1	height: 1, width: 1
	SRC_Y_X_START	0
	SRC_HEIGHT2_WIDTH2	height: 1, width: 1
	SRC_CNTL	direction: left to right, trajectory: unbounded Y
Host Data	HOST_CNTL	0
Pattern	PAT_REG0	0
	PAT_REG1	0
	PAT_CNTL	0
Scissor	SC_LEFT	0
	SC_TOP	0
	SC_BOTTOM	(mode y resolution) - 1
	SC_RIGHT	(mode pitch) - 1
Data Path	DP_BKGD_CLR	0 (normally Black)
	DP_FRGD_CLR	0xFFFFFFFF (normally White)
	DP_WRITE_MASK	0xFFFFFFFF
	DP_MIX	foreground: SRC, background: DST
	DP_SRC	foreground: foreground, background: background, mono: always ‘1’
Color Compare	CLR_CMP_CLR	0
	CLR_CMP_MASK	0xFFFFFFFF
	CLR_CMP_CNTL	compare: false, key: destination

In addition to the above, the registers DP_PIX_WIDTH and DP_CHAIN_MASK need to be set differently depending on the pixel depth of the display.

Table 5-3 Pixel Depth-Dependent Register Initialization

Pixel Depth-Dependent Register Initialization		
Pixel Depth	**DP_PIX_WIDTH**	**DP_CHAIN_MASK**
4	host, source, destination: all 4 bpp, pixel order: MSB to LSB	0x8888
8	host: 8 bpp, source: 8 bpp, destination: 8 bpp	0x8080
15	host: 15 bpp, source: 15 bpp, destination: 15 bpp	0x4210
16	host: 16 bpp, source: 16 bpp, destination: 16 bpp	0x8410
24	host: 8 bpp, source: 8 bpp, destination: 8 bpp	0x8080
32	host: 32 bpp, source: 32 bpp, destination: 32 bpp	0x8080

Finally, when the above registers have been set, generate a **wait_for_idle ( )** call to complete initialization.

5.5.2 InitEngine Example

The following example is the implementation of the engine initialization function **init_engine ( )** in the *mach64* sample code.

Example Code for Initializing The Engine

```
void init_engine (void)
{
   unsigned long pitch_value, xres, yres;

   // determine modal information from global mode structure
   xres = (unsigned long)(MODE_INFO.xres);
   yres = (unsigned long) (MODE_INFO.yres);
   pitch_value = (unsigned long) (MODE_INFO.pitch);

   if (MODE_INFO.bpp == 24)
   {
   // In 24 bpp, the engine is in 8 bpp - this requires that all
   // horizontal coordinates and widths must be adjusted
   pitch_value = pitch_value * 3;
   }
   // Reset engine, enable, and clear any engine errors
   reset_engine();
   // Ensure that vga page pointers are set to zero - the upper
      // page pointers are set to 1 to handle overflows in the
```

```
// lower page
iow32 (MEM_VGA_WP_SEL, 0x00010000);
iow32 (MEM_VGA_RP_SEL, 0x00010000);

// ---- Setup standard engine context ----

// All GUI registers here are FIFOed - therefore, wait for
// the appropriate number of empty FIFO entries
wait_for_fifo(14);

// enable all registers to be loaded for context loads
regw(CONTEXT_MASK, 0xFFFFFFFF);

// set destination pitch to modal pitch, set offset to zero
regw(DST_OFF_PITCH, (pitch_value / 8) << 22);

// zero these registers (set them to a known state)
regw(DST_Y_X,  0);
regw(DST_HEIGHT,  0);
regw(DST_BRES_ERR,  0);
regw(DST_BRES_INC,  0);
regw(DST_BRES_DEC,  0);

// set destination drawing attributes
regw(DST_CNTL, DST_LAST_PEL  |  DST_Y_TOP_TO_BOTTOM  |
               DST_X_LEFT_TO_RIGHT);

// set source pitch to modal pitch, set offset to zero
regw(SRC_OFF_PITCH,  (pitch_value / 8)  <<  22);

// set these registers to a known state
regw(SRC_Y_X,  0);
regw(SRC_HEIGHT1_WIDTH1,  1);
regw(SRC_Y_X_START,  0);
regw(SRC_HEIGHT2_WIDTH2,  1);

// set  source pixel retrieving attributes
regw(SRC_CNTL,  SRC_LINE_X_LEFT_TO_RIGHT);

// set host attributes
```

```
    wait_for_fifo (13);
    regw(HOST_CNTL,  0);

    // set pattern attributes
    regw(PAT_REG0,  0);
    regw(PAT_REG1,  0);
    regw(PAT_CNTL,  0);

    // set scissors to modal size
    regw(SC_LEFT,  0);
    regw(SC_TOP,  0);
    regw(SC_BOTTOM,  yres-1);
    regw(SC_RIGHT,  pitch_value-1);

    // set background color to minimum value (usually BLACK)
    regw(DP_BKGD_CLR,  0);

    // set foreground color to maximum value (usually WHITE)
    regw(DP_FRGD_CLR,  0xFFFFFFFF);

    // set write mask to effect all pixel bits
    regw(DP_WRITE_MASK,  0xFFFFFFFF);

    // set foreground mix to overpaint and background mix to
    // no-effect
    regw(DP_MIX, FRGD_MIX_S  |  BKGD_MIX_D);

    // set primary source pixel channel to foreground color
    // register
    regw(DP_SRC,  FRGD_SRC_FRGD_CLR);

    // set compare functionality to false (no-effect on
    // destination)
    wait_for_fifo(3);
    regw(CLR_CMP_CLR,  0);
    regw(CLR_CMP_MASK,  0xFFFFFFFF);
    regw(CLR_CMP_CNTL,  0);

    // set pixel depth
  switch(MODE_INFO.bpp)
```

```
{
      case 4 :
         wait_for_fifo(2);
         regw(DP_PIX_WIDTH, HOST_4BPP |  SRC_4BPP  |
                  DST_4BPP  |  BYTE_ORDER_MSB_TO_LSB);
         regw(DP_CHAIN_MASK, 0x8888);
         break;
      case 8 :
         wait_for_fifo(2);
         regw(DP_PIX_WIDTH, HOST_8BPP | SRC_8BPP | DST_8BPP |
                  BYTE_ORDER_LSB_TO_MSB);
         regw(DP_CHAIN_MASK, 0x8080);
         break;
      case 15:
      case 16:
         if (MODE_INFO.depth == 555)
         {
            wait_for_fifo(2);
            regw (DP_PIX_WIDTH, HOST_15BPP | SRC_15BPP |
                  DST_15BPP |  BYTE_ORDER_LSB_TO_MSB);
            regw (DP_CHAIN_MASK, 0x4210);
         }
         else
         {
            wait_for_fifo(2);
            regw (DP_PIX_WIDTH, HOST_16BPP | SRC_16BPP |
                  DST_16BPP |  BYTE_ORDER_LSB_TO_MSB);
            regw (DP_CHAIN_MASK, 0x8410);
         }
         break;
      case 24:
         wait_for_fifo(2);
         regw (DP_PIX_WIDTH, HOST_8BPP | SRC_8BPP |
                  DST_8BPP | BYTE_ORDER_LSB_TO_MSB);
         regw (DP_CHAIN_MASK, 0x8080);
         break;
```

```
                case 32:
                    wait_for_fifo(2);
                    regw (DP_PIX_WIDTH, HOST_32BPP | SRC_32BPP |
                              DST_32BPP |  BYTE_ORDER_LSB_TO_MSB);
                    regw (DP_CHAIN_MASK, 0x8080);
                    break;
            }
            wait_for_idle ( ); // insure engine is idle before leaving
        }
```

This page intentionally left blank.

Chapter 6
Engine Operations

6.1 Introduction

This chapter demonstrates standard *mach64* accelerator operations.

6.2 Background Information

6.2.1 Details About the Registers

The following is a summary of *mach64* register groups and their functions and purpose.

6.2.1.1 Accelerator CRTC and DAC Registers

Clock Control: Used for controlling the frequency synthesizer.

CRTC: Used for setting up the Cathode Ray Tube Controller.

DAC: Used for programming the Digital-to-Analog Converter.

Hardware Cursor: Used for programming the hardware cursor.

Overscan: Used for setting up the color in the overscan display area.

6.2.1.2 Setup and Control Registers

Bus Control: Used for Bus-specific access.

Configuration: Used for configuring the *mach64* engine.

Memory Control: Used for setting up the apertures.

Scratch Pad: Used by the BIOS at boot time.

Test: Used when the *mach64* is put into diagnostic mode.

6.2.1.3 Draw Engine Control Registers

Color Compare: Used in setting up the color comparator circuit.

Context Control: Used with display contexts.

Data Path: Used to set up the pixel data path.

Engine Status: Provides information on the current status of the draw engine.

FIFO Status: Provides information on the current status of the FIFO Queue.

Host Data: Used to provide data from the host to the *mach64* engine.

Pattern: Used when drawing patterned lines or rectangles.

Scissor: Used to inhibit the draw engine outside a specified region.

6.2.1.4 Draw Engine Trajectory Registers

Destination Draw Engine: Used to set up the destination trajectory for the draw operation.

Source Draw Engine: Used to set up the source trajectory for the draw operation.

6.2.2 Logical Pixel Data Path

This section describes the internal architecture of the *mach64* graphics coprocessors. This powerful architecture is referred to as the **Pixel Data Path**. It provides great flexibility in the way the coprocessor may use color data from different sources to modify the destination pixels. The accompanying figure shows a block diagram of the pixel data path. The pixel data path is central to all the coprocessor drawing operations. As such, a thorough understanding of this architecture and the operation of its components is essential to programming the coprocessor.

Before the pixel data path is described, a number of key terms will be defined. **Destination** refers to a region in on-screen or off-screen display memory that will be affected by a drawing operation. **Source** refers to the provider of the data used during a drawing operation to affect a destination pixel region. The source data may be set to specific colors (solid rectangle fills, solid lines, etc.); supplied by the host through data registers (pattern or host registers); read from a region in on-screen or off-screen display memory through source trajectories (bitblits). A **multiplexer**, or **mux**, is a switching device that uses one or more control lines to select a unique output from several inputs. An

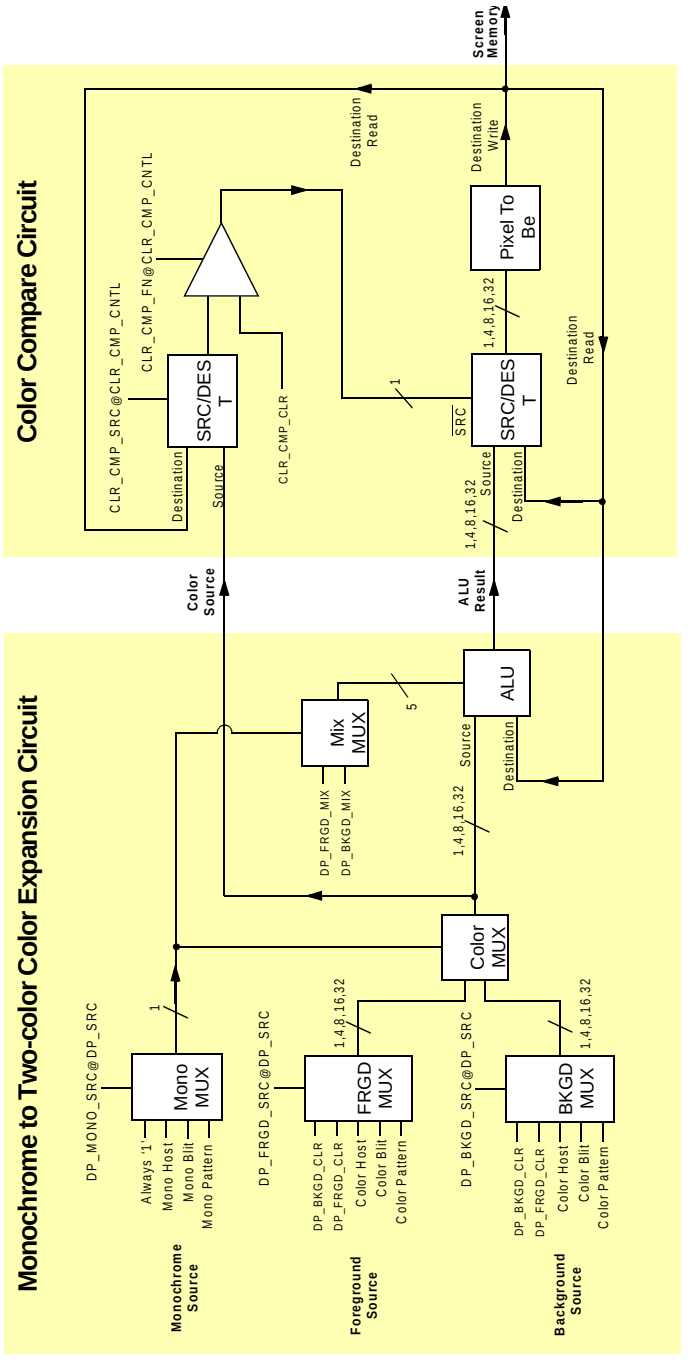
Arithmetic Logic Unit, or **ALU**, is a processor that performs a Boolean logic or arithmetic operation on two or more operands to produce an output. The operation performed by the ALU is referred to as the mix.

The logical pixel data path gives a basic understanding of how the combination of input values stored in the GUI registers result in screen output. It can be broken up into two distinct units: The monochrome to two-color color expansion circuit, and the color compare circuit. The physical data path is actually 64 bits wide for all color depths. Therefore, in 8 bpp modes, eight pixels are processed simultaneously; in 16 bpp modes, four pixels are processed simultaneously, and so on.

6.2.2.1 Monochrome To Two-color Color Expansion Circuit

This is the first half of the logical pixel data path. Its purpose is to generate source data. Various registers are set to define what the source data will be. Specifications set include color, pattern, trajectory, mixing logic, etc. In the pixel data path architecture, one of two color source units may provide the source data used to modify the destination pixels. These color sources are referred to as the **foreground source** and **background source**. During each drawing operation, one of these two sources is always selected for the source of each pixel area.

The selections and control of the source all begin with the DP_SRC register. Referring to the accompanying diagram, the MONO MUX, FRGD MUX, and BKGD MUX are all controlled by DP_SRC. These controls determine what the mono, foreground, and background source will be.



Note: These two blocks are VERY IMPORTANT in understanding the *mach64*.

Figure 6-2. Color Expansion and Color Compare Circuits

Mono Source: The mono MUX specifies the source of the mono source (control bit stream). This bit stream contains values of either '0' or '1', each corresponding to a pixel. Each pixel has one and only one value. This value is used to control the color mux and mix mux. If the value is '0', the background color and mix is used. If the value is '1', the foreground color and mix is used. There are four possible settings for the mono source, as set out by the DP_SRC register:

- **Always '1'**: This is a trivial source and is used for simple blits and drawing functions. It forces the foreground source and foreground mix to always be used. The background source and mix is ignored.
- **Mono Pattern**: The source here becomes the 8x8 fixed mono pattern. The PAT_CNTL register enables the mono pattern, and the PAT_REG registers define it. See Section *6.2.2.4 Pattern Consumption* for details on how the pattern registers are interpreted.
- **Mono Host**: Setting the mono source to mono host forces the input to come from the HOST_DATA registers. The data must be set up previously, and the HOST_CNTL register must also be initialized. HOST_CNTL controls the consumption of host data on 1bpp and 4bpp data. If HOST_BYTE_ALIGN@ HOST_CNTL is set, host consumption advances to the nearest byte boundary whenever the destination trajectory advances in the Y directions. The host pixel depth must be set to monochrome with DP_PIX_WIDTH, and the monochrome data is color expanded into foreground and background color sources. Destination pixel depth may be set to any valid pixel depth. Section *6.2.2.3 Host Data Consumption* gives a detailed explanation on the consumption of the host data.
- **Mono Blit**: This source selects one of four possible source trajectories (see trajectories sect...) using the SRC_CNTL register. The source pixel depth must be set to monochrome with DP_PIX_WIDTH, and the monochrome data is color expanded into foreground and background color sources. Destination pixel depth may be set to any valid pixel depth. The source is defined by the source trajectory registers. More in trajectories can be found in section 6.1.3 Trajectories.

Foreground and Background Source: The foreground and background sources are identical in their characteristics. They contain color information only, and are referred to as the color source. One of five settings can be independently selected for the foreground and background source. These settings are described as follows:

- **DP_BKGD_CLR**: 1 to 32 bit value corresponding to a color located in the background color register. The number of bits used varies depending on the graphics mode used.
- **DP_FRGD_CLR**: Same as DP_BKGD_CLR, but using the foreground color register.
- **Color Blit**: Similar to Mono Blit, except DP_PIX_WIDTH is set to the appropriate pixel depth. Again, the source is defined by the source trajectory registers.

- **Color Pattern**: The source is the 4x2 and 8x1 fixed color patterns set up in PAT_REG. These patterns are only useful in 8bpp draw mode. PAT_CNTL is used to select which of the color patterns is to be used.
- **Color Host**: Similar to the Mono Host, the source is taken from HOST_DATA and controlled by HOST_CNTL. If the Color Host is selected for one of the color sources, host data will be consumed for every pixel, regardless of whether the color MUX selects that color source.

Note that the host pixel depth must be set to the same pixel depth as the destination pixel depth with DP_PIX_WIDTH

Color MUX: The Color MUX is fairly straightforward. The bit stream generated by the mono MUX determines whether the output of the foreground MUX or background MUX is used to generate the output color. As indicated in Mono Source, '0' for background source and '1' for foreground source.

Mix MUX: The Mix MUX is controlled by the same bit stream as the Color MUX and determines how the output gets mixed before it is displayed. Once again, the control bits are interpreted as '0' for background mix and '1' for foreground mix.

ALU: The ALU performs the actual mixing of the color source (source input) and destination read (destination input) based on the Mix MUX output. The result of the mix is passed on to the logical color compare circuit.

6.2.2.2 Color Compare Circuit

The second half of the Logical Pixel Data Path is the color compare circuit. It is useful in performing operations such as transparent blits. It is driven by the CLR_CMP_CNTL register. Source data is inputted to this circuit and the color compare registers determine if the source data (output of the ALU) gets displayed to the screen or not (current pixel in video memory is re-outputted).

The comparator (located at the center of the diagram) is the heart of the color compare circuit. This determines what will be outputted to the screen. Its control is CLR_CMP_FCN@CLR_CMP_CNTL. This field determines how the source data is compared to CLR_CMP_CLR, and can be set to TRUE, FALSE, EQUAL, or NOT EQUAL)

Trivial Cases (TRUE or FALSE): Trivial cases are when this field is either true or false. If it is set to false, the output of the comparator is always false, resulting in the source data always being outputted to the screen. In essence, it's as if the color compare circuit were not even there, and the ALU result from the monochrome to two-color expansion circuit

were outputted directly to the screen. If the field is set to true, the output of the comparator is always true, and the bottom SRC/DEST MUX only allows the destination read to pass. Here, the destination is outputted to itself, and the screen contents don't change. The source data is basically ignored.

Non-Trivial Cases (EQUAL or NOT EQUAL): Non-trivial cases are when CLR_CMP_FCN is set to CLR_CMP_CLR or not (CLR_CMP_CLR). In these cases the result of the top left SRC/DEST MUX is compared with the contents of CLR_CMP_CLR.

The top left SRC/DEST MUX is controlled by CLR_CMP_SRC@CLR_CMP_CNTL. It basically specifies whether CLR_CMP_CLR is compared with the source or destination. It is only used if CLR_CMP_FCN is set to a non-trivial value. The output of this MUX is compared with CLR_CMP_CLR (EQUAL or NOT_EQUAL), and the result of the comparator is either true or false. This result gets fed into the bottom MUX and determines if the source (output of the ALU circuit) or destination (current pixel in video memory) gets outputted to the screen.

6.2.2.3 Host Data Consumption

The following tables illustrate the order in which pixels are consumed from the HOST_DATA register. The shaded numbers indicate the bit position within the HOST_DATA register. The numbers in the table indicate the order of pixel consumption, starting from zero.

	HOST_DATA															
	31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Monochrome or 1 bpp, left-to-right, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=0	24	25	26	27	28	29	30	31	16	17	18	19	20	21	22	23
	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
Monochrome or 1 bpp, right-to-left, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=0	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
	23	22	21	20	19	18	17	16	31	30	29	28	27	26	25	24
Monochrome or 1 bpp, left-to-right, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=1	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Monochrome or 1 bpp, right-to-left, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

	HOST_DATA							
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
4 bpp, left-to-right, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=0	6	7	4	5	2	3	0	1
4 bpp, right-to-left, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=0	1	0	3	2	5	4	7	6
4 bpp, left-to-right, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=1	7	6	5	4	3	2	1	0
4 bpp, right-to-left, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH=1	0	1	2	3	4	5	6	7
8 bpp, left-to-right	3		2		1		0	
8 bpp, right-to-left	0		1		2		3	
16 bpp, left-to-right	1				0			
16 bpp, right-to-left	0				1			

Notes:

- Host data consumption for 32 bits per pixel is self-evident.
- Host data consumption for line draws is the same as for left-to-right trajectories.
- Pixel consumption in 15 bpp modes is the same as 16 bpp modes.
- Packed 24 bpp mode is essentially 8 bpp mode. R, G, and B component data must be fed in individually in 8-bit units.
- The HOST_BYTE_ALIGN@HOST_CNTL bit may affect pixel consumption for 1 bpp and 4 bpp modes. When it is set, pixel consumption advances to the next nearest byte boundary whenever the destination advances in the Y direction. Line draws are unaffected.
- If too much host data is written to the HOST_DATA register, the extra data will be ignored.
- If not enough data is written to the HOST_DATA register, any subsequent write to a FIFOed register will cause the draw engine to *panic*; that is, the draw operation will complete with a garbage color.

6.2.2.4 Pattern Consumption

Pattern consumption for the various fixed patterns is shown in the tables below. P0 and P1 indicate PAT_REG0 and PAT_REG1 respectively. The numbers in parentheses are the bits within the pattern registers, which are used according to the destination pixel location.

Monochrome 8x8 fixed pattern, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH = 0								
(DST_Y mod 8)	(DST_X mod 8)							
	0	1	2	3	4	5	6	7
0	P0(7)	P0(6)	P0(5)	P0(4)	P0(3)	P0(2)	P0(1)	P0(0)
1	P0(15)	P0(14)	P0(13)	P0(12)	P0(11)	P0(10)	P0(9)	P0(8)
2	P0(23)	P0(22)	P0(21)	P0(20)	P0(19)	P0(18)	P0(17)	P0(16)
3	P0(31)	P0(30)	P0(29)	P0(28)	P0(27)	P0(26)	P0(25)	P0(24)
4	P1(7)	P1(6)	P1(5)	P1(4)	P1(3)	P1(2)	P1(1)	P1(0)
5	P1(15)	P1(14)	P1(13)	P1(12)	P1(11)	P1(10)	P1(9)	P1(8)
6	P1(23)	P1(22)	P1(21)	P1(20)	P1(19)	P1(18)	P1(17)	P1(16)
7	P1(31)	P1(30)	P1(29)	P1(28)	P1(27)	P1(26)	P1(25)	P1(24)

Monochrome 8x8 fixed pattern, DP_BYTE_PIX_ORDER@DP_PIX_WIDTH = 1								
(DST_Y mod 8)	(DST_X mod 8)							
	0	1	2	3	4	5	6	7
0	P0(0)	P0(1)	P0(2)	P0(3)	P0(4)	P0(5)	P0(6)	P0(7)
1	P0(8)	P0(9)	P0(10)	P0(11)	P0(12)	P0(13)	P0(14)	P0(15)
2	P0(16)	P0(17)	P0(18)	P0(19)	P0(20)	P0(21)	P0(22)	P0(23)
3	P0(24)	P0(25)	P0(26)	P0(27)	P0(28)	P0(29)	P0(30)	P0(31)
4	P1(0)	P1(1)	P1(2)	P1(3)	P1(4)	P1(5)	P1(6)	P1(7)
5	P1(8)	P1(9)	P1(10)	P1(11)	P1(12)	P1(13)	P1(14)	P1(15)
6	P1(16)	P1(17)	P1(18)	P1(19)	P1(20)	P1(21)	P1(22)	P1(23)
7	P1(24)	P1(25)	P1(26)	P1(27)	P1(28)	P1(29)	P1(30)	P1(31)

8 bpp, 4x2 fixed pattern				
(DST_Y mod 2)	(DST_X mod 4)			
	0	1	2	3
0	P0(7:0)	P0(15:8)	P0(23:16)	P0(31:24)
1	P1(7:0)	P1(15:8)	P1(23:16)	P1(31:24)

8 bpp, 8x1 fixed pattern							
(DST_X mod 8)							
0	1	2	3	4	5	6	7
P0(7:0)	P0(15:8)	P0(23:16)	P0(31:17)	P1(7:0)	P1(15:8)	P1(23:16)	P1(31:24)

6.2.3 Trajectories

A **trajectory** is the path traversed through display memory while reading source data or drawing destination data during a raster operation. A trajectory is defined by a set of parameters that describe its location, dimensions, and attributes, such as its starting point in memory, width and height if a rectangular trajectory, and the direction in which pixel data is read or written. A trajectory may extend through a linear, continuous region in memory starting from a specific location. Alternatively, its route may map a rectangular region relative to a coordinate system whose origin in memory and pitch between consecutive lines are defined as part of the trajectory's parameters.

The trajectories used by the *mach64* accelerator functions fall into two categories: **source trajectories**, and **destination trajectories**. A source trajectory describes a region in memory from which source data is read for a raster operation. A destination trajectory describes the region where a raster operation will draw pixel data.

Several drawing operations require data from one region of memory to be transferred to another (and in some cases, the source data and destination data may need to be combined in some fashion). For example, a bitblt (bit block transfer) operation must copy data from a rectangular region in on-screen or off-screen memory to another rectangular region in on-screen or off-screen memory. The rectangular source region may be defined by a rectangular source trajectory, while the destination region may be defined by a rectangular destination trajectory. As the destination trajectory advances through its path, it writes the data that is read through the source trajectory as it moves along its path.

Many examples showing how to set up and use source and destination trajectories are presented in subsequent sections where common accelerator functions are described. Meanwhile, the following table and sections describe the source and destination

trajectories and explain the criteria for establishing them.

The **Trajectory Registers** column specifies all the registers that need to be initialized for the desired trajectory.

The **Initiator** registers are registers that, once set, will initiate the draw operation. This means that all source trajectory registers (if any) and any other destination trajectory registers that need to be set must already be set. (i.e., this register is always the last one set.) This initiator register also indicated to the engine what the destination trajectory is.

The **Enable** bits are located in the SRC_CNTL register. They indicate what the source trajectory will be. They are usually, but not necessarily, set first. The bits are examined in the following order: SRC_LINEAR, SRC_PATT_ENA, and SRC_PATT_ROT.

	Trajectory	Trajectory Registers	Enable/Initiate	
Destination	Rectangle	DST_OFFSET, DST_PITCH, DST_X, DST_Y, DST_WIDTH, DST_HEIGHT, DST_X_DIR@DST_CNTL, DST_Y_DIR@DST_CNTL	DST_WIDTH or DST_HEIGHT_WIDTH or DST_X_WIDTH	Initiators
	Line	DST_OFFSET, DST_PITCH, DST_X, DST_Y, DST_BRES_LNTH, DST_BRES_ERR, DST_BRES_INC, DST_BRES_DEC, DST_X_DIR@DST_CNTL, DST_Y_DIR@DST_CNTL, DST_Y_MAJOR@DST_CNTL	DST_BRES_LNTH	
Source	Strictly Linear	SRC_OFFSET If destination is line, then SRC_LINE_X_DIR@SRC_CNTL, else DST_X_DIR@DST_CNTL	SRC_LINEAR@SRC_CNTL==1	Enables
	Unbounded Y	SRC_OFFSET, SRC_PITCH, SRC_X, SRC_Y, SRC_WIDTH1 If destination is line, then also SRC_LINE_X_DIR@SRC_CNTL	SRC_PATT_ENA@SRC_CNTL==0 SRC_PATT_ROT@SRC_CNTL==0 SRC_LINEAR@SRC_CNTL==0	
	General Pattern	SRC_OFFSET, SRC_PITCH, SRC_X, SRC_Y, SRC_WIDTH1, SRC_HEIGHT1 If destination is line, then also SRC_LINE_X_DIR@SRC_CNTL	SRC_PATT_ENA@SRC_CNTL==1 SRC_PATT_ROT@SRC_CNTL==0 SRC_LINEAR@SRC_CNTL==0	
	General Pattern with Rotation	SRC_OFFSET, SRC_PITCH, SRC_X, SRC_Y, SRC_WIDTH1, SRC_HEIGHT1, SRC_X_START, SRC_Y_START, SRC_WIDTH2, SRC_HEIGHT2 If destination is line, then also SRC_LINE_X_DIR@SRC_CNTL	SRC_PATT_ENA@SRC_CNTL==1 SRC_PATT_ROT@SRC_CNTL==1 SRC_LINEAR@SRC_CNTL==0	

Notes:

- DP_PIX_WIDTH, SRC_OFF_PITCH and SRC_LINEAR@SRC_CNTL should be written before any other source registers because they do not force a recalculation of the source memory address. SRC_Y should be written to in order to force a recalculation before doing a draw operation with a blit source. Similarly, for DP_PIX_WID and DST_OFF_PITCH, a destination address recalculation can be forced by writing to DST_Y.
- SRC_WIDTH1, SRC_WIDTH2, and DST_WIDTH should never be set to zero. This is an invalid condition.

6.2.3.1 Destination Trajectory 1, Rectangular

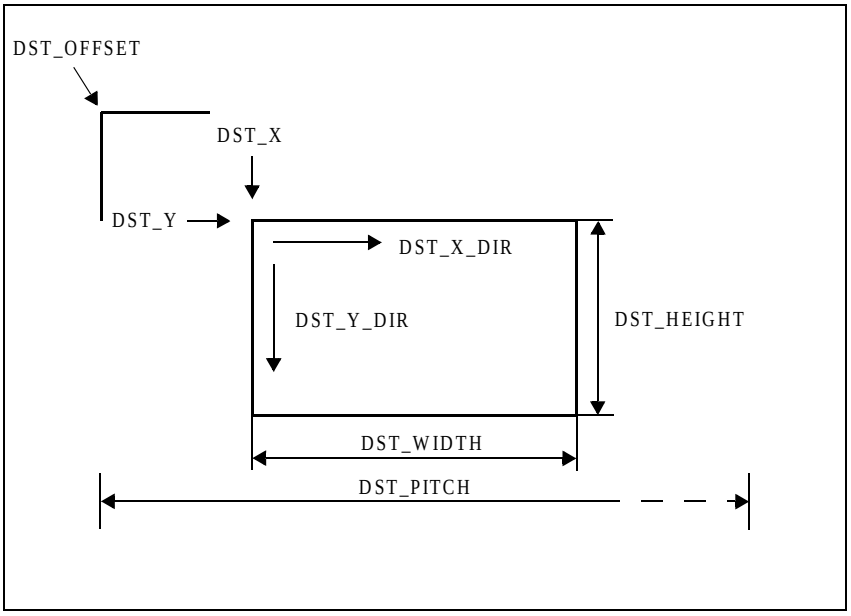
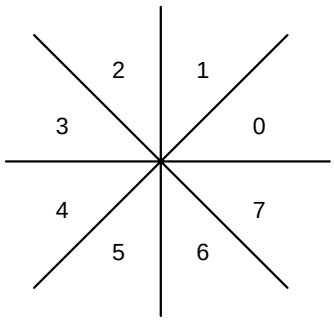


Figure 6-2. Destination Trajectory 1

Description: The trajectory begins at the initial DST_X, DST_Y location. The trajectory traverses in a left-to-right or right-to-left direction depending on DST_X_DIR@DST_CNTL, until DST_WIDTH pixels have been drawn. DST_X is then reset to the original DST_X value, and DST_Y is advanced in a top-to-bottom or bottom-to-top direction depending on DST_Y_DIR@DST_CNTL. The operation continues until DST_HEIGHT lines have been drawn.

Initiator: DST_WIDTH or DST_HEIGHT_WIDTH or DST_X_WIDTH

6.2.3.2 Destination Trajectory 2, Line



Octant	DST_X_DIR	DST_Y_MAJOR	DST_Y_DIR
0	1	0	0
1	1	1	0
2	0	1	0
3	0	0	0
4	0	0	1
5	0	1	1
6	1	1	1
7	1	0	1

Figure 6-3. Destination Trajectory 2

Description: The line drawing pseudocode that describes the draw trajectory is based on Bresenham's Algorithm:

```
for (i=0; i<DST_BRES_LNTH; i++)
{
   WritePixel(DST_X, DST_Y)
   // Advance in the major axis direction.
   if (DST_Y_MAJOR) {
      if (DST_Y_DIR) DST_Y += 1
      else DST_Y -= 1
    } else {
      if (DST_X_DIR) DST_X += 1
      else DST_X -= 1
    }
   if (DST_BRES_ERR < 0 || (DST_BRES_SIGN && DST_BRES_ERR==0))
    {
      // Axial step.
      DST_BRES_ERR += DST_BRES_INC
    }
    else
    {
      // Diagonal step.
      DST_BRES_ERR += DST_BRES_DEC
      // Advance in the minor axis direction also.
      if (DST_Y_MAJOR)
```

```
            {
               if (DST_X_DIR) DST_X += 1
               else DST_X -= 1
            }
            else
            {
               if (DST_Y_DIR) DST_Y += 1
               else DST_Y -= 1
            }
         }
      }
      if (DST_LAST_PEL) WritePixel(DST_X, DST_Y)
```

The octant bits and DST_LAST_PEL reside in DST_CNTL.

The Bresenham parameters are calculated as follows:

DST_BRES_ERR = 2 * min(|dx|,|dy|) - max(|dx|,|dy|)

DST_BRES_INC = 2 * min(|dx|,|dy|)

DST_BRES_DEC = 2 * min(|dx|,|dy|) - 2 * max(|dx|,|dy|)

DST_BRES_LNTH = max(|dx|,|dy|) + 1

Initiator: DST_BRES_LNTH

Comments: The DST_BRES_SIGN bit is used to determine whether a zero value for the Bresenham error term is considered to be positive or negative. This is important for drawing lines with the same endpoints identically no matter which direction the line draw proceeds in. It is up to the host application to set a convention (right/left or top/bottom) for using this bit.

6.2.3.3 Source Trajectory 1, Strictly Linear

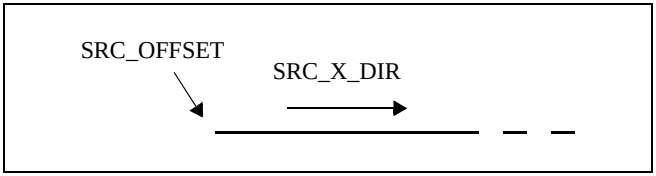


Figure 6-4. Source Trajectory 1

Description: This source trajectory traverses linearly in memory starting at SRC_OFFSET. Pixels are consumed until the destination trajectory has halted.

Criterion: SRC_LINEAR@SRC_CNTL==1

Comments: Source offset and SRC_X_DIR are the only parameters used to set up the source trajectory. SRC_X_DIR tracks DST_X_DIR@DST_CNTL in the case of blits. For lines, SRC_X_DIR equals SRC_LINE_X_DIR@SRC_CNTL. SRC_X_DIR should be set to go from left to right.

6.2.3.4 Source Trajectory 2, Unbounded Y

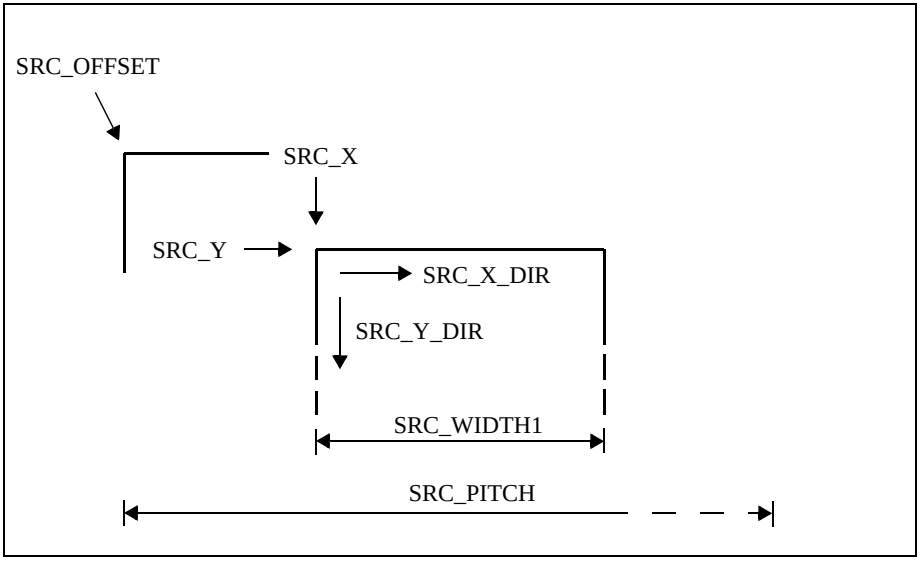


Figure 6-5. Source Trajectory 2

Description: This source trajectory begins at SRC_X, SRC_Y. This trajectory traverses in a left-to-right or right-to-left direction depending on SRC_X_DIR (equal to DST_X_DIR@DST_CNTL for destination rectangles, or SRC_LINE_X_DIR@SRC_CNTL for destination lines). When SRC_WIDTH1 pixels have been consumed, SRC_X is reset to its original value and SRC_Y is advanced in a top-to-bottom or bottom-to-top direction depending on SRC_Y_DIR (which is equal to DST_Y_DIR@DST_CNTL). Pixels are consumed until the destination trajectory has halted.

Criterion: SRC_PATT_EN@SRC_CNTL==0 and SRC_PATT_ROT@SRC_CNTL==0 and SRC_LINEAR@SRC_CNTL==0

Comments: If the destination trajectory is rectangular, SRC_X_DIR and SRC_Y_DIR track DST_X_DIR@DST_CNTL and DST_Y_DIR@DST_CNTL. For lines, SRC_LINE_X_DIR@SRC_CNTL is used and the source trajectory does not advance in the Y direction.

6.2.3.5 Source Trajectory 3, General Pattern

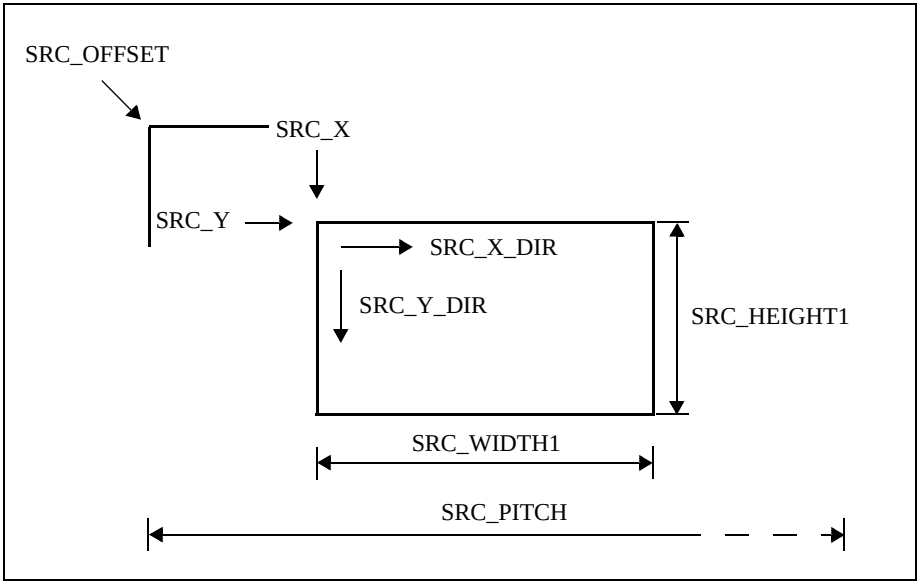


Figure 6-6. Source Trajectory 3

Description: This source trajectory begins at SRC_X, SRC_Y. This trajectory traverses in a left-to-right or right-to-left direction depending on SRC_X_DIR (equal to DST_X_DIR@DST_CNTL for destination rectangles, or SRC_LINE_X_DIR@SRC_CNTL for destination lines). When

SRC_WIDTH1 pixels have been consumed, SRC_X is reset to its original value. When the destination advances in the Y direction, SRC_X is reset to its original value and SRC_Y is advanced in a top-to-bottom or bottom-to-top direction depending on SRC_Y_DIR (which is equal to DST_Y_DIR@DST_CNTL). When SRC_HEIGHT1 lines have been consumed, SRC_Y is reset to its original value. Pixels are consumed until the destination trajectory has halted.

Criterion: SRC_PATT_EN@SRC_CNTL==1 and SRC_PATT_ROT@SRC_CNTL==0 and SRC_LINEAR@SRC_CNTL==0

Comments: If the destination trajectory is rectangular, SRC_X_DIR and SRC_Y_DIR track DST_X_DIR@DST_CNTL and DST_Y_DIR@DST_CNTL. For lines, SRC_LINE_X_DIR@SRC_CNTL is used and the source trajectory does not advance in the Y direction.

6.2.3.6 Source Trajectory 4, General Pattern With Rotation

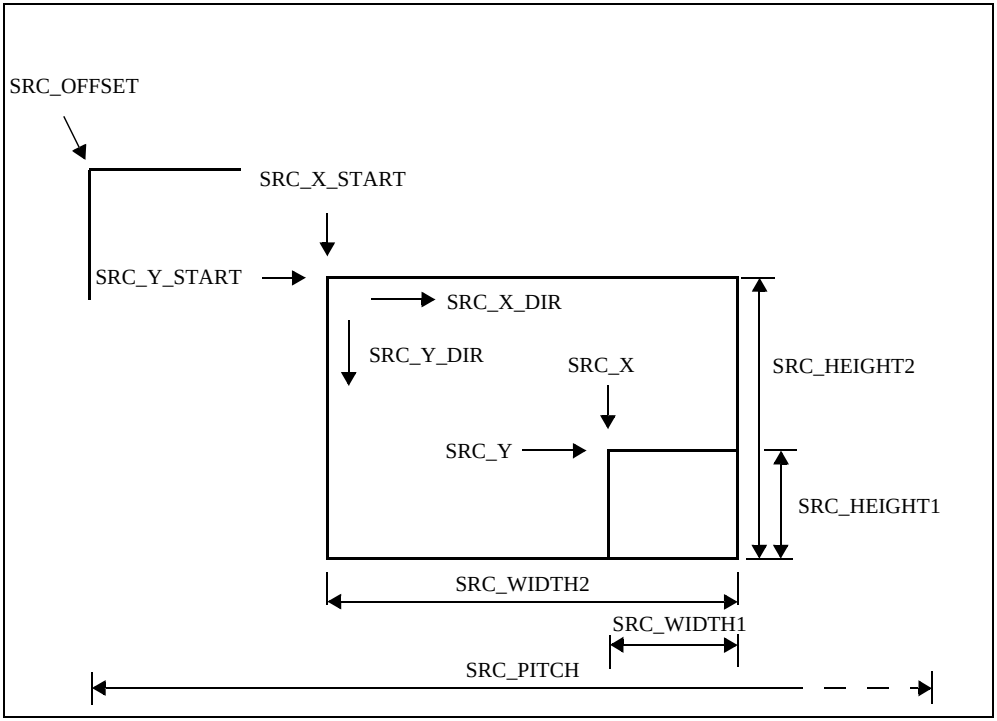


Figure 6-7. Source Trajectory 4

Description: This source trajectory begins at SRC_X, SRC_Y. This trajectory traverses in a left-to-right or right-to-left direction depending on SRC_X_DIR (equal to DST_X_DIR@DST_CNTL for destination rectangles, or SRC_LINE_X_DIR@SRC_CNTL for destination lines). When SRC_WIDTH1 pixels have been consumed, SRC_X is reset to SRC_X_START. When the destination advances in the Y direction, SRC_X is reset to SRC_X_START and SRC_Y is advanced in a top-to-bottom or bottom-to-top direction depending on SRC_Y_DIR (which is equal to DST_Y_DIR@DST_CNTL). All further traversals in the X direction use SRC_WIDTH2, instead of SRC_WIDTH1 and reset to SRC_X_START. When SRC_HEIGHT1 lines have been consumed, SRC_Y is reset to SRC_Y_START. All further traversals use SRC_HEIGHT2 instead of SRC_HEIGHT1 and reset to SRC_Y_START when the count is exhausted. Pixels are consumed until the destination trajectory has halted.

Criterion: SRC_PATT_EN@SRC_CNTL==1 and SRC_PATT_ROT@SRC_CNTL==1 and SRC_LINEAR@SRC_CNTL==0

Comments: If the destination trajectory is rectangular, SRC_X_DIR and SRC_Y_DIR track DST_X_DIR@DST_CNTL and DST_Y_DIR@DST_CNTL. For lines, SRC_LINE_X_DIR@SRC_CNTL is used and the source trajectory does not advance in the Y direction.

6.2.3.7 Trajectory Modifier 1, SRC_BYTE_ALIGN

When SRC_BYTE_ALIGN@SRC_CNTL is set, the source pointer skips to the next byte boundary when the destination trajectory advances in the Y direction. There is a similar bit for host data called HOST_BYTE_ALIGN@HOST_CNTL. These bits are only meaningful for 1 bpp or 4 bpp data. See Section *6.2.2.3*: *Host Data Consumption* for the pixel ordering.

6.2.3.8 Trajectory Modifier 2, DST_POLYGON_EN

The DST_POLYGON_EN affects both lines and blits.

When drawing a line, only a single pixel is drawn per scan line (this only affects X major lines). Horizontal lines are not drawn. Lines whose trajectory goes left of the left scissor are saturated to the left scissor.

When blitting, at the beginning of each destination line, an internal polygon fill flag is reset. If the polygon fill flag is reset, drawing is inhibited at the destination. For each pixel, an implicit 1 bpp polygon boundary source (this is neither a monochrome nor a color source, but an implicit third source) is read. If the result is '1' (a polygon edge) the polygon

fill flag is toggled. Both left and right edges of the polygon are inclusive. The right edge is optionally exclusive on the *mach64*CT family.

6.2.3.9 Trajectory Modifier 3, DP_BYTE_PIX_ORDER

The DP_BYTE_PIX_ORDER@DP_PIX_WIDTH bit affects the pixel order of both 1 bpp and 4 bpp data within a byte. This affects the source area, destination area, and host data consumption. When set, left-to-right pixel order proceeds from the least significant bit or nibble to the most significant bit or nybble within a byte. The bytewise order is unaffected. See *Section 6.2.2.3: Host Data Consumption* for the pixel ordering.

6.2.4 Side Effects Of Trajectories

A side effect is a change in the draw engine state after a draw operation has been completed. Typically, it refers to the trajectory pointers (the source and destination coordinates).

- The source pointer is always reset to the original SRC_X, SRC_Y after completion of a draw operation.
- The destination pointer is set according to the DST_X_TILE and DST_Y_TILE bits after completion of a blit operation. If DST_X_TILE is set, then DST_X = original_DST_X + DST_WIDTH for left-to-right destination trajectories, or DST_X = original_DST_X – DST_WIDTH for right-to-left destination trajectories; otherwise, it is reset to the original DST_X value from before the draw. This is also applicable for the DST_Y_TILE bit (with DST_Y and DST_HEIGHT).
- For lines, the final DST_X, DST_Y rest on the last pixel of the line. The LAST_PEL_ON bit specifies whether the last pixel on that line is drawn.

6.2.5 Source And Destination Alignment

Sources may have one of two possible alignments:

- **Source alignment**
- **Destination alignment**

Source alignment means that the top left corner of the source area is aligned to the top left corner of the destination area, as illustrated below:

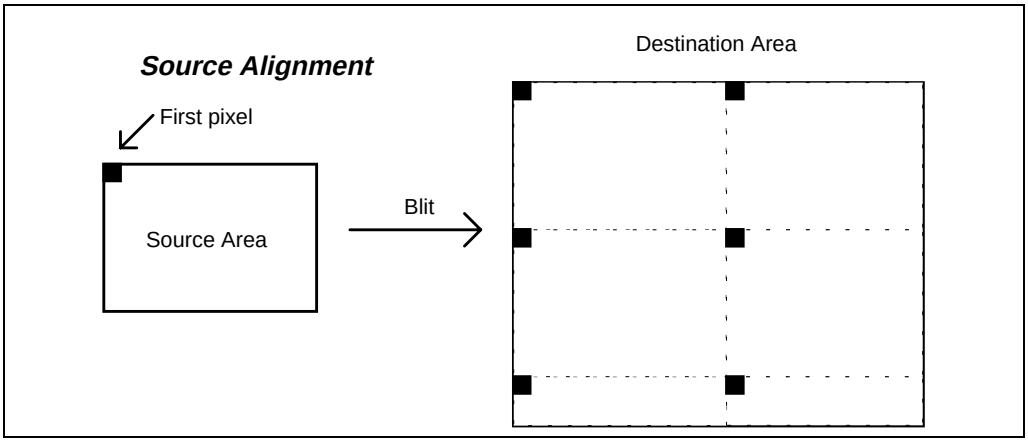


Figure 6-10. Source Alignment

Destination alignment to the Nth pixel means that the top left corner of the source area is aligned to (X mod N)==0 and (Y mod N)==0. There may in fact be different N values for the horizontal and vertical destination alignment.

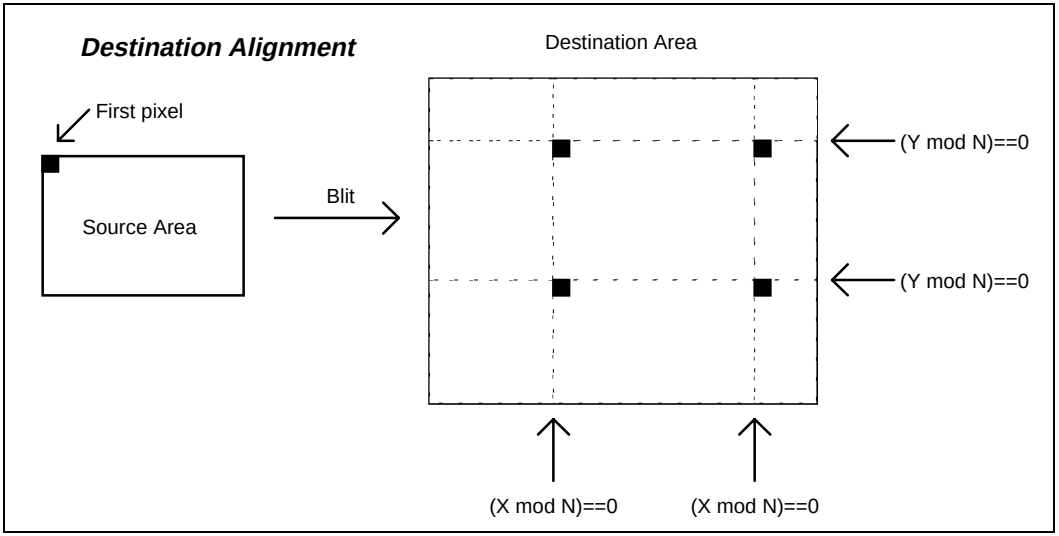


Figure 6-11. Destination Alignment

Various sources and their implicit alignments are listed below:

Table 6-1 Source Alignment

Source	Alignment
DP_FRGD_CLR	Destination aligned
DP_BKGD_CLR	Destination aligned
Fixed 8x8 mono pattern	Destination aligned X8 Y8
Fixed 4x2 color pattern	Destination aligned X4 Y2
Fixed 8x1 color pattern	Destination aligned X8
Mono host	Source aligned
Color host	Source aligned
Any blit source (strictly linear, unbounded Y, general pattern, general pattern with rotation)	Source aligned

The strict definition of **source alignment** is that a QWORD (or DWORD depending on memory type and size) for a source-aligned source is rotated to align with the destination. No rotation occurs for destination-aligned sources.

6.2.6 Source and Destination Mixing Logic

A source and destination pixel may be mixed in two ways:

- A logical operation or an averaging function may be performed on the source and destination to produce a composite pixel. The process may be referred to as an ALU function, a mix function, or a ROP (raster operation).
- The color source pixel (before ALU processing) or the destination pixel can be compared to a color compare register. If the result is FALSE, the result of the ALU is written; otherwise, the destination pixel is written back to the destination (no pixel is drawn). In this manner, the source pixel can be selectively inhibited from writing to the destination.

ALU functions and compare functions may be used at the same time, but the ALU will only operate on pixels for which the compare function returns FALSE.

The available mix functions and compare functions are listed in the tables below. The ALU will mix the source and destination data with any of the functions listed. More complex functions may be accomplished with multiple passes.

The comparison functions compare a color register against the destination data at the current pixel.

- If the result of the comparison is FALSE, the result of the ALU is written to the destination; otherwise, the destination data is written to the destination.

Table 6-2 Mix and Comparison Functions

Mix Functions	
0	not D
1	0
2	1
3	D
4	not S
5	D xor S
6	(not D) xor S
7	S
8	(not D) or (not S)
9	D or (not S)
A	(not D) or S
B	D or S
C	D and S
D	(not D) and S
E	D and (not S)
F	(not D) and (not S)
17	(D+S) >> 1

Comparison Functions	
0	FALSE
1	TRUE
2	Reserved
3	Reserved
4	Pixel != CLR_CMP_COLOR
5	Pixel == CLR_CMP_COLOR
6	Reserved
7	Reserved

Function 17h additionally requires the DP_CHAIN_MASK register to be set. Each ‘1’ in the mask will prevent the carry bit from that bit position from adding to the next bit.

6.2.7 Remarks On Pixel Depth

Not all pixel depths are created equal:

- 1 bpp mode is supported by the drawing engine but not by the CRTC. Therefore, 1 bpp mode can only be used in off-screen memory.
- Pitch is normally specified in multiples of 8 pixels. An additional restriction is that it must also fall on a 64-bit boundary. That implies that pitch for 1 bpp mode must be a multiple of 64 pixels, and pitch for 4 bpp mode must be a multiple of 16 pixels.
- The DP_BYTE_PIX_ORDER@DP_PIX_WIDTH bit only affects pixel ordering within a byte. Therefore, only 1 bpp and 4 bpp modes are affected.
- All pixel depths above 8 bpp are direct color modes. 4 bpp and 8 bpp modes are pseudocolor modes.
- Packed 24 bpp mode is actually 24 bpp CRTC mode and 8 bpp draw mode with

special rotations done on DP_FRGD_CLR, DP_BKGD_CLR, DP_WRITE_MASK, and fixed 8x8 mono patterns. See *Drawing in Packed 24 Bit Per Pixel Mode* in Section *6.4.1*.

- DP_CHAIN_MASK must be manually set for the destination pixel depth (this register only affects the mix function 17h, the averaging function). The following table lists the settings:

Table 6-3 DP_CHAIN_MASK Setting

Pixel Depth	DP_CHAIN_MASK
1 bpp	N/A
4 bpp pseudocolor	0x8888
8 bpp	0x8080
15 bpp, aRGB 1555	0x4210
16 bpp, RGB 565	0x8410
24 bpp, RGB 888	0x8080
32 bpp, RGBa 8888	0x8080

- 15 bpp and 16 bpp modes are identical draw modes, but different DAC modes must be set (use BIOS services for mode switching so the application does not have to handle it). 15 bpp mode is always RGB 555, and 16 bpp mode is always RGB 565.
- Although pixel depths for source area, destination area, and host may be set independently, the only pixel depth conversion available is 1 bpp to any pixel depth monochrome expansion. Behavior is undefined for any other mixing and matching of pixel depths.

6.3 Draw Operations

This section provides specific examples of how to set up the *mach64* engine for various trajectories. Section *6.2.1* demonstrates how to set up the destination trajectory and initiate the draw operation. Section *6.2.2* demonstrates how to set up the four basic types of source trajectory. The remaining two sections show various useful draw operations.

6.3.1 Color Source

The solid color source is the simplest form of source data. The color for drawing the line or rectangle comes from DP_FRGD_CLR alone. This is done by setting the mono source to always ‘1’. The destination trajectory registers must also be set.

6.3.1.1 Drawing Lines

Line draws are performed using an 18-bit Bresenham line draw engine.

To draw a line:

1. Set up the draw context with either a context load or many register writes.

2. Determine the direction octant so that the line trajectory will be drawn and set the DST_X_DIR, DST_Y_DIR and DST_Y_MAJOR bits accordingly. Also set the LAST_PEL_ON bit as desired (this bit only determines whether the last pixel in the line is drawn; it has no effect on the actual DST_X, DST_Y trajectory).

From the start and endpoints of the line, calculate all the Bresenham parameters and write them out to the registers.

DST_BRES_ERR = 2 * min(|dx|,|dy|) - max(|dx|,|dy|)
DST_BRES_INC = 2 * min(|dx|,|dy|)
DST_BRES_DEC = 2 * [min(|dx|,|dy|) - max(|dx|,|dy|)]

3. Write out the desired number of pixels drawn to DST_BRES_LNTH.

Line drawing is not supported in packed 24 bpp modes.

Example Code for Drawing Lines (normal or polygon outline)

```
//
------------------------------------------------------------
// DrawLine - draw a line from (x1, y1) to (x2, y2)
//
// The drawing of the last pixel in the line is determined by
// the current setting of the DST_CNTL register (LAST_PEL bit).
// The engine does not support lines in 24 bpp modes.

void DrawLine (short x1, short y1, short x2, short y2)
{
   short dx, dy;
   long minDelta, maxDelta;
   short x_dir, y_dir, y_major;

   dx = abs(x2 - x1);
   dy = abs(y2 - y1);
   minDelta = __min(dx,dy);
   maxDelta = __max(dx,dy);
```

```
// Determine the octant.
if (x1 < x2) x_dir = 1;
else x_dir = 0;
if (y1 < y2) y_dir = 0x0802;// use top/bottom for Bresenham
                            // zero sign convention
else y_dir = 0;
if (dx < dy) y_major = 4;
else y_major = 0;

// Assume that the context registers have already been set up
//   somewhere else.
// Set the line trajectory registers and initiate.
WaitForFifo(6);
// The register read of DST_CNTL is not FIFOed, so the application
//  must guarantee that there isn't a DST_CNTL somewhere in the
//  write FIFO.
// If the application cannot guarantee this, then the
//  application must provide a known value for DST_CNTL or
//  insert a WaitForIdle( ) here (a wait_for_idle will slow
//  overall performance).
WaitForIdle();
regw(DST_CNTL,(regr(DST_CNTL) & ~0x7) |
               (ULONG)(y_major | y_dir |  x_dir));
regw(DST_Y_X, ((ULONG)x1 << 16) | y1);
regw(DST_BRES_ERR, 2 * minDelta - maxDelta);
regw(DST_BRES_INC, 2 * minDelta);
regw(DST_BRES_DEC, 2 * (minDelta - maxDelta));
regw(DST_BRES_LNTH, maxDelta + 1);
}
```

6.3.1.2 Drawing Rectangles

Drawing rectangles is one of the simplest of the *mach64* operations. It is also quite versatile. Below is a sample routine to draw a rectangle. You will notice that the source is not specified in the routine itself. This allows the routine to be used for solid rectangles or pattern filled rectangles. The source registers to draw a solid rectangle is given here in the main routine. Note that 24bpp is not supported in this example, but is in the SDK sample code.

Example Code for Drawing Solid Rectangles

```
//main rectangle draw
//assume engine is initialized and mode is already set
int rcolor;     //color of rectangle
int rx;         //top left x coordinate of rectangle
int ry;         //top left y coordinate of rectangle
int rwidth;     //width of rectangle
int rheight;    //height of rectangle

WaitForFifo(2);
regw(DP_FRGD_CLR, get_color_code(rcolor);
regw(DP_SRC, BKGD_SRC_BKGD_CLR | FRGD_SRC_FRGD_CLR |
                                  MONO_SRC_ONE);
draw_rect(rx, ry, rwidth, rheight);
//end of main code


void draw_rect (int x, int y, int width, int height)
{
   WaitForFifo (4);

   // perform rectangle fill
   regw (DST_X, (unsigned long) x);
   regw (DST_Y, (unsigned long) y);
   regw (DST_HEIGHT, (unsigned long) height);
   regw (DST_WIDTH, (unsigned long) width);
}
```

Two more examples demonstrate drawing a rectangle filled with solid color data and with data provided through the HOST_DATA registers.

Example Code for Initiating a Solid Rectangle Fill

```
// Setup the draw engine context manually.
WaitForFifo(12);
regw(DP_FRGD_CLR, 0xFFFFFFFF);    // white
regw(DP_WRITE_MASK, 0xFFFFFFFF);  // enable all planes
regw(DP_PIX_WIDTH, 0x00020202);   // 8 bpp
// Note: Background mix should be set to leave_alone when not
//   being used (mono source is always_'1') because this is one
//   of the conditions for block write to be enabled.
// If the memory supports block write, the rectangle fill will
//   draw much faster.
regw(DP_MIX, 0x00070003);   // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000100);   // mono:always_'1',
                            //   frgd:DP_FRGD_CLR
regw(CLR_CMP_CNTL, 0x00000000);   // disable
regw(GUI_TRAJ_CNTL, 0x00000003);  // left-to-right,
top-to-bottom
regw(SC_LEFT_RIGHT, 0x03FF0000);  // 0 to 1023
regw(SC_TOP_BOTTOM, 0x03FF0000);  // 0 to 1023

// Setup the draw trajectory and initiate (write DST_OFF_PITCH
// first).
regw(DST_OFF_PITCH, ((ULONG)pitch << 22) | offset);
regw(DST_Y_X, ((ULONG)x << 16) | y);
regw(DST_HEIGHT_WIDTH, ((ULONG)width << 16) | height);
```

Example Code for Initiating a Rectangle Filled with Host Data

```
// Setup the draw engine context manually.
WaiForFifo(12);
regw(DP_FRGD_CLR, 0xFFFFFFFF);    // white
regw(DP_WRITE_MASK, 0xFFFFFFFF);  // enable all planes
regw(DP_PIX_WIDTH, 0x00020202);   // 8 bpp
// If the foreground mix for a color host operation is set to
//   paint (7), you might as well use the aperture because it
//   would be faster.
// It is only worthwhile to use host operations when the ALU
//   function is not trivial, or for monochrome host operations.
regw(DP_MIX, 0x00050003);   // frgd:xor, bkgd:leave_alone
regw(DP_SRC, 0x00000200);   // mono:always_'1',
```

```
                                    // frgd:color_host
regw(CLR_CMP_CNTL, 0x00000000);     // disable
regw(GUI_TRAJ_CNTL, 0x00000003);    // left-to-right,
top-to-bottom
regw(SC_LEFT_RIGHT, 0x03FF0000);    // 0 to 1023
regw(SC_TOP_BOTTOM, 0x03FF0000);    // 0 to 1023

// Setup the draw trajectory and initiate (write DST_OFF_PITCH
// first).
regw(DST_OFF_PITCH, ((ULONG)pitch << 22) | offset);
regw(DST_Y_X, ((ULONG)x << 16) | y);
regw(DST_HEIGHT_WIDTH, ((ULONG)width << 16) | height);

// Calculate the amount of data to output.
numberOfPixels = (ULONG)width * height;
numberOfDwords = numberOfPixels / pixelsPerDword;
if ((numberOfPixels % pixelsPerDword)!=0) numberOfDwords++;

// Output host data.
for (i=0; i<numberOfDwords*pixelsPerDword; i+=pixelsPerDword) {
   // This inner loop can be optimized to burst in data 16 DWORDs
   //   at a time. Only one DWORD is written at a time for
   //   simplicity. When bursting in data, first wait for 16
   //   free FIFO entries, then use REP MOVSD to HOST_DATA0
   //   through HOST_DATA16
   WaitForFifo(1);

   // Output 4 pixels of 8 bpp (byte) data in left-to-right
   // order.
   regw(HOST_DATA0, pixel[i] | ((ULONG)pixel[i+1] << 8)
                             | ((ULONG)pixel[i+2] << 16)
                             | ((ULONG)pixel[i+3] << 24));
}

// If too much data is written, the extra data will be ignored.
// If not enough data is written, then the next write to a FIFOed
//   register other than a HOST_DATA register will cause the draw
//   engine to panic, i.e. the rectangle fill will complete with
//   a garbage color.
```

It is left as an exercise for the programmer to fill a rectangle with monochrome host data (set DP_MONO_SRC@DP_SRC to “host data” and set DP_FRGD_SRC@DP_SRC and DP_BKGD_SRC@DP_SRC to any two valid color sources except for “host data”, i.e. not color_host).

6.3.2 Standard BitBlt Source

A bitblt is a **rectangle fill** that specifically uses a color blit source. There are four types of blit source trajectory, as described in Section *6.2.3*: *Trajectories*. Note that the source trajectory direction always tracks the destination trajectory direction. Blit sources are always source-aligned.

6.3.2.3 Simple 1 to 1

The DP_SRC register specifies the simple 1-1 bitblit. It is also important to set the SRC_CNTL to unbounded y (simple 1-1 bitblit).

Example Code for Initiating a Simple Blit (Unbounded Y)

```
// Use an unbounded Y source trajectory to do a rectangular blit.

// Set up the context manually.
WaitForFifo(7);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres
regw(DP_WRITE_MASK, 0xFFFFFFFF); // enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp (depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint,
                     // bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x00000000); // disable
// Set up the source trajectory (remember to write SRC_OFF_PITCH
// and SRC_CNTL first).
WaitForFifo(8);
regw(SRC_OFF_PITCH, 0x20000000); // pitch:1024(depends on
                     // mode),offset:0
regw(SRC_CNTL, 0x00000000); // unbounded Y
regw(SRC_Y_X, (srcX << 16) | srcY);
regw(SRC_WIDTH1, srcWidth);

// Set up the destination trajectory and initiate blit (write
// DST_OFF_PITCH first).
regw(DST_OFF_PITCH, 0x20000000); //pitch:1024(depends on mode),
                     // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
regw(DST_Y_X, (dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, (dstWidth << 16) | dstHeight);
```

6.3.2.4 General Pattern

Using General Pattern source implies that the source and destination are different sizes. SRC_CNTL is now set to general pattern.

Example Code for Initiating a Rectangle Filled with a General 2D Pattern

```
// Use a general pattern source trajectory to fill a rectangle
// with an area pattern.
// The source area should be smaller than the destination area for
// a visible effect.

// Set up the context manually.
WaitForFifo(7);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yes
regw(DP_WRITE_MASK, 0xFFFFFFFF); // enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp (depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x00000000); // disable

// Set up the source trajectory (remember to write SRC_OFF_PITCH
// and SRC_CNTL first).
WaitForFifo(8);
regw(SRC_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                     // offset:0
regw(SRC_CNTL, 0x00000001); // general pattern
regw(SRC_Y_X, ((ULONG)srcX << 16) | srcY);
regw(SRC_HEIGHT1_WIDTH1, ((ULONG)srcWidth << 16) | srcHeight);

// Set up the destination trajectory and initiate blit (write
// DST_OFF_PITCH first).
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                     // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, ((ULONG)dstWidth << 16) | dstHeight);
```

6.3.2.5 General Pattern With Rotation

General pattern with rotation is similar to general pattern, but allows for pattern alignment. This requires a few extra registers to be set.

Example Code for Initiating a Rectangle Filled with a Rotated 2D Pattern

```
// Use a general pattern source trajectory to fill a rectangle
// with a rotated area pattern.
// The source area should be smaller than the destination area for
// a visible effect.

// Set up the context manually.
WaitForFifo(7);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres
regw(DP_WRITE_MASK, 0xFFFFFFFF);// enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp (depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x00000000); // disable

// Set up the source trajectory (remember to write SRC_OFF_PITCH
// and SRC_CNTL first).
// srcXStart and srcYStart denote the top left corner of the
pattern
// srcX and srcY offset into that pattern
// srcWidth2 and srcHeight2 specify the pattern size
// srcWidth1 and scrHeight1 specify the size of the rectangle
bound
// by srcX,
// srcY, and the bottom right corner of the pattern
WaitForFifo(10);
regw(SRC_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                    // offset:0
regw(SRC_CNTL, 0x00000003); // general pattern with rotation
regw(SRC_Y_X_START, ((ULONG)srcXStart << 16) | srcYStart);
regw(SRC_HEIGHT2_WIDTH2, ((ULONG)srcWidth << 16) | srcHeight);
regw(SRC_Y_X, ((ULONG)srcX << 16) | srcY);
regw(SRC_HEIGHT1_WIDTH1, ((ULONG)(srcXStart+srcWidth-srcX) <<
              16) | (srcYStart+srcHeight-srcY));
```

```
// Set up the destination trajectory and initiate blit (write
// DST_OFF_PITCH first).
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                        // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, ((ULONG)dstWidth << 16) | dstHeight);
```

6.3.2.6 Strictly Linear

A very simple source is the strictly linear source. The following code is very straightforward.

Example Code for Initiating a Blit with a Linear Source

```
// Use a linear source trajectory to fill a rectangle.
// The source area would usually be packed in an offscreen
// memory area.

// Set up the context manually.
WaitForFifo(7);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres
regw(DP_WRITE_MASK, 0xFFFFFFFF);// enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp (depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x00000000); // disable

// Set up the source trajectory (remember to set SRC_WIDTH1 to
// a non-zero value).
WaitForFifo(7);
regw(SRC_OFF_PITCH, 0x20000000 | offset);  //pitch:1024(depends
                            // on mode)
regw(SRC_CNTL, 0x00000004); // linear

// Set up the destination trajectory and initiate blit (write
// DST_OFF_PITCH first).
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                        // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
```

```
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, ((ULON G)dstWidth << 16) | dstHeight);
```

6.3.3 Specialized BitBlt Source

The following examples show various examples of bitblt source while exercising the various capabilities of the color expansion circuitry in the *mach64* engine.

6.3.3.1 Monochrome Expansion

Monochrome expansions are especially useful for font caching. Monochrome expansion bitblits are very efficient in terms of storing the source information. Not only is the information packed into a linear segment of memory, but each on-screen pixel only uses one bit to store its information.

Example Code for Initiating a Monochrome Expansion Blit

```
// Assume that there is monochrome data (eg text) stored linearly
// in off-screen memory. The data is to be expanded to a
foreground
// color, the background is to be transparent.


// Set up the context manually.
WaitForFifo(8);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to 1023
regw(SC_TOP_BOTTOM, 0x03FF0000);  // 0 to 1023
regw(DP_WRITE_MASK, 0xFFFFFFFF); // enable all bit planes
regw(DP_FRGD_CLR, 0xFFFFFFFF); // white
regw(DP_PIX_WIDTH, 0x00020002); // SRC:1 bpp,
                     // DST:8bpp(depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00030100); // mono:blit,
                     // frgd:DP_FRGD_CLR
regw(CLR_CMP_CNTL, 0x00000000); // disable

// Set up the source trajectory (remember to set SRC_WIDTH1 to
// a non-zero value).
WaitForFifo(7);
regw(SRC_OFF_PITCH, 0x20000000 | offScreenOffset); //
pitch:1024
regw(SRC_CNTL, 0x00000004);           // linear

// Set up the destination trajectory and initiate blit.
```

```
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                      // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, ((ULONG)dstWidth << 16) | dstHeight);
```

6.3.3.2 General Pattern Lines

When the destination trajectory is a line, the source trajectory behaves in almost the same fashion as for a rectangular destination trajectory. The only differences are:

- The source trajectory never advances in the Y direction (the source height is implicitly equal to one).
- The source trajectory X direction is independent of the destination X direction, and can be set by the SRC_LINE_X_DIR@SRC_CNTL.

Example Code for Drawing Lines With a General Pattern

```
// Use a general pattern source to do a line pattern. Note that
// the source does not advance in the Y direction when the
// destination is a line.

// Set up the context manually.
WaitForFifo(7);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres
regw(DP_WRITE_MASK, 0xFFFFFFFF); // enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp (depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x00000000); // disable

// Set up the source trajectory (remember to write SRC_OFF_PITCH
// and SRC_CNTL first).
WaitForFifo(11);
regw(SRC_OFF_PITCH, 0x20000000 | offset); //pitch:1024(depends
                          // on mode)
regw(SRC_CNTL, 0x00000001); // general pattern
regw(SRC_Y_X, ((ULONG)srcX << 16) | srcY);
regw(SRC_HEIGHT1_WIDTH1, ((ULONG)pattLength << 16) | 1);
```

```
// Draw the line.
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
                            // mode), offset:0
regw(DST_CNTL, octant | lineOptions);
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_BRES_ERR, dstBresErr);
regw(DST_BRES_INC, dstBresInc);
regw(DST_BRES_DEC, dstBresDec);
regw(DST_BRES_LNTH, lineLength);
```

6.3.3.3 Transparent BitBlts

A transparent blit is simply a blit where a designated color (background color) from the source is inhibited from being drawn to the destination. This kind of blit is useful for copying odd-shaped objects onto a bitmapped background (games, for example). A simple blit with source compare enabled will do a transparent blit.

Example Code for a Transparent Blit

```
// Use a linear source trajectory with a transparent color to fill
// a rectangle.
// The source area would usually be packed in an offscreen memory
// area.

// Set up the context manually.
WaitForFifo(9);
regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres
regw(DP_WRITE_MASK, 0xFFFFFFFF); // enable all bit planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp(depends on mode)
regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
regw(DP_SRC, 0x00000300); // mono:always_'1', frgd:blit
regw(CLR_CMP_CNTL, 0x01000005); // source compare, equality
regw(CLR_CMP_MASK, 0xFFFFFFFF); // enable all planes for
                      // comparison
regw(CLR_CMP_CLR, transparentColor);  // color to be transparent

// Set up the source trajectory (remember to set SRC_WIDTH1 to a
// non-zero value).
WaitForFifo(7);
regw(SRC_OFF_PITCH, 0x20000000 | offset); //pitch:1024(depends
```

```
                                    // on mode)
regw(SRC_CNTL, 0x00000004);         // linear

// Set up the destination trajectory and initiate the blit.
regw(DST_OFF_PITCH, 0x20000000); // pitch:1024(depends on
mode),
                        // offset:0
regw(DST_CNTL, 0x00000003); // left-to-right, top-to-bottom
regw(DST_Y_X, ((ULONG)dstX << 16) | dstY);
regw(DST_HEIGHT_WIDTH, ((ULONG)dstWidth << 16) | dstHeight);
```

6.3.4 Pattern Source

Pattern sources derive their pixel data from the contents of the pattern registers PAT_REG0 and PAT_REG1.

6.3.4.1 Fixed Patterns

Three types of fixed pattern are available:

- 4x2 color pattern.
- 8x1 color pattern.
- 8x8 monochrome pattern.

The fixed color patterns are only supported in 8 bpp mode. Fixed patterns are always destination-aligned. See Section *6.2.2* for a depiction of pattern consumption. he destination draw trajectories in the following code can be replaced by the draw rectangle routine or line draw routine in Section *6.3.1*. The important registers to set are the PAT_CNTL (in union with GUI_TRAJ_CNTL) and PAT_REGs.

Example Code for Rectangle Fills Using Fixed Patterns

```
// 8x8 mono pattern

// Setup the draw engine context manually.
WaitForFifo(12);
regw(DP_FRGD_CLR, 0xFFFFFFFF); // white
regw(DP_BKGD_CLR, 0x00000000); // black
regw(DP_WRITE_MASK, 0xFFFFFFFF);// enable all planes
regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp(depends on mode)
regw(DP_MIX, 0x00070007); // frgd:paint, bkgd:paint
regw(DP_SRC, 0x00010100); // mono:pattern,
```

```
                        // frgd:DP_FRGD_CLR,
                        // bkgd:DP_BKGD_CLR
    regw(PAT_REG0, patternData0); // pattern data
    regw(PAT_REG1, patternData1); // pattern data
    regw(CLR_CMP_CNTL, 0x00000000); // disable
    regw(GUI_TRAJ_CNTL, 0x01000003); // enable 8x8 mono patterns
                        // left-to-right, top-to-bottom
    regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
    regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres

    // Setup the draw trajectory and initiate.
    WaitForFifo(3);
    regw(DST_OFF_PITCH, ((ULONG)pitch << 22) | offset);
    regw(DST_Y_X, ((ULONG)x << 16) | y);
    regw(DST_HEIGHT_WIDTH, ((ULONG)width << 16) | height);

    // 4x2 color pattern

    // Setup the draw engine context manually.
    WaitForFifo(13);
    regw(DP_WRITE_MASK, 0xFFFFFFFF);// enable all planes
    regw(DP_PIX_WIDTH, 0x00020202); // 8 bpp(depends on mode)
    regw(DP_MIX, 0x00070003); // frgd:paint, bkgd:leave_alone
    regw(DP_SRC, 0x00000400); // mono:always_'1', frgd:pattern
    regw(PAT_REG0, patternData0); // pattern data
    regw(PAT_REG1, patternData1); // pattern data
    regw(CLR_CMP_CNTL, 0x00000000);    // disable
    regw(GUI_TRAJ_CNTL, 0x02000003); // enable 4x2 color patterns
                        // left-to-right, top-to-bottom
    regw(SC_LEFT_RIGHT, 0x03FF0000); // 0 to >= xres
    regw(SC_TOP_BOTTOM, 0x03FF0000); // 0 to >= yres

    // Setup the draw trajectory and initiate.
    regw(DST_OFF_PITCH, ((ULONG)pitch << 22) | offset);
    regw(DST_Y_X, ((ULONG)x << 16) | y);
    regw(DST_HEIGHT_WIDTH, ((ULONG)width << 16) | height);
```

The 8x1 color pattern is left as an exercise for the programmer.

6.4 Miscellaneous Operations

6.4.1 Drawing In Packed 24 Bit Per Pixel Mode

There is no 24-bit packed draw engine mode, but there is a 24-bit packed display mode. Drawing in this mode is accomplished by setting the engine in 8 bit per pixel mode and manipulating the DST_24_ROT and DST_24_ROT_EN bits. The following rules must be followed for drawing in this mode:

- Source and destination pitches must be set to three times the display pitch.
- All X coordinates and widths must be specified at three times the normal value. Remember that left-to-right operations begin on an R value, and right-to-left operations begin on a B value. That means that for left-to-right operations, the initial DST_X is expressed as (X * 3) and for right-to-left DST_X is (X * 3 + 2).
- Before any draw operation is initiated, the DST_24_ROT_EN@DST_CNTL must be enabled, and DST_24_ROT@DST_CNTL must be set to ((DST_X / 4) mod 6), where DST_X is the starting DST_X value as described above.

DST_X (8 bpp)																										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	....
X (24 bpp)																												
0			1			2			3			4			5			6			7			8			....	
DWORD																												
0				1				2				3				4				5				6			....	
DST_24_ROT Value																												
0		0	0		1	1		2	2		2	3		3	3		4	4		5	5		5	0		0	....	
COLOR COMPONENTS																												
R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	....

In the above table:

- **X** is the desired X coordinate in packed 24 bpp mode.
- **DST_X** is the value that you actually write to the draw engine (remember to start on an R component on left-to-right operations, and on a B component for right-to-left operations).
- The **DWORD** and **color components** rows show how memory is actually laid out in relation to pixel data.
- The **DST_24_ROT** row shows the value to place in the DST_24_ROT@DST_CNTL field before initiating a draw operation. Use the leftmost DST_24_ROT number in the

column for left-to-right operations, and the rightmost number for right-to-left operations.

- The **DST_24_ROT** value is simply the (DWORD-value-of-the-starting-byte mod 6).

Notes:

- The **rotation enable bit** only affects DP_FRGD_CLR, DP_BKGD_CLR, DP_WRITE_MASK, and fixed 8x8 mono patterns. Colors and masks are rotated appropriately, keying on the DST_24_ROT value.
- The line draw engine does not function in 24 bpp packed mode.
- Any other monochrome source other than fixed 8x8 monochrome patterns are only supported if the application sets up that monochrome source such that each bit in the monochrome source is expanded to 3 bits (one for each of R, G, and B).
- Polygons are only supported if the host manually draws the polygon boundary lines, only drawing one *pixel component* (the leftmost one -- R) per scan line, as opposed to one pixel per scan line.

Example Code for Drawing a Solid Rectangle in Packed 24 Bit Mode

```
// This procedure will fill a packed 24 bpp rectangle with a
// 24 bit color.

VIOD FillRect24(x, y, width, height, color)
   short x,y;
   USHORT width, height;
   ULONG color;
{
   USHORT rotation;

   // Setup the draw engine context manually.
    WaitForFifo(12);
   regw(DP_FRGD_CLR, color); // set rectangle color
   regw(DP_WRITE_MASK, 0x00FFFFFF); // enable all planes
   regw(DP_PIX_WIDTH, 0x00020202); // must be set to 8 bpp
   regw(DP_MIX, 0x00070003); // frgd:paint,
                             // bkgd:leave_alone
   regw(DP_SRC, 0x00000100); // mono:always_'1',
                             // frgd:DP_FRGD_CLR
   regw(CLR_CMP_CNTL, 0x00000000); // disable
   regw(SC_LEFT_RIGHT, 0x0BFF0000); // 0 to (1024 * 3 - 1)
   regw(SC_TOP_BOTTOM, 0x0BFF0000); // 0 to (1024 * 3 - 1)
```

```
   // Calculate the initial rotation factor (this is for
left-to-right;
   // to do right-to-left, calculate ((x * 3 + 2) / 4) % 6)).
   rotation = ((x * 3) / 4) % 6;


   // Setup the draw trajectory and initiate.
   regw(DST_CNTL, 0x00000083 | (rotation << 8));  //
left-to-right
                              // top-to-bottom
                              // rotation enabled
   regw(DST_OFF_PITCH, (((ULONG)pitch * 3) << 22) | offset);
   regw(DST_Y_X, (((ULONG)x * 3) << 16) | y);
  regw(DST_HEIGHT_WIDTH, (((ULONG)width * 3) << 16) | height);
}
```

6.4.2 Scissoring and Masking

Drawing may be inhibited outside a rectangular region by setting the scissor registers — SC_LEFT, SC_RIGHT, SC_TOP, and SC_BOTTOM. Scissors are inclusive on all edges. Therefore, to include the whole screen, left and top scissors should be set to 0, and right and bottom scissors should be set to (xres - 1) and (yres - 1) respectively. Note that a scissored draw operation draws at the same speed as an unscissored one. Drawing behavior is undefined for any objects drawn outside the device coordinate space, whether they are scissored or not. The device coordinate space is –4096 to +4095 in the X direction, and –16384 to +16383 in the Y direction.

Bits within a particular pixel may be selectively inhibited by setting the DP_WRITE_MASK register. This function can be useful for manipulating (or leaving alone) a pixel alpha channel.

6.4.3 Hardware Cursor

The *mach64* hardware cursor is similar in function to the *mach32* hardware cursor. Each cursor pixel is defined by a 2-bit field with the definition below:

Pixel Value	Meaning
00	Cursor color 0
01	Cursor color 1
10	Transparent
11	Complement - Note that in pseudocolor modes, some *mach64* board implementations will complement the index and others will complement the LUT lookup value.

Note that if the DAC supports a hardware cursor, it is preferable to use the DAC's cursor. Consult the manufacturer's DAC specification for programming information.

Cursor pitch 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 Intel order. The first pixel is defined in the low-order 2 bits of the low-order byte in memory. Each cursor scan line definition resides back-to-back in memory.

Cursor colors are defined by CUR_CLR0 and CUR_CLR1. Note that for *pseudo color modes*, the colors are specified in color indices, and for *direct color modes*, the colors are specified in 24-bit true color. The meaning of other registers is illustrated below:

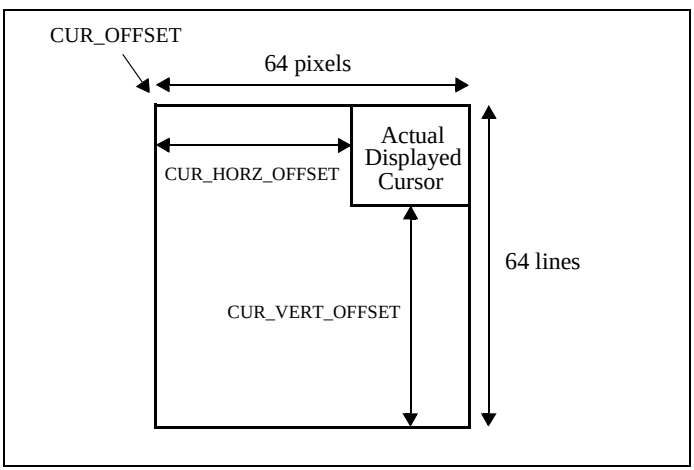


Figure 6-2. Hardware cursor position

The screen position of the top left corner of the displayed cursor is specified by CUR_HORZ_VERT_POSN. Care must be taken when the cursor hot spot is not the top left corner and the physical cursor position becomes negative. The *mach64* will not display the cursor at all if either the horizontal or vertical cursor position is negative.

- **If X becomes negative**, the cursor manager must adjust the CUR_HORZ_OFFSET to a larger number and saturate CUR_HORZ_POSN to zero.
- **If Y becomes negative**, CUR_VERT_OFFSET must be adjusted to a larger number, CUR_OFFSET must be adjusted to point to the appropriate line in the cursor definition, and CUR_VERT_POSN must be saturated to zero.

Example Code for Enabling, Disabling and Moving the Hardware Cursor

```
//
-------------------------------------------------------------
// EnableHWCursor - turn on the hardware cursor
VOID EnableHWCursor(VOID)
{
    iow16(GEN_TEST_CNTL, GEN_TEST_CNTL_0,
          ior16(GEN_TEST_CNTL, GEN_TEST_CNTL_0) | 0x80);
}
// -------------------------------------------------------------
// DisableHWCursor - turn off the hardware cursor
VOID DisableHWCursor(VOID)
{
    iow16(GEN_TEST_CNTL, GET_TEST_CNTL_0,
          ior16(GEN_TEST_CNTL, GEN_TEST_CNTL_0) & (~0x80L));
}
//
-------------------------------------------------------------
// SetHWCursorPos - set the hardware cursor position relative to
// hotspot
// It is assumed that the cursor has been previously defined
// linearly in off-screen memory with a pitch of 64 pixels (16
// bytes, or 2 QWORDs).
// CUR_OFFSET   = QWORD offset of cursor definition in graphics
// memory
// CUR_HORZ_OFF = 64 - cursorWidth
// CUR_VERT_OFF = 64 - cursorHeight
VOID SetHWCursorPos(short x, short y)
{
    USHORT curHorzOff, curVertOff;
```

```
    ULONG curOffset;
    static BOOL prevViolation=FALSE;
    BOOL violation = FALSE;

    curOffset = cur.offset;

    // Check for coordinate violations.
    if ((x - cur.hotSpot.x) < 0) {
        curHorzOff = 64 - cur.width - (x - cur.hotSpot.x);
        x = 0;
        violation = TRUE;
    } else curHorzOff = 64 - cur.width;
    if ((y - cur.hotSpot.y) < 0) {
        curVertOff = 64 - cur.height - (y - cur.hotSpot.y);
        curOffset = cur.offset + (cur.hotSpot.y - y) * 2;
        y = 0;
        violation = TRUE;
    } else curVertOff = 64 - cur.height;
    if (violation || prevViolation) {
        regw(CUR_OFFSET, curOffset);
        regw(CUR_HORZ_VERT_OFF, ((ULONG)curVertOff << 16) |
                                        curHorzOff);
    }
    prevViolation = violation;
    // Set the cursor position.
    regw(CUR_HORZ_VERT_POSN, ((ULONG)y << 16) | x);
}
```

Chapter 7
Advanced Topics

7.1 Introduction

This chapter contains several advanced topics on using the mach64.

7.2 Polygons

The *mach64* uses an alternate-fill algorithm for polygon filling. Polygon fills are simply rectangle fills with the DST_POLYGON_EN@DST_CNTL bit set. At the beginning of each destination scan line, an internal polygon fill flag is reset. Whenever this flag is in a reset state, drawing is inhibited. The polygon boundary source (this source is implicit and is established by using the blit source registers) is consumed, providing polygon boundary data. Whenever a polygon edge is detected, the internal polygon fill flag is toggled. Only rectangular destinations proceeding in a left-to-right and top-to-bottom direction are supported for polygon filling.

Polygon edges are inclusive on both left and right sides when filling. On the *mach64*CT, the right edge may be optionally inclusive or exclusive.

Note that any monochrome or color sources may be selected in the pixel data path except for blit sources (because the blit source registers are used to configure the polygon source trajectory) when polygon filling. Polygon boundary source is only meaningful when configured to 1 bpp pixel depth (set this with DP_SRC_PIX_WIDTH@DP_PIX_WIDTH).

Polygon boundaries are created by drawing lines in 1 bpp mode with the DST_POLYGON_EN@DST_CNTL bit set. This bit causes a maximum of one pixel per scan line to be drawn (horizontal lines are not drawn at all), and lines exceeding the left scissor boundary are saturated to the left scissor. Note that the pitch for the 1 bpp polygon outlines must be aligned along 64-pixel boundary.

To draw a polygon:

1. Clear the off-screen area where the 1 bpp polygon outlines are to be drawn.

2. Set the mix to XOR (this takes care of the degenerate case where two polygon boundary lines culminate in a vertical peak), enable the DST_POLYGON_EN bit, and draw all the polygon outline lines in 1 bpp from top to bottom with LAST_PEL_OFF.

3. Set up the blit source registers to point to the polygon outline area.

4. Fill the polygon bounding rectangle.

Example Code for Drawing a General Polygon

```
// Procedure to draw a polygon from a set of vertices. It is
//  assumed that the vertices form an open-ended polygon (which
//   will be closed by the procedure).

typedef struct tagPOINT {
    short x,y;
} POINT;

typedef struct tagBOX {
    short x, y;
    USHORT width, height;
} BOX;

// This routine will fill a polygon with a solid color. The host
//  application may in fact use any mono/color source combination
//   except for blit sources.
VOID DrawPolygon(lpPoints, nPoints, color)
   POINT lpPoints[];        // list of vertices
   USHORT nPoints;          // number of vertices
   ULONG color;             // color to fill the polygon with
{
   BOX bound;
   USHORT pitch, i, nextPoint;

   // First get the bounding box of the polygon vertices.
   GetBoundingBox(lpPoints, nPoints, &bound);

   // Calculate 1 bpp pitch.
   pitch = bound.width / 8;
   if ((bound.width % 8)!=0) pitch++;   // round up nearest
                                        //   multiple of 8
   while ((pitch % 8)!=0) pitch++;      // in 1 bpp mode, pitch
                                        //  must be a multiple
                                        //   of 64 pixels
```

```
// Clear a 1 bpp area of off-screen memory.
WaitForFifo(11);
regw(GUI_TRAJ_CNTL, 0x00000003);     // left-to-right,
                                     //   top-to-bottom
regw(DP_WRITE_MASK, 0xFFFFFFFF);     // enable all planes
regw(DP_PIX_WIDTH, 0x00000000);      // 1 bpp
regw(DP_MIX, 0x00010001);            // frgd:zero
regw(DP_SRC, 0x00000100);            // mono:always_'1',
                                     //   frgd:DP_FRGD_CLR
regw(CLR_CMP_CNTL, 0x00000000);      // disable
regw(SC_LEFT_RIGHT, ((ULONG)(bound.width-1) << 16));
regw(SC_TOP_BOTTOM, ((ULONG)(bound.height-1) << 16));
regw(DST_OFF_PITCH, ((ULONG)pitch << 22) | offScreenOffset);
regw(DST_Y_X, 0x00000000);
regw(DST_HEIGHT_WIDTH, ((ULONG)bound.width << 16) |
                                          bound.height);

// Set the context for polygon line drawing.
WaitForFifo(3);
regw(DST_CNTL, 0x00000040);          // DST_POLYGON_EN,
                                     //   DST_LAST_PEL_OFF
regw(DP_MIX, 0x00050005);            // D xor S
regw(DP_FRGD_CLR, 0xFFFFFFFF);       // white

// Draw the polygon outlines.
for (i=0; i<(nPoints-1); i++) {
   nextPoint = (i+1) % nPoints;

   // Draw only top to bottom lines.
   if (lpPoints[i].y > lpPoints[nextPoint].y) {
      DrawLine(lpPoints[nextPoint].x - bound.x,
               lpPoints[nextPoint].y - bound.y,
               lpPoints[i].x - bound.x,
               lpPoints[i].y - bound.y);
   } else {
      DrawLine(lpPoints[i].x - bound.x,
               lpPoints[i].y - bound.y,
               lpPoints[nextPoint].x - bound.x,
               lpPoints[nextPoint].y - bound.y);
   }
```

```
    }

    // Set the context for the polygon blit.
    WaitForFifo(14);
    regw(DST_CNTL, 0x00000043);    // DST_POLYGON_EN,
                                   //   DST_LAST_PEL_OFF
    regw(DP_MIX, 0x00070007);      // frgd:paint, bkgd:paint
    regw(DP_SRC, 0x00000100);      // mono:always_'1',
                                   //   frgd:DP_FRDG_CLR
    regw(DP_PIX_WIDTH, 0x00020002);// src:1 bpp, dst:8 bpp
    regw(DP_FRGD_CLR, color);      // set polygon color
    regw(SC_LEFT_RIGHT,((ULONG)(bound.x+bound.width-1) << 16)
                                        | bound.x);
    regw(SC_TOP_BOTTOM,((ULONG)(bound.y+bound.height-1) << 16)
                                        | bound.y);

    // Set the source trajectory to point to outline area
    //   (unbounded Y).
    regw(SRC_CNTL, 0x00000000);
    regw(SRC_OFF_PITCH, ((ULONG)pitch << 22) | offScreenOffset);
    regw(SRC_Y_X, 0x00000000);
    regw(SRC_WIDTH1, bound.width);

    // Blit it.
    regw(DST_OFF_PITCH, ((ULONG)dstPitch << 22) | screenOffset);
    regw(DST_Y_X, ((ULONG)bound.x << 16) | bound.y);
    regw(DST_HEIGHT_WIDTH, ((ULONG)bound.width << 16)
                                        | bound.height);
}
```

7.3 Scrolling and Panning

Scrolling and panning of the display area to the limits of the draw area can be simply done by changing the value of CRTC_OFFSET@CRTC_OFF_PITCH.

Note that offset has a granularity of 64 bits, which means that horizontal panning will be more “jerky” at lower pixel depths than at higher pixel depths.

Example Code for Calculating CRTC_OFFSET from X and Y Coordinates

```
// A display area shows a window to a larger desktop.
// dispOffset is the QWORD offset to the top left corner of the
//   desktop pitch*8 is the width of the desktop in pixels
//  x,y is the coordinate pair which offsets into the desktop.
// This desktop coordinate pair will be the top left corner of the
//   display region.

// Calculate the new CRTC offset from x,y. X must fall on a QWORD
//   boundary.
crtcOffset = dispOffset + (y * pitch*8 + x) / pixelsPerQword;
regw(CRTC_OFF_PITCH, ((ULONG)pitch << 22) | crtcOffset);
```

7.4 CRT Synchronization and Animation

For smooth animation, it is necessary to inhibit drawing to areas of the screen that are currently being scanned by the CRT controller. Failure to take necessary precautions will cause flickering or tearing effects on the animated object. Outlined below are several possible strategies that can be used for smooth animation.

7.4.1 Double Buffering (Memory)

Two areas of screen memory are allocated, each big enough for an entire display screen. While one memory area is being displayed, the other is updated, thus avoiding any collision between the CRTC and the draw engine. The system timer or the CRTC vertical line counter can be used to generate interrupts at constant time intervals.

In the interrupt service routine:

1. Wait-for-idle to ensure that the draw engine is not in the middle of drawing.

2. Set CRT_OFFSET to toggle to the memory area to display. The display will not change until the CRTC vertical counter resets to the top of the display area.

3. Wait for the display to change, i.e. wait for the CRTC vertical counter to reset to zero. If a CRTC vertical line count interrupt is used, then this step may be omitted.

4. Signal the mainline application that buffers have toggled.

In the mainline application:

1. Disable interrupts.

2. Draw the new frame into the draw buffer area. The application may use its own strategy to do this, either clearing the draw buffer and drawing from scratch, or updating the frame deltas.

3. Enable interrupts.

4. Wait for a signal from the interrupt service routine that buffers have toggled.

The buffer switching may also be done in the main line application, and using the system timer to switch buffers is optional. The advantage to using the system timer is a constant frame rate.

During application development, the programmer can omit steps 1 and 3 in the mainline to determine whether or not the desired frame rate can be accomplished. If not, flickering will occur.

7.4.2 Double Buffering (Palette)

The palette-driven double buffer is just a specialized case of the double buffer scheme described above. In 8 bpp mode, two memory areas can be allocated and overlaid on top of each other, each 4 bpp deep. The palette must be defined such that the lower four bits and the upper four bits specify the same 16 colors. The same algorithm is used as above, except that DAC_MASK is used to switch the displayed area (instead of CRTC_OFFSET), and DP_WRITE_MASK is used to write to the non-displayed area (instead of DST_OFFSET).

7.4.3 Single Buffering (Synchronized)

Simple animations (small update areas) may be accomplished with a single buffer with no flickering or tearing by refraining from drawing until the CRTC vertical line count is within a certain range. The vertical line count can be polled by reading CRTC_CRNT_VLINE@CRTC_VLINE_CRNT_VLINE or it can be interrupt-driven by setting CRTC_VLINE_INT_EN@CRTC_INT_CNTL and

CRTC_VLINE@CRTC_VLINE_CRNT_VLINE. Once the CRTC is scanning the desired range, the application must attempt to draw all that it must draw before the CRTC scan encroaches upon the draw area.

This method does not use up that much memory, but cannot update large areas of the screen without flicker.

Interrupts from the *mach64* chip are not recommended because ISA systems cannot share interrupts, and commonly run out of IRQ levels. Any program that uses interrupts must have a fall back mechanism for interrupt disabled configurations.

7.4.4 Single Buffering (Delta Framing)

Delta framing is a method of achieving flicker-free animation without CRT synchronization. Only the changes from one frame to the next are drawn on the screen. The animation will be flicker-free because no undrawing is ever done. Tearing will occur, but the effects will be minimal given the draw rate.

1. Calculate the bounding box of a region on the screen that is changing.

2. Construct this region for the next frame in off-screen memory.

3. Blit the region to on-screen.

This method becomes very complex if many (overlapping) regions are changing from one frame to the next.

7.5 Manual Mode Switching And Custom CRT Modes

7.5.1 Manual Mode Switching

Mode switching by manual means is not recommended. If for some reason this cannot be avoided, perform the following:

1. *mach64* subsystems must always be configured with a non-volatile storage system for storing mode and monitor information. The application programmer must detect what kind of non-volatile storage is on board and access it appropriately to retrieve mode information. The most common configuration uses an EEPROM. Consult the manufacturer's EEPROM data sheet. See *Appendix B, EEPROM Map*, for a possible storage mapping of the EEPROM and section 7.8 for information on how to access the EEPROM.

2. Set the accelerator CRTC using the information retrieved in step 1.

3. Detect the type of DAC used by reading the CONFIG_STAT0 register. Additional detection may be required for DACs that are upward compatible with the supported DAC types. Consult the manufacturer's DAC data sheet.

4. Initialize the DAC to the appropriate pixel depth and mode using DAC_CNTL and DAC_REGS. Consult the appropriate manufacturer's DAC data sheet.

5. Switch from VGA mode to accelerator mode by setting the CRTC_EXT_DISP_EN bit in the CRTC_GEN_CNTL register.

Additional material suitable for developers of non-DOS operating system drivers is available from ATI's Developer Relations group. Please call the number on the front of this manual.

7.5.2 Designing A Custom CRT Mode

The following illustration shows how the CRTC and overscan registers correspond to an actual video mode. The actual addressable display area is bounded by H_DISP and V_DISP. All registers are referenced to the upper left corner of the display area.

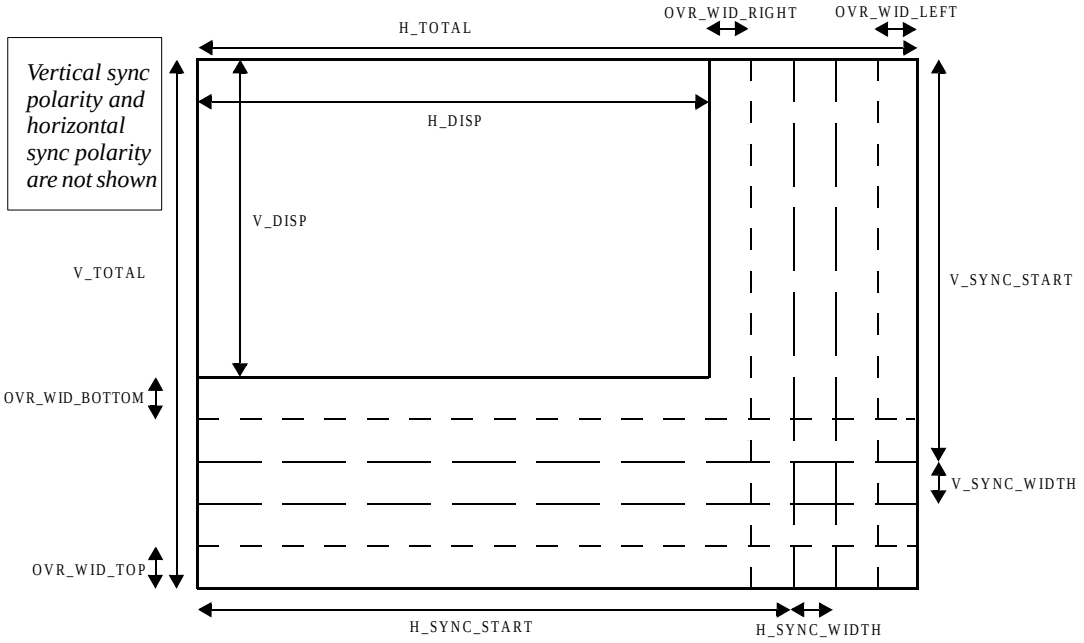


Figure 7-1. Actual video mode

The relationships between CRTC and monitor parameters are listed in the following tables:

Symbol Definitions	
PCLK	pixel clock rate (Hz)
T_{PCLK}	pixel clock period (sec)
H_{RES}	horizontal displayed resolution (pixels)
H_{SYNC}	horizontal sync rate (Hz)
H_{FP}	horizontal front porch (sec)
H_{BP}	horizontal back porch (sec)
H_{SWID}	horizontal sync width (sec)
H_{ACTIVE}	horizontal active time (sec)
H_{BLANK}	horizontal blank time (sec)

Symbol Definitions (Continued)	
V_{RES}	vertical displayed resolution (pixels)
V_{SYNC}	vertical sync rate (Hz)
V_{FP}	vertical front porch (sec)
V_{BP}	vertical back porch (sec)
V_{SWID}	vertical sync width (sec)
V_{ACTIVE}	vertical active time (sec)
V_{BLANK}	vertical blank time (sec)

Monitor Parameter to CRTC Parameter Conversions	
H_DISP	H_{RES} / 8 - 1
H_TOTAL	PCLK / H_{SYNC} / 8 – 0.5
H_SYNC_WID	H_{SWID} * PCLK / 8 + 0.5
H_SYNC_STRT	(H_{RES} + H_{FP} * PCLK + 0.5) / 8 – 1
V_DISP	V_{RES} – 1
V_TOTAL	H_{SYNC} / V_{SYNC} – 0.5
V_SYNC_WID	V_{SWID} * H_{SYNC} + 0.5
V_SYNC_STRT	V_{RES} + V_{FP} * H_{SYNC} – 0.5

CRTC Parameter to Monitor Parameter Conversions	
H_{RES}	(H_DISP + 1) * 8
H_{SYNC}	PCLK / (H_TOTAL + 1) / 8
H_{SWID}	H_SYNC_WID * 8 / PCLK
H_{FP}	(H_SYNC_STRT – H_DISP) * 8 / PCLK
H_{BP}	(H_TOTAL – H_SYNC_STRT – H_SYNC_WID) * 8 / PCLK
H_{BLANK}	(H_TOTAL – H_DISP) * 8 / PCLK
H_{ACTIVE}	(H_DISP + 1) * 8 / PCLK
V_{RES}	V_DISP + 1
V_{SYNC}	H_{SYNC} / (V_TOTAL + 1)
V_{SWID}	V_SYNC_WID / H_{SYNC}
V_{FP}	(V_SYNC_STRT – V_DISP) / H_{SYNC}
V_{BP}	(V_TOTAL – V_SYNC_STRT – V_SYNC_WID) / H_{SYNC}
V_{BLANK}	(V_TOTAL – V_DISP / H_{SYNC}
V_{ACTIVE}	(V_DISP + 1) / H_{SYNC}

Note that PCLK, H_DISP, H_TOTAL, H_SYNC_WID, H_SYNC_STRT, V_DISP, V_TOTAL, V_SYNC_WID, V_SYNC_STRT, HRES, and VRES are **integer** values. All the other parameters are **real**.

Refer to *Appendix C, CRTC Parameters* for listings of parameters for standard display modes.

Pixel clocks may be chosen from the ATI18818 clock chip. Refer to *Appendix D, Clock Chip Reference* for more details.

Example CRTC Calculation for 640x480 60 Hz Non-interlaced

```
Given parameters:
---------------
Hres  = 640
Hsync = 31.469 KHz
Hswid = 3.813 usec
Hfp   = 0.953 usec
Vres  = 480
Vsync = 59.94 Hz
Vswid = 0.064 msec
Vfp   = 0.350 msec
Pclk  = 50.35/2 = 25.18MHz(ATI1881X clock chip selection 4)
```

```
Hpol  = negative polarity
Vpol  = negative polarity

CRTC calculations:
-----------------
H_TOTAL     = (Pclk / Hsync / 8) - 0.5
            = (25.18 MHz / 31.469 KHz / 8) - 0.5
            = 99.52 = 63h

H_DISP      = Hres / 8 - 1 = 640 / 8 - 1
            = 79 = 4fh

H_SYNC_STRT = (Hres + Hfp * Pclk + 0.5) / 8 - 1
            = (640 + 0.953 usec * 25.18 MHz + 0.5) / 8 - 1
            = 82.06 = 52h

H_SYNC_WID  = (Hswid * Pclk) / 8 + 0.5
            = (3.813 usec * 25.18 MHz) / 8 + 0.5
            = 12.50 = 0ch -> 0ch + 20h (- polarity) = 2ch

V_TOTAL     = (Hsync / Vsync) - 0.5
            = (31.469 KHz / 59.94 Hz) - 0.5
            = 524.51 = 20ch

V_DISP      = Vres - 1 = 479 = 1dfh

V_SYNC_STRT = Vres + Vfp * Hsync - 0.5
            = 480 + 0.350 msec * 31.469 KHz - 0.5
            = 490.51 = 1eah

V_SYNC_WID  = (Vswid * Hsync) + 0.5
            = (0.064 msec * 31.469 KHz) + 0.5
            = 2.51 = 02h -> 02h + 20h (- polarity) = 22h

CLOCK_CNTL  = 14h (clock chip selection 4, divide by 2)
```

Note that the clock chip selection value depends on the type of clock chip used on the *mach64* card.

7.6 Interrupts

The *mach64* is able to generate hardware interrupts under a variety of conditions:

- Interrupt on command FIFO overflow (BUS_CNTL)
- Interrupt on host data error (BUS_CNTL)
- Interrupt on CRTC vertical blank (CRTC_INT_CNTL)
- Interrupt on CRTC vertical line count == CRTC_VLINE (CRTC_INT_CNTL)

To enable interrupts, the application must follow the steps below:

1. Disable interrupt generation with a CLI instruction.

2. Re-vector the interrupt vector to the interrupt service routine, remembering to save the old interrupt vector. Prior knowledge of which IRQ line the *mach64* board is wired to is required. Typically, the cascaded IRQ 2 is used (which is actually IRQ 9), so interrupt 0x71 must be re-vectored in the vector table. This particular IRQ level is not guaranteed and may in fact be another IRQ or disabled altogether.

3. Read the interrupt mask from the 8259 interrupt controller and save it. This value must be restored on program termination. Enable the appropriate IRQ in the mask (by zeroing the corresponding bit) and write this value back to the 8259. Remember that if IRQ 2-cascade is used, both the primary and secondary 8259 interrupt masks must be programmed (bit 2 of the primary, and bit 1 of the secondary).

4. Enable interrupts with an STI instruction.

5. Clear the appropriate acknowledge bit of the desired interrupt source and enable the interrupt (in BUS_CNTL or CRTC_INT_CNTL).

In the interrupt service routine:

1. Read the appropriate interrupt status bit to determine what caused the interrupt. If a cause cannot be found, then chain the interrupt to the old interrupt vector, otherwise proceed with the appropriate action.

2. Acknowledge the *mach64* (BUS_CNTL or CRTC_INT_CNTL).

3. Acknowledge the 8259 interrupt controller. Remember that if IRQ 2-cascade is used, both primary and secondary controllers must be acknowledged.

To disable interrupts:

1. Disable the *mach64* interrupt.
2. Disable interrupts with CLI.
3. Restore the 8259 interrupt masks.
4. Restore the interrupt vector table.
5. Enable interrupts with STI.

It is not recommended that interrupts be used in retail software applications because ISA-based systems tend to be fully loaded with hardware-interruptible devices, and ISA interrupts are not shareable. Also, some *mach64* boards may not be interrupt configurable. Any application that uses interrupts must have a fall back mechanism that does not use interrupts (i.e. polling).

7.7 Off-Screen Memory Management

Off-screen memory management is a requirement for any real application that directly uses the accelerator. Hardware cursor definitions, context save areas, font caches, and bitmap caches are all kept in off-screen memory. Independent source and destination pitches and offsets, and a linear source trajectory facilitate implementation of an off-screen memory manager.

Memory can be allocated in linear chunks, aligned to 64-bit boundaries.

A simple cache manager is shown below:

Example Code for an Off-screen Memory Manager

```
#define CACHE_INITIALIZE 0x0001
#define CACHE_ZERO       0x0002

typedef struct tagCacheInfo {
   ULONG qOffset;     // QWORD offset
   ULONG qSize;       // size in QWORDS
   ULONG nPixels;     // size in pixels (may be less than Qsize)
   BOOL empty;        // is item empty or full?
   struct tagCacheInfo FAR *nextCache; // next cache item
} CacheInfo;
```

```
// Host application must initialize pixPerQword, pitch,
// cacheQOffset, cacheQSize, cacheQRemain.
USHORT pixPerQword;         // pixels per QWORD for the current
                            //   mode
USHORT pitch;               // graphics mode pitch
ULONG cacheQOffset;         // offset to beginning of cache area
ULONG cacheQSize;           // total cache size in QWORDS
ULONG cacheQRemain;         // remaining unused cache in QWORDS
CacheInfo *cacheHead=NULL;  // pointer to first cache element
CacheInfo *cacheTail=NULL;  // pointer to last cache element
char FAR errorString[256]="";// put error messages here

//
-------------------------------------------------------------
// AllocCache
//
// Description:
//  Allocate an area in off-screen memory for application usage.
//
// Parameters:
//    nPixels         Number of pixels requested.
//    lpPixels        Pointer to pixel data (must be compatible
//                      with device).
//    cFlags    Cache flags.
//                    CACHE_INITIALIZE   Load cache just
//                                         allocated with pixel
//                                         data.
//                    CACHE_ZERO         Zero the cache area
//                                         just allocated.
//
// Return value:
//    On success, returns the QWORD offset of the cache area
//    relative to the base of graphics memory. Returns
//    0xFFFFFFFF if the call failed.
//
// Comments:
//    This routine will allocate a cache area rounded up to the
//    nearest scan line.
```

```
ULONG AllocCache(nPixels, lpPixels, cFlags)
   ULONG nPixels;
   VOID HUGE *lpPixels;
   USHORT cFlags;
{
   CacheInfo *cachePtr;
   ULONG qSize;// size in QWORDS of cache area
   ULONG dSize;// size in DWORDS of cache area requested
   ULONG lSize;// size in scan lines of cache area
   ULONG qPerLine;// number of QWORDS per line

   // Calculate the number of QWORDS needed.
   qSize = nPixels / pixPerQword;
   if ((nPixels % pixPerQword)!=0) qSize++;
   dSize = qSize * 2;

   // Round up to the nearest number of lines (this is optional;
   // this needs to be done if the application is going to blit
   // from screen to off-screen cache, because it's easier to
   // manage, as there are no linear destination trajectories,
   // only rectangular ones).
   qPerLine = ((pitch * 8) / pixPerQword);
   lSize = qSize / qPerLine;
   if ((qSize % qPerLine)!=0) lSize++;
   // Calculate how many qwords that is.
   qSize = lSize * qPerLine;

   // First, check to see if there are any empty entries in
   // the chain.
   for (cachePtr=cacheHead; cachePtr!=NULL;
       cachePtr=cachePtr->nextCache) {
       if (cachePtr->empty && qSize<=cachePtr->qSize) {
           cachePtr->nPixels = nPixels;
           cachePtr->empty = FALSE;
          if ((cFlags & CACHE_INITIALIZE) && lpPixels!=NULL) {
              Host2Screen(lpPixels, cachePtr->qOffset, dSize);
          } else if (cFlags & CACHE_ZERO) {
              FillRect (pitch, cachePtr->qOffset, 0L, 0, 0,
                        pitch*8, (USHORT)lSize);
          }
```

```
                return cachePtr->qOffset;
            }
        }
        if (cacheQRemain < qSize) {
            sprintf(errorString,"AllocCache: Not enough off-screen
             memory\n");
            return 0xFFFFFFFF;
        }

        // Create a new cache entry.
        if (cacheHead==NULL) {
            cacheHead = malloc(sizeof(CacheInfo));
            if (cacheHead==NULL) {
            sprintf(errorString,"AllocCache: Out of heap space\n");
                return 0xFFFFFFFF;
            }
            cacheTail = cacheHead;
        } else {
            cacheTail->nextCache = malloc(sizeof(CacheInfo));
            if (cacheTail->nextCache==NULL) {
            sprintf(errorString,"AllocCache: Out of heap space\n");
                return 0xFFFFFFFF;
            }
            cacheTail = cacheTail->nextCache;
        }
        cacheTail->qOffset = cacheQSize - cacheQRemain +
                                            + cacheQOffset;
        cacheTail->qSize = qSize;
        cacheTail->nPixels = nPixels;
        cacheTail->empty = FALSE;
        cacheTail->nextCache = NULL;
        cacheQRemain -= qSize;
        if ((cFlags & CACHE_INITIALIZE) && lpPixels!=NULL) {
            Host2Screen(lpPixels, cacheTail->qOffset, dSize);
        } else if (cFlags & CACHE_ZERO) {
            FillRect(pitch, cacheTail->qOffset, 0L, 0, 0,
                     pitch*8, (USHORT)lSize);
        }
        return cacheTail->qOffset;
    }
```

```
//
-------------------------------------------------------------
// FreeCache
//
// Description:
//    Releases a cache area previously allocated with AllocCache
//
// Parameters:
//    qOffset   QWORD offset of cache area from base of graphics
//    memory.
//
// Comments:
//    This routine just tags the area as empty and available for
//    re-use.
//    Garbage collection is not done here.
VOID FreeCache(qOffset)
   ULONG qOffset;
{
   CacheInfo *cachePtr;

   for   (cachePtr=cacheHead; cachePtr!=NULL;
                           cachePtr=cachePtr->nextCache) {
         if (cachePtr->qOffset==qOffset) {
            cachePtr->empty = TRUE;
            return;
         }
   }
}

//
-------------------------------------------------------------
// LargestCacheBlock
//
// Description:
//    Returns the size in QWORDS of the largest empty cache
//    block.
ULONG LargestCacheBlock(VOID)
{
   CacheInfo *cachePtr;
   ULONG biggest=0;
```

```
    for   (cachePtr=cacheHead; cachePtr!=NULL;
                              cachePtr=cachePtr->nextCache) {
          if (cachePtr->empty && cachePtr->qSize > biggest) {
             biggest = cachePtr->qSize;
          }
    }
    if (cacheQRemain > biggest) {
       biggest = cacheQRemain;
    }
    return biggest;
}
```

It is left as an exercise for the programmer to devise a garbage collect, flush cache (free all), and modify cache data routine.

7.8 Boot -Time Initialization

This section describes the registers required to initialize a *mach64* after power-up. All boot-time initialization is performed in the adapter ROM.

- The scratch registers, **SCRATCH_REG0** and **SCRATCH_REG1**, may be used at the adapter ROM's discretion, with the exception of the lower 7 bits of SCRATCH_REG1. These bits are used to communicate ROM segment location to applications, and must be initialized at boot-time. Typically, installed mode information and other flags are stored in the other bits.
- **BUS_CNTL** is used to configure the *mach64* bus interface unit and to control FIFO error and host error interrupts. At boot time, all interrupts should be disabled and the bus interface unit must be programmed appropriately for the type of host expansion bus. In determining the appropriate initialization values, the safest values should be used first, and incrementally reduced until minimum safe values are discovered.
- **MEM_CNTL** is used to configure the memory interface unit. Memory size must be determined by the adapter ROM and written appropriately. Initial memory boundary information should be stored in the non-volatile storage area. All other configuration bits are first determined empirically using the methods described for BUS_CNTL, and later hard-coded for particular memory configurations.
- **GEN_TEST_CNTL** is used for accessing an external EEPROM, enabling overscan to external DACs, enabling the hardware cursor, resetting the draw engine, enabling VRAM block write memory cycles, and chip diagnostic functions. At boot time, overscan and block write must be initialized. The hardware cursor must be disabled and the draw engine must be reset and enabled.

- **CONFIG_CNTL** is used for initializing the linear aperture and small apertures, setting the card ID, and disabling the VGA. The apertures should be disabled on power-up, and should only be initialized when an application calls the ROM for aperture services. The card ID should be set to zero in single-card systems. The VGA disable bit should never be touched.
- **CONFIG_CHIP_ID**, **CONFIG_STAT0**, and **CONFIG_STAT1** are used to determine board configuration for initialization, for ROM query functions, or for hardware debugging.

Of all the registers listed above, only CONFIG_CHIP_ID, GEN_TEST_CNTL, BUS_CNTL, and SCRATCH_REG1 may be touched by applications. CONFIG_CHIP_ID is used to identify a specific class and revision of accelerator, and GEN_TEST_CNTL is used to enable the hardware cursor and reset the draw engine. No other bits should be touched in GEN_TEST_CNTL. BUS_CNTL is used to configure interrupts for application debugging. SCRATCH_REG1 is used to determine the ROM segment location for calling ROM service routines.

7.9 Performance Issues

Performance is a complex issue that requires a clear definition of the terminology and an explanation of the factors affecting graphics performance.

7.9.1 Redundancy

Redundancy is the duplication of information. Most draw operations are redundant in that the same pixel or pattern of pixels is repeatedly written into memory. Since host expansion buses (ISA, EISA, MCA, VLB, PCI) tend to be slow, draw operations performed by the host CPU tend to be slow as well. Graphics accelerators improve performance by reducing the amount of redundant information travelling across the host expansion bus by simply specifying the type of pixel information to be written and the draw trajectory.

Any operation whose draw information cannot easily be reduced (such as a host-to-screen bitmap transfer) should do direct memory writes into the linear frame buffer instead of being drawn by the draw engine because draw setup overhead will slow the operation.

7.9.2 Draw Speed

Draw speed is a raw measure of how fast the draw engine can put pixels to memory. This is measured in pixels per second. Many benchmark programs do not measure draw speed correctly because they do not factor in concurrency.

7.9.3 Concurrency

Concurrency is the inherent ability of graphics accelerators to perform a draw operation at the same time that the host CPU is doing something else. An accelerator is a fixed-function processor that performs dedicated tasks and relieves the CPU to do other tasks. Concurrency and reduction of redundancy are the primary reasons why graphics accelerators are faster than dumb frame buffer devices, such as the VGA.

7.9.4 Efficiency

Efficiency is a measure of concurrency. Maximum efficiency in a software process is achieved when the host is never idle and the draw engine is never idle (this never happens). Efficiency will be affected by draw speed, CPU speed, FIFO depth, and the order of draw operations (for example, a draw engine operation followed by a linear frame buffer access requires a wait-for-engine-idle in between, which causes the CPU to idle, thus decreasing efficiency).

Note that graphics benchmark programs are atypical because they have inherently low efficiency.

Performance should be measured on both slow and fast CPUs because efficiency differs radically from system to system.

7.9.5 Expansion Buses

There are currently four different expansion bus standards for x86 platforms:

- ISA
- EISA
- VLB
- PCI

Each differs in maximum and typical throughput. Bus type will only affect the performance of host-to-screen and screen-to-host transfers. Most other draw operations have very low redundancy and bus transfer times are very small.

7.9.6 Block Write

Block write is a high speed color fill feature of VRAMs and some specialized types of DRAMs. Four consecutive addresses can be filled with a solid color in the time it takes to do a single memory access.

The *mach64* uses block write if GEN_BLOCK_WR_EN@GEN_TEST_CNTL is enabled, the foreground mix is set to paint (function 7), the background mix is set to transparent (function 3), the color compare function is set to FALSE, WRT_MASK is set to all '1's, destination pixel size is 8, 15, 16, or 32 bpp, and DST_24_ROT_EN@DST_CNTL is disabled. Any monochrome source may be used. It is the adapter ROM's responsibility to enable GEN_BLOCK_WR_EN@GEN_TEST_CNTL at boot time if a compatible type of memory is detected.

7.9.7 Memory Bandwidth

Memory bandwidth is a measure of the number of memory accesses per second, which is easily quantifiable. On the *mach64*, a memory access to a current page costs two cycles and a page faulted memory access costs seven cycles.

A **page** is defined as 512 addresses, where the data width may be 32 bits or 64 bits wide depending on memory configuration. The frequency of page faulting depends on the burst rates of the various devices contending for the memory bus.

The table below contains information that is required to perform a memory bandwidth calculation on all *mach64* chips.

Table 7-1 Chip Characteristics Affecting Performance

Feature	VT	3D RAGE	3D RAGE II	3D RAGE II+	3D RAGE IIC	3D RAGE PRO
Memory data width < 2M, bits	32	32	32	32	32	n/a
Memory data width ≥ 2M, bits	64	64	64	64	64	64
Source FIFO size	8x32	8x32	16x32	32x32	32x32	32x32
Page hit (memory cycles)	2	2	2	1	1	1
Page miss (memory cycles)	7	7	7	7	7	7
Page size	512	512	512	512	512	512
Maximum memory speed, MHz	66	66	83	83	83	100
Blockwrite (SGRAM only)	no	no	no	yes	yes	yes

Notes on memory bandwidth analysis:

- Every memory access will either page-hit (the data at the requested memory address is on the same page as the previous memory access) or page-miss (the requested address is on a different page as the previous memory access).
- Write-only operations use 1 memory access per QWORD (or DWORD).
- Read-modify-write operations use 2 memory accesses per QWORD (or DWORD). A read-modify-write will occur if one of the following is true: (1) The WRITE_MASK is not all 1's, (2) A non-trivial ALU operation is selected, i.e. an ALU operation that requires a destination read, (3) The destination compare function is non-trivial, i.e. not TRUE nor FALSE, (4) The left or right edge boundaries are not exactly aligned to a QWORD (or DWORD); the edge words will be read-modify-write and all other words will conform to the criteria 1-3.
- Blit operations are either read-write (2 memory accesses) or read-modify-write (3 memory accesses) according to the destination read-modify-write conditions outlined above. A page miss will occur at a minimum rate of the size of the source FIFO.
- DRAM boards will be affected by the CRTC which is periodically fetching data to display. The bandwidth used up is roughly equal to the pixel clock rate of the video mode times the pixel width in bytes. Remember that extra page faulting will occur because of this fetching. The page faulting frequency can be approximated by computing the percent bandwidth the CRTC will use and using this ratio to approximate the page fault rate given the calculated draw speed.

Example of a memory bandwidth calculation: A screen-to-screen blit with dimensions 160x120, a destination mix of XOR at 30 frames per second in 8 bpp mode, and a pitch of 1024 pixels; assume a memory clock of 50MHz, and data width of 64 bits.
Number of QWORDs in 160x120 area: (160x120 pixels / (8 pixels / QWORD)) = 2400 QWORDs.
Number of memory accesses per QWORD: source-read + dest-read + dest-write = 3 accesses/QWORD
Note that the source read occurs because it is a screen-to screen-operation, and the destination read occurs because it is a read-modify-write destination mix.
Number of memory accesses: 2400 * 3 = 7200 accesses.
Memory page size: (512 x 64 bits) / (8 bits/pixel) / (1024 pixels/line) = 4 lines
Page faulting from operation size is so infrequent that we will ignore this factor. Page faults from muxing source-reads and destination-read-modify-writes should occur every four memory accesses. Therefore, average access time is: (3 * 2 + 1 * 7) / 4 = 3.25 cycles/access
Number of memory cycles needed for a single blit: 7200 * 3.25 = 23400 cycles.

Draw speed = (160x120 pixels) / (23400 cycles / 50000000 cycles/sec) = 41 Mpixels/second.
Percent memory bandwidth used: (23400 cycles/frame * 30 frames/sec) / 50000000 cycles/sec * 100% = 1.4%.
Notes:
The average access time calculation will vary depending on a number of factors. Not aligning a source or destination edge to a QWORD boundary will increase the average access time by a small amount. A write-only operation will page fault much less than the read-read-modify-write operation in the example above.
Also note that these calculations are only applicable to large draw operations. Draw engine setup overhead becomes much more significant for small operations.

7.9.8 Performance

Performance is a complex measure, and depends greatly upon host configuration, accelerator configuration, and application efficiency. Performance cannot be quantified in a single measure.

- System performance can be improved with faster host and accelerator configurations.
- Application performance on a fixed hardware configuration can be improved by reducing redundancy and improving efficiency on a particular target system.

Chapter 8
mach64VT/GT Specific Features

8.1 Introduction

This chapter will focus on the special features that are available on the *mach64*VT and *mach64*GT (3D RAGE) (ATI264VT/GT) variants. As the *mach64*VT/GT is fully upward compatible with the *mach64*CT, any driver that was written for the *mach64*CT should work on the *mach64*VT/GT without modification.

8.2 Summary of Additional Features

The *mach64*VT has several new hardware features that are useful when doing capture and playback of motion video data. The new major subsystem is the hardware overlay/scaler.

- Hardware Overlay
- Hardware Scaler
- Hardware Color Keyer
- Hardware Color Source Converter
- Hardware Color Interpolation
- New Register Block Access

The *mach64*GT (3D RAGE) has some unique features that are covered in this chapter:

- Front End Scaler/3D Pipeline
- Bus Mastering

There presented an example that explains the details of how to use front end scaler for color space conversion. Low level programming for 3D operations is not discussed. There are also two examples of bus mastering; one using the bus mastering capabilities of the 3D RAGE PRO graphics controller to transfer a bitmap from system memory to the frame buffer, and a second example that shows how to queue a series of engine register writes and bus master them to the GUI.

8.3 *mach64*VT/GT Register Access

The register mapping for the *mach64*VT follows the same convention as the *mach64*CT registers. All registers are mapped relatively to the top of the defined memory aperture. For the *mach64*CT, a 1KB block at the top of the aperture was defined for all registers. This upper 1KB block is known as block 0. The *mach64*VT/GT expands this register space by adding a new 1KB register block below the original block0. This new block is referred to as block 1, and it contains registers that are specific to the *mach64*VT/GT. Note that these new registers are memory mapped only.

8.3.1 Memory Map

The following diagram gives a pictorial representation of the register blocks' location relative to the graphics aperture and system memory.

Typical Organization Of *mach64* Aperture Within Host Address Space (PC-compatible)

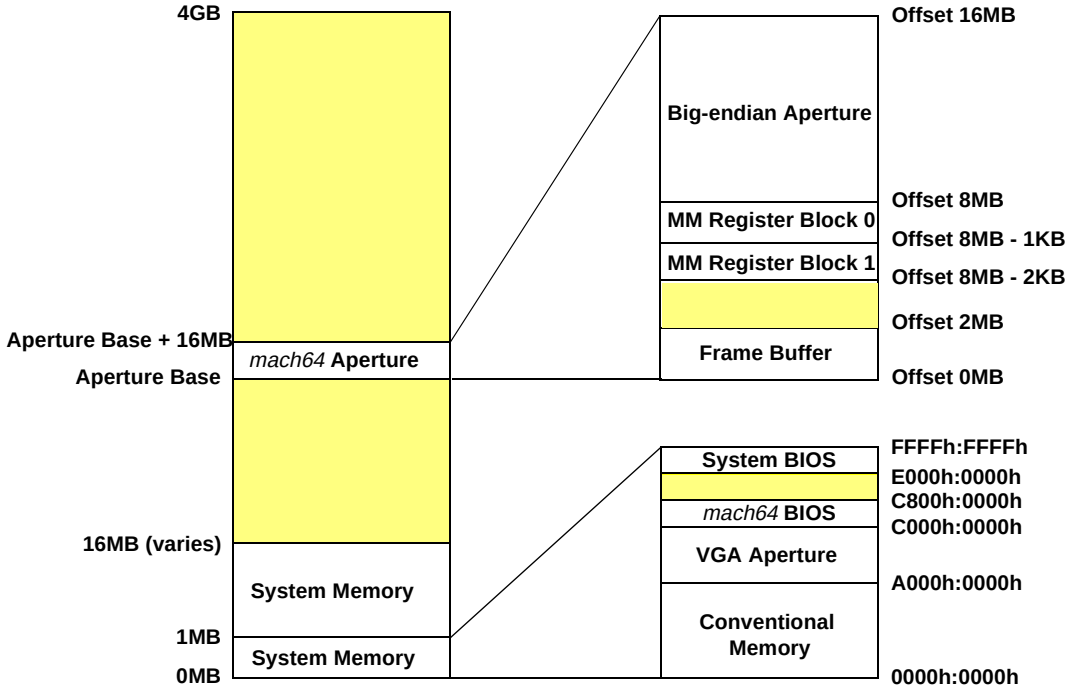


Figure 8-1. Aperture Within Host Address Space (PC-compatible)

8.3.2 Determining Register Address

All *mach64*VT/GT registers in register block 1 are memory mapped only. These registers are all 32 bits wide.

- If the small aperture are enabled, the memory mapped registers in block 1 may be accessed through a 1KB area at a segmenet:offset of B000h:F800h.
- If the big aperture is enabled, the memory mapped registers in block 1 occur the address space located at the base address of the aperture, plus an offset of 7FFC00h for an 8MB aperture configuration. A method of accessing extended memory is required to access the registers at this location. Note that the *mach64*VT/GT does not have a 4MB aperture configuration.

Referring to the *mach64 Register Reference Guide*, the **DWORD Offset** or **Memory Map (MM) select** is given to describe the register's address. The following notation is used:

```
MM:block#_offset
```

where **block#** identifies the register block and **offset** is the DWORD offset *within* the accosted block. For example, OVERLAY_SCALE_CNTL is located at MM:1_09h, i.e., DWORD offset 9 within register block 1. If the register block# is omitted then the register is assumed to be in block 0.

If access through the small apertures is desired, the physical address for register block 1 can be determined by the following equation:

```
physical memory address = (MM select << 2) + B000h:F800h
```

For example, if the **MM select** = 1_09 (OVERLAY_SCALE_CNTL), the physical address would be B000h:f824h.

If the big aperture is enabled, the equation for register block 1 becomes:

```
physical memory address = (MM select << 2) + aperture base +
                          memmap offset
```

where **memmap offset** is 7FF800h. Using the example above, if the aperture base address is A0000000h, the aperture size is 8MB (offset 7FF*00h) and the **MM select** = 1_09 (OVERLAY_SCALE_CNTL), the physical memory address would be A07FF824h.

8.3.3 Enabling Register Block 1

In order to access register block 1, it must first be enabled. This is accomplished by setting BUS_EXT_REG_EN@BUS_CNTL.

Example Code for Enabling Register Block 1

```
//
void enable_block_1 (void)
{
   WaitForIdle ();
   //Enable register block 1 by setting BUS_CNTL[27]
   regw (BUS_CNTL, regr (BUS_CNTL) | 0x08000000);
}
```

8.4 Hardware Overlay/Scaler

The *mach64*VT/GT adds a single pass back end vertical/horizontal scaling unit. Scaling is defined by a starting display coordinate that is relative to the start of the **active** display area. Scaling output is enabled based on the color key and color key function selected. An overlay area is redefined as the area to be scaled in the display output. the maximum input width of the input source to be scaled up or down is 384 pixels with revision A variants of the *mach64*VT/GT, and 720 pixels for revision B asics and beyond.

Scaling is orthogonal to the GUI engine operations. As such, engine operations can occur concurrently and independent of scaling operations.

Source scaling formats supported:

- RBG 555
- RGB 565
- RGB 8888
- YUV9 planar
- UYV12 planar
- YUVU 422 packed
- VYUY 422 packed

Destination format supported is RGB24bpp direct to the DAC.

- Edge effect for scaling are on the right side (end pixels) of the display
- The scaler can operate on packed data, YUV9/12 scaling or direct RGB scaling with

RGB/YUV 422 packed output

- YUV modes support pixel lending when scaling. For RGB input to the scaler, only pixel replication is allowed.
- Support for independent Video input
- Page flipping based on display/video_in trigger conditions

8.4.1 Overlay

The overlay starting and ending coordinates (OVERLAY_Y_X) **must** always be in the active region of video or else undetermined side effects will occur. In addition, the ending overlay coordinates must be greater than the starting coordinates. If the ending coordinates are outside the active region, clipping will not occur.

The overlay can operate at a different pixel depth than the graphics display, but the overlay is restricted to direct color modes.

When ECP_DEV@PLL_VCLK_CNTL = VCLK/2, the overlay has the following programming restrictions:

1. Overlays starting on an even pixel must end on an even pixel. Failure to do so will result in one less pixel being displayed.

2. Overlays starting on an odd pixel must end on an odd pixel. Failure to do so will result in one less pixel being displayed.

8.4.2 Scaler

The buffers used for scaler input are restricted to being quadword aligned. Both the initial offset of the buffers being used and the pitch must fall on quadword boundaries.

The scaler source is limited to lines no longer than 384 pixels in length with revision A variants of the mach64VT/GT, and 720 pixels for revision B asics and beyond.

The 3D RAGE PRO also introduces some new registers that affect the scaler/overlay operation. Five scaler co-efficient registers have been introduced that control the peaking of the horizontal scaler. A function has been introduced that programs these registers to acceptable default values.

8.4.3 Color Keyer

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

OVERLAY_KEY_CNTL, OVERLAY_VIDEO_CLR_KEY, OVERLAY_VIDEO_CLR_MSK, OVERLAY_GRAPHICS_CLR_KEY, OVERLAY_GRAPHICS_CLR_MSK.

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). Both key color registers are 24 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 - 23 should be masked out, as we will not be using those bits when comparing the source data against the destination.

VIDEO_KEY_FN @ OVERLAY_KEY_CNTL and GRAPHICS_KEY_FN @ OVERLAY_KEY_CNTL determine how the color keys are applied. It is also possible to compare the graphics and video outputs by using OVERLAY_CMP_MIX @ OVERLAY_KEY_CNTL. A programing example is provided with the source code that accompanies this document.

8.4.4 Color Interpolator/ Alpha Blender

The alpha blender is a 5 bit multiplier which is used in both vertical and horizontal scaling.

Vertical mode multiplies all components from the current line (1 - alpha), and adding that to the result of the alpha multiplied by the next line.

Vertical and horizontal modes apply an alpha derived from the high 5 fractional bits of the vertical and horizontal DDAs to a pixel and its vertically/horizontally adjacent pixel. The exact equation is shown below:

$$\text{blendedPixel} = (1 - \alpha) * \text{currentPixel} + \alpha * \text{nextPixel}$$

The display scaler can upscale or downscale in both the horizontal and vertical directions. When downscaling is involved, the following scaling algorithms are applied to the vertical and horizontal scaler based on what the next line or pixel to be fetched is. If the (next line/pixel) $\geq$ (current line/pixel + 2), the current value of "α" will be ignored in favor of either a 50-50 blend or "α" = 0 (fixed alpha). It should be noted that the next line or pixel to fetch is determined by the current integer portion of the accumulator. The limitations incurred by downscaling are a result of the architectural limitations of the *mach64*VT in the vertical blending stage. For YUV scaling, it is worth noting that there are cases when the Y component may be downscaling while the UV components may require upscaling.

This is due to the fact that in the YUV modes supported, the UV pixels are subsampled. For example, in YVU9, the pixel subsampling is 1:4. Thus, the scaling factor for UV is 1/4 that programmed for Y. A downscaling factor of Y ($1 \leq sf \leq 4$) will result in an upscaling factor of ($0.25 \leq sf \leq 1$) for the corresponding UV pixels in YVU9 mode.

The scaler is limited to source lines $\leq$ 384 pixels in length with revision A variants of the mach64VT/GT, and $\leq$ 720 pixels for revision B asics and beyond.

8.4.5 Color Space Converter

Color conversion equations for YUV (CCIR-601) to RGB with a color temperature of 9300K:

$R = 9Y/8 + 25V/16 - 218$

$G = 9Y/8 - 13V/16 - 25U/64 + 136$

$B = 9Y/8 + 2U - 274$

The red equation (CCIR-601) with color temperature of 6500K:

$R = 9Y/8 + 25V/8 - 418$

For optimal results, the incoming YUV is **pre-saturated** to $16 \leq Y \leq 235$, and $16 \leq$ (U and V) ≤ 240 prior to being converted to RGB space.

The Y2R conversions (R2Y/Y2R) can be manually overridden and disabled if desired.

8.5 Packed Pixel Modes

The following packed pixel formats are supported by the mach64VT/GT overlay/scaler hardware:

Packed Pixel Mode				
Mode	**B3 (31:42)**	**B2 (23:16)**	**B1 (15:8)**	**B0 (7:0)**
15bpp RGB 1555	$a_1R_1R_1R_1R_1R_1G_1G_1$	$G_1G_1G_1B_1B_1B_1B_1B_1$	$a_0R_0R_0R_0R_0R_0G_0G_0$	$G_0G_0G_0B_0B_0B_0B_0B_0$
16bpp RGB 565	$R_1R_1R_1R_1R_1G_1G_1G_1$	$G_1G_1G_1B_1B_1B_1B_1B_1$	$R_0R_0R_0R_0R_0G_0G_0G_0$	$G_0G_0G_0B_0B_0B_0B_0B_0$
32bpp RGB a888	a	R	G	B
YUV422: (11) VYUY	V	Y_1	U	Y_0
YUV422: (12) YVYU	Y_1	V	Y_0	U

YUYV mode is considered a 16bpp mode (since each "unit" is YUYV), thus the pitch/offset for YUYV should be set in terms of 16bpp pixel (even though each YUYV unit contains two pixels, with UV being shared).

8.6 Planar Pixel Modes

YVU9 is 4:1:1 subsampled for U and V in both horizontal and vertical directions. For every 4 Y pixels in the horizontal direction, there is a corresponding U/V pixel as a result of the horizontal subsampling. For every 4 Y lines in the vertical direction, there is a single U/V line due to the vertical subsampling. In essence, for each 4x4 block of Y pixels, there is a single U/V pixel associated with it. The Y/U/V pitches are specified in terms of pixels. The Y/U/V offsets are specified in terms of bytes. The U and V dimensions must be exactly 1/4 of the Y dimensions.

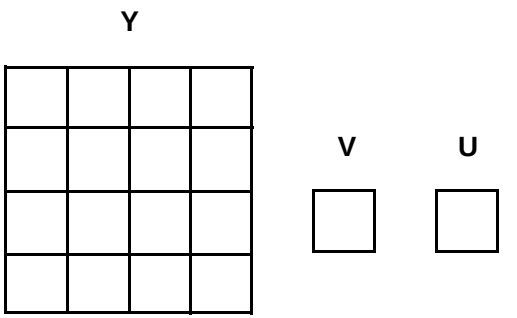


Figure 8-2. YVU9 4:1:1 Ratios

YVU12 is 4:2:2 subsampled for U and V in both horizontal and vertical directions. For every 2 Y pixels in the horizontal direction, there is a corresponding U/V pixel as a result of the horizontal subsampling. For every 2 Y lines in the vertical direction, there is a single U/V line due to the vertical subsampling. In essence, for every 2x2 block of Y pixels, there is a single U/V pixel. The Y/U/V pitches are specified in terms of pixels. The Y/U/V offsets are byte offsets. The U and V dimensions must be exactly 1/2 of the Y dimensions for YVU12.

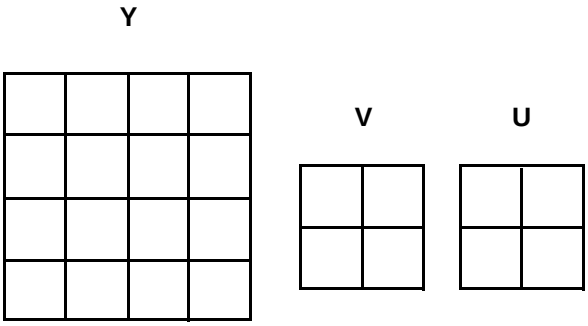


Figure 8-3. YVU12 4:2:2 Ratios

8.7 Unpacker / Dynamic Range Corrector

Source pixels of low pixel depth should be dynamically range corrected when expanded to 24 bit pixels. The unpacker also has a bypass mode for YUV sources, and a zero extend mode for when dynamic range correction is not desired. RGB modes are internally expanded to 24-bit modes prior to scaling. The scaled image is then dithered down to the destination output mode. The "a888" mode is internally processed as a "565" mode. Thus, only the most significant bits of each color component (RGB) are used when scaling 24-bit mode.

Table 8-1 Pixel Expansions

Zero Extended and Dynamic Range Corrected Pixel Expansions							
		aRGB 1555		RGB 565		aRGB a888	
		Zero	Dyn	Zero	Dyn	Zero	Dyn
Red	R(7)	D(14)	D(14)	D(15)	D(15)	D(23)	D(23)
	R(6)	D(13)	D(13)	D(14)	D(14)	D(22)	D(22)
	R(5)	D(21)	D(12)	D(13)	D(13)	D(21)	D(21)
	R(4)	D(11)	D(11)	D(12)	D(12)	D(20)	D20)
Red	R(3)	D(10)	D(10)	D(11)	D(11)	D(19)	D(19)
	R(2)	0	D(14)	0	D(15)	0	D(23)
	R(1)	0	D(13)	0	D(14)	0	D(22)
	R(0)	0	D(12)	0	D(13)	0	D(21)
Green	G(7)	D(9)	D(9)	D(10)	D(10)	D(15)	D(15)
	G(6)	D(8)	D(8)	D(9)	D(9)	D(14)	D(14)
	G(5)	D(7)	D(7)	D(8)	D(8)	D(13)	D(13)
	G(4)	D(6)	D(6)	D(7)	D(7)	D(12)	D(12)
	G(3)	D(5)	D(5)	D(6)	D(6)	D(11)	D(11)
	G(2)	0	D(9)	D(5)	D(5)	D(10)	D(10)
	G(1)	0	D(8)	0	D(10)	0	D(15)
	G(0)	0	D(7)	0	D(9)	0	D(14)
Blue	B(7)	D(4)	D(4)	D(4)	D(4)	D(7)	D(7)
	B(6)	D(3)	D(3)	D(3)	D(3)	D(6)	D(6)
	B(5)	D(2)	D(2)	D(2)	D(2)	D(5)	D(5)
	B(4)	D(1)	D(1)	D(1)	D(1)	D(4)	D(4)
	B(3)	D(0)	D(0)	D(0)	D(0)	D(3)	D(3)
	B(2)	0	D(4)	0	D(4)	0	D(7)
	B(1)	0	D(3)	0	D(3)	0	D(6)
	B(0)	0	D(2)	0	D(2)	0	D(5)

8.8 Overlay Programming

8.8.1 Overlay Scaling

The scaling operation allows vertical and horizontal single pass scaling for up/down to the DAC. The graphics and video streams are combined based on source/destination keying operation. In addition, scaling can take place from a single buffer or can be double buffered. The register programming outlined below assumes that the host uses a convention of 12 fractional bits and 4 integer bits (8 bits on a 3D RAGE PRO) in its vertical scale accumulator and increment variable.

1. Determine the horizontal/vertical scale increments

$$V\_INC = \frac{sourceHeight \ll 12}{destinationHeight}$$

$$H\_INC = \frac{sourceWidth \ll 12}{destinationWidth}$$

It is recommended that V_INC/H_INC be truncated after the 12th decimal place to avoid running out of source pixels (due to error) as opposed to rounding up to the nearest value.

2a. Initialize scalar vertical/horizontal registers.

xxxx@OVERLAY_SCALE_CNTL = Configure scaling options

VERT_INC@OVERLAY_SCALE_INC = V_INC

HORZ_INC@OVERLAY_SCALE_INC = H_INC

2b. (i) For Single Buffer Scaling from Buffer 0:

SCALER_IN@VIDEO_FORMAT = input source format for scaler

BUF0_OFFSET = byte address/offset of BUF0

BUF0_PITCH = pitch of BUF0

SCALER_HEIGHT_WIDTH = width and height of the buffer for scaling

SCALER_BUF@CAPTURE_CONFIG = Buffer 0 (set scaler to buffer 0)

(ii) For Double Buffering add:

BUF1_OFFSET/PITCH = address offset/pitch of BUF1

xxxx@CAPTURE_CONFIG = set trigger conditions

3. Scale to DAC: (24bpp fixed)

OVERLAY_Y_X = (x, y) coordinates of overlay relative to (0, 0) top left corner of **active** display

OVERLAY_Y_X_END = ending coordinates of overlay window (bottom, right corner)

OVERLAY_KEY_CLR/MSK = Overlay key color and mask settings

OVERLAY_KEY_CNTL = Determines how overlay will use key color

4. Enable scaling/overlay window

SCALE_EN@OVERLAY_SCALE_CNTL = enable

OVERLAY_EN@OVERLAY_SCALE_CNTL = enable

This will enable continuous scaling of the "video" data to the overlay or video port.

8.8.2 UV Interpolation

In YUV modes YUV422, YVU12, and YUV9, the UV data is subsampled. In YUV422, UV is subsampled in the horizontal direction only (1:2 subsampling), while YVU12 and YVU9 subsample UV both in the vertical and horizontal directions (1:2 and 1:4, respectively). For background purposes, the following possible subsampling types can exist:

Table 8-2 Possible Subsampling Types

Subsample Algorithm	Y Pixels	UV Subsampling (1:2)	UV Subsampling (1:4)
UV Even	y0, y1, y2, y3, y4...	1. u0, u2, u4, u6...	1. u0, u4, u8, u12... 2. u2, u6, u10, u14...
UV Even Blend + + 1/2	y0, y1, y2, y3, y4...	2. 1/2(u0+ u1), 1/2(u2+ u3), 1/2(u4+ u5)...	3. 1/2(u0+ u1), 1/2(u4+ u5), 1/2(u8+ u9)... 4. 1/2(u2+ u3), 1/2(u6+ u7), 1/2(u10+ u11)...
UV Odd	y0, y1, y2, y3, y4...	3. u1, u3, u5, u7...	5. u1, u5, u9, u13... 6. u3, u7, u11, u15...
UV Odd Blend + + 1/2	y0, y1, y2, y3, y4...	4. 1/2(u1+ u2), 1/2(u3+ u4), 1/2(u5+ u6)...	7. 1/2(u1+ u2), 1/2(u5+ u6), 1/2(u9+ u10)... 8. 1/2(u3+ u4), 1/2(u7+ u8), 1/2(u11+ u12)...

The mach64VT is equipped to best handle subsampled data for case (1) of UV (1:2) and case (1) for UV (1:4).The chroma component of all subsampled YUV sources is

interpolated. For other subsampling types, the UV data sampled for YUV422/YVU12/YVU9 cannot be centered according to input subsampling. The interpolation for the UV pixels is fixed as follows:

YUV422/YVU12: U0, (U0+U1)/2 and V0, (V0+V1)/2

YVU9: U0, (3U0 + U1)/4, (U0+U1)/2, (U0+3U1)/4
and V0, (3V0+V1)/4, (V0+V1)/2, (V0+3V1)/4

8.9 Front End Scaler Programming

8.9.1 Front End Scaler Operation

The 3D RAGE II/II+/IIC and the 3D RAGE PRO chips all have a front end scaler / 3D engine pipeline that provides support for horizontal and vertical scaling, interpolation and color space conversion of source images.

Scaled pixel data is processed through a 2 tap, 4 bit co-efficient fixed linear filter. Horizontal and vertical scaling are done in a single pass. For each destination line, two lines of source data are read, and then color expanded to 24 bpp for vertical blending. The resultant data is then blended horizontally, converted to RGB if necessary and then packed to the destination pixel type and dithered.

8.9.2 Performing a Blt Using the Front End Scaler

To configure the front end scaler for a bit block transfer (blt), follow these steps:

1. Activate the front end scaler

SCALE_3D_FNC@SCALE_3D_CNTL

2. Initialize appropriate registers *(for the 3D RAGE PRO only)*

ALPHA_TST_CNTL = 0 (turn off any default alpha blending states)

TEX_CNTL = 0 (turn off any default lighting states)

3. Configure the front end scaler registers for the blt

SCALE_OFF = offset in the frame buffer of the scaler source data

SCALE_PITCH = the appropriate pitch for the scaler source data

SCALE_WIDTH = the width of the scaler data

SCALE_HEIGHT = the height of the scaler data

4. Set the appropriate scaling factors

SCALE_X_INC = X scaling factor (this register follow a 12 bit fractional, 8 bits unsigned integer format)

SCALE_Y_INC = Y scaling factor (this register follow a 12 bit fractional, 8 bits unsigned integer format)

**** for the 3D RAGE II/II+ only:***

SCALE_UV_HACC = UV scaling factor, should you wish to scale U and V independent of Y.

5. Set the engine up to use the front end scaler for the blt

DP_FRGD_SRC@DP_SRC = 5, to use the front end scaler data

DP_SCALE_PIX_WIDTH@DP_PIX_WIDTH

DP_DST_PIX_WIDTH@DP_PIX_WIDTH

enable the appropriate bits @DP_WRITE_MSK

set the desired mix display mixing settings @DP_MIX

set the appropriate draw engine trajectory @GUI_TRAJ_CNTL

DST_X@DST_X

DST_Y@DST_Y

DST_HEIGHT@DST_HEIGHT

DST_WIDTH@DST_WIDTH - this initiates the blt

8.10 Bus Master Programming

8.10.1 Bus Master Operation

The 3D RAGE II/II+/C and 3D RAGE PRO chips all have the ability to act as a bus master. The bus mastering capabilities of these chips allow you to transfer data from system memory to the frame buffer and vice versa with minimal CPU usage. There are basically two types of transfers that the graphics chip will perform: system and GUI transfers. A system transfer involves moving memory between system memory and frame buffer memory (either way), while a GUI transfer involves moving data from system memory to the frame buffer through the GUI (or engine). A typical use of a system transfer would be moving a bitmap that is loaded into system memory into the frame buffer. You could also 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. A typical use of a GUI transfer (also known as a "virtual FIFO") would be 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 is constantly performing the same type of blt or screen setup, it may be beneficial to use the bus master in this case.

8.10.2 Creating a Descriptor Table

The bus master is instructed where to retrieve data through the use of descriptor tables. A descriptor entry consists of 4 DWORDs, with the following values:

Table 8-3 Descriptor Entry

	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
DWORD 2	BM_COMMAND	11:0 30 31	Count of bytes to transfer (4 kb maximum) Disable incrementing frame buffer offset End of descriptor list
DWORD 3	Reserved	31:0	

Transfers use the same byte offsets for both 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.

Note that a maximum of 4096 bytes of data can be transferred per descriptor. As a result,

if you are transferring an image that is larger than 4 kb, you must 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 as well as the physical memory address of the head of the table must be known.

When using the bus master hardware for GUI register writes (as a "virtual FIFO"), you must write the data to the BM_ADDR register. When using this method, the hardware will know that the first DWORD is the address of the register (in the MM offset format), and the following DWORD is the data for that register. In this case, because you wish to send all the data to the same register (BM_ADDR), you must inhibit incrementing the frame buffer offset. This is done by setting bit 30 of BM_COMMAND descriptor entry to a 1.

A programming example is provided for setting up a descriptor table, as well as performing a GUI bus master.

PSEUDO CODE TO SET UP A DESCRIPTOR:

loop:

- write the frame buffer destination offset address to BM_FRAME_BUFF_OFFSET
- write the physical address of the memory to be transferred to SYSTEM_MEM_ADDR
- write the amount of bytes to be transferred to BM_COMMAND (4096 bytes maximum)
    - if this is the last descriptor entry, set bit 31 to 1.
    - if you are writing to one memory address (e.g. for a GUI transfer), set bit 30 to 1.
- write a 0 for the reserved DWORD
- 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.

8.10.3 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 3D RAGE can be used to allow the CPU to perform other tasks while the 3D RAGE moves the data into the frame buffer.

The steps required to set up the 3D RAGE to perform a bus master operation from system memory to the frame buffer are outlined below. We are assuming that the descriptor table has already been set up, and the physical memory address of the descriptor table is paragraph aligned.

1. Set BUS_EXT_REG_EN@BUS_CNTL to enable the multimedia registers.

2. Set BUS_MASTER_DIS@BUS_CNTL to 0 to enable bus mastering

3. Set BUSMASTER_EOL_INT_AK@CRTC_INT_CNTL to 1 to clear the bus master end of transfer interrupt.

4. Set BUSMASTER_EOL_INT_EN@CRTC_INT_CNTL to enable the interrupt

5. Set SYSTEM_TRIGGER@BM_SYSTEM_TABLE to the desired transfer method (0 in this case), then OR this with SYSTEM_TABLE_ADDR@BM_SYSTEM_TABLE (which is the physical memory address of the head of the descriptor table - the first descriptor entry), and write this to BM_SYSTEM_TABLE. Writing to BM_SYSTEM_TABLE initiates the bus master operation.

At this point, you can allow the CPU to perform other tasks. To find out if the bus master transfer is complete, read BUSMASTER_EOL_INT@CRTC_INT_CNTL to see if it is set to 1. This indicates that the transfer is complete. Once BUSMASTER_EOL_INT has been acknowledged (set to 1), a 1 should be written to this bit to clear the interrupt.

8.10.4 Setting up a GUI Master Operation

As mentioned, the bus master hardware on the 3D RAGE can be configured to act as a virtual FIFO. You can queue up a number of register writes and use the bus master hardware to perform the writes in a single pass, thus freeing up the CPU to perform other tasks. The descriptor table is sent by the hardware to a circular buffer. The size of this buffer is determined by CIRCULAR_BUF_SIZE@BM_GUI_TABLE. Buffer sizes are 16, 32, 64 and 128 kb. For this reason, the physical memory address of the descriptor table must be aligned to the size of the circular buffer selected. That is, if you select a 16 kb circular buffer, the memory that you allocate for your descriptor table must start on a 16 kb boundary. If your queue of commands actually exceeds 16 kb, the buffer simply wraps around on the 16 kb address, thus making a "circular" buffer.

When using the bus master hardware as a virtual FIFO, the data that is to be transferred takes the following format:

DWORD (register address in MM offset format)

DWORD (data to be written to the register)

...

DWORD (register address in MM offset format)

DWORD (data to be written to the register)

The descriptor must be created so that the BM_SYSTEM_MEM_ADDR points to the beginning of this chain of register address/data alternating DWORDs.

Here are the steps to setup the bus master hardware to work as a virtual FIFO:

1. Set BUS_EXT_REG_EN@BUS_CNTL to enable the multimedia registers.

2. Set BUS_MASTER_DIS@BUS_CNTL to 0 to enable bus mastering

3. Set FRAME_BUF_OFFSET to BM_ADDR + the memory mapped register offset from the beginning of the aperture (0x7FFC00 in an 8 Mb aperture). BM_ADDR must be in the MM offset format, which is 0x92.

4. Set SYSTEM_MEM_ADDR to the physical memory address of the data to be transferred.

5. Set BM_COMMAND to the amount of bytes to be transferred. Also, set bit 30 to 1 to indicate that the frame buffer offset should NOT be incremented. Also, if this is the last descriptor, set bit 31 to 1 to indicate the end of the descriptor table.

6. Set the reserved DWORD to 0.

7. Repeat steps 3 to 6 for each descriptor required.

8. Logically OR the physical address of the GUI descriptor table (GUI_TABLE_ADDR@BM_GUI_TABLE) with the circular buffer size you wish to set up (CIRCULAR_BUF_SIZE@BM_GUI_TABLE), and write this value to BM_GUI_TABLE.

9. Set SRC_BM_ENABLE, SRC_BM_SYNC, and BUS_MASTER_OP@SRC_CNTL to the active settings. BUS_MASTER_OP = 3 for a system memory to bus master host data register transfer.

10. Initiate a GUI operation (write DST_WIDTH or DST_HEIGHT_WIDTH). The value you write to this register does not matter.

To determine if the transfer is complete, you must wait for engine idle.

If an application is writing to consecutive registers (i.e. register MM offsets are consecutive), the bus master hardware can be set up to use the first DWORD as the address of the starting register, then continue for n registers. The value of n-1 (writing 0 means that 1 register will be written) is written to GUIREG_COUNTER@BM_ADDR, and the hardware will automatically increment the register address on each write. Thus the data would be formatted:

DWORD (register address in MM offset format = "n")

DWORD (data to be written to register n)

DWORD (data to be written to register (n + 1))

DWORD (data to be written to register (n + 2))

...

This page intentionally left blank.

Appendix A
BIOS Services

A.1 Introduction

All accelerator services are provided in the accelerator ROM segment at offset 64h. The ROM segment can be determined by reading the SCRATCH_REG1 register and calculating:

```
SEGMENT = (SCRATCH_REG1 & 0x7F) * 0x80 + 0xC000
```

where SCRATCH_REG1 is I/O register 11h.

The ROM segment can also be found by using the method described in *Section 3.3: mach64 Detection*.

A.2 Services

All functions return with error code in AH		
AH	= 0	; No error
AH	= 1	; Function complete with error
AH	= 2	; Function not supported

Function AL=0, Load Coprocessor CRTC Parameters		
CL[3:0]	= Color depth	
	= 1	; 4 bpp
	= 2	; 8 bpp
	= 3	; 15 bpp (555)
	= 4	; 16 bpp (565)
	= 5	; 24 bpp (in RGB format if available, else in BGR)
	= 6	; 32 bpp (in RGBx format if available, else whatever 32 bpp that is ; supported)
CL[4]	= 1	; Enable gamma correction if 15 bpp and above ; Set the RAMDAC to 8 bits if in 8 bpp mode for support of ; 256 color grey scale
CL[7:6]	= Pitch size	
	= 0	; 1024
	= 1	; Don't change
	= 2	; Pitch size is the same as horizontal display

Function AL=0, Load Coprocessor CRTC Parameters (Continued)		
CH	= Resolution	
	= 12h	; 640x480
	= 6Ah	; 800x600
	= 55h	; 1024x768
	= 80h	; Load table from offset of external storage (EEPROM) in BX
	= 81h	; Load table according to data in DX:BX
	= 82h	; OEM specific mode
	= 83h	; 1280x1024
	= 84h	; 1600x1200
	= E1h	; 640x400
	= E2h	; 320x200
	= E3h	; 320x240
	= E4h	; 512x384
	= E5h	; 400x300
	= E6h	; 640x350
DX:BX	= Pointer to parameter table if CH = 81h	
BX	= Offset into EEPROM table if CH = 80h	

Function AL=1, Set Display Mode		
CL[0]	= 0	; VGA and set the DAC to 6 bits
	= 1	; Coprocessor
CL[5]	= 0	;
CL[7]	= 1	; Enable 8-bit DAC or Gamma Correction ; This bit is OR'ed with CL[4] in function AL = 0
In Return		
CL[5]	= 0	; H CRTC parameters are normal
	= 1	; H CRTC parameters doubled by hardware If programmed by application directly, CRTC pitch needs to be divided by 2.

Function AL=2, Load Coprocessor CRTC Parameters and Set Display Mode
Same arguments as AL = 0

Function AL=3, Read EEPROM Data	
BX	= Index
Returns	
BX	= Data

Function AL=4, Write EEPROM Data	
BX	= Index
DX	= Data

Function AL=5, Memory Aperture Service		
CL	= 0	; Disable memory aperture
CL[0]	= 1	; Enable memory aperture
CL[2]	= 1	; Enable VGA memory aperture
CL[7]	= 1	; Set memory aperture location
BX	= Aperture location in MByte. Only implemented with major revision 1 or higher.	

Function AL=6, Short Query Function		
AL[4:0]	= Aperture configuration	
	= 0 = 1 = 2	; Disable ; 4M ; 8M
AL[5]	= 1	; VGA disable
AL[6]	= 0 = 1	; Aperture address is user configurable ; Aperture address is predefined or hard coded in BIOS
AL[7]	= 0 = 1	; Aperture address is in 4G range ; Aperture address is in 128M range
BX	= Aperture address	
CH	= Color depth support (see offset 13 in the Query Structure table)	
CL	= Memory size	
DX	= ASIC identification, [7:0] = revision, [15:8] = type	

Function AL=7, Return Hardware Capability List	
In Return	
DX:BX	= Offset into a table specifying the maximum dot clock information. Each mode is organized as a row of information in the table, terminated by a zero after the last row, as shown below:
DX:[BX-1]	= Number of bytes per row
DX:[BX-2]	= Format type
AL	= Format type
	= 0, 1

H_DISP	DACTYPE (if format == 0) RAMTYPE (if format == 1)	MEMREQ	MAX DOTCLOCK	PIXEL WIDTH
0 (end of table)				

Function AL=7, Return Hardware Capability List

In Return

DX:CX = Pointer into a table specifying the maximum dot clock information (only if the value in CX has been modified. Set CX = 0FFFFh and check if the value is changed after calling). Application program should check this table first to determine if the video mode is supported. Each mode is organized as a row of information in the table, terminated by a zero after the last row, as shown below:

H_DISP	DACTYPE (if format == 0) RAMTYPE (if format == 1)	MEMREQ	MAX DOTCLOCK	PIXEL WIDTH
0 (end of table)				

H_DISP	Horizontal resolution in number of characters.
DACMASK	(1 Shift-Left DACTYPE)
RAMMASK	(1 Shift-Left RMATYPE)
DACTYPE	DACTYPE including the subtype information
MEMREQ	The minimum memory required to support the specified resolution and color depth. (DRAM requirement Shift-Left 4) or (VRAM requirement)
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) &
       (DACMASK & (1 shl dactype))          &
   (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
```

Function AL=8, Return Query Device Data Structure in Bytes

On Entry:		
CL[0]	= 0	; Buffer size for header information only
	= 1	; Buffer size for header information and mode tables
Returns:		
CX	= Number of bytes	

Function AL=9, Query Device

DX:BX	= Pointer to buffer	
CL[0]	= 0	; Return header information only
	= 1	; Return header information and mode table

Function AL=0Ah, Return Clock Chip Frequency Table

AL	= Clock chip type	
DX:BX	= Offset pointing to the 16 words containing the pre-programmed dot clock frequency; unit is in KHz/10 (four significant digits)	
DX:CX	= Offset pointing to the table containing clock chip information in the following format:	
	db	Clock chip type
	db	Frequency table identification
	dw	Minimum PCLK frequency (in KHz/10)
	dw	Maximum PCLK frequency (in KHz/10)
	db	Extended coprocessor mode PCLK entry if != 0ffh
	db	Extended VGA mode PCLK entry if != 0ffh
	dw	Reference clock frequency (in KHz/10)
	dw	Reference clock divider
	dw	Hardware specific information
	dw	MCLK frequency in power-down mode
	dw	MCLK frequency in normal mode for DRAM boards

Function AL=0Ah, Return Clock Chip Frequency Table (Continued)		
	dw	MCLK frequency in normal mode for VRAM boards
	dw	SCLK frequency
	db	MCLK entry number
	db	SCLK entry number
	dw	Coprocessor mode MCLK frequency if ! = 0
	dw	Reserved
	dw	0ffh

Function AL=0Bh, Program a Specified Clock Entry		
CL[2-0]	= 0	; PCLK, dot clock
	= 1	; MCLK, memory clock
	= 2	; SCLK, serial clock
	= 3	; internal clock
CH	= Entry in the frequency table for programming PCLK	
BX	= Unit is in KHz/10	
in return		
AL	= Clock chip type	
BX	= Programming word depending on type	

Function AL=0Ch, DPMS Service, Set DPMS Mode		
CL[1-0]	= 0	; Active
	= 1	; Standby
	= 2	; Suspend
	= 3	; Off
	= 4	; Blank the display (this is not a DPMS state)

Function AL=0Dh, Return Current DPMS State in CL

Function AL=0Eh, Set Graphics Controller's Power Management State		
C[1-0]	= 0	; Active
	= 1	; Standby
	= 2	; Suspend
	= 3	; Off

Function AL=0Fh, Return Graphics Controller's Current Power Management State in CL

Function AL=10h, Set DAC to Different States		
CL	=80h	; reserved
	=0	; set DAC to normal mode
	=1	; set DAC to sleep mode

Function AL=11h, Return External Storage Device Information (INSTALL should use this information to dynamically configure the data structure.)		
CL	= External data structure information	
CL[7]	= 1	; No external data storage can be used; Write EEPROM will not work
CL[6-4]	= 000	; External data is readable and writable
	= 001	; External data storage is readable but not writable
	= 011	; External data storage is not readable and writable
	= 100	; External data storage is readable and writable; writing must be; handled by the application program based on device type in CL[3-0]
CL[3-0]	= 0	; device type
CH	= Number of read only CRT table in the storage device after the writable entry	
DL	= Last 16 bit writable entry in the storage device	
DH[7]	= 1	; the BIOS has built-in CRTC parameters
DH[5]	= 1	; the BIOS supports extended function AL = 15h
BL	= Offset into the CRTC parameter table	
BH	= Size of the CRTC parameter table; if the number is smaller than the one in the CRTC table, then discard the bottom ones	

For **INSTALL.EXE**,

```
if CL[7] ==0; normal mach64 operation
if ( (CL[7] ==1 ) & (DH[6] == 0) )
      the refresh information is predefined or handled by OEM's
      own program
if ( (CL[7] ==1 ) & (DH[7-6] == 11b )
      the refresh information can be handled through extended
      function AL=15h
```

Function AL=12h, Short Query	
CX	= 0 (to ensure backward compatibility)
On Return	
AX	= Reserved
BX	= Reserved
CX	= 0 for fixed I/O (2ECh, 1CCh, 1C8h) = 1 for relocatable or block I/O
DX	= IO base address

Function AL=13h, Display Data Channel Support (DDC)		
BL	= 0	; Return DDC format supported by graphics controller and monitor
On Return		
	BX= 0	; DDC not supported
	BX[0] = 1	; DDC1 supported by monitor
	BX[1] = 1	; DDC2B supported by monitor
	AL[0] = 1	; DDC1 supported by BIOS
	AL[1] = 1	; DDC2B supported by BIOS
	AL[2] = 1	; DDC2AB supported by BIOS
	AL[6] = 1	; BIOS support detailed EDID timing at power-up
	AL[7] = 1	; BIOS can use EDID information to set up the board at power-up
BL	= 1	; Read EDID data (supports DDC1/DDC2B only, first EDID block for DDC2B)
		CX = Buffer size
		DX:DI = Pointer to buffer
BL	= 2	; Read buffer (only supports DDC2B or DDC2AB)
		CX = Buffer size
		DX:DI = Pointer to buffer (monitor address in first byte of DX:DI when calling)
BL	= 3	; Write buffer (only supports DDC2B or DDC2AB)
		CX = Buffer size
		DX:DI = Pointer to buffer
		DX:[DI] .. DX:[DI+CX-1] = data to write
		DX:[DI+CX] = number of bytes to read after write
		DX:[DI+CX+1] = delay in msec between write and read
		DX:[DI+CX+2] = destination address
		DX:[DI+CX+3] = source address
		in return
		DX:DI = data read if required

Function AL=14h, Save and Restore Graphics Controller States		
CL	= 0	; Return buffer size required in number of bytes
		CX = Buffer size
		BX = Save and restore mechanism used
		BX[0] = 1; can pass in segment point to last 64K of VGA or linear aperture
		BX[1] = 1 ; can pass in segment pointer pointing to 0:0 with full access

Function AL=14h, Save and Restore Graphics Controller States (Continued)		
		BX[2] = 1 ; can pass in segment pointer pointing to beginning of memory aperture
CL	= 1	; Save controller states
		DX:DI = Pointer to buffer
		BX = Save and restore mechanism used
		if (BX[0] = 1 in the CL=0 function) SI = segment pointer to last 64K of VGA or linear aperture
		if (BX[1] = 1 in the CL=0 function) SI = segment pointer to 0:0 with full access
		if (BX[2] = 1 in the CL=0 function) SI = segment pointer to memory aperture
CL	= 2	; Restore controller states
		DX:DI = Pointer to buffer
		BX = Save and restore mechanism used
		if (BX[0] = 1 in the CL=0 function) SI = segment pointer to last 64K of VGA or linear aperture
		if (BX[1] = 1 in the CL=0 function) SI = segment pointer to 0:0 with full access
		if (BX[2] = 1 in the CL=0 function) SI = segment pointer to memory aperture

Function AL=15h, Refresh Rate Support			
BL	= 0	; Get current refresh rate information	
	= 1	; Change current refresh rate information	
	= 2	; Save refresh rate information	
		DX:DI = Pointer to buffer (minimum 20 bytes required and it is terminated by 0FFFFh)	
		Offset (word)	Content
		0	12h (640x480), refresh mask
			bit 6 = 72Hz
			bit 5 = 75Hz
			if bits 5, 6 = 0; 60Hz
		1	6Ah (800x600), refresh mask
			bit 3 = 56Hz
			bit 2 = 60Hz
			bit 1 = 72Hz
			bit 0 = 75Hz
		2	55h (1024x768), refresh mask
			bit 3 = 87Hz interlaced
			bit 2 = 60Hz
			bit 1 = 70Hz
			bit 0 = 75Hz
		3	83h (1280x1024), refresh mask
			bit 4 = 43Hz
			bit 3 = 47Hz
			bit 2 = 60Hz
			bit 1 = 70Hz
			bit 0 = 75Hz
		4	84h (1600x1200), refresh mask
		(last)	0FFFFh
	= 3	; Set current external CRT table state	
		BH = 1	; use external CRTC table
		BH = 0	; don't use external CRTC table
	= 4	; Current external CRT table state	
		AL[0] = 1	; external CRTC table will be used by the BIOS
		AL[0] = 0	; external CRTC table will not be used by the BIOS

A.3 Query Structure

Query Structure	
Byte Offset	**Description**
0:1h	Size of structure in bytes
2h	Revision of query structure
3h	Number of mode tables
4:5h	Offset in bytes to mode tables
6h	Size of each mode table in bytes
7h	VGA Type: 0= disabled 1= enabled
	ASIC identification bits 15:0 ASIC type D700 = GX-C D701 = GX-D D702 = GX-E 57xx = CX 43xx = CT 45xx = ET
Ah	VGA Boundary 0h= full access 1h= 256K 2h= 512K 3h= 768K 4h= 1M 10h= no access through VGA
Bh	Memory Size 0= 512K 1= 1M 2= 2M 3= 4M 4= 6M 5= 8M

Query Structure (Continued)	
Byte Offset	**Description**
Ch	Bits 3:0DAC type; Bits 7:4DAC subtype 00h = Internal DAC XXXX 1XXXb = Internal DAC 01h = IBM RGB514 02h = TLC34075 / ATI68875 72h = TVP3026 03h = Brooktree BT476/8 04h = Brooktree BT481 14h = AT&T20C490, AT&T20C491, AT&T20C493, SC15025/15026, IMS-G174, MU9C4910, MU9C1880 05h = ATI68860 Rev B 15h = ATI68860 Rev C 75h = TVP3026 06h = STG1700 16h = AT&T20C498 07h = STG1702 17h = SC15021 27h = AT&T21C498 37h = STG1703 47h = Chrontel CH8398 57h = AT&T20C408

Query Structure (Continued)	
Byte Offset	**Description**
Dh	Memory Type bit 7 0 = the memory may support blockwrite 1 = the memory does not support blockwrite bits 3:0 For **GX** controller 0 = DRAM 256Kx16 1 = VRAM 256K x4 2 = VRAM 256K x16 3 = DRAM 256Kx4 5 = VRAM 256x4 special 6 = VRAM 256x16 special For **CX** controller 0 = DRAM symmetric RAS/CAS 1 = EDO DRAM 3 = DRAM asymmetric RAS/CAS For **CT, ET, GT, LT, ST** and **VT** controllers 1 = DRAM 2 = EDO DRAM 3 = BRRAM 4 = SDRAM 5 = SGRAM
Eh	Bus Type 0 = ISA 1 = EISA 2 - 4 = Reserved 5 = VLB non-multiplexed 6 = VLB 7 = PCI
Fh	Bit 7 - Enable composite sync Bit 6 - Enable sync on green
10h:11h	Aperture address in megabytes (0-4095)
12h	Aperture configuration (see extended BIOS function AL=6)

Query Structure (Continued)	
Byte Offset	**Description**
13h	Color Depth Support Bit Definition 0 = 1; support 16 bpp, 565 1 = 1; support 16 bpp, 555 2 = 1; support RGB in 24 bpp 3 = 1; support BGR in 24 bpp 4 = 1; support 32 bpp (unpack 24 bpp in xBGR, x is byte 0) 5 = 1; support 32 bpp (unpack 24 bpp in RGBx, R is byte 0) 6 = 1; support 32 bpp (unpack 24 bpp in BGRx, B is byte 0) 7 = 1; support 32 bpp (unpack 24 bpp in xRGB, x is byte 0)
14h	RAMDAC support feature 4 = 1; support sleep mode 5 = 1; support 256 grey scale 6 = 1; support gamma correction 7 = 1; support sync on green
15h	Bit 0 = I/O address type (see extended function AL = 12h)
16h:17h	Offset into current mode table if non-zero (not implemented)
18h:19h	I/O base address
1Ah:1Bh	Offset into DAC Hsync pipeline delay adjust information
1Ch:1Fh	Reserved

A.4 Mode Table Structure

Mode tables immediately follow the device status table. Use the forward pointer to reference mode tables, as the device status table may expand in the future. It is possible to have no modes installed. Typically, between 2 and 7, mode tables will be returned.

Mode Table Structure	
Offset	**Description**
Installed Mode Table 1	
0:1h	Horizontal display resolution, in pixels
2:3h	Vertical display resolution, in scanlines
4h	Maximum pixel depth
5h	Reserved
6:7h	Offset into EEPROM = 0;Table is generated from VGA parameters <>0; Offset into EEPROM table
8:9h	Reserved
A:Bh	Reserved
C:Dh	Bit 15= Reserved Bit 14= Use external crystal if ATI 18818 is used Bit 13= Enable mux mode Bit 12= Enable Composite Sync Bit 11= Enable Hsync delay in BIOS Bit 10= Reserved, used for TLC34075 Bit 9= Enable interlace Bit 8= Enable double scan Bits 7:4= Mode table type — 0 for external, 1 for internal Bits 3:0= Reserved
Eh	CRTC_H_TOTAL
Fh	CRTC_H_DISP
10h	CRTC_H_SYNC_STRT
11h	CRTC_H_SYNC_WID
12:13h	CRTC_V_TOTAL
14:15h	CRTC_V_DISP
16:17h	CRTC_V_SYNC_STRT
18h	CRTC_V_SYNC_WID
19h	CLOCK_CNTL
1A:1Bh	Dot clock for coprocessor mode, for programmable clock chip

Mode Table Structure (Continued)	
Offset	**Description**
1C:1Dh	Bits 15:12= Reserved Bits 11:8= CRTC_H_SYNC_DLY Bits 7:4= OVR_WID_RIGHT Bits 3:0= OVR_WID_LEFT
1E:1Fh	OVR_WID_T0P, OVR_WID_BOTTOM
20:21h	OVR_CLR_B, OVR_CLR_8
22:23h	OVR_CLR_G, OVR_CLR_R
Installed Mode Table 2	
24:47h	Entries definition same as Mode Table 1.
. . .	
Installed Mode Table n	
N*24h- (N*24+23h)	Entries definition same as Mode Table 1.

A.5 Scratch Registers Information

Note that the following information is subject to change, with the exception of the low byte of SCRATCH_REG1, which conveys ROM location information to applications.

SCRATCH_REG0 (42ECh)		
	Bit 1:0	Graphics controller power management states
SCRATCH_REG0 + 1 (42EDh)		800x600 refresh rate information
	Bit 7	External CRTC table indicator
	Bits 6:0	800x600 refresh mask
SCRATCH_REG0 + 2 (42EEh)		Reserved (can be 1280x1024)
	Bit 7	DDC2 detected state
	Bit 6	640x480 72Hz
	Bit 5	640x480 75Hz
	Bits 4:0	1280x1024 refresh mask

SCRATCH_REG0 + 3 (42EFh)	1024x768 refresh rate information
Bit 7	Not used
Bits 6:0	1024x768 refresh mask
SCRATCH_REG1 (46ECh)	ROM location
SCRATCH_REG1 + 1 (46EDh)	
Bits 7:4	RAMDAC subtype
Bit 3	Not used
Bit 2	If set, disable the programming of DAC to VGA mode when INT 10h is called, for CT, CX only.
Bit 1	Reserved
Bit 0	Sync on green enable
SCRATCH_REG1 + 2 (46EEh)	
Bits 7:6	CRTC pitch size
Bit 5	Mux mode
Bit 4	Enable gamma correction or 256 color greyscale
Bit 3	32bpp color orientation information
Bit 2	TLC34075 output clock select or TVP3026 15/16bpp information
Bit 1	32bpp color orientation information
Bit 0	Current gamma correction or 256 color state_cation
SCRATCH_REG1 + 3 (46EFh)	Programmable dot clock information
1CE/BB (VGA enable, GX/CX only)	
Bits 7:6	640x480 refresh rate information
Bits 5:4	Monochrome mode, color information
Bit 1	If set, use VGAWONDER compatible paging mechanism in packed pixel mode
Bit 0	If set, disable the programming of DAC to VGA mode when INT 10h is called.

Appendix B
Extended BIOS (LT Specific)

B.1 Return Panel Type and Controller Supported Information

Function 80h - Return Panel Type and Controller Supported Information

This function is intended for diagnostic support and may not have a real application purpose. The tables included are OEM specific and the information returned depends on the controller and panel used by the OEM.

To Call:	AL	= 80h	Return panel type and controller supported information
Returns:	DX:DI	=	Pointer to a data structure that would identify the capabilities of the controller and types of panel that can be supported in the BIOS and their corresponding identification code.

The following tables are OEM specific and the information returned will depend on the controller and a panel used by the OEM.

B.1.1 Header Information

Table B-1 Header Information

Offset (byte)	Content
0 - 1	Data structure type, will be 0
2 - 9	ATI signature string
10 - 17	OEM signature string

Table B-1 Header Information (Continued)

Offset (byte)	Content
18 - 19	bit 0 = 0 Reserved bit 1 = 0 Reserved bit 2 = 1 If inverse video is supported bit 3 = 1 If shading control is supported bit 4 = 1 If contrast control is supported bit 5 = 1 If brightness control is supported bit 6 = 1 If positioning is supported bit 7 = 1 If expansion is supported bit 8 = 1 If text cursor size control is supported bit 9 = 1 If text cursor blinking control is supported bit 10 = 1 If hardware icon is supported bit 11 = 1 If color dithering is supported bits 15 - 12 = Reserved
20 - 21	Offset into the panel information table with panel ID 0, DX is the segment
22 - 23	Offset into the panel information table with panel ID 1, DX is the segment
24 - 25	Offset into the panel information table with panel ID 2, DX is the segment
. . .	. . .
82 - 83	Offset into the panel information table with panel ID 31, DX is the segment
84 - 95	Reserved

B.1.2 Panel Information

Table 0-1 Panel Information

Offset (byte)	Content
0	Panel identification (000h - 01Fh)
1 - 24	Panel identification string
25 - 26	Horizontal size in pixels
27 - 28	Vertical size in lines
20 - 30	Flat panel type bit 0 = 0 MonoCHrome = 1 Color bit 1 = 0 Single panel construction = 1 Dual (split) panel construction bits 7 - 2 = 0 STN (passive matrix) = 1 TFT (active matrix) = 2 Active addressed STN = 3 EL = 4 Plasma bits 15 - 8 = Reserved
31	Red bits per primary
32	Green bits per primary
33	BLue bits per primary
34	Reserved bits per primary
35 - 38	Size in KB of off screen memory required for frame buffer
39 - 42	Pointer to reserved off screen memory for frame buffer
43 - 56	Reserved

Table 0-1 Panel Information (Continued)

Offset (byte)	Content		
57 - 60	bits 2 - 0	Panel format:	
		For split-panel color STN panels	
		= 000	PACK6 (12-bit interface, 6-bit to upper panel, 6-bit to lower panel)
		= 001	PACK8 (16-bit interface, 8-bit to upper panel, 8-bit to lower panel)
		= 010	PACK12 (24-bit interface, 12-bit to upper panel, 12-bit to lower panel)
		For single-panel color STN panels	
		= 000	PACK12 (12-bit interface)
		= 001	PACK16 (16-bit interface)
		For TFT panels	
		= 000	8-color panel - (111 RGB)
		= 001	512-color panel - (333 RGB)
		= 010	4096-color panel - (444 RGB)
		= 100	18-bit/pixel panel - (666 RGB)
		= 101	24-bit/pixel panel - (888 RGB)
	bit 3	= Reserved	
	bit 7 - 4	Panel type	
		= 0001	Split panel STN color
		= 0011	Single panel STN color
		= 0111	Color TFT (1 pixel per CLock)
		= 1111	Color TFT (2 pixels per CLock)
	bits 10 - 8	Grey scale level	
		= 000	Indicates no frame modulation should be done (applies only to TFT panels)
		= 001	2 levels of grey support (applies only to TFT panels)
		= 010	4 levels of grey support (applies only to TFT panels)
		= 011	8 levels of grey support (applies only to STN panels)
		= 100	16 levels of grey support (applies only to STN panels)
		= 101	32 levels of grey support (applies only to monoCHrome STN panels)
		= 110	64 levels of grey support (applies only to STN panels)

Table 0-1 Panel Information (Continued)

Offset (byte)	Content
57 - 60	bits 12 - 11 MOD signal toggle rate = 00 Toggle MOD signal every frame = 01 Toggle MOD signal every 7 lines = 10 Toggle MOD signal every 13 lines = 11 Toggle MOD signal every 19 lines bit 14 - 13 Cursor BLink rate = 00 Same as CRT = 01 Blink every 32 frames = 10 Blink every 48 frames = 11 Blink every 64 frames bits 15 Reserved bit 16 Active frame pulse / VSYNC = 0 Active high frame pulse / VSYNC = 1 Active low frame pulse / VSYNC bit 17 Active line pulse / HSYNC = 0 Active high line pulse / HSYNC = 1 Active low line pulse / HSYNC bit 18 Active display enable / MOD = 0 Active high display enable / MOD = 1 Active low display enable / MOD bit 19 Active shift CLock / PCLK = 0 Active high shift CLock / PCLK = 1 Active low shift CLock / PCLK bit 21 - 20 Dithering = 00 Disable dithering = 01 Dither to 4 bits = 10 Dither to 5 bits = 11 Dither to 6 bits

Table 0-1 Panel Information (Continued)

Offset (byte)	Content
57 - 60	bit 22 Inverse video = 0 Disable inverse video = 1 Enable inverse video bit 24 - 23 Backlight modulation CLock selection = 00 32 KHz = 01 32 KHz divided by 2 = 10 32 KHz divided by 3 = 11 32 KHz divided by 4 bits 27 - 25 Backlight brightness level = 00 Dimmest = 11 Brightest
	bits 31 - 28 HSYNC delay for the LCD panel = 000 No delay = 001 Delay by 1 VCLK = 010 Delay by 2 VCLKs = 011 Delay by 3 VCLKs = 100 Delay by 4 VCLKs = 101 Delay by 5 VCLKs = 110 Delay by 6 VCLKs = 111 Delay by 7 VCLKs
61	bit 0 = 1 If LVDS interface is used bits 7 - 1 = Reserved

Table 0-1 Panel Information (Continued)

Offset (byte)	Content
62 - 63	Minimum pixel CLock supported
64 - 65	Maximum pixel CLock supported
66 - 67	Reserved
68 - 69	Reserved
70 - 127	Array of offsets into mode tables, DX is the segment, and the end of the array will have an offset of 00000h

B.1.3 Mode Table Structure

Table 0-2 Mode Table Structure

Offset (byte)	Content
0 - 1	Horizontal display resolution in pixels
2 - 3	Vertical display resolution in lines
4 - 5	bit 0 = 1 If mode table is for VGA mode bit 1 = 1 If mode table is for coprocessor mode bit 2 = 1 If ImpacTV is supported bits 15 - 3 = Reserved
6 - 7	Offset into parameter table for expansion
8 - 9	Pixel CLock
10 - 11	Pixel CLock adjustment
12 - 15	bits 10 - 0 = FP_POS bit 11 = Reserved bits 22 - 12 = LOWER_PANEL_VPOS bit 23 = Reserved bits 25 - 24 = LP_VPOS_ADJUST0 bits 27 - 26 = LP_VPOS_ADJUST1 bits 29 - 28 = LP_VPOS_ADJUST2 bit 31 - 30 = Reserved
16 - 17	CRTC_H_TOTAL

Table 0-2 Mode Table Structure (Continued)

Offset (byte)	Content
18 - 19	CRTC_H_DISP
20 - 21	CRTC_H_SYNC_STRT_DLY
22	CRTC_H_SYNC_WID
23	OVR_WID_LEFT
24	OVR_WID_RIGHT
26 - 25	CRTC_V_TOTAL
27 - 28	CRTC_V_DISP
29 - 30	CRTC_V_SYNC_STRT
31	CRTC_V_SYNC_WID
32 - 33	OVR_WID_TOP
34 - 35	OVR_WID_BOTTOM

B.1.4 Additional Mode Table Structure for DSTN Panel

Table 0-3 Additional Mode Table Structure for DSTN Panel

Offset (byte)	Content
36 - 37	Pixel CLock
38 - 39	Pixel CLock adjustment
40 - 43	bits 10 - 0 = FP_POS bit 11 = Reserved bits 22 - 12 = LOWER_PANEL_VPOS bit 23 = Reserved bits 25 - 24 = LP_VPOS_ADJUST0 bits 27 - 26 = LP_VPOS_ADJUST1 bits 29 - 28 = LP_VPOS_ADJUST2 bit 31 - 30 = Reserved
44 - 45	CRTC_H_TOTAL
46 - 47	CRTC_H_DISP
48 - 49	CRTC_H_SYNC_STRT_DLY
50	CRTC_H_SYNC_WID
51	OVR_WID_LEFT
52	OVR_WID_RIGHT
53 - 54	CRTC_V_TOTAL
55 - 56	CRTC_V_DISP
57 - 58	CRTC_V_SYNC_STRT
59	CRTC_V_SYNC_WID
60 - 61	OVR_WID_TOP
62 - 63	OVR_WID_BOTTOM

B.1.5 Expansion Mode Table Structure

Table 0-4 Expansion Mode Table Structure

Offset (byte)	Content
0 - 1	Pixel clock
2 - 3	Pixel clock adjustment
4 - 7	bits 10 - 0 = FP_POS bit 11 = Reserved bits 22 - 12 = LOWER_PANEL_VPOS bit 23 = Reserved bits 25 - 24 = LP_VPOS_ADJUST0 bits 27 - 26 = LP_VPOS_ADJUST1 bits 29 - 28 = LP_VPOS_ADJUST2 bit 31 - 30 = Reserved
8 - 9	CRTC_H_TOTAL
10 - 11	CRTC_H_DISP
12 - 13	CRTC_H_SYNC_STRT_DLY
14	CRTC_H_SYNC_WID
15	OVR_WID_LEFT
16	OVR_WID_RIGHT
17 - 18	CRTC_V_TOTAL
19 - 20	CRTC_V_DISP
21 - 22	CRTC_V_SYNC_STRT
23	CRTC_V_SYNC_WID
24 - 25	OVR_WID_TOP
26 - 27	OVR_WID_BOTTOM
28 - 31	HORZ_BLEND_RATIO
32 - 35	Vertical stretCHing for VGA mode bits 9 - 0 = VERT_STRETCH_RATIO0 bits 19 - 10 = VERT_STRETCH_RATIO1 bits 29 - 20 = VERT_STRETCH_RATIO2 bits 31 - 30 Reserved
36 - 37	Vertical stretCHing for coprocessor mode bits 9 - 0 = VERT_STRETCH_RATIO0 bits 15 - 10 = Reserved

B.2 Return Panel Identity Information

Function 81h – Return Panel Identity Information

This function allows checking for current attached flat panel device identification.

To Call:	AL	=	81h	Return Panel Identity Information
Returns:	CL [4 - 0]	=	Panel identity	(see Function 80h)
	CL [7 - 5]	=	000b	Reserved
	DX:DI	=	Pointer to the panel definition	(see Function 80h)

B.3 VBE / FP Functions

Function 82h – VESA BIOS Extensions / Flat Panel Functions

Reserved values should always be set to the value zero (0).

Subfunction 01h – Return Flat Panel Information

To Call:	AL	=	82h	VBE / FP Functions
	BL	=	01h	Return flat panel information
Returns:	DX:DI	=	Pointer to flat panel information structure	

Comments: This subfunction returns information about the current attached flat panel device.

Table 0-5 Flat Panel Information Structure

Offset (byte)	Content		
0 - 1	Horizontal size in pixels		
2 - 3	Vertical size in lines		
4 - 5	Flat panel type		
	bit 0	= 0	Monochrome
		= 1	Color
	bit 1	= 0	Single panel construction
		= 1	Dual (split) panel construction

Table 0-5 Flat Panel Information Structure (Continued)

Offset (byte)	Content
4 - 5	bits 7 - 2 = 0 STN (passive matrix) = 1 TFT (active matrix) = 2 Active addressed STN = 3 EL = 4 Plasma bits 15 - 8 = Reserved
6	Red bits per primary
7	Green bits per primary
8	BLue bits per primary
9	Reserved bits per primary
10 - 13	Size in KB of off screen memory required for frame buffer
14 - 17	Pointer to reserved off screen memory for frame buffer
18 - 31	Reserved

Subfunction 02h – Return/Select Inverse Video

This Subfunction provides for checking/setting the current state of screen inversion being an ability to display black text/graphics on a white background.

To Call:

AL	=	82h	VBE/FP function
BL	=	02h	Return/select inverse video
BH	=	00h	Return request

Returns:

BL = Current polarity state

BL [0]	= 1	Text modes inverted
BL [1]	= 1	Graphics modes inverted
BL [7 - 2]	= 000000b	Reserved

BH = Available polarity settings

BH [0]	= 1	Text inverse available
BH [1]	= 1	Graphics inverse available
BH [2]	= 1	Text and Graphics inverse must be the same
BH [7 - 3]	= 00000b	Reserved

Comments: Available settings must be tested prior to using “set” command.

To Call:

AL	=	82h	VBE/FP function
BL	=	02h	Return/select inverse video
BH	=	01h	Select request
CL	=	Active polarity to set	
		CL [0] = 1	Inverse text modes
		CL [1] = 1	Inverse graphics modes
		CL [7 - 2] = 000000b	Reserved

Comments: When bit 2 in “available settings” is set (BH [2] = 1), bits 0 and 1 must be set accordingly. Otherwise the inverse will be turned “off”.

Subfunction 03h – Return/Select Flat Panel Shading Options

This Subfunction provides for an end user the ability to select from a range of OEM supplied shading options.

To Call:

AL	=	82h	VBE/FP function
BL	=	03h	Return/select flat panel shading options
BH	=	00h	Get number of shading options request

Returns:

CL	=	Number of shading options
CH	=	Current shading option

To Call:

AL	=	82h	VBE/FP function
BL	=	03h	Return/select flat panel shading options
BH	=	01h	Select option request
CH	=	Shading option number to set	

Comments: Valid shading option to set is 1 to the number of shading options returned by the “Get number of shading options”.

Subfunction 04h – Return / Select Flat Panel Contrast

This Subfunction allows for the selection of flat panel contrast levels when OEM has provided a software interface for adjusting the voltage to the biasing circuitry on the panel.

This is not a frame rate control.

To Call:

AL	=	82h	VBE/FP function
BL	=	04h	Return / Select Flat Panel Contrast
BH	=	00h	Return Range Request

Returns:

CH	=	Upper Limit
CL	=	Current flat panel contrast

To Call:

AL	=	82h	VBE/FP function
BL	=	04h	Return / Select Flat Panel Contrast
BH	=	01h	Select Request
CL	=	Flat panel contrast to set	

Comments: Valid flat panel contrast to set is 0 to the upper limit returned by the “Return Range Request”

Subfunction 05h – Return / Select Flat Panel Brightness

This Subfunction allows for the selection of flat panel brightness levels when OEM has provided a SW interface for adjusting the voltage to the blacklight.

To Call:

AL	=	82h	VBE/FP function
BL	=	05h	Return / Select Flat Panel Brightness
BH	=	00h	Return Range Request

Returns:

CH	=	Upper Limit
CL	=	Current flat panel brightness

To Call:

AL	=	82h	VBE/FP function
BL	=	05h	Return / Select Flat Panel Brightness
BH	=	01h	Select Request
CL	=	Flat panel brightness to set	

Comments: Valid flat panel brightness option to set is 0 to the upper limit returned by the “Return Range Request”

Subfunction 06h – Return / Select Vertical and Horizontal Positioning

The displayed portion of the mode is positioned by means of this Subfunction which represents a global hardware setting. However, its effectiveness may be mode dependent.

To Call:

AL	=	82h	VBE/FP function
BL	=	06h	Return / Select Vertical and Horizontal Positioning
BH	=	00h	Return Request

Returns:

BL = Current vertical position

= 0	Top
= 1	Center
= 2	Bottom

All other values are reserved

BH = Available vertical position settings

BH [0]	= 1	Top
BH [1]	= 1	Center
BH [2]	= 1	Bottom
BH [7 - 3]	= 00000b	Reserved

CL = Current horizontal position

= 0	Left
= 1	Center
= 2	Right

All other values are reserved.

CH = Available horizontal position settings

BH [0	= 1	Left
BH [1]	= 1	Center
BH [2]	= 1	Right
BH [7 - 3]	= 00000b	Reserved

Comments: The returned are hardware values being set, not the mode dependent information.

To Call:

AL	= 82h	VBE/FP function
BL	= 06h	Return / Select Vertical and Horizontal Positioning
BH	= 01h	Select Request
CH	= Vertical position to set	
	= 0	Top
	= 1	Center
	= 2	Bottom
	All other values are reserved.	
CL	= Horizontal position to set	
	= 0	Left
	= 1	Center
	= 2	Right
	All other values are reserved.	

Subfunction 07h – Return/Select Vertical and Horizontal Expansion

This Subfunction allows for the displayed portion of a mode to be expanded.

To Call:

AL	=	82h	VBE/FP function
BL	=	07h	Return / Select Vertical and Horizontal Expansion
BH	=	00h	Return Request

Returns:

BL	=	Current vertical expansion		
	BL [0]	= 0	Text expansion disable	
		= 1	Text expansion enable	
	BL [1]	= 0	Graphics expansion disable	
		= 1	Graphics expansion enable	
	BL [7 - 2]	= 000000b	Reserved	
BH	=	Available vertical expansion settings		
	BH [0]	= 0	Text expansion not available	
		= 1	Text expansion available	
	BH [1]	= 0	Graphics expansion not available	
		= 1	Graphics expansion available	
	BH [2]	= 1	Horizontal and vertical expansion must be enabled/disabled simultaneously	
	BH [7 - 3]	= 00000b	Reserved	
CL	=	Current horizontal expansion		
	CL [0]	= 0	Text expansion disabled	
		= 1	Text expansion enabled	
	CL [1]	= 0	Graphics expansion disabled	
		= 1	Graphics expansion enabled	
	CL [7 - 2]	= 000000b	Reserved	

CH	=	Available horizontal expansion settings		
		CH [0]	= 0	Text expansion not available
			= 1	Text expansion available
		CH [1]	= 0	Graphics expansion not available
			= 1	Graphics expansion available
		CH [2]	= 1	Horizontal and vertical expansion must be enabled/disabled simultaneously
		CH [7 - 2]	= 00000b	Reserved

Comments: The returned are hardware values being set, not the mode dependent information.

To Call:

AL	=	82h	VBE/FP function	
BL	=	06h	Return / Select Vertical and Horizontal Expansion	
BH	=	01h	Select Request	
CL	=	Horizontal expansion		
		CL [0]	= 0	Disable text expansion
			= 1	Enable text expansion
		CL [1]	= 0	Disable graphics expansion
			= 1	Enable graphics expansion
		CL [7 - 2]	= 000000b	Reserved
CH	=	Vertical expansion		
		CH [0]	= 0	Disable text expansion
			= 1	Enable text expansion
		CH [1]	= 0	Disable graphics expansion
			= 1	Enable graphics expansion
		CH [7 - 2]	= 000000b	Reserved

B.4 Monitor / TV Detection

Function 83h – Monitor / TV Detection

This Function allows to detect what display is attached to the computer and what its current status is.

To Call:	AL	= 83h	LCD / monitor / TV detection
	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 [7 - 2]	= 000000b	Reserved
Returns:	CL [1 - 0]	= Monitor	
		= 0	No monitor
		= 1	MonoCHrome monitor
		= 2	Color monitor
	CL [3 - 2]	= TV	
		= 0	No TV attaCHed
		= 1	TV attaCHed to composite connector
		= 2	TV attaCHed to S-Video connector
		= 3	TV attaCHed to both composite and S-Video connectors
		All other values are reserved.	

B.5 Return / Select Active Display

Function 84h – Return / Select Active Display

This Function allows application software to determine information about currently attached active display and set the necessary among the available active display modes.

To Call:

AL	=	84h	Return / Select Active Display
BH	=	00h	Return request

Returns:

BL = Current display

BL [0]	= 1	Flat panel
BL [1]	= 1	CRT
BL [2]	= 1	TV
BL [7 - 3]	= 00000b	Reserved

CL = Requested display

CL [0]	= 1	Flat panel
CL [1]	= 1	CRT
CL [2]	= 1	TV
CL [3]	= 1	Auto-switCH
CL [7 - 4]	= 0000b	Reserved

BH = Available display

BH [0]	= 1	Flat panel
BH [1]	= 1	CRT
BH [2]	= 1	TV
BH [7 - 3]	= 00000b	Reserved

To Call:

AL	=	84h	Return / Select Active Display
BH	=	01h	Select request
CL	=	Requested display	

CL [0]	= 1	Flat panel
CL [1]	= 1	CRT
CL [2]	= 1	TV
CL [3]	= 1	Auto-switCH
CL [7 - 4]	= 0000b	Reserved

Returns:

BL = Current display

BL [0]	= 1	Flat panel
BL [1]	= 1	CRT
BL [2]	= 1	TV
BL [7 - 3]	= 00000b	Reserved

B.6 Return / Select Power Management Mode

Function 85h – Return / Select Power Management Mode

This Function allows the ability to check / set between different power management and counter values for timer modes.

To Call:

AL	=	85h	Return / Select power management mode
BH	=	00h	Return request

Returns:

CL = Power management mode

CL [0]	= 0	Power management disabled
	= 1	Power management enabled
CL [2 - 1]	= 0	Pin mode
	= 1	Register mode
	= 2	Timer mode
CL [7 - 3]	= 00000b	Reserved

CH = Counter value for timer mode

CH [3 - 0] = Standby counter value in minutes
CH [7 - 4] = Suspend counter value in minutes

To Call:

AL	=	85h	Return / Select power management mode
BH	=	01h	Enable / Disable power management request
CL	=	Power management mode	
CL [0]	= 0		Disable power management
	= 1		Enable power management
CL [7 - 1]	= 0000000b		Reserved

To Call:

AL	=	85h	Return / Select power management mode
BH	=	02h	Set counter value request (for timer mode only)
CL	=	Counter value for timer mode	

CL [3 - 0] = Standby counter value in minutes
CL [7 - 4] = Suspend counter value in minutes

B.7 In and Out Suspend State

Function 86h – In and Out Of Suspend State

When this Function is called, interrupt should be disabled. The graphics subsystem is ready to put into suspend mode or ready to get out of suspend mode. It is assumed that no other graphics operation will be initiated after this call and suspend procedure or resume procedure should start immediately.

To Call: AL = 086h In and Out Of Suspend State

CL [3 - 0]	= 1	Suspend start, call before the hardware pin is put to suspend, ready to suspend when exit
	= 2	Suspend complete, call after the hardware pin is put to suspend
	= 3	Ready to get out of suspend, call before the hardware pin is put to normal
	= 4	Out of suspend is complete, call after the hardware pin is put to normal

All other values are reserved.

CH [7 - 4] = 0000b Reserved

This page intentionally left blank.

Appendix C
CRTC Parameters

C.1 Introduction

Note that all clock selects in the following tables assume an ATI18818 clock chip.

C.2 CRTC Parameters for 640x480

640x480 60Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		25.18MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x63	V_TOTAL	0x20C
Screen Display	H_DISP	0x4F	V_DISP	0x1DF
Sync Start	H_SYNC_STRT	0x51	V_SYNC_STRT	0x1E9
Sync Width	H_SYNC_WID	0x2C	V_SYNC_WID	0x22
Resolution	640		480	
Scan Frequency	31.469KHz		59.94Hz	
Polarity	(–)		(–)	
Sync Width	3.813us	12 chars	0.064ms	2 lines
Front Porch	0.636us	2 chars	0.318 ms	10 lines
Back Porch	1.907us	6 chars	1.048 ms	33 lines
Active Time	25.422us	80 chars	15.253ms	480 lines
Blank Time	6.356us	20 chars	1.430ms	45 lines

640x480 72Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		31.20MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x67	V_TOTAL	0x207
Screen Display	H_DISP	0x4F	V_DISP	0x1DF
Sync Start	H_SYNC_STRT	0x52	V_SYNC_STRT	0x1E8
Sync Width	H_SYNC_WID	0x25	V_SYNC_WID	0x23
Resolution	640		480	
Scan Frequency	37.500KHz		72.12Hz	
Polarity	(–)		(–)	
Sync Width	1.282us	5 chars	0.080ms	3 lines
Front Porch	0.769us	3 chars	0.240ms	9 lines
Back Porch	4.103us	16 chars	0.747ms	28 lines
Active Time	20.513us	80 chars	12.800ms	480 lines
Blank Time	6.154us	24 chars	1.067ms	40 lines

640x480 75Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		31.50MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x68	V_TOTAL	0x01F3
Screen Display	H_DISP	0x4F	V_DISP	0x01DF
Sync Start	H_SYNC_STRT	0x51	V_SYNC_STRT	0x01E0
Sync Width	H_SYNC_WID	0x28	V_SYNC_WID	0x23
Resolution	640		480	
Scan Frequency	37.500KHz		75.00Hz	
Polarity	(–)		(–)	
Sync Width	2.032us	8 chars	0.080ms	3 lines
Front Porch	0.508us	2 chars	0.027ms	1 lines
Back Porch	3.810us	15 chars	0.427ms	16 lines
Active Time	20.317us	80 chars	12.800ms	480 lines
Blank Time	6.349us	25 chars	0.533ms	20 lines

640x480 90Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		39.91MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x67	V_TOTAL	0x214
Screen Display	H_DISP	0x4F	V_DISP	0x01DF
Sync Start	H_SYNC_STRT	0x53	V_SYNC_STRT	0x01F8
Sync Width	H_SYNC_WID	0x25	V_SYNC_WID	0x2E
Resolution	640		480	
Scan Frequency	47.969KHz		90.00Hz	
Polarity	(–)		(–)	
Sync Width	1.002us	5 chars	0.292ms	14 lines
Front Porch	0.902us	4 chars	0.521ms	25 lines
Back Porch	2.907us	15 chars	0.292ms	14 lines
Active Time	16.036us	80 chars	10.007ms	480 lines
Blank Time	4.811us	24 chars	1.105ms	53 lines

640x480 100Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		44.90MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x69	V_TOTAL	0x0212
Screen Display	H_DISP	0x4F	V_DISP	0x01DF
Sync Start	H_SYNC_STRT	0x53	V_SYNC_STRT	0x01F5
Sync Width	H_SYNC_WID	0x30	V_SYNC_WID	0x2C
Resolution	640		480	
Scan Frequency	52.948KHz		99.71Hz	
Polarity	(–)		(–)	
Sync Width	2.851us	16 chars	0.227ms	12 lines
Front Porch	0.801us	4 chars	0.416ms	22 lines
Back Porch	0.981us	6 chars	0.322ms	17 lines
Active Time	14.254us	80 chars	9.065ms	480 lines
Blank Time	4.633us	26 chars	0.963ms	51 lines

C.3 CRTC Parameters for 800x600

800x600 48Hz Interlaced

CRTC_GEN_CNTL		0X02		
DOT_CLOCK		36.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x84	V_TOTAL	0x2BD
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x6D	V_SYNC_STRT	0x262
Sync Width	H_SYNC_WID	0x10	V_SYNC_WID	0xC
Resolution	800		600	
Scan Frequency	33.835KHz		96.39Hz	
Polarity	(+)		(+)	
Sync Width	3.556us	16 chars	0.177ms	12 lines
Front Porch	2.222us	10 chars	0.163ms	11 lines
Back Porch	1.555us	7 chars	1.167ms	79 lines
Active Time	22.222us	100 chars	8.867ms	600 lines
Blank Time	7.333us	33 chars	1.507ms	102 lines

800x600 56Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		36.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x7F	V_TOTAL	0x270
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x66	V_SYNC_STRT	0x258
Sync Width	H_SYNC_WID	0x9	V_SYNC_WID	0x2
Resolution	800		600	
Scan Frequency	35.156KHz		56.25Hz	
Polarity	(+)		(+)	
Sync Width	2.000us	9 chars	0.057ms	2 lines
Front Porch	0.667us	3 chars	0.028ms	1 lines
Back Porch	3.555us	16 chars	0.626ms	22 lines
Active Time	22.222us	100 chars	17.067ms	600 lines
Blank Time	6.222us	28 chars	0.711ms	25 lines

800x600 60Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		40.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x83	V_TOTAL	0x273
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x68	V_SYNC_STRT	0x258
Sync Width	H_SYNC_WID	0x10	V_SYNC_WID	0x4
Resolution	800		600	
Scan Frequency	37.879KHz		60.32Hz	
Polarity	(+)		(+)	
Sync Width	3.200us	16 chars	0.106ms	4 lines
Front Porch	1.000us	5 chars	0.026ms	1 lines
Back Porch	2.200us	11 chars	0.607ms	23 lines
Active Time	20.000us	100 chars	15.840ms	600 lines
Blank Time	6.400us	32 chars	0.739ms	28 lines

800x600 70Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		44.90MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x7D	V_TOTAL	0x27B
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x66	V_SYNC_STRT	0x260
Sync Width	H_SYNC_WID	0x12	V_SYNC_WID	0x2C
Resolution	800		600	
Scan Frequency	44.544KHz		70.04Hz	
Polarity	(+)		(–)	
Sync Width	3.207us	18 chars	0.269ms	12 lines
Front Porch	0.535us	3 chars	0.202ms	9 lines
Back Porch	0.891us	5 chars	0.337ms	15 lines
Active Time	17.817us	100 chars	13.470ms	600 lines
Blank Time	4.633us	26 chars	0.808ms	36 lines

800x600 72Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	50.00MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x81	V_TOTAL	0x299
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x6A	V_SYNC_STRT	0x27C
Sync Width	H_SYNC_WID	0xF	V_SYNC_WID	0x6
Resolution	800		600	
Scan Frequency	48.090KHz		72.19Hz	
Polarity	(+)		(+)	
Sync Width	2.400us	15 chars	0.125ms	6 lines
Front Porch	1.120us	7 chars	0.769ms	37 lines
Back Porch	1.280us	8 chars	0.478ms	23 lines
Active Time	16.000us	100 chars	12.477ms	600 lines
Blank Time	4.800us	30 chars	1.372ms	66 lines

800x600 75Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	49.50MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x83	V_TOTAL	0x0270
Screen Display	H_DISP	0x63	V_DISP	0x0257
Sync Start	H_SYNC_STRT	0x65	V_SYNC_STRT	0x0258
Sync Width	H_SYNC_WID	0x0A	V_SYNC_WID	0x03
Resolution	800		600	
Scan Frequency	46.875KHz		75.00Hz	
Polarity	(+)		(+)	
Sync Width	1.616us	10 chars	0.064ms	3 lines
Front Porch	0.323us	2 chars	0.021ms	1 lines
Back Porch	3.232us	20 chars	0.448ms	21 lines
Active Time	16.162us	100 chars	12.800ms	600 lines
Blank Time	5.172us	32 chars	0.533ms	25 lines

800x600 90Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		56.64MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x7B	V_TOTAL	0x27A
Screen Display	H_DISP	0x63	V_DISP	0x257
Sync Start	H_SYNC_STRT	0x64	V_SYNC_STRT	0x25F
Sync Width	H_SYNC_WID	0x08	V_SYNC_WID	0x0B
Resolution	800		600	
Scan Frequency	57.097KHz		89.92Hz	
Polarity	(+)		(+)	
Sync Width	1.130us	8 chars	0.193ms	11 lines
Front Porch	0.071us	1 chars	0.140ms	8 lines
Back Porch	2.189us	15 chars	0.280ms	16 lines
Active Time	14.124us	100 chars	10.508ms	600 lines
Blank Time	3.390us	24 chars	0.613ms	35 lines

800x600 100Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		67.50MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x86	V_TOTAL	0x0270
Screen Display	H_DISP	0x63	V_DISP	0x0257
Sync Start	H_SYNC_STRT	0x63	V_SYNC_STRT	0x025E
Sync Width	H_SYNC_WID	0x08	V_SYNC_WID	0x04
Resolution	800		600	
Scan Frequency	62.500KHz		100.00Hz	
Polarity	(+)		(+)	
Sync Width	0.948us	8 chars	0.064ms	4 lines
Front Porch	0.000us	0 chars	0.112ms	7 lines
Back Porch	3.200us	27 chars	0.224ms	14 lines
Active Time	11.852us	100 chars	9.600ms	600 lines
Blank Time	4.148us	35 chars	0.400ms	25 lines

C.4 CRTC Parameters for 1024x768

1024x768 43Hz Interlaced

CRTC_GEN_CNTL		0x02		
DOT_CLOCK		44.90MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x9D	V_TOTAL	0x330
Screen Display	H_DISP	0x7F	V_DISP	0x2FF
Sync Start	H_SYNC_STRT	0x80	V_SYNC_STRT	0x300
Sync Width	H_SYNC_WID	0x16	V_SYNC_WID	0x8
Resolution	1024		768	
Scan Frequency	35.522KHz		86.96Hz	
Polarity	(+)		(+)	
Sync Width	3.920us	22 chars	0.113ms	8 lines
Front Porch	0.178us	1 chars	0.014ms	1 lines
Back Porch	1.247us	7 chars	0.563ms	40 lines
Active Time	22.806us	128 chars	10.810ms	768 lines
Blank Time	5.345us	30 chars	0.690ms	49 lines

1024x768 60Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		65.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xA7	V_TOTAL	0x325
Screen Display	H_DISP	0x7F	V_DISP	0x2FF
Sync Start	H_SYNC_STRT	0x82	V_SYNC_STRT	0x302
Sync Width	H_SYNC_WID	0x31	V_SYNC_WID	0x26
Resolution	1024		768	
Scan Frequency	48.363KHz		60.00Hz	
Polarity	(–)		(–)	
Sync Width	2.092us	17 chars	0.124ms	6 lines
Front Porch	0.369us	3 chars	0.062ms	3 lines
Back Porch	2.462us	20 chars	0.601ms	29 lines
Active Time	15.754us	128 chars	15.880ms	768 lines
Blank Time	4.923us	40 chars	0.786ms	38 lines

1024X768 70Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		75.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xA5	V_TOTAL	0x325
Screen Display	H_DISP	0x7F	V_DISP	0x2FF
Sync Start	H_SYNC_STRT	0x82	V_SYNC_STRT	0x302
Sync Width	H_SYNC_WID	0x31	V_SYNC_WID	0x26
Resolution	1024		768	
Scan Frequency	56.476KHz		70.07Hz	
Polarity	(–)		(–)	
Sync Width	1.813us	17 chars	0.106ms	6 lines
Front Porch	0.320us	3 chars	0.053ms	3 lines
Back Porch	1.921us	18 chars	0.514ms	29 lines
Active Time	13.653us	128 chars	13.599ms	768 lines
Blank Time	4.053us	38 chars	0.673ms	38 lines

1024x768 72Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		75.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xA0	V_TOTAL	0x325
Screen Display	H_DISP	0x7F	V_DISP	0x2FF
Sync Start	H_SYNC_STRT	0x82	V_SYNC_STRT	0x302
Sync Width	H_SYNC_WID	0x31	V_SYNC_WID	0x26
Resolution	10224		768	
Scan Frequency	58.230KHz		72.245Hz	
Polarity	(–)		(–)	
Sync Width	1.813us	17 chars	0.103ms	6 lines
Front Porch	0.320us	3 chars	0.052ms	3 lines
Back Porch	1.387us	13 chars	0.498ms	29 lines
Active Time	13.653us	128 chars	13.189ms	768 lines
Blank Time	3.520us	33 chars	0.653ms	38 lines

1024x768 75Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		78.75MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xA3	V_TOTAL	0x031F
Screen Display	H_DISP	0x7F	V_DISP	0x02FF
Sync Start	H_SYNC_STRT	0x81	V_SYNC_STRT	0x0300
Sync Width	H_SYNC_WID	0x0C	V_SYNC_WID	0x03
Resolution	1024		768	
Scan Frequency	60.023KHz		75.03Hz	
Polarity	(+)		(+)	
Sync Width	1.219us	12 chars	0.050ms	3 lines
Front Porch	0.203us	2 chars	0.017ms	1 lines
Back Porch	2.235us	22 chars	0.466ms	28 lines
Active Time	13.003us	128 chars	12.795ms	768 lines
Blank Time	3.657us	36 chars	0.533ms	32 lines

1024x768 90Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		100.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xA3	V_TOTAL	0x34C
Screen Display	H_DISP	0x7F	V_DISP	0x2FF
Sync Start	H_SYNC_STRT	0x7F	V_SYNC_STRT	0x314
Sync Width	H_SYNC_WID	0x2C	V_SYNC_WID	0x2F
Resolution	1024		768	
Scan Frequency	76.220KHz		90.20Hz	
Polarity	(–)		(–)	
Sync Width	0.960us	12 chars	0.197ms	15 lines
Front Porch	0.000us	0 chars	0.276ms	21 lines
Back Porch	1.920us	24 chars	0.537ms	41 lines
Active Time	10.240us	128 chars	10.076ms	768 lines
Blank Time	2.880us	36 chars	1.010ms	77 lines

1024x768 100Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		110.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xAD	V_TOTAL	0x0317
Screen Display	H_DISP	0x7F	V_DISP	0x02FF
Sync Start	H_SYNC_STRT	0x7F	V_SYNC_STRT	0x02FF
Sync Width	H_SYNC_WID	0x2B	V_SYNC_WID	0x28
Resolution	1024		768	
Scan Frequency	79.023KHz		99.78Hz	
Polarity	(–)		(–)	
Sync Width	0.800us	11 chars	0.101ms	8 lines
Front Porch	0.000us	0 chars	0.000ms	0 lines
Back Porch	2.545us	35 chars	0.202ms	16 lines
Active Time	9.309us	128 chars	9.719ms	768 lines
Blank Time	3.345us	46 chars	0.304ms	24 lines

C.5 CRTC Parameters for 1152x864

1152x864 43Hz Interlaced

CRTC_GEN_CNTL	0x02			
DOT_CLOCK	65.0MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xB0	V_TOTAL	0x041E
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x98	V_SYNC_STRT	0x03AD
Sync Width	H_SYNC_WID	0x10	V_SYNC_WID	0x09
Resolution	1152		864	
Scan Frequency	45.904KHz		87.02Hz	
Polarity	(+)		(+)	
Sync Width	1.969us	16 chars	0.098ms	9 lines
Front Porch	1.062us	9 chars	0.850ms	78 lines
Back Porch	1.031us	8 chars	1.133ms	104 lines
Active Time	17.723us	144 chars	9.411ms	864 lines
Blank Time	4.062us	33 chars	2.080ms	191 lines

1152X864 47Hz Interlaced

CRTC_GEN_CNTL	0x02			
DOT_CLOCK	65.0MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xB4	V_TOTAL	0x03B2
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x9A	V_SYNC_STRT	0x037D
Sync Width	H_SYNC_WID	0x10	V_SYNC_WID	0x09
Resolution	1152		864	
Scan Frequency	44.890KHz		94.80Hz	
Polarity	(+)		(+)	
Sync Width	1.969us	16 chars	0.100ms	9 lines
Front Porch	1.415us	11 chars	0.334ms	30 lines
Back Porch	1.170us	10 chars	0.490ms	44 lines
Active Time	17.723us	144 chars	9.624ms	864 lines
Blank Time	4.554us	37 chars	0.924ms	83 lines

1152X864 60Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		80.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xB5	V_TOTAL	0x0393
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x97	V_SYNC_STRT	0x0365
Sync Width	H_SYNC_WID	0x0E	V_SYNC_WID	0x05
Resolution	1152		864	
Scan Frequency	54.945KHz		59.98Hz	
Polarity	(+)		(+)	
Sync Width	1.400us	14 chars	0.091ms	5 lines
Front Porch	0.800us	8 chars	0.109ms	6 lines
Back Porch	1.600us	16 chars	0.746ms	41 lines
Active Time	14.400us	144 chars	15.725ms	864 lines
Blank Time	3.800us	38 chars	0.946ms	52 lines

1152X864 70Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		100.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xBC	V_TOTAL	0x03B0
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x94	V_SYNC_STRT	0x036C
Sync Width	H_SYNC_WID	0x13	V_SYNC_WID	0x0B
Resolution	1152		864	
Scan Frequency	66.138KHz		69.99Hz	
Polarity	(+)		(+)	
Sync Width	1.520us	19 chars	0.166ms	11 lines
Front Porch	0.390us	5 chars	0.197ms	13 lines
Back Porch	1.690us	21 chars	0.862ms	57 lines
Active Time	11.520us	144 chars	13.064ms	864 lines
Blank Time	3.600us	45 chars	1.225ms	81 lines

1152X864 75Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		110.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xB6	V_TOTAL	0x03E9
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x92	V_SYNC_STRT	0x038C
Sync Width	H_SYNC_WID	0x12	V_SYNC_WID	0x08
Resolution	1152		864	
Scan Frequency	75.137KHz		74.99Hz	
Polarity	(+)		(+)	
Sync Width	1.309us	18 chars	0.106ms	8 lines
Front Porch	0.245us	3 chars	0.599ms	45 lines
Back Porch	1.282us	18 chars	1.132ms	85 lines
Active Time	10.473us	144 chars	11.499ms	864 lines
Blank Time	2.836us	39 chars	1.837ms	138 lines

1152X864 80Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		110.0MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xB3	V_TOTAL	0x03BD
Screen Display	H_DISP	0x8F	V_DISP	0x035F
Sync Start	H_SYNC_STRT	0x91	V_SYNC_STRT	0x037D
Sync Width	H_SYNC_WID	0x0E	V_SYNC_WID	0x07
Resolution	1152		864	
Scan Frequency	76.389KHz		79.74Hz	
Polarity	(+)		(+)	
Sync Width	1.018us	14 chars	0.092ms	7 lines
Front Porch	0.127us	2 chars	0.393ms	30 lines
Back Porch	1.473us	20 chars	0.747ms	57 lines
Active Time	10.473us	144 chars	11.311ms	864 lines
Blank Time	2.618us	36 chars	1.231ms	94 lines

C.6 CRTC Parameters for 1280x1024

1280x1024 43Hz Interlaced

CRTC_GEN_CNTL		0x02		
DOT_CLOCK		80.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xC7	V_TOTAL	0x47C
Screen Display	H_DISP	0x9F	V_DISP	0x3FF
Sync Start	H_SYNC_STRT	0xA9	V_SYNC_STRT	0x431
Sync Width	H_SYNC_WID	0xA	V_SYNC_WID	0xA
Resolution	1024		1024	
Scan Frequency	50.000KHz		87.03Hz	
Polarity	(+)		(+)	
Sync Width	1.000us	10 chars	0.100ms	10 lines
Front Porch	1.000us	10 chars	0.500ms	50 lines
Back Porch	2.000us	20 chars	0.650ms	65 lines
Active Time	16.000us	160 chars	10.240ms	1024 lines
Blank Time	4.000us	40 chars	1.250ms	125 lines

1280x1024 47Hz Interlaced

CRTC_GEN_CNTL		0x02		
DOT_CLOCK		80.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xC7	V_TOTAL	0x41C
Screen Display	H_DISP	0x9F	V_DISP	0x3FF
Sync Start	H_SYNC_STRT	0xA9	V_SYNC_STRT	0x400
Sync Width	H_SYNC_WID	0xA	V_SYNC_WID	0xA
Resolution	1280		1024	
Scan Frequency	50.000KHz		94.97Hz	
Polarity	(+)		(+)	
Sync Width	1.000us	10 chars	0.100ms	10 lines
Front Porch	1.000us	10 chars	0.010ms	1 line
Back Porch	2.000us	20 chars	0.180ms	18 lines
Active Time	16.000us	160 chars	10.240ms	1024 lines
Blank Time	4.000us	40 chars	0.290ms	29 lines

1280x1024 60Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	108.00MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xD2	V_TOTAL	0x429
Screen Display	H_DISP	0x9F	V_DISP	0x3FF
Sync Start	H_SYNC_STRT	0xA5	V_SYNC_STRT	0x400
Sync Width	H_SYNC_WID	0x0E	V_SYNC_WID	0x03
Resolution	1280		1024	
Scan Frequency	63.981KHz		60.02Hz	
Polarity	(+)		(+)	
Sync Width	1.037us	14 chars	0.047ms	3 lines
Front Porch	0.444us	6 chars	0.015ms	1 line
Back Porch	2.297us	31 chars	0.594ms	38 lines
Active Time	11.852us	160 chars	16.005ms	1024 lines
Blank Time	3.778us	51 chars	0.656ms	42 lines

1280x1024 70Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	126.00MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xD2	V_TOTAL	0x429
Screen Display	H_DISP	0x9F	V_DISP	0x3FF
Sync Start	H_SYNC_STRT	0xA9	V_SYNC_STRT	0x400
Sync Width	H_SYNC_WID	0xE	V_SYNC_WID	0x5
Resolution	1280		1024	
Scan Frequency	74.645KHz		70.02Hz	
Polarity	(+)		(+)	
Sync Width	0.889us	14 chars	0.067ms	5 lines
Front Porch	0.635us	10 chars	0.013ms	1 lines
Back Porch	1.714us	27 chars	0.483ms	36 lines
Active Time	10.159us	160 chars	13.718ms	1024 lines
Blank Time	3.238us	51 chars	0.563ms	42 lines

1280x1024 74Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		135.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xD5	V_TOTAL	0x427
Screen Display	H_DISP	0x9F	V_DISP	0x3FF
Sync Start	H_SYNC_STRT	0xA3	V_SYNC_STRT	0x3FF
Sync Width	H_SYNC_WID	0x12	V_SYNC_WID	0x1E
Resolution	1280		1024	
Scan Frequency	78.855KHz		74.11Hz	
Polarity	(+)		(+)	
Sync Width	1.067us	18 chars	0.380ms	30 lines
Front Porch	0.237us	4 chars	0.000ms	0 lines
Back Porch	1.896us	32 chars	0.127ms	10 lines
Active Time	9.481us	160 chars	12.986ms	1024 lines
Blank Time	3.200us	54 chars	0.507ms	40 lines

1280x1024 75Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		135.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xD2	V_TOTAL	0x429
Screen Display	H_DISP	0x9F	V_DISP	0x03FF
Sync Start	H_SYNC_STRT	0xA1	V_SYNC_STRT	0x0400
Sync Width	H_SYNC_WID	0x12	V_SYNC_WID	0x03
Resolution	1280		1024	
Scan Frequency	79.976KHz		75.02Hz	
Polarity	(+)		(+)	
Sync Width	1.067us	18 chars	0.038ms	3 lines
Front Porch	0.119us	2 chars	0.012ms	1 line
Back Porch	1.837us	31 chars	0.475ms	38 lines
Active Time	9.481us	160 chars	12.804ms	1024 lines
Blank Time	3.022us	51 chars	0.525ms	42 lines

C.7 CRTC Parameters for 1600x1200

1600x1200 60Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	156.00MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0xFF	V_TOTAL	0x4F1
Screen Display	H_DISP	0xC7	V_DISP	0x4AF
Sync Start	H_SYNC_STRT	0xCB	V_SYNC_STRT	0x4B9
Sync Width	H_SYNC_WID	0x34	V_SYNC_WID	0x28
Resolution	1600		1200	
Scan Frequency	76.200KHz		60.00Hz	
Polarity	(–)		(–)	
Sync Width	1.026us	20 chars	0.105ms	8 lines
Front Porch	0.205us	4 chars	0.131ms	10 lines
Back Porch	1.636us	32 chars	0.682ms	52 lines
Active Time	10.256us	200 chars	15.748ms	1200 lines
Blank Time	2.872us	56 chars	0.866ms	66 lines

1600x1200 66Hz Non-Interlaced

CRTC_GEN_CNTL	0x00			
DOT_CLOCK	172.00MHz			
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x103	V_TOTAL	0x4DB
Screen Display	H_DISP	0xC7	V_DISP	0x4AF
Sync Start	H_SYNC_STRT	0xCC	V_SYNC_STRT	0x4B2
Sync Width	H_SYNC_WID	0x31	V_SYNC_WID	0x23
Resolution	1600		1200	
Scan Frequency	82.700KHz		66.00Hz	
Polarity	(–)		(–)	
Sync Width	0.791us	17 chars	0.036ms	3 lines
Front Porch	0.233us	5 chars	0.036ms	3 lines
Back Porch	1.767us	38 chars	0.567ms	47 lines
Active Time	9.302us	200 chars	14.512ms	1200 lines
Blank Time	2.791us	60 chars	0.532ms	44 lines

1600x1200 76Hz Non-Interlaced

CRTC_GEN_CNTL		0x00		
DOT_CLOCK		198.00MHz		
	Horizontal		Vertical	
Screen Total	H_TOTAL	0x103	V_TOTAL	0x4E1
Screen Display	H_DISP	0xC7	V_DISP	0x4AF
Sync Start	H_SYNC_STRT	0xCC	V_SYNC_STRT	0x4B2
Sync Width	H_SYNC_WID	0x31	V_SYNC_WID	0x25
Resolution	1600		1200	
Scan Frequency	95.200KHz		76.00Hz	
Polarity	(–)		(–)	
Sync Width	0.687us	17 chars	0.052ms	5 lines
Front Porch	0.202us	5 chars	0.032ms	3 lines
Back Porch	1.535us	38 chars	0.441ms	42 lines
Active Time	8.081us	200 chars	12.606ms	1200 lines
Blank Time	2.424us	60 chars	0.525ms	50 lines

This page intentionally left blank.

Appendix D
Clock Chip Reference

D.1 Clock Chip

ATI18818	
Select	**PCLK (MHz)**
0h	50.35 or 25.18
1h	56.64 or 28.32
2h	63.00 or 31.50
3h	72.00 or 36.00
4h	40.00
5h	44.90
6h	49.50
7h	50.00
8h	Reserved
9h	110.00
Ah	126.00
Bh	135.00
Ch	Reserved
Dh	80.00
Eh	75.00
Fh	65.00

Note: The first four selections on the 18818 clock chip are programmable. The values stated in the table for those selections are one of two possible values programmed by the adapter BIOS at boot time.

To determine the type of clock chip present on the adapter, query the adapter BIOS and deduce which chip is being used from the enumerated mode table information. Otherwise, the host must set up a dummy video mode with one of the differing frequency selections, and determine the time between vertical sync pulses.

This page intentionally left blank.

Appendix E
Register Summary

E.1 Introduction

The *mach64* ASIC has the following five register classes:

- VGA registers
- Setup and control registers
- Accelerator CRTC and DAC registers
- Draw engine context control registers
- Draw engine trajectory control registers

E.2 Memory Mapping

All memory-mapped registers reside at offset 3FFC00h from the base aperture address on 4M boards or 7FFC00h on 8M boards. If VGA is enabled, these registers are aliased at offset FC00h from segment B000h.

All memory offsets in the tables following are DWORD offsets.

E.3 I/O Mapping

There are two I/O addressing types: standard and relocatable. The standard type I/O base address will be 2ECh, 1CCh or 1C8h. The physical I/O address is calculated as (I/O select <<10) + I/O base address. Relocatable I/O (PCI only), if enabled, uses the MM select to describe the register's physical I/O address. It is calculated as (MM select << 2) + I/O base address.

E.4 VGA Registers

VGA registers are completely segregated from the accelerator registers. Their functions are mutually exclusive. They are addressed at I/O ports 1CE-1CFh, 3B0-3BFh, 3C0-3CFh, 3D0-3DFh. (*See mach64 VGA Register Guide for more details*.)

E.5 Setup and Control Registers

Setup and control registers are memory-mapped, and are also aliased at the I/O base address. Most of these registers are initialized once only, at boot time.

Setup and Control Registers			
Name	**I/O Select**	**Memory Offset**	**R/W**
SCRATCH_REG0	10h	20h	R/W
SCRATCH_REG1	11h	21h	R/W
BUS_CNTL	13h	28h	R/W
MEM_CNTL	14h	2Ch	R/W
MEM_VGA_WP_SEL	15h	2Dh	R/W
MEM_VGA_RP_SEL	16h	2Eh	R/W
GEN_TEST_CNTL	19h	34h	R/W
CONFIG_CNTL	1Ah	37H*	R/W
CONFIG_CHIP_ID	1Bh	38h	R
CONFIG_STAT	1Ch	39h	R
CONFIG_STAT	1Dh	3Ah	R

* Except for *mach64*CX, GX-E and earlier.

E.6 Accelerator CRTC and DAC registers

The CRTC and DAC registers are memory-mapped, and are also aliased at the I/O base address. The accelerator CRTC registers are separate from the VGA CRTC registers.

Accelerator CRTC and DAC Registers			
Name	**I/O Select**	**Memory Offset**	**R/W**
CRTC_H_TOTAL_DISP	0h,1Fh	0h	R/W
CRTC_H_SYNC_STRT_WID	1h	1h	R/W
CRTC_V_TOTAL_DISP	2h	2h	R/W
CRTC_V_SYNC_START_WID	3h	3h	R/W
CRTC_VLINE_CRNT_VLINE	4h	4h	R/W
CRTC_OFF_PITCH	5h	5h	R/W
CRTC_INT_CNTL	6h	6h	R/W
CRTC_GEN_CNTL	7h	7h	R/W
OVR_CLR	8h	10h	R/W
OVR_WID_LEFT_RIGHT	9h	11h	R/W

Accelerator CRTC and DAC Registers (Continued)			
Name	**I/O Select**	**Memory Offset**	**R/W**
OVR_WID_TOP_BOTTOM	Ah	12h	R/W
CUR_CLR0	Bh	18h	R/W
CUR_CLR1	Ch	19h	R/W
CUR_OFFSET	Dh	1Ah	R/W
CUR_HORZ_VERT_POSN	Eh	1Bh	R/W
CUR_HORZ_VERT_OFF	Fh	1Ch	R/W
CLOCK_SEL_CNTL	12h	24h	R/W
DAC_REGS	17h	30h	R/W
DAC_CNTL	18h	31h	R/W

E.7 Draw Engine Context Control Registers

Draw engine context control registers are memory-mapped. They are used to set up the draw engine data path and destination mixing logic.

Draw Engine Context Control and Status Registers			
Name	**Offset**	**R/W**	**Description**
PAT_REG0	A0h	R/W	Pattern register 0, for fixed patterns
PAT_REG1	A1h	R/W	Pattern register 1, for fixed patterns
PAT_CNTL	A2h	R/W	Pattern control, for enabling fixed patterns
SC_LEFT	A8h	R/W	Scissor left
SC_RIGHT	A9h	R/W	Scissor right
SC_LEFT_RIGHT	AAh	W	Scissor left and right
SC_TOP	ABh	R/W	Scissor top
SC_BOTTOM	ACh	R/W	Scissor bottom
SC_TOP_BOTTOM	ADh	W	Scissor top and bottom
DP_BKGD_CLR	B0h	R/W	Background color
DP_FRGD_CLR	B1h	R/W	Foreground color
DP_WRITE_MASK	B2h	R/W	Write mask
DP_CHAIN_MASK	B3h	R/W	ALU carry chain mask (only used for (D+S)/2)
DP_PIX_WIDTH	B4h	R/W	Pixel width, for setting source, destination, and host pixel widths
DP_MIX	B5h	R/W	Mix, for setting foreground and background mixes
DP_SRC	B6h	R/W	Source, for setting mono source, and foreground and background sources
CLR_CMP_CLR	C0h	R/W	Compare color

Draw Engine Context Control and Status Registers (Continued)			
Name	**Offset**	**R/W**	**Description**
CLR_CMP_MSK	C1h	R/W	Compare mask
CLR_CMP_CNTL	C2h	R/W	Compare control, for setting compare function and compare source
FIFO_STAT	C4h	R	FIFO status
CONTEXT_MASK	C8	R/W	Context load mask, for selectively loading draw engine registers
CONTEXT_SAVE_PTR	CAh	R/W	Context save pointer, in units of 64 DWORDs. Writing to this register saves the engine context
CONTEXT_LOAD_CNTL	CBh	R/W	Context load pointer, for restoring an engine context with an option to perform a draw operation
GUI_TRAJ_CNTL	CCh	R/W	Trajectory control, for setting DST_CNTL, SRC_CNTL, HOST_CNTL, and PAT_CNTL with a single memory access
GUI_STAT	CEh	R	Engine status

E.8 Draw Engine Trajectory Control Registers

Draw engine trajectory control registers are memory-mapped. They are used to set up the source and destination trajectories and to initiate draw operations.

Draw Engine Trajectory Control Registers				
Name	**Offset**	**R/W**	**Init.**	**Description**
DST_OFF_PITCH	40h	R/W		Destination offset and pitch
DST_X	41h	R/W		Destination X
DST_Y	42h	R/W		Destination Y
DST_Y_X	43h	W		Destination Y and X
DST_WIDTH	44h	R/W	Yes	Destination width
DST_HEIGHT	45h	R/W		Destination height
DST_HEIGHT_WIDTH	46h	W	Yes	Destination height and width.
DST_X_WIDTH	47h	W	Yes	Destination X and width
DST_BRES_LNTH	48h	R/W	Yes	Bresenham line length
DST_BRES_ERR	49h	R/W		Bresenham error term
DST_BRES_INC	4Ah	R/W		Bresenham axial step term
DST_BRES_DEC	4Bh	R/W		Bresenham diagonal step term

Draw Engine Trajectory Control Registers (Continued)				
Name	**Offset**	**R/W**	**Init.**	**Description**
DST_CNTL	4Ch	R/W		Destination control, for setting destination trajectory direction, destination side effects, line and polygon options, and packed 24 bit initial rotation value
SRC_OFF_PITCH	60h	R/W		Source offset and pitch
SRC_X	61h	R/W		Source X
SRC_Y	62h	R/W		Source Y
SRC_Y_X	63h	W		Source Y and source X
SRC_WIDTH1	64h	R/W		Source width1 — for setting the source width for unbounded Y and general pattern sources, for setting the minor source width for general patterns with rotation
SRC_HEIGHT1	65h	R/W		Source height1 — for setting the source height for general pattern sources, for setting the minor source height for general patterns with rotation
SRC_HEIGHT1_WIDTH1	66h	W		Source height1 and source width1
SRC_X_START	67h	R/W		Source X start — for setting the starting X location for general patterns with rotation
SRC_Y_START	68h	R/W		Source Y start — for setting the starting Y location for general patterns with rotation
SRC_Y_X_START	69h	W		Source Y start and source X start
SRC_WIDTH2	6Ah	R/W		Source width2 — for setting the major source width for general patterns with rotation
SRC_HEIGHT2	6Bh	R/W		Source height2 — for setting the major source height for general patterns with rotation
SRC_HEIGHT2_WIDTH2	6Ch	W		Source height 2 and source width2
SRC_CNTL	6Dh	R/W		Source control — for setting source trajectory type and source trajectory modifiers
HOST_DATA[15:0]	80:8Fh	W		Host registers 0 to 15 are identical but mapped to 16 separate locations
HOST_CNTL	90h	R/W		Host control — for setting host consumption modifiers

This page intentionally left blank.

Appendix F
Programming PLL Registers in mach64 CT Family

F.1 Introduction

CLOCK_CNTL [R/W] (MM:24, I/O:12, 49C8, 49CC, 4AEC)		
Field Name	**Bit(s)**	**Description**
CLOCK_SEL	3:0	Non-VGA mode video clock frequency select. Internal clock synthesizer (PLL) uses only bits 1 and 0. External clock chip uses all four bits. In VGA mode, clock select is determined by GENMO(3:2).
(Reserved)	5:4	
CLOCK_STROBE	6	Controls STROBE signal to external clock synthesizer.
(Reserved)	7:8	
PLL_WR_EN	9	Internal clock synthesizer (PLL) register write enable. 0 = PLL_DATA is read-only 1 = PLL_DATA is read/write
PLL_ADDR	13:10	Selects register in internal clock synthesizer (PLL) to read or write.
(Reserved)	15:14	
PLL_DATA	23:16	Internal clock synthesizer (PLL) read/write data. (see PLL_WR_EN)
(Reserved)	31:24	

F.2 PLL Registers

The PLL registers on the next page are accessed indirectly through the CLOCK_CNTL register above. Example reads and writes of the PLL registers are given below. The address CLOCK_CNTL0 represents bits 7:0, CLOCK_CNTL1 bits 15:8, and CLOCK_CNTL2 bits 23:16.

PLL Register Read

iow8 CLOCK_CNTL1 PLL_ADDR; PLL address to read (PLL_WR_EN = 0)
ior8 CLOCK_CNTL2 PLL_DATA; data is put into variable PLL_DATA

PLL Register Write

iow8 CLOCK_CNTL1 PLL_ADDR | PLL_WR_EN; PLL address to write and PLL_WR_EN = 1

iow8 CLOCK_CNTL2 PLL_DATA; PLL data to write

Note that only 8-bit I/O or memory read and write operations are recommended for PLL register reads and writes.

PLL Registers				
Addr	**Register Name**	**Field**	**Bits**	**Function**
0	Reserved			
1	PLL_MACRO_CNTL			Controls to analog PLL macro (default = D4h)
		PLL_PC_GAIN	2:0	Charge-pump gain setting
		PLL_VC_GAIN	4:3	VCGEN gain setting
		PLL_DUTY_CYC	7:5	Duty cycle control for pixel clock PLL
2	PLL_REF_DIV		7:0	Reference divider setting (default = 36h) Note: There are only 6 bits in SGS CT-C2.
3	PLL_GEN_CNTL			MCLK and general control (default = 4Fh)
		PLL_OVERRIDE	0	Power-down PLL or ext. bias PLL
		PLL_MCLK_RST	1	Reset MCLK PLL
		OSC_EN	2	Oscillator enable
		EXT_CLK_EN	3	Force EXTFREQ0 to input
		MCLK_SRC_SEL	6:4	
		EXT_CLK_CNTL	7	Enable CLKSEL and CLKSTRb outputs for external clock chip. Note: EXT_CLK_CNTL not in SGS CT-C2. EXT_CLK_EN does both functions.
4	MCLK_FB_DIV		7:0	MCLK feedback divider (default = 97h, 40MHz)
5	PLL_VCLK_CNTL			Pixel clock control (default = 04h)
		VCLK_SRC_SEL	1:0	00 : VCLK = CPUCLK 01 : VCLK = EXTFREQ0 10 : VCLK = XTALIN 11 : VCLK = PLLVCLK
		PLL_VCLK_RST	2	Reset VCLK PLL
		VCLK_INVERT	3	Invert VCLK to get opposite duty cycle

PLL Registers (Continued)				
Addr	**Register Name**	**Field**	**Bits**	**Function**
5	PLL_VCLK_CNTL	Reserved	7:4	
6	VCLK_POST_DIV			Post dividers for VCLK 0-3 (default = 6Ah)
		VCLK0_POST	1:0	Post divider for VCLK setting 0
		VCLK1_POST	3:2	Post divider for VCLK setting 1
		VCLK2_POST	5:4	Post divider for VCLK setting 2
		VCLK3_POST	7:6	Post divider for VCLK setting 3
7	VCLK0_FB_DIV		7:0	Feedback divider for VCLK 0 (default = BEh)
8	VCLK1_FB_DIV		7:0	Feedback divider for VCLK 1 (default = D6h)
9	VCLK2_FB_DIV		7:0	Feedback divider for VCLK 2 (default = EEh)
10	VCLK3_FB_DIV		7:0	Feedback divider for VCLK 3 (default = 88h)
11:13	Reserved			
14	PLL_TEST_CRTL		7:0	PLL test mode control (forced to 00h when not in PLL test mode from GEN_TEST_CTRL register).
15	PLL_TEST_COUNT		7:0	PLL test mode counter (read only, no default)

Notes:

1. PLL_MACRO_CNTL settings control gain and duty cycle of analog PLL's. Gain bits affect lock and jitter of PLL's. This register should only be adjusted by the BIOS.
2. The reference divider setting must be in the range of 2h to FFh.
3. Oscillator enable is only supported in NEC foundry due to limitations in oscillator macro cells. Oscillator will always run in other foundries, no matter how this bit is set.
4. Suggested range for feedback dividers is 80h to FFh. Lower settings result in coarser control of output frequency and possibility of clock jitter. Feedback dividers below 02h will not function.
5. Pixel clock (VCLK) post-divider values are: 00=divide-by-1; 01=divide-by-2; 10=divide-by-4; 11=divide-by-8.
6. All clock sources can be programmed to exceed the frequency limitations of the hardware. Do not attempt to program the PLL registers without a good understanding of the frequency limitations of all clock nets.
7. PLL_TEST_CTRL and PLL_TEST_COUNT are used only during manufacturing tests of analog PLL's.

F.3 Clock Sources

All clock signals in *mach64*CT are derived from three master clocks — Bus Clock (CPUCLK), MCLK and VCLK. MCLK and VCLK each has four different source choices. These include internal PLLs (PLLMCLK and PLLVCLK), external clock pins (CPUCLK and EXTFREQ0 or EXTFREQ1), XTALIN pin and the PLL reference (PLLREFCLK) which XTALIN/reference divider setting. When RESETb goes active, all clocks will switch to using CPUCLK as their source. After reset, either the test vectors will select external sources or the BIOS will select internal sources.

F.4 External Clock Support

The external clock sources are supported by *mach64*CT, primarily for testing but also on a board if required. The control signals for the external clock chip are multiplexed on the feature connector pins. The feature connector may not be used when the external clock sources are active.

Switching to external clocks is done as follows:

1. Disable the feature connector (DAC_FEA_CON_EN@DAC_CNTL, defaults to disabled).

2. Set EXT_CLK_EN@PLL_GEN_CNTL = 1 to enable external clock support pins (defaults to high).

3. Make sure the external clock signals are being driven into the chip.

4. Set MCLK_SRC_SEL@PLL_GEN_CNTL = 101 for EXTFREQ1 as MCLK. Also set VCLK_SRC_SEL@PLL_VCLK_CNTL = 01 for EXTFREQ0 as VCLK.

Switching to internal clocks at boot time is done as follows:

1. Program reference, feedback and VCLK post dividers to the desired settings.

2. Write to PLL_GEN_CNTL, setting PLL_OVERRIDE = 0, PLL_MCLK_RST = 0 and OSC_EN = 1.

3. Write to PLL_VCLK_CNTL, setting PLL_VCLK_RST = 0.

4. Allow 5ms for internal PLL to lock frequencies.

5. Set MCLK_SRC_SEL@PLL_GEN_CNTL = 001.

6. Set VCLK_SRC_SEL@PLL_VCLK_CNTL = 11.

7. Wait a few cycles (1 microsecond).

8. Set EXT_CLK_EN@PLL_GEN_CNTL = 0 to disable external clock support pins.

9. Enable the feature connector (DAC_FEA_CON_EN@DAC_CNTL = 1).

F.5 Frequency Limits

The design of *mach64*CT imposes the following limits on the clock source frequencies:

- MCLK may not exceed 68MHz or the limit imposed by memory type.
- VCLK is limited by the current display mode:
  - In VGA, it may not exceed 80MHz.
  - In 4bpp & 8bpp, it may be up to 135MHz.
  - In 15 to 32bpp, it may not exceed 80MHz.
- CPUCLK may not exceed 33MHz.

The clock going out the feature connector (DCLK) may not exceed 40MHz according to the VESA specification. In practice, a higher limit (possibly 80MHz) will be attempted. When VCLK is set to exceed the limit, then DAC_FEA_CON_EN@DAC_CNTL must be set low to turn off the feature connector.

F.6 Frequency Synthesis Description

To generate a specific output frequency, the reference (M), feedback (N), and post dividers (P) must be loaded with the appropriate divide-down ratios. The internal PLLs for CT and ET are optimized to lock to output frequencies in the range from 135 MHz to 68 MHz. The PLLs for other members of the *mach64*CT family are optimized to lock with output frequencies from 100 MHz to 200 MHz. Setting the PLLs to lock outside these ranges can result in increased jitter or total mis-function (no lock).

The PLLREFCLK lower limit is found based on the upper limit of the PLL lock range (e.g. 135 MHz) and the maximum feedback divider (255) as follows:

Minimum PLLREFCLK = 135 MHz / (2 * 255) = 265 kHz

This is then used to find the reference divider based on the XTALIN frequency.

XTALIN is normally 14.318 MHz and the maximum reference divider M is found by:

M = Floor[14.318 MHz / 265 kHz] = 54
(the Floor function means round down)

Using the maximum reference divider allowed (in this case is 54) ensures the best clock step resolution. However, lower reference dividers might be used to improve clock jitter.

Feedback dividers (N) should kept in the range 80h to FFh. The effective feedback divider is twice the register setting due to the structure of the internal PLL. The post divider (P) may be either 1, 2, 4, or 8.

To determine the N and P values to program for a target frequency, follow the procedure below (where R is the frequency of XTALIN and T is the target frequency):

1. Calculate the value of P. Find the value of Q from the equation below and use it to find P in the following table:

Q = (T * M) / (2 * R)

Q Range	Result
more than 255	M too big
127.5 to 255	P = 1
63.5 to 127.5	P = 2
31.5 to 63.5	P = 4
16 to 31.5	P = 8
less than 16	M too small

2. Calculate the value of N by using the value of P obtained in step 1. N is given by:

N = Q * P
The result N is rounded to the nearest whole number.

3. Determine the actual frequency. Given P and the rounded-off N, the actual output frequency is found by:

Output_Frequency = (2 * R * N) / (M * P)

For example:

If R = 14.318 MHz and M = 54, then Q = 75.43 (if the desired frequency is 40MHz). The table indicates P = 2 for this Q value. The calculation of N = Q*P gives 150.85 and rounding up gives N = 151. The final output frequency is therefore 40.04MHz.

The maximum frequency that can be synthesized is the upper limit of PLL lock range for the specific version of *mach64*CT. It may be 135, 160, 200, or 240 MHz. The minimum

frequency that can be synthesized depends on the largest post divider available. For VCLK, P = 8 is always available and minimum VCLK = (2*R*128)/(M*8). For MCLK, post divider settings of 4 and 8 are not available on some versions of the controller. The minimum frequency setting for MCLK is limited to the correspondingly higher values for these controllers.

Sample divider settings for typical Pixel and Memory clock frequencies when R = 14.318 MHz and M = 54:

Target Freq. (MHz)	Post Divider P	Feedback Register N	Feedback Register N	Actual Freq. (MHz)	Percent Error (%)
135	1	255	FFh	135.23	0.17
126	1	238	EEh	126.21	0.17
110	1	207	CFh	109.77	0.21
100	1	189	BDh	100.23	0.23
92.4	1	174	AEh	92.27	0.14
80	1	151	97h	80.08	0.1
75	1	141	8Dh	74.77	0.31
65	2	245	F5h	64.96	0.06
56.6	2	213	D5h	56.48	0.21
50.2	2	189	BDh	50.11	0.19
49.95	2	188	BCh	49.85	0.2
45	2	170	AAh	45.08	0.18
44.95	2	170	AAh	45.08	0.29
40	2	151	97h	40.04	0.1
36	2	136	88h	36.06	0.17
32.97	4	249	F9h	33.01	0.12
32	4	241	F1h	31.95	0.16
31.5	4	238	EEh	31.55	0.16
28.322	4	214	D6h	28.37	0.17
25.175	4	190	BEh	25.19	0.06

F.7 Duty Cycle Control

The DAC clock (VCLK) is the fastest clock on a *mach64*CT chip. When displayed in 1280x1024 or higher resolutions, VCLK will exceed 100 MHz. The DAC circuitry is sensitive to the duty cycle of VCLK in this range. Duty cycle adjustment for VCLK is available through PLL_DUTY_CYC@PLL_MACRO_CNTL and VCLK_INVERT@PLL_VCLK_CNTL. The CT also has VCLK_D_CYC@PLL_VCLK_CNTL, but these bits should not be used (leave at 00).

The optimal settings for the duty cycle control bits have been determined by ATI during testing under extreme conditions of temperature and voltage. The BIOS sets the proper values for each version of *mach64*CT. There should be no need to change these settings.

F.8 PLL Gain Settings

The internal PLLs have two settings that affect their gain characteristics. These are set by PLL_PC_GAIN and PLL_VC_GAIN in the PLL_MACRO_CNTL register. They will affect optimal lock ranges and jitter characteristics. ATI has determined the optimal settings for these bits under extreme operating conditions. The BIOS sets these bits to optimal values for each version of *mach64*CT. There should not be any need to modify these values.

Books

Foley, James D., van Dam, Andries, Feiner, Steven K. and Hughes, John F., *Computer Graphics: Principles and Practice* (2nd ed.), Reading, Massachusetts: Addison-Wesley, 1990, ISBN 0-201-12110-7

While not directly related to PC graphics programming, Foley/van Dam provide a good overview into the fundamentals of Computer Graphics as a general subject. The text is mostly theoretical with some pseudocode but no working code examples. There is, however, a fairly good derivation of Bresenham's Line drawing algorithm which is used by most hardware graphics accelerators.

Ferraro, Richard F., *Programmer's Guide to the EGA, VGA, and Super VGA Cards* (3rd ed.), Reading, Massachusetts: Addison-Wesley, 1994, ISBN 0-201-62490-7

A very handy book in understanding the details of programming for VGA and SVGA cards. The third edition also covers programming for some Graphics Accelerator boards including the IBM 8514/A and ATI's own *mach32* series. This book provides very good descriptions of all of the VGA's registers and contains numerous small code examples in both C and 80x86 Assembly language. Highly recommended.

Abrash, Michael, *Zen of Graphics Programming*, Scottsdale, Arizona: Coriolis Group Books, 1995, ISBN 1-883577-08-X

Abrash, who is known for his earlier book *The Zen of Code Optimization* as well as for his column in *Dr. Dobb's Journal*, discusses optimized programming techniques for VGA cards. The book comes with a diskette full of examples. Although he specifically avoids SVGA and accelerators his coverage of the plain VGA's full capabilities is thorough. This book also contains a section on Mode X programming.

Wilton, Richard, *Programmer's Guide to PC Video Systems* (2nd ed.), Redmond, Washington: Microsoft Press, 1994, ISBN 1-55615-641-3

One of the classic references on PC Graphics Adapters at the hardware level. The second addition also contains topics covering VGA 256-color graphics programming, animation, 32-bit graphics programming, and the VESA BIOS Extension (VBE) for SVGA graphics programming. The book also comes with a companion diskette with source code examples.

This page intentionally left blank.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V